ADVANCES IN APPLIED MATHEMATICS 12, 337–357 (1991)

A Time-Efficient, Linear-Space Local Similarity Algorithm*

XIAOQIU HUANG

Department of Computer Science, Michigan Technological University, Houghton, Michigan 49931

AND

WEBB MILLER

Department of Computer Science, The Pennsylvania State University, University Park, Pennsylvania 16802

Dynamic programming algorithms to determine similar regions of two sequences are useful for analyzing biosequence data. This paper presents a time-efficient algorithm that produces k best "non-intersecting" local alignments for any chosen k. The algorithm's main strength is that it needs only $O(M + N + K)$ space, where M and N are the lengths of the given sequences and K is the total length of the computed alignments. © 1991 Academic Press, Inc.

1. INTRODUCTION

Local alignment algorithms locate matching segments within two sequences (Sellers [10]); this contrasts with global alignment methods, which align entire sequences including unconserved regions. A number of dynamic programming algorithms have been designed to produce local alignments (Goad and Kanehisa [2], Hall and Myers [4], Sellers [11], Smith

*This publication was supported in part by Grant R01 LM05110 from the National Library of Medicine.

0196-8858/91 $7.50

and Waterman [13], Waterman and Eggert [15]). Waterman [14] offers the opinion that such an algorithm is "probably the most useful dynamic programming algorithm for current problems in biology."

In many contexts, a major drawback of current dynamic programming algorithms for local sequence alignment is the need for space proportional to the product of the two sequence lengths. For *global* alignments, there exists a dynamic programming algorithm that requires only space proportional to the sum of the sequence lengths (Hirschberg [5], Myers and Miller [7]).

Very recently, Huang *et al.* [6] used the Myers–Miller global alignment algorithm as part of a local alignment algorithm. The resulting method is a "linear-space algorithm" in the sense that it needs only space proportional to the sum of the input size and the output size. Huang *et al.* illustrate the method's utility by computing 100 best non-intersecting local alignments of a 73,360-nucleotide sequence containing the human β-like globin cluster and a corresponding 44,594-nucleotide sequence from rabbit. Such a problem is completely outside the scope of quadratic-space algorithms, i.e., those requiring space proportional to the product of the sequence lengths, since 73360×44595 is over three billion. Huang *et al.* also point out certain advantages of their method over the widely-used and faster, but less rigorous, LFASTA program (Pearson and Lipman [9]). Unfortunately, the simple method that they present is very slow. For each of the k alignments it makes an $O(MN)$-time sweep of the entire dynamic-programming matrix, so the total time is $O(kMN)$. For example, each of the 100 local alignments for the β-globin sequences required 5 or 6 h on a Sun 4 workstation.

Here we improve the algorithm of Huang *et al.* by lowering its time requirement, while retaining its space efficiency. Under reasonable assumptions, the new algorithm's running time is $O(MN + \Sigma_{n=1}^{k} L_n^2)$, where L_n is the length of the nth computed alignment. This represents a considerable savings when both k is large and most of the computed alignments are short compared to the original sequences. For instance, the new algorithm finds the 100 local alignments for the β-globin sequences mentioned above in about 15 h on a Sun 4, a definite improvement over the three weeks required by the simpler algorithm.

The remainder of the paper is organized as follows. In Section 2 we review the local similarity algorithm of Smith and Waterman [13] as extended by Waterman and Eggert [15]. To prepare for later developments, this method is explained in terms of an algorithm for finding optimal paths in a certain graph, following Myers and Miller [8]. Section 3 presents the new linear-space algorithm. An example is given in Section 4 to verify the need for particular care in the algorithm's formulation. Section 5 closes by discussing a program that implements the algorithm.

An implementation in the C language of the program described in this paper is freely available from the authors. The simplest way to obtain the program is by electronic mail from huang@cs.mtu.edu or webb@cs.psu.edu.

2. Waterman's Approach

The sequences A and B consist of symbols chosen from an alphabet Σ. Let ε, a unique symbol not in Σ, denote the sequence of zero symbols. Then ε is the identity element with respect to the concatenation of sequences, i.e., if v and w are sequences, then $v\varepsilon w = vw$.

An *aligned pair* has the form $\left[\begin{smallmatrix} a \\ b \end{smallmatrix}\right]$, where $a, b \in \Sigma \cup \{\varepsilon\}$. An *alignment* is a finite sequence of aligned pairs. An alignment *S aligns A and B* if A is the concatenation of upper elements of S and B is the concatenation of lower elements. For example, the sequence of four aligned pairs $\left[\begin{smallmatrix} p \\ x \end{smallmatrix}\right]\left[\begin{smallmatrix} \varepsilon \\ y \end{smallmatrix}\right]\left[\begin{smallmatrix} q \\ \varepsilon \end{smallmatrix}\right]\left[\begin{smallmatrix} r \\ z \end{smallmatrix}\right]$ aligns pqr and xyz. The *null pair*, $\eta = \left[\begin{smallmatrix} \varepsilon \\ \varepsilon \end{smallmatrix}\right]$, acts as the identity element for the concatenation of alignments, i.e., if α and β are alignments, then $\alpha\eta\beta$ and $\alpha\beta$ are considered equal. Without loss of generality, assume that alignments consist of non-null aligned pairs.

An aligned pair with upper entry ε is called an *insertion pair*, and one with lower entry ε is a *deletion pair*. Within a given alignment, an *insertion gap* is a contiguous subsequence of insertion pairs delimited by non-insertion pairs or an end of the alignment. A *deletion gap* is similarly defined, and collectively such blocks are called *gaps*. For example, the alignment $\left[\begin{smallmatrix} \varepsilon \\ x \end{smallmatrix}\right]\left[\begin{smallmatrix} \varepsilon \\ y \end{smallmatrix}\right]\left[\begin{smallmatrix} q \\ \varepsilon \end{smallmatrix}\right]\left[\begin{smallmatrix} r \\ z \end{smallmatrix}\right]$ has an insertion gap of length two and a deletion gap of length one.

A score is assigned to an alignment based on a user-specified scoring function σ that assigns a real-valued cost to each possible non-null aligned pair, and on a *gap penalty* $g > 0$. The score of an alignment S is simply the sum of the costs of each aligned pair in it minus a penalty g for each gap, i.e., score $(S) = \Sigma\{\sigma(\pi)\colon \pi$ is an aligned pair of $S\} - g \times$ (number of gaps in S). For local similarity problems, the "average" weight of an aligned pair should be negative so that a random extension of an alignment will lower the score. For instance, with DNA sequences, which have a four-letter alphabet, we might set $\sigma\left(\left[\begin{smallmatrix} a \\ a \end{smallmatrix}\right]\right) = 1$ for all $a \in \Sigma$ and $\sigma\left(\left[\begin{smallmatrix} a \\ b \end{smallmatrix}\right]\right) = -1$ if $b \neq a$. In addition, assume that all insertion and deletion pairs have negative weight.

The above definition is slightly more general than the traditional approach to affine gap penalties (Gotoh [3]). Beside charging g for opening a gap, we assess potentially different penalties for each kind of insertion or

deletion pair; traditionally, each pair in the gap is charged a fixed "gap-extension" penalty.

For sequences $A = a_1 a_2 \cdots a_M$ and $B = b_1 b_2 \cdots b_N$, the *alignment graph*, $G_{A,B}$, is an edge-labeled directed graph, defined as follows. G has $3(M+1)(N+1)$ vertices, denoted $(i,j)_C$, $(i,j)_D$, and $(i,j)_I$, where $i \in [0, M]$ and $j \in [0, N]$. (We use $[x, y]$ to denote the set of integers t such that $x \le t \le y$.) The following edges, and only these edges, are in $G_{A,B}$:

1. If $i \in [1, M]$ and $j \in [0, N]$, then there is a *deletion* edge $(i-1, j)_D \to (i, j)_D$ labeled $\begin{bmatrix} a_i \\ \varepsilon \end{bmatrix}$.

2. If $i \in [1, M]$ and $j \in [0, N]$, then there is a *deletion initiation* edge $(i-1, j)_C \to (i, j)_D$ labeled $\begin{bmatrix} a_i \\ \varepsilon \end{bmatrix}$.

3. If $i \in [0, M]$ and $j \in [1, N]$, then there is an *insertion* edge $(i, j-1)_I \to (i, j)_I$ labeled $\begin{bmatrix} \varepsilon \\ b_j \end{bmatrix}$.

4. If $i \in [0, M]$ and $j \in [1, N]$, then there is an insertion initiation edge $(i, j-1)_C \to (i, j)_I$ labeled $\begin{bmatrix} \varepsilon \\ b_j \end{bmatrix}$.

5. If $i \in [1, M]$ and $j \in [1, N]$, then there is a *substitution* edge $(i-1, j-1)_C \to (i, j)_C$ labeled $\begin{bmatrix} a_i \\ b_j \end{bmatrix}$.

6. If $i \in [0, M]$ and $j \in [0, N]$, then there is a *null* edge $(i, j)_D \to (i, j)_C$ labeled η.

7. If $i \in [0, M]$ and $j \in [0, N]$, then there is a *null* edge $(i, j)_I \to (i, j)_C$ labeled η.

Note that the D-vertices in row 0 and the I-vertices in column 0 are not reachable from other vertices.

Figure 1 illustrates $G_{ab,baa}$. Rows of the graph after the top-most row are labeled with successive entries of the first sequence; columns after the left-most column are labeled with entries of the second sequence. All edges not annotated are null edges labeled η.

Let $A[m..i]$ denote the sequence $a_{m+1} a_{m+2} \cdots a_i$ and let $B[n..j]$ denote $b_{n+1} b_{n+2} \cdots b_j$. A path in $G_{A,B}$ from $(m, n)_c$ to $(i, j)_C$ is said to *spell* the alignment obtained by concatenating its edge labels. Each path from $(m, n)_C$ to $(i, j)_C$ spells an alignment between $A[m..i]$ and $B[n..j]$, and every such alignment is spelled by some path. However, there may be many paths from $(m, n)_C$ to $(i, j)_C$ spelling the same alignment. Thus the correspondence between paths and alignments is not one-to-one unless one restricts attention to a canonical subset of the paths. A path is *normal* if and only if it does not contain subpaths of the form $(i-1, j)_D \to (i-1, j)_C \to (i, j)_D$ or $(i, j-1)_I \to (i, j-1)_C \to (i, j)_I$. An exercise, not proven here, shows that alignments between $A[m..i]$ and $B[n..j]$ are in one-to-one correspondence with normal paths from $(m, n)_C$ to $(i, j)_C$.

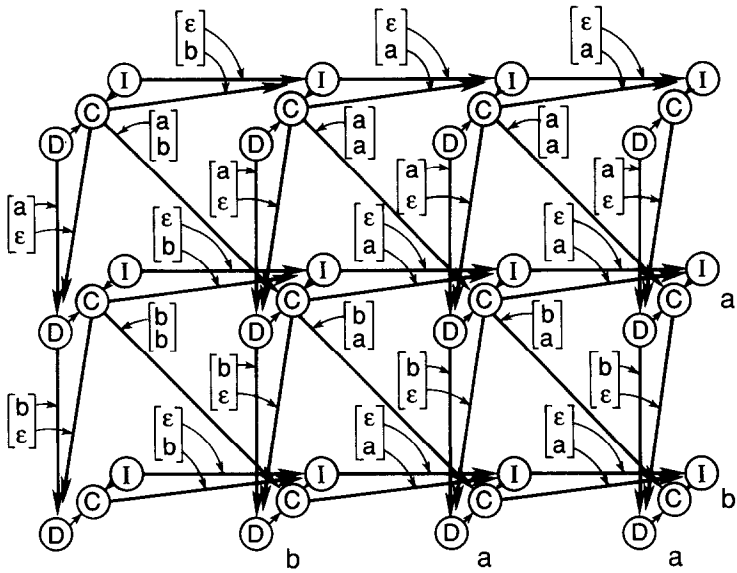

FIG. 1. $G_{A,B}$ for $A = ab$ and $B = baa$.

Our first goal is an algorithm that computes the score and final vertex $(i,j)_C$ of a highest-scoring alignment among all alignments between a substring $A[m..i]$ of A and a substring $B[n..j]$ of B. Assign weights to edges of $G_{A,B}$ as follows. Deletion initiation and insertion initiation edges are weighted $\sigma(\pi) - g$, where π is the aligned pair labeling the edge. Null edges are weighted 0 and all other edges are weighted $\sigma(\pi)$. The weight or *score* of a path, i.e., the sum of its edge weights, thus equals the sum of the scores of its non-null labels minus g times the number of initiation edges. For normal paths this is exactly the score of the corresponding alignment since each initiation edge corresponds to the leftmost aligned pair in a gap. Hence the problem is to locate a maximum-weight normal path between any two nodes of G. However, for every non-normal path there is a normal path of greater weight spelling the same alignment, since a subpath such as $(i-1,j)_D \to (i-1,j)_C \to (i,j)_D$ can be replaced by $(i-1,j)_D \to (i,j)_D$ for a net weight gain of $g > 0$. Thus the problem is to locate a maximum-weight path (it must be normal) in G. Since insertion and deletion edges have negative weights, there is a maximum-weight path that begins and ends with a substitution edge, and hence connects C-vertices. A path connecting C-vertices will be called a *C-path*.

Because $G_{A,B}$ is acyclic, a maximum-weight path can be determined in a single pass over its vertices, so long as they are taken in a *topological*

```
D(0,0) ← I(0,0) ← −∞
C(0,0) ← 0
score_max → i_best ← j_best ← 0
for j ← 1 to N do
  {D(0,j) ← −∞
   I(0,j) ← σ([ε; b_j]) − g
   C(0,j) ← 0
  }
for i ← 1 to M do
  {I(i,0) ← −∞
   D(i,0) ← σ([a_i; ε]) − g
   C(i,0) ← 0
   for j ← 1 to N do
     {D(i,j) ← max{D(i − 1,j), C(i − 1,j) − g} + σ([a_i; ε])
      I(i,j) ← max{I(i,j − 1), C(i,j − 1) − g} + σ([ε; b_j])
      C(i,j) ← max{0, D(i,j), I(i,j), C(i − 1,j − 1) + σ([a_i; b_j])}
      if C(i,j) > score_max then
        {i_best ← i
         j_best ← j
         score_max ← C(i,j)
        }
     }
  }
write "A best local alignment with score" score_max "ends at" (i_best, j_best)
```

FIG. 2. The Smith–Waterman local alignment algorithm.

order, i.e., an ordering of the vertices with the property that every edge is directed from a vertex to a successor in the ordering. One topological order for $G_{A,B}$'s vertices is to treat the rows in order, sweeping left to right within a row. For fixed (i, j), $(i, j)_C$ is treated after $(i, j)_D$ and $(i, j)_I$.

In the algorithm of Fig. 2, the values $D(i, j)$, $I(i, j)$, and $C(i, j)$ are, respectively, the maximum weight of the path from any C-vertex to $(i, j)_D$, to $(i, j)_I$, and to $(i, j)_C$. Each of these values is found by considering the edges entering the vertex, where for $C(i, j)$ we also consider the zero-edge path from $(i, j)_C$ to itself. For instance, when $i > 0$, there are two edges into $(i, j)_D$, i.e., one from $(i - 1, j)_D$ with weight $\sigma\left(\left[\begin{smallmatrix} a_i \\ \varepsilon \end{smallmatrix}\right]\right)$ and one from $(i - 1, j)_C$ with weight $\sigma\left(\left[\begin{smallmatrix} a_i \\ \varepsilon \end{smallmatrix}\right]\right) - g$. Since a maximum weight path to $(i, j)_D$ must end with one of these edges and since the portion of the path preceding that last edge must be optimal, $D(i, j)$ is the larger of

$D(i-1,j)+\sigma\left(\left[\begin{smallmatrix} a_i \\ \varepsilon \end{smallmatrix}\right]\right)$ or $C(i-1,j)+\sigma\left(\left[\begin{smallmatrix} a_i \\ \varepsilon \end{smallmatrix}\right]\right)-g$. Values at unreachable vertices, i.e., $D(0,j)$ and $I(i,0)$, are defined to be $-\infty$, a suitably large negative constant.

Waterman and Eggert [15] extend the Smith-Waterman algorithm to compute k best "non-intersecting" local alignments, where $k > 0$ can be chosen arbitrarily. More formally, let $P_1, P_2, \ldots, P_k$ be C-paths of G in non-increasing order of scores. $P_1, P_2, \ldots, P_k$ are *k best non-intersecting paths* if for each $n \in [1,k]$, P_n is a maximum-score path in G_n, where G_n is obtained by removing the substitution (i.e., diagonal) edges of $P_1, P_2, \ldots, P_{n-1}$ from G. This definition has the following "non-uniqueness" property. It is possible to have two sequences of k best non-intersecting paths $\{P_n\}$ and $\{Q_n\}$ such that for some n, P_n and Q_n have different scores. An example of this phenomenon is given in Section 4.

The basic idea of the Waterman–Eggert algorithm is that once all entires $D_n(i,j)$, $I_n(i,j)$, and $C_n(i,j)$ for paths in G_n are found, the corresponding values in G_{n+1} can be determined quite efficiently. Indeed, $D_{n+1}(i,j) = D_n(i,j)$ for "most" positions (i,j) (and similarly for I and C). Only values at positions "near" P_n need be recomputed. See the Waterman–Eggert paper for more details; Waterman [14] gives a complete implementation in the C language.

3. A Linear-Space Algorithm

A serious drawback of the Waterman–Eggert algorithm is the need to save all values $D_n(i,j)$, $I_n(i,j)$, and $C_n(i,j)$ from the current graph, G_n. (A subscript n will always indicate that the value is relative to G_n) Huang *et al.* [6] give a local alignment algorithm that needs only "linear space," i.e., space proportional to $M+N+K$ (instead of $MN+K$), where K is the total length of all computed alignments. Unfortunately, the technique is quite slow: for each $n \in [1,k]$ it computes all entries of D_n, I_n, and C_n from scratch, so $O(kMN)$ time is needed to find k alignments. Our goal is to obtain a speedup comparable to that from the Waterman–Eggert approach, while maintaining the linear space bound and guaranteeing optimality of the computed solution.

In outline, our approach is as follows. We perform the Smith–Waterman algorithm (Fig. 2), saving k best scores instead of just one. Care is required since typically the k largest values $C(i,j)$ will occur at vertices (i,j) that are packed tightly together. To avoid this problem, we save values $C(i_1,j_1)$ and $C(i_2,j_2)$ only if there are *disjoint* optimal paths ending at (i_1,j_1) and (i_2,j_2). However, this approach introduces another problem. It may be that the terminal vertex of, say, the ith overall best

non-intersecting path is also the endpoint of a higher scoring path that shares its startpoint with local alignment #1. It is only after we remove the substitution edges of the best path that the Smith–Waterman method can discern this ith best path. In other words, the k best non-intersecting paths that can be found in one Smith–Waterman pass over the C-matrix may not include the k overall best paths. Huang *et al.* show that this phenomenon is important in practice.

Once we have determined the best path, the problem is to discover any high-scoring paths that were hidden from the first Smith–Waterman pass. This can be done by making a "backward" Smith–Waterman pass over a limited portion of the graph to discover the "region of influence" of the path being removed, followed by a forward Smith–Waterman pass over this region to find the newly-exposed paths. It is not immediately apparent how one can determine a region that is guaranteed to be sufficiently large; see Lemma 3, below, for our solution. In general, we need one complete pass over the matrix, followed by limited forward and backward passes for each removed path (except for the final path). Our record of the k best non-intersecting paths is updated whenever we find a path that outscores our current kth best path. With this outline, we are ready to consider the details.

The first observation we need is that the Smith–Waterman algorithm (Fig. 2) can be modified to compute the first vertex of an optimal path. As will be shown in Section 4, it is critical how we select this first vertex in the case of ties, i.e., when there are several optimal paths to $(i, j)_C$ with different initial vertices. In what follows, let $<_G$ denote any topological order on the C-vertices of G. This ordering can be the same as the order in which vertices are treated in Fig. 2, or it can be different. $\text{First}_n(i, j)_D$ is defined to be the last C-vertex in this ordering such that there is a path in G_n from that vertex to $(i, j)_D$ of score $D(i, j)$. $\text{First}_n(i, j)_I$ and $\text{First}_n(i, j)_C$ are defined similarly.

Figure 2 can be augmented so that vertices $\text{First}(i, j)$ are computed. The basic idea is simply to inherit *First* from whichever incoming path is found to be optimal. (For 0-edge optimal paths set $\text{First}(i, j)_C = (i, j)_C$.) For example, the line

$$D(i, j) \leftarrow \max\{D(i-1, j), C(i-1, j) - g\} + \sigma\left(\begin{bmatrix} a_i \\ \varepsilon \end{bmatrix}\right)$$

becomes

if $D(i-1, j) > C(i-1, j) - g$ **then**
$\{D(i, j) \leftarrow D(i-1, j) + \sigma\left(\begin{bmatrix} a_i \\ \varepsilon \end{bmatrix}\right)$
$\text{First}(i, j)_D \leftarrow \text{First}(i-1, j)_D$
$\}$

else if $D(i-1,j) < C(i-1,j) - g$ **then**
$\{D(i,j) \leftarrow C(i-1,j) + \sigma\left(\left[\begin{smallmatrix} a_i \\ \varepsilon \end{smallmatrix}\right]\right) - g$
$\text{First}(i,j)_D \leftarrow \text{First}(i-1,j)_C$
$\}$
else /*tie!*/
$\{D(i,j) \leftarrow D(i-1,j) + \sigma\left(\left[\begin{smallmatrix} a_i \\ \varepsilon \end{smallmatrix}\right]\right)$
$\text{First}(i,j)_D \leftarrow \max_{<_G}\{\text{First}(i-1,j)_D, \text{First}(i-1,j)_C\}$
$\}$

LEMMA 1. *Fix $n \in [1,k]$ and let u be a C-vertex such that G_{n+1} is formed from G_n by removing the substitution edges of an optimal path from* $\text{First}_n(u)$ *to u. If v is a C-vertex with* $\text{First}_n(v) \neq \text{First}_n(u)$, *then* $C_{n+1}(v) = C_n(v)$ *and* $\text{First}_{n+1}(v) = \text{First}_n(v)$.

Proof. Let P be the path from $\text{First}_n(u)$ to u whose substitution edges are removed, and let Q be an optimal path from $\text{First}_n(v)$ to v. We first show that P and Q have no vertex in common. Suppose otherwise and let s be a vertex lying on P and Q. The prefix of P ending at s and the prefix of Q ending at s must have the same score (sum of edge weights). To see this, suppose that the prefix of P ending at s had lower score. Form a path P' from $\text{First}_n(v)$ to u by appending the suffix of P beginning at s to the prefix of Q ending at s. Then $\text{score}(P') > \text{score}(P)$, contradicting the optimality of P.

Without loss of generality, suppose $\text{First}_n(u) <_G \text{First}_n(v)$. Then the path P' constructed as above satisfies $\text{score}(P') = \text{score}(P)$, and P' starts at a vertex that follows $\text{First}_n(u)$ in the ordering $<_G$. This contradicts the definition of $\text{First}_n(u)$ and proves that P and Q are disjoint.

Since P and Q have no vertex in common, they have no edge in common. Thus Q is also a path in G_{n+1}, so $C_{n+1}(v) \geq C_n(v)$. But since $C_{n+1}(w) \leq C_n(w)$ for all vertices w, we have $C_{n+1}(v) = C_n(v)$. There cannot exist an optimal path in G_{n+1} from a $<_G$-successor of $\text{First}_n(v)$ to v, since such a path would also be a path in G_n, so $\text{First}_{n+1}(v) = \text{First}_n(v)$. □

As described in the previous section, let G_1 be the original graph $G_{A,B}$ and for each $n \in [1, k-1]$, let G_{n+1} be determined by locating a C-vertex u that maximizes $C_n(u)$ and removing from G_n the substitution edges of an optimal path from $\text{First}_n(u)$ to u. Define a relation E_n over the C-vertices of G by: uE_nv if and only if $\text{First}_n(u) = \text{First}_n(v)$. E_n is an equivalence relation, and hence E_n partitions the vertices of G into equivalence classes. For each equivalence class S of E_n, define $\text{score}_n(S) = \max\{C_n(u): u \in S\}$. Our local alignment algorithm begins by determining the k best (i.e., highest-scoring) equivalence classes for E_1. Each of these k equiva-

lence classes S is represented by a 7-tuple:

$\langle C, F, u, T, B, L, R \rangle$, where
$C = \text{score}(S)$,
$F = \text{First}(s)$ for all $s \in S$,
$\text{First}(u) = F$ and $C(u) = \text{score}(S)$, and
$[T, B] \times [L, R]$ contains all $s \in S$ with $C(s) > W$.

In this definition, W is the minimum score of the retained equivalence classes, and we say that $[T, B] \times [L, R]$ contains $(i, j)_C$ if $T \leq i \leq B$ and $L \leq j \leq R$. Initially, C values and First vertices are relative to G_1; as the algorithm proceeds, the tuples are updated so as to be valid for $G_2, G_3, \ldots, G_{k-1}$. Henceforth, we use *tuple* to designate such a 7-tuple, and refer to the entries of tuple S by $S.C, S.F, \ldots, S.R$.

Once k best classes of G_1 are determined, we select and delete a best class S, then remove the diagonal edges of an optimal path from $S.F$ to $S.u$ to get G_2. Lemma 1 implies that if vertex v satisfies $\text{First}_1(v) \neq S.F$, then $C_2(v) = C_1(v)$ and $\text{First}_2(v) = \text{First}_1(v)$. Thus, to find the $k - 1$ best classes in G_2 we need only re-evaluate C and *First* for vertices in S with score $> W$, i.e., vertices contained in $[S.T, S.B] \times [S.L, S.R]$. This is because vertices in other equivalence classes are already assigned to the proper *First* vertex, and vertices with score $\leq W$ are no longer interesting, since we already have enough non-intersecting paths with score at least W. However, each vertex in $[S.T, S.B] \times [S.L, S.R]$ needs to be re-scored in G_{n+1}, because we may find a new path to it of score $> W$.

The $k - n + 1$ best tuples for G_n are kept in a data structure, called LIST, that supports the following operations:

find(f): Return the tuple whose *First* vertex is f, or *null* if no such tuple exists.

insert(S): Add tuple S to LIST.

maxtuple ( ): Remove and return a maximum-score tuple in LIST.

minscore ( ): Return the minimum score of a tuple in LIST.

replace(S): Replace a minimum-score tuple in LIST by S.

size( ): Return the number of tuples in LIST.

Candidates for implementing LIST include a linear list, a balanced search tree (Aho *et al.* [1]) or a splay tree (Sleator and Tarjan [12]). The data structure used in our alignment software is described near the end of this section.

To maintain LIST, we use the function *enter* of Fig. 3. Let $i(u)$ and $j(u)$ denote the i-component and j-component of vertex u. Enter(c, f, u, w, l) performs the following tasks. If there is already a tuple S in LIST whose

```
function enter(c, f, u, w, l)
    S ← find(f)
    if S ≠ null then
     {if S.C < c then
        {S.C ← c
         S.u ← u
        }
      S.T ← min(S.T, i(u))
      S.B ← max(S.B, i(u))
      S.L ← min(S.L, j(u))
      S.R ← max(S.R, j(u))
    }
    else
      if size( ) = l then
            replace(⟨c, f, u, i(u), i(u), j(u), j(u)⟩)
      else
            insert(⟨c, f, u, i(u), i(u), j(u), j(u)⟩)
    if size( ) = l then
            return minscore( )
    else
            return w
```

FIG. 3. The *enter* function.

First vertex is f, then the other attributes of S are adjusted if necessary. Otherwise, the tuple $\langle c, f, u, i(u), i(u), j(u), j(u)\rangle$ either replaces a minimum-score tuple in LIST or is inserted into LIST, depending on whether or not LIST is full (i.e., contains l tuples). Finally, the minimum tuple score is returned if LIST is full.

Notice that *enter* guarantees that any two tuples in LIST have distinct *First* values. That is, if S and S' are in LIST and $S \neq S'$, then $S.F \neq S'.F$. The rectangle $[T, B] \times [L, R]$ is properly maintained so long as *enter* is called whenever $C(u) > W$.

Figure 4 outlines our algorithm for computing k best non-intersecting paths. The first part of the algorithm (two nested **for** loops) computes tuples representing k highest-scoring classes in G_1, while the second part reports k best non-intersecting paths one at a time. The nth iteration of the **for** loop in the second part selects a highest-scoring tuple, reports its optimal path, removes the diagonal edges of the path to form G_{n+1}, and computes the $k - n$ best tuples in G_{n+1}. The algorithm mentions only computation of $C(i, j)$ and abbreviates $\text{First}(i, j)_C$ by $\text{First}(i, j)$; computation of $D(i, j)$, $\text{First}(i, j)_D$, $I(i, j)$, and $\text{First}(i, j)_I$ is implicit. In another effort to simplify the presentation, we assume in what follows that there are at least k distinct values of *First* in G_1 with paths of positive score.

```
W ← 0
for i ← 0 to M do
    for j ← 0 to N do
        {Compute C(i, j) and First(i, j)
         if C(i, j) > W then
            W ← enter(C(i, j), First(i, j), (i, j), W, k)
        }
for n ← 1 to k do
  {S ← maxtuple( )
   alignment(S)
   if n ≠ k then
       {Determine T ≤ S.T and L ≤ S.L so that no C-path starting outside [T, S.B] ×
        [L, S.R] and ending inside [S.T, S.B] × [S.L, S.R] has score greater than W
        for i ← T to S.B do
          for j ← L to S.R do
            {Compute C(i, j) and First(i, j) relative to [T, S.B] × [L, S.R]
             if C(i, j) > W and (i, j) is in [S.T, S.B] × [S.L, S.R] then
                W ← enter(C(i, j), First(i, j), (i, j), W, k − n)
            }
       }
  }
```

FIG. 4. Outlined of the linear-space algorithm.

The procedure alignment(S) employs the global alignment of Myers and Miller to find an optimal path from $S.F$ to $S.u$ and then removes the diagonal edges of the path from G_n. In practice, these "removed" edges are recorded in a data structure so that they will not be used again in determining T and L, in the final nested **for** loops, or in subsequent calls to *alignment*. Specifically, when evaluating

$$C(i,j) \leftarrow \max\left\{0, D(i,j), I(i,j), C(i-1,j-1) + \sigma\left(\begin{bmatrix} a_i \\ b_j \end{bmatrix}\right)\right\}$$

after entering the main **for** loop of Fig. 4, we need to check whether the edge from $(i-1, j-1)_C$ to $(i,j)_C$ has been removed; if it has, then we maximize over just the first three terms. We now prove correctness of the local alignment algorithm, postponing a description of our method for computing T and L until the end of this section.

LEMMA 2. *At the start of the* nth *iteration of the main* **for** *loop of Fig.* 4, LIST *contains* $k - n + 1$ *tuples that satisfy the following properties*:

(1) $\min\{S.C\colon S \in \text{LIST}\} = W_n$, where W_n is the value of W at the start of the nth iteration,

(2) for each C-vertex u that satisfies $C_n(u) > W_n$, there is a S in LIST such that $\text{First}_n(u) = S.F$, $C_n(u) \leq S.C$, and $u \in [S.T, S.B] \times [S.L, S.R]$,

(3) for each tuple S in LIST, $\text{First}_n(S.u) = S.F$ and $C_n(S.u) = S.C$ and

(4) maxtuple delivers a tuple S such that an optimal path from $S.F$ to $S.u$ is an optimal path in G_n.

Proof. (Induction on n). For $n = 1$, the claim follows readily from the definition of *enter* in Fig. 3. (Recall our assumption that there are at least k distinct values of *First* in G_1 with paths of positive score.) Assuming that the claim is true for n, we will show that it is true for $n + 1$. Consider the nth iteration of the loop. The assignment $S \leftarrow$ maxtuple ( ) reduces the size of LIST from $k - n + 1$ to $k - n$. With every call to *enter* during that iteration, the last argument in the call is $k - n$, so the condition "size( ) = l" is true within *enter*. It follows that *enter* calls *replace* rather than *insert*. Thus the size of LIST at the start of the next iteration is still $k - n = k - (n + 1) + 1$, and hence LIST contains the claimed number of tuples. It likewise follows that *enter* returns the minimum tuple score, which is assigned to W. This verifies condition 1.

Consider condition 2, and let u satisfy $C_{n+1}(u) > W_{n+1}$. We have already noted that *insert* is not called once execution reaches the main **for** loop; instead minimum-score tuples are replaced by ones of higher score. It follows that successive values of W are non-decreasing throughout the k iterations of the main loop. Thus, $C_n(u) \geq C_{n+1}(u) > W_{n+1} \geq W_n$. The induction hypothesis guarantees that at the start of the nth iteration there is a S' in LIST such that $\text{First}_n(u) = S'.F$, $C_n(u) \leq S'.C$, and $u \in [S'.T, S'.B] \times [S'.L, S'.R]$. Let S be the tuple selected and removed from LIST by *maxtuple*.

First suppose that $S' \neq S$. Then $S'.C \geq C_n(u) > W_{n+1}$ at the start of the nth iteration, so S' is not replaced during the nth iteration. *Enter* is written so that $S'.C$ can only increase in value and the rectangle associated with S' can only increase in extent. Also, $\text{First}_{n+1}(u) = \text{First}_n(u)$ and $C_{n+1}(u) = C_n(u)$ by Lemma 1, so u satisfies condition 2 with n replaced by $n + 1$.

Next suppose that $S' = S$. Since $C_{n+1}(u) > W_{n+1} \geq W_n$, the definition of T and L in Fig. 4 guarantees that all paths in G_{n+1} to u of cost greater than W_n are entirely contained in $[T, S.B] \times [L, S.R]$. The nested **for** loop at the end of Fig. 4 discovers all those paths and updates LIST to contain the desired tuple. This completes the verification of condition 2.

Now consider condition 3. Let S be in LIST at the start of the $(n + 1)$th iteration. If S is placed in LIST during the nth iteration, or if $S.C$ and $S.u$ are assigned new values during that iteration, then c, f, and u for that call

to *enter* satisfy $c = C_{n+1}(u)$ and $f = \text{First}_{n+1}(u)$, verifying condition 3. Otherwise, Lemma 1 and the induction hypothesis imply that $\text{First}_{n+1}(S.u) = \text{First}_n(S.u) = S.F$ and $C_{n+1}(u) = C_n(u) = S.C$.

It remains to verify condition 4. Let S be the tuple selected by *maxtuple* in the nth iteration of the main **for** loop of Fig. 4. It follows from assertions 2 and 3 of Lemma 2 that $C_n(S.u) = S.C = \max\{C_n(v)\colon v$ is a vertex of $G\}$ and $\text{First}_n(S.u) = S.F$. Thus any optimal path from $S.F$ to $S.u$ is a highest-scoring path in G_n. □

THEOREM 1. *The algorithm of Fig.* 4 *computes k best non-intersecting local alignments.*

Proof. Correctness follows immediately from condition 4 of Lemma 2. □

To compute values T and L needed in the main **for** loop of Fig. 4, we use the procedures *locate* and *disjoint* of Fig. 5. Namely, the algorithm of Fig. 4 performs the call locate($S.T, S.B, S.L.S.R$) to find T and L such that no C-path starting outside $[T, S.B] \times [L, S.R]$ and ending inside $[S.T, S.B] \times [S.L, S.R]$ has score greater than W. For the most part, the computation is merely a "reverse" local similarity method that proceeds from lower right to upper left. (The differences are that no analog of LIST is maintained and that any tie-breaking rule can be used.) Values C', D', I' and vertices *Last* in the reverse computation correspond to the C, D, I, and *First* in the forward computation. For the remainder of the section, X represents any of C', D', or I'. The notation $\text{Last}(v) \geq (T', L')$ means that both $i(\text{Last}(v)) \geq T'$ and $j(\text{Last}(v)) \geq L'$. The **while** loop in *locate* stops when either no optimal path beginning on row T and column L ends inside $[T', b] \times [L', r]$ or $T = L = 0$. The **repeat** loop in *locate* terminates when either *disjoint* returns **true** or $T = L = 0$. Lastly, disjoint(t, b, l, r; **var** t', l') returns **false** if there is no S in LIST such that $[S.T, S.B] \times [S.L, S.R]$ shares any vertex with either $[t, b] \times [l, l' - 1]$ or $[t, t' - 1] \times [l, r]$; it returns **true** with t' and l' adjusted if there is such an S.

LEMMA 3. Locate(t, b, l, r) *determines T and L so that any path starting outside* $[T, b] \times [L, r]$ *and ending inside* $[t, b] \times [l, r]$ *has score at most W, where W is the minimum tuple score in* LIST.

Proof. To smooth the way for a more formal proof, we first develop an intuitive understanding based on Fig. 6. If *locate* terminates with $T = L = 0$, then the result is trivial, so suppose otherwise. Indeed, suppose that the final values of T and L are both positive; we leave the case that exactly one of T or L is 0 to the reader. At termination, *rflag* = *cflag* = **false**, so for each vertex v on the top and right boundaries of $[T, B] \times [L, R]$ the

```
FUNCTION locate(t, b, l, r).
    Perform a reverse local alignment computation on [t, b] × [l, r] and save C', D', I', and
Last for vertices of the forms (t, j)_X for j ∈ [l, r] or (i, l)_X for i ∈ [t, b]
    T ← T' ← t
    L ← L' ← l
    repeat
      {rflag ← cflag ← true
       while (rflag and T > 0) or (cflag and L > 0) do
          {if rflag and T > 0 then
             {rflag ← false
              T ← T − 1
              Determine C', D', I', and Last at (T, j)_X for j ← r down to L
              if Last(T, j)_X ≥ (T', L') for some j ∈ [L, r] and X then
                   rflag ← ture
              if Last(T, L)_X ≥ (T', L') for some X then
                   cflag ← ture
             }
           if cflag and L > 0 then
             {cflag ← false
              L ← L − 1
              Determine C', D', I', and Last at (i, L)_X for i ← b downto T
              if Last(i, L)_X ≥ (T', L') for some i ∈ [T, b] and X then
                   cflag ← true
              if Last(T, L)_X ≥ (T', L') for some X then
                   rflag ← true
             }
          }
      }
    until disjoint(T, b, L, r, T', L') or T = L = 0
    return (T, L)

FUNCTION disjoint(t, b, l, r; var t', l').
    for each S in LIST do
          if S.T ≤ b and S.L ≤ r and S.B ≥ t and S.R ≥ l and (S.T < t' or S.L < l') then
            { t' ← min(t', S.T)
              l' ← min(l', S.L)
              return false
            }
    return true
```

FIG. 5. The *locate* and *disjoint* functions.

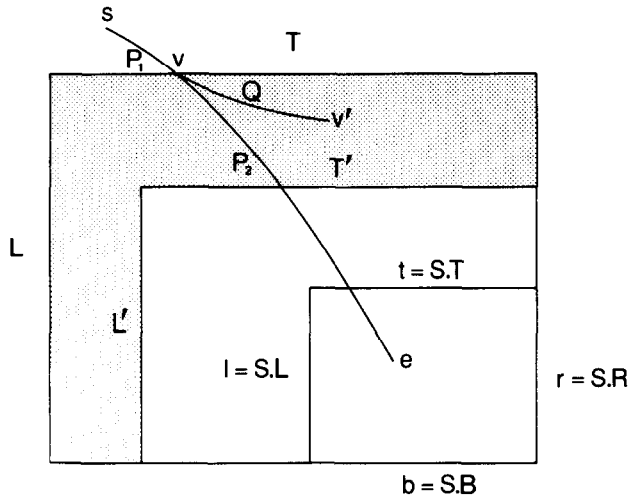

FIG. 6. Graphical interpretation of T and L.

condition "Last(v) $\geq (T', L')$" is false. Pictorially, Last(v) is in the shaded region in Fig. 6. Also disjoint(T, b, L, r, T', L') = **true**, so no S in LIST has an associated rectangle that intersects the shaded region in Fig. 6. Part 2 of Lemma 2 then guarantees that all vertices s in the shaded region satisfy $C_n(s) \leq W$.

More formally, let P be a C-path that starts outside $[T, b] \times [L, r]$ at vertex s and ends in $[t, b] \times [l, r]$ at vertex e. Let v be the first vertex on P that has the form either $(T, j)_X$ for $j \in [L, r]$ or $(i, L)_X$ for $i \in [T, b]$. Also, let v' = Last(v). Let P_1 be the prefix of P ending at v and let P_2 be the suffix of P starting at v. Then v' must be in $[t, b] \times [l, l' - 1]$ or $[t, t' - 1] \times [l, r]$, so v' is not in the rectangle $[S.T, S.B] \times [S.L, S.R]$ for any S in LIST. Thus $C_n(v') \leq W$ by part 2 of Lemma 2. Let Q be the path from v to v' and form P' by appending Q to P_1. Since P' ends at v', score(P') $\leq C_n(v')$, so score(P) = score(P_1) + score(P_2) $\leq$ score(P_1) + score(Q) = score(P') $\leq C_n(v') \leq W$. □

Notice that if no path in G_{n+1} that ends in $[t, b] \times [l, r]$ has score greater than W, then no value of C' will exceed W in the reverse pass. In that case, the forward pass at the end of Fig. 4 can be omitted.

Our preferred implementation of LIST uses an array of k records, one per tuple. Special care is taken to accelerate the frequent operations *size*, *find*, *minscore*, and *replace*. The number of tuples in LIST is kept in a variable so that size takes only constant time. Since vertices in the same equivalence class form a connected region, find(f) tends to be invoked

with the same f for a period of time. Maintaining a pointer to the most recently accessed record allows find(f) to usually be served in constant time. To speed up *minscore*, a pointer to a minimum score record is kept, which also makes *replace* efficient. Whenever this minimum-score record is replaced or modified, the pointer needs be recomputed by searching the entire array. Experimental results quoted in Section 5 show that this simple implementation strategy is entirely adequate.

The algorithm is designed to require only linear space. In the complete forward sweep of G_1, it suffices to save only the most recently computed row of each matrix. (For implementation details, see Huang *et al.* [6]). In the reverse pass to determine a rectangle, the most recently computed column of each matrix is saved as well. Recomputation of rectangles is handled like the complete forward sweep. Thus, $O(M + N)$ space is sufficient for values $C_n(i, j)$, $\text{First}_n(i, j)_C$, etc. The *alignment* procedure of Myers and Miller [7] uses space $O(M + N + L_n)$, where L_n is the length of the computed alignment. $O(K)$ space is needed to save the aligned pairs of computed alignments, where $K = \sum_{n=1}^{k} L_n$. Lastly, the data structure for LIST takes $O(k)$ space, where $k \ll K$. Thus, the algorithm requires $O(M + N + K)$ space.

A completely rigorous and realistic analysis of the algorithm's time complexity would be difficult, but the general trend is easy to grasp. The complete forward sweep clearly takes $O(MN)$ time in the worst case, exclusive of calls to manipulate LIST, and as just noted, manipulation of LIST takes negligible time in practice. The remainder of the algorithm may take $O(kMN)$ time in the worst case that all computed alignments are nearly as long as the original sequences. However, in practice, at most a few alignments are large, and the performance of the algorithm is much better than the worst case. Execution time is strongly affected by the choice of scoring function σ and gap penalty g. When mismatches and gaps are penalized very lightly, the rectangles requiring recomputation are large and the algorithms' efficiency suffers accordingly. When reasonable weights are used, sides of the recomputation rectangle are only a few times longer than the length L_n of the local alignment, so the recomputations take $O(\sum_{n=1}^{k} L_n^2)$ time in expectation.

One final implementation note deserves mention. Some users may prefer a cutoff score to k. That is, instead of computing k best alignments, all non-intersecting alignments with scores greater than the cutoff score are computed. Our algorithm can easily be modified to do this. In Fig. 4, replace $W \leftarrow 0$ by $W \leftarrow$ cutoff, and replace

for $n \leftarrow 1$ **to** k **do**

by:

while LIST is not empty **do**

In addition, the function *enter* is modified accordingly: no bound is placed on the size of LIST, *insert* is always called, and no minimum core is returned. The difficulty in working with cutoffs is that a setting just slightly below the optimal cutoff value may bring forth a deluge of insignificant local alignments.

4. An Example

Lemma 1 is central to our approach. It implies that we need only save endpoints of k paths from a single complete sweep that computes all entries of D, I, and C. Lemma 1 guarantees that we will never "break" any of the retained paths by removing diagonal edges of a higher-scoring path. This section contains an example showing that a less meticulous definition of *First* may well not work properly for this purpose.

For simplicity of this example, suppose that the gap-open penalty g is 0. Then for each i and j the vertices $(i, j)_X$ for $X = D$, I, and C can be thought of a coalescing into a single node. (Details can be found in Myers and Miller [8].) Consider the tie-breaking rule that is applied when the diagonal edge entering a node gives a path-weight identical to a horizontal or vertical edge. In particular, suppose that such a tie is broken in favor of the diagonal edge, i.e., First(i, j) is set equal to First$(i - 1, j - 1)$. Assume the following matrix, σ, of substitution costs.

	a	b	c	d	e
a	2	1			
b	1	2	1		
c		1	4		1
d				2	
e			1		

Values of σ not shown (including deletions and insertions) equal -1.

Let $A = abcd$ and $B = aced$. A best local alignment of A and B is $\left[\begin{smallmatrix} a \\ a \end{smallmatrix}\right]\left[\begin{smallmatrix} b \\ c \end{smallmatrix}\right]\left[\begin{smallmatrix} c \\ e \end{smallmatrix}\right]\left[\begin{smallmatrix} d \\ d \end{smallmatrix}\right]$, which scores 6. The second best local alignment that shares no aligned pairs with the first alignment is $\left[\begin{smallmatrix} b \\ a \end{smallmatrix}\right]\left[\begin{smallmatrix} c \\ c \end{smallmatrix}\right]$, which scores 5.

Figure 7 depicts $G_{A,B}$. Edges with positive weights are shown; edges weighted -1 are either not shown or dotted. First(u) and First(v) are shown for the rule that prefers diagonal edges. There is an alternative optimal path from First(u) to u that breaks the only optimal path from First(v) to v. The corresponding alignment is $\left[\begin{smallmatrix} a \\ a \end{smallmatrix}\right]\left[\begin{smallmatrix} b \\ \varepsilon \end{smallmatrix}\right]\left[\begin{smallmatrix} c \\ c \end{smallmatrix}\right]\left[\begin{smallmatrix} \varepsilon \\ e \end{smallmatrix}\right]\left[\begin{smallmatrix} d \\ d \end{smallmatrix}\right]$, which scores 6 but contains the pair $\left[\begin{smallmatrix} c \\ c \end{smallmatrix}\right]$. This would affect our algorithm because

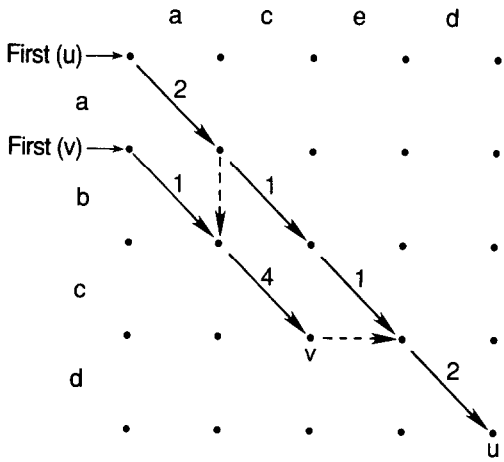

FIG. 7. $G_{A,B}$ for $A = abcd$ and $B = aced$.

when the Myers–Miller procedure is invoked to find an optimal path between two given vertices, there is no effective way of controlling which optimal path is found.

If *First* is defined as in Section 3 (using $<_G$ to break ties), then First(v) = First(u) = (1, 0). Then the only optimal path from First(u) to u spells the alignment $\left[\begin{smallmatrix} b \\ a \end{smallmatrix}\right]\left[\begin{smallmatrix} c \\ c \end{smallmatrix}\right]\left[\begin{smallmatrix} e \\ e \end{smallmatrix}\right]\left[\begin{smallmatrix} d \\ d \end{smallmatrix}\right]$, which scores 6. The second best non-intersecting alignment is spelled by a path extending $\frac{3}{4}$ of the way down the main diagonal of G, i.e., $\left[\begin{smallmatrix} a \\ a \end{smallmatrix}\right]\left[\begin{smallmatrix} b \\ c \end{smallmatrix}\right]\left[\begin{smallmatrix} c \\ e \end{smallmatrix}\right]$, which scores 4.

Figure 7 also provides an example of the non-uniqueness phenomenon mentioned at the end of Section 2, i.e., where two sequences of k best non-intersecting paths have different scores. For $k = 2$, the paths

$$(0,0) \to (1,1) \to (2,2) \to (3,3) \to (4,4)$$
$$(1,0) \to (2,1) \to (3,2)$$

constitute two best non-intersecting paths, the second of which scores 5. The paths

$$(1,0) \to (2,1) \to (3,2) \to (3,3) \to (4,4)$$
$$(0,0) \to (1,1) \to (2,2) \to (3,3)$$

also constitute two best non-intersecting paths, the second of which scores 4.

5. Discussion

This section compares the execution time of the new algorithm and that of the algorithm of Huang *et al.* [6]. Both algorithms were implemented in the C programming language. DNA sequences containing the human and rabbit β-like globin gene clusters were used as test sequences. These sequences share many similar regions, which tends to favor the old algorithm. Use of random sequences would perhaps unfairly favor the new algorithm because such sequences usually contain only short regions of similarity, hence the new algorithm would spend very little time recomputing parts of the matrix.

In all tests, a matching aligned pair scored 1.0 and a mismatched pair scored -1.5. The running time of the old algorithm is not significantly affected by the choice of weights. Thus, we conducted only one run using one set of weights for the old algorithm. On a 73,360-nucleotide sequence containing the human β-globin gene cluster and a corresponding 44,594-nucleotide rabbit sequence, the old algorithm took about 23 days to compute 100 best non-intersecting alignments with a gap-open penalty $g = 6.0$ and and a gap-extension penalty $e = 0.2$. (That is, each deletion or insertion pair is scored -0.2.) On the 11,950-nucleotide human δ-β region and the 12,400-nucleotide rabbit $\delta - \beta$ region, the old algorithm took $5\frac{1}{2}$ h to compute 20 best non-intersecting alignments with $g = 6.0$ and $e = 0.2$.

Times for the new program are shown in Tables I and II. Table I reports the time to compute 100 best non-intersecting alignments between the human and rabbit sequences for various gap penalties. Table II reports the time to compute 20 best non-intersecting alignments between the shorter $\delta - \beta$ regions. The *total* column shows the total time (in hours) taken by our program. The time distribution over the five major parts of the program is also given, where *main* is the complete forward pass, *locate* is the backward pass determining a rectangle, *update* is recomputation of the rectangle, *list* is maintanence of LIST, and *align* is the Myers–Miller global alignment procedure. Notice that using small gap penalties can

TABLE I
Time to Compute 100 Best Alignments of β-Globin Regions

g	e	Total	Main	Locate	Update	List	Align
3.0	0.5	15.5	71.3%	12.9%	11.9%	1.4%	2.3%
3.0	1.0	12.5	91.7%	4.2%	2.1%	0.4%	1.5%
6.0	0.2	15.9	66.7%	14.3%	12.6%	2.2%	3.8%
8.0	0.2	13.3	82.7%	7.5%	6.8%	1.6%	1.2%

TABLE II
Time to Compute 20 Best Alignments of the $\delta - \beta$ Regions

g	e	Total	Main	Locate	Update	List	Align
3.0	0.5	2.05	25.5%	34.2%	32.6%	2.6%	4.6%
3.0	1.0	0.70	73.5%	12.4%	9.5%	1.3%	3.1%
6.0	0.2	2.08	23.8%	27.6%	27.2%	3.0%	17.9%
8.0	0.2	0.82	61.6%	14.9%	14.7%	4.3%	3.7%

substantially increase the sizes of recomputed regions, which has a noteworthy impact on the total execution time.

References

1. A. V. Aho, J. E. Hopcroft, and J. D. Ullman, "The Design and Analysis of Computer Algorithms," Addison-Wesley, Reading, MA, 1974.
2. W. B. Goad and M. I. Kanehisa, Pattern recognition in nucleic acid sequences I: A general method for finding local homologies and symmetries, *Nucl. Acids Res.* **10** (1982), 247–263.
3. O. Gotoh, An improved algorithm for matching biological sequences, *J. Mol. Biol.* **162** (1982), 705–708.
4. J. D. Hall and E. W. Myers, A software tool for finding locally optimal alignments in protein and nucleic acid sequence, *CABIOS* **4** (1988), 35–40.
5. D. S. Hirschberg, A linear space algorithm for computing maximal common subsequences, *Commun. ACM* **18** (1975), 341–343.
6. X. Huang, R. C. Hardison, and W. Miller, A space-efficient algorithm for local similarities, *CABIOS*, **6** (1990), 373–381.
7. E. W. Myers and W. Miller, Optimal alignments in linear space, *CABIOS* **4** (1988), 11–17.
8. E. W. Myers and W. Miller, Approximate matching of regular expressions, *Bull. Math. Biol.* **51** (1989), 5–37.
9. W. R. Pearson and D. J. Lipman, Improved tools for biological sequence comparison, *Proc. Nat. Acad. Sci. U.S.A.* **88** (1988), 2444–2448.
10. P. H. Sellers, The theory and computation of evolutionary distances: pattern recognition, *J. Algorithms* **1** (1980), 359–373.
11. P. H. Sellers, Pattern recognition in genetic sequences by mismatch density, *Bull. Math. Biol.* **46** (1984), 501–514.
12. D. D. Sleator and R. E. Tarjan, Self-adjusting binary search trees, *J. Assoc. Comput. Mach.* **32** (1985), 652–686.
13. T. F. Smith and M. S. Waterman, Identification of common molecular sequences, *J. Mol. Biol.* **147** (1981), 195–197.
14. M. S. Waterman, Sequence alignments, *in* "Mathematical Methods for DNA Sequences" (M. S. Waterman, Ed.), pp. 53–92, CRC Press, Boca Raton, FL, 1988.
15. M. S. Waterman and M. Eggert, A new algorithm for best subsequence alignments with application to tRNA–rRNA comparisons, *J. Mol. Biol.* **197** (1987), 723–728.