Optimal alignments in linear space

*Eugene W. Myers† and Webb Miller**

Abstract

Space, not time, is often the limiting factor when computing optimal sequence alignments, and a number of recent papers in the biology literature have proposed space-saving strategies. However, a 1975 computer science paper by Hirschberg presented a method that is superior to the newer proposals, both in theory and in practice. The goal of this note is to give Hirschberg's idea the visibility it deserves by developing a linear-space version of Gotoh's algorithm, which accommodates affine gap penalties. A portable C-software package implementing this algorithm is available on the BIONET free of charge.

Introduction

Consider the problem: Given sequences $A = a_1 a_2 \cdots a_M$ and $B = b_1 b_2 \cdots b_N$, find a set of ''evolutionary operations'' that converts A to B and minimizes the sum of the operations' costs. The allowed operations are (1) replace one symbol by another, (2) delete k consecutive symbols, or (3) insert k consecutive symbols. In addition, the problem statement requires that every symbol of A must be either replaced or deleted. Replacement costs are specified by a table, w, where $w(a, b)$ gives the cost of replacing a by b. Note that a symbol of A is effectively left unedited if it is replaced by itself at no cost, i.e., $w(a, a) = 0$. Two non-negative constants, g and h, specify an affine function, $gap(k) = g + hk$, for the cost of a k-symbol indel (insertion or deletion). Informally, opening up a gap costs g and each symbol in the gap costs h.

The problem is often formulated as maximizing the similarity score of an alignment, rather than minimizing the difference score of a conversion. A bonus $\sigma(a, b)$ is added for every aligned pair (a, b) and a ''gap penalty'' $q + rk$ is subtracted for every k-symbol gap. This formulation is converted to a difference problem by the transformations

$$w(a, b) = \sigma_{\max} - \sigma(a, b) \text{ for all pairs } (a, b)$$
$$g = q$$
$$h = r + \tfrac{1}{2}\sigma_{\max}$$

where $\sigma_{\max} = \max_{(a, b)} \sigma(a, b)$ (Smith, *et al.*, 1981). Thus, to produce an alignment that maximizes the similarity score, first apply these transformations and then run the program described in this paper with the resulting w, g, and h. If the minimum conversion score is C, then the corresponding maximum alignment score is $\tfrac{1}{2}(M + N)\sigma_{\max} - C$.

Gotoh (1982) gave an algorithm that solves such problems in $O(MN)$ time. If only the minimum cost is desired, then it is easy to implement the algorithm in $O(N)$ space, where N can be taken as the shorter sequence length. If one also desires a set of operations attaining the

†Department of Computer Science, University of Arizona, Tucson, AZ 85721. The work of this author was supported in part by NSF Grant DCR-8511455.

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

minimum cost, then straightforward implementations need $O(MN)$ space. In practice, this space requirement often limits the method's applicability, and several papers (Taylor, 1984; Watanabe, *et al.*, 1985; Altschul and Erickson, 1986; Gotoh, 1986; Gotoh, 1987) have presented strategies that reduce space consumption by constant factors. These papers fail to note that Hirschberg (1975) showed how to produce an optimal conversion or alignment in $O(N)$ space. When only a single optimal alignment of A and B is desired, Hirschberg's approach is superior to the others. For example, in one megabyte of memory, our program based on Hirschberg's method can align two sequences of length 62,500. Altschul and Erickson (1986) propose keeping 7 bits for each of MN entries, so the limit for their method is $7N^2 \leq 8\times10^6$, or $N < 1070$. Moreover, any program that packs and unpacks bits or uses disk storage is doomed to be slow and, probably, non-portable.

$O(MN)$-space methods permit the construction of all optimal alignments. However, the number of alignments that attain the minimum cost is often astronomical, in part because a brute force enumeration lists many arrangements whose differences are insignificant to the user. Moreover, when one is searching for a particular ''biologically meaningful'' arrangement, it may be necessary to consider slightly sub-optimal alignments (Waterman, 1983; Waterman and Byers, 1985). One alternative to explicitly constructing all optimal alignments is to modify our linear-space program to produce ''left-most'' and ''right-most'' optimal alignments that delineate the range of possibilities. In any case, it is important to understand that a single optimal alignment can be found in far less space than is needed to record ''traceback'' information for finding all optimal alignments.

Hirschberg's original presentation treats a simpler alignment problem, known as the longest common subsequence problem, where $w(a, b) = 1$ if $a \neq b$, $w(a, a) = 0$, and $gap(k) = k$, However, the approach is quite general. To the best of our knowledge, any sequence comparison algorithm whose ''cost-only'' version runs in $O(N)$ space can be adapted to produce an optimal alignment in $O(N)$ space. For example, Myers (1986) accomplished this for a ''greedy'' alignment algorithm that is quite different from the traditional dynamic programming approach. Miller and Myers (1988) applied Hirschberg's technique to a concave gap penalty algorithm that subsumes Gotoh's algorithm as a special case.

In this note we apply Hirschberg's technique to Gotoh's algorithm. Limiting consideration to a relatively simple method yields a simple and novel development that we hope will bring Hirschberg's idea to a wider audience. Moreover, for affine indel costs, the more general concave-weights software (Miller and Myers, 1988) runs 3.0 times slower and uses 3.5 times more space than the program described in this paper.

System and Methods

C software implementing the algorithm was written and tested on a Vax 11/780 running 4.3 BSD Unix. The program is portable: setting an appropriate compilation constant adapts the software to a machine with a different memory capacity. The only requirement is an ANSI-standard C compiler and accompanying standard I/O library.

The Algorithm

1. Computing the Cost in Linear Space

Let A_i denote the i-symbol prefix $a_1 a_2 \cdots a_i$ of A and let B_j denote $b_1 b_2 \cdots b_j$. Define

$C(i, j)$ = minimum cost of a conversion of A_i to B_j
$D(i, j)$ = minimum cost of a conversion of A_i to B_j that deletes a_i
$I(i, j)$ = minimum cost of a conversion of A_i to B_j that inserts b_j

Note that $D(i, j)$ is properly defined only when $i > 0$, and $I(i, j)$ only for $j > 0$. Gotoh (1972) showed how to compute the C, D, and I matrices in $O(MN)$ time. Below we present Gotoh's method, where we have treated the boundary conditions carefully by defining $D(0, j)$ and $I(i, 0)$ appropriately.

The values $C(i, j)$ satisfy the recurrence relations:

$$C(i,j) = \begin{cases} \min\{D(i,j),\ I(i,j),\ C(i-1,j-1) + w(a_i, b_j)\} & \text{if } i > 0 \text{ and } j > 0 \\ gap(j) & \text{if } i = 0 \text{ and } j > 0 \\ gap(i) & \text{if } i > 0 \text{ and } j = 0 \\ 0 & \text{if } i = 0 \text{ and } j = 0 \end{cases} \quad [*]$$

For $i, j > 0$, an optimal conversion of A_i to B_j ends with either (1) a delete, (2) an insert, or (3) the replacement of a_i by b_j. Thus, the first line above follows readily. For $j > 0$, an optimal conversion of A_0 (the empty sequence) to B_j must insert all j symbols, so $C(0, j) = gap(j)$ and the second line follows. The remaining two lines follow similarly. In the recurrence, and in Figures 1A and 1B, certain lines are starred because they are subsequently modified.

As noted earlier, we are free to pick a definition of $D(0, j)$. It is convenient to set $D(0, j) = C(0, j) + g$ for $j > 0$. Moreover, we need not compute $D(i, 0)$ for $i \geq 0$, since other quantities do not depend on these values. Then

$$D(i,j) = \begin{cases} \min\{D(i-1,j),\ C(i-1,j) + g\} + h & \text{if } i > 0 \text{ and } j > 0 \\ C(0,j) + g & \text{if } i = 0 \text{ and } j > 0 \end{cases}$$

If $i > 1$, then extending an optimal conversion of A_{i-1} to B_j so that it deletes a_i adds $g + h$ to its cost, or h if it ends by deleting a_{i-1}. This reasoning confirms the first line for $i > 1$. For the case where $i = 1$, an optimal conversion of A_1 to B_j ending with a delete must convert A_0 to B_j and then delete a_1. Thus, $D(1, j) = C(0, j) + gap(1)$, which is exactly the assignment implied by the recurrence because $D(0, j) = C(0, j) + g$.

I is handled like D. Thus, if we define $I(i, 0) = C(i, 0) + g$ for $i > 0$ and ignore $I(0, j)$ for $j \geq 0$, then

$$I(i,j) = \begin{cases} \min\{I(i,j-1),\ C(i,j-1) + g\} + h & \text{if } i > 0 \text{ and } j > 0 \\ C(i,0) + g & \text{if } i > 0 \text{ and } j = 0 \end{cases}$$

The recurrence relations for C, D, and I lead to the algorithm of Figure 1A, which uses a variable t that runs through the sequence of values $gap(1), gap(2), \cdots$.

```
arrays C[0..M, 0..N], D[0..M, 0..N], I[0..M, 0..N]
scalar t

C(0, 0) ← 0
t ← g
for j ← 1 to N do
  { C(0, j) ← t ← t + h
    D(0, j) ← t + g
  }
[*] t ← g
for i ← 1 to M do
  {
    C(i, 0) ← t ← t + h
    I(i, 0) ← t + g
    for j ← 1 to N do
      { I(i, j) ← min {I(i, j−1), C(i, j−1) + g} + h
        D(i, j) ← min {D(i−1, j), C(i−1, j) + g} + h


        C(i, j) ← min {D(i, j), I(i, j), C(i−1, j−1) + w(a_i, b_j)}
      }
  }
write "cost is" C(M, N)
```

Figure 1A: Gotoh's algorithm

```
vectors CC[0..N], DD[0..N]
scalars e, c, s, t

CC(0) ← 0
t ← g
for j ← 1 to N do
  { CC(j) ← t ← t + h
    DD(j) ← t + g
  }
[*] t ← g
for i ← 1 to M do
  { s ← CC(0)
    CC(0) ← c ← t ← t + h
    e ← t + g
    for j ← 1 to N do
      { e ← min {e, c + g} + h
        DD(j) ← min {DD(j), CC(j) + g} + h
        c ← min {DD(j), e, s + w(a_i, b_j)}
        s ← CC(j)
        CC(j) ← c
      }
  }
write "cost is" CC(N)
```

Figure 1B: O(N) space cost-only version

Values in the i^{th} rows of C and D depend only on values in rows i and $i-1$, while values in the i^{th} row of I depend only on values in row i. This means that a handful of row-sized vectors are adequate to compute successive rows. In fact, with a little care, two vectors suffice: if CC and DD contain the $i-1^{st}$ rows of C and D, then the i^{th} rows may be computed by overwriting values for the $i-1^{st}$ rows in a left-to-right sweep with the aid of three scalars, e, c, and s. Specifically, if $i, j > 0$, then immediatedly before $C(i, j)$, $D(i, j)$, and $I(i, j)$ are assigned to $CC(j)$, $DD(j)$, and e, respectively, we have:

$$CC(k) = \begin{cases} C(i, k) & \text{if } k < j \\ C(i-1, k) & \text{if } k \geq j \end{cases}$$

$$DD(k) = \begin{cases} D(i, k) & \text{if } k < j \\ D(i-1, k) & \text{if } k \geq j \end{cases}$$

$$e = I(i, j-1)$$
$$c = C(i, j-1)$$
$$s = C(i-1, j-1)$$

With this loop-invariant condition in mind, the $O(N)$ space cost-only variation of Figure 1B is readily understood.

Example 1. Let $w(a, b) = 1$ if $a \neq b$, $w(a, a) = 0$, and $gap(k) = 2 + 0.5k$. The unique optimal conversion of *agtac* to *aag* deletes *gt* and replaces *c* by *g*, for a total cost of 4. When

applied to these two sequences, the algorithm of Figure 1A computes the values in Table 1. Entries denoted '*' are undefined.

C-matrix:

0.0	2.5	3.0	3.5	
2.5	0.0	2.5	3.0	
3.0	2.5	1.0	2.5	
3.5	3.0	3.5	2.0	
4.0	s 3.5	3.0	4.5	
4.5	c 4.0	4.5	4.0	
CC				

D-matrix

*	4.5	5.0	5.5	
*	5.0	5.5	6.0	
*	2.5	5.0	5.5	
*	3.0	3.5	5.0	
*	3.5	4.0	4.5	
*	4.0	4.5	5.0	
DD				

I-matrix;

*	*	*	*	
4.5	5.0	2.5	3.0	
5.0	5.5	5.0	3.5	
5.5	6.0	5.5	6.0	
6.0	6.5	6.0	5.5	
6.5	e 7.0	6.5	7.0	

Table 1: Arrays C, D, and I computed by Fig. 1A for sequences agtac and aag.

In place of the three arrays, the algorithm of Figure 1B keeps only a vector for *C*, a vector for *D*, and scalars for $I(i, j-1)$, $C(i, j-1)$, and $C(i-1, j-1)$. At the top of the inner loop when $i = 5$ and $j = 2$, the contents of *CC* and *DD* are the values enclosed in boxes in Table 1, while the contents of *e*, *c*, and *s* are circled.

2. Delivering a Conversion in Linear Space

Hirschberg (1975) presented a recursive divide-and-conquer algorithm for delivering a longest common subsequence in linear space. Generalizing his specific treatment, the central idea is to find the ''midpoint'' of an optimal conversion using a ''forward'' and a ''reverse'' application of the linear space cost-only variation. Then an optimal conversion can be delivered by recursively determining optimal conversions on both sides of this midpoint.

Suppose $M > 1$ and $N > 0$. Let $i^* = \lfloor M/2 \rfloor$, so row i^* properly bisects the *C* matrix. In a forward phase, the linear space cost-only algorithm is applied to the strings A_{i^*} and *B*, resulting in vectors *CC* and *DD* satisfying:

$CC(j)$ = minimum cost of a conversion of A_{i^*} to B_j
$DD(j)$ = minimum cost of a conversion of A_{i^*} to B_j that ends with a delete

Let $rev(A)$ denote the reverse of *A*, i.e., $a_M a_{M-1} \cdots a_1$, and let A_i^T denote the suffix $a_{i+1} a_{i+2} \cdots a_M$ of *A*. (Recall that $A_i = a_1 a_2 \cdots a_i$.) Define $rev(B)$ and B_j^T similarly. Note that $rev(rev(A)_{M-i}) = A_i^T$. In a reverse phase, the linear space cost-only algorithm is applied to $rev(A)_{M-i^*}$ and $rev(B)$, with a new pair of vectors, *RR* and *SS*, filling the roles of *CC* and *DD*. Upon completion, $RR(j)$ is the minimum cost of a conversion of $rev(A)_{M-i^*}$ to $rev(B)_j$ and $SS(j)$ is the minimum cost of a conversion of $rev(A)_{M-i^*}$ to $rev(B)_j$ that ends with a delete.

But the reverse of a conversion of $rev(A)_{M-i}$ to $rev(B)_{N-j}$ is a conversion of A_i^T to B_j^T. Thus, the resulting vectors satisfy:

$RR(N-j)$ = minimum cost of a conversion of A_{i*}^T to B_j^T
$SS(N-j)$ = minimum cost of a conversion of A_{i*}^T to B_j^T that *begins* with a delete

Recall that the algorithm of Figure 1B does not compute $DD(0)$ and $SS(0)$, which are needed below. This is easily rectified by observing that $DD(0) = CC(0)$ and $SS(0) = RR(0)$.

Given the vectors above, the midpoint of an optimal conversion can be found using the following observation. For any conversion of A to B, there exists a $j \in [0, N]$ such that the conversion is the concatenation of either (1) a conversion of A_{i*} to B_j and a conversion of A_{i*}^T to B_j^T or (2) a conversion of A_{i*} to B_j ending with a delete and and a conversion of A_{i*}^T to B_j^T beginning with a delete, in which case the deletions bracketing the concatenation point must be coalesced into a single operation. For fixed j, the minimum cost of a type 1 conversion with midpoint $(i*, j)$ is $CC(j) + RR(N-j)$, i.e., the minimum cost of a conversion of A_{i*} to B_j plus the minimum cost of a conversion of A_{i*}^T to B_j^T. Similarly, the minimum cost of a type 2 conversion is $DD(j) + SS(N-j) - g$, where g is subtracted because bracketing deletes are coalesced into a single operation, i.e., $gap(x+y) = gap(x) + gap(y) - g$. Thus, the optimal cost of converting A to B is

$$\min_{j \in [0, N]} \{ \min(CC(j) + RR(N-j),\ DD(j) + SS(N-j) - g) \}$$

If the minimum is attained at $j*$, then $(i*, j*)$ is an *optimal midpoint* for the problem. When several values attain the minimum, the method of breaking ties determines whether the ‘‘left-most’’ or ‘‘right-most’’ optimal alignment is selected.

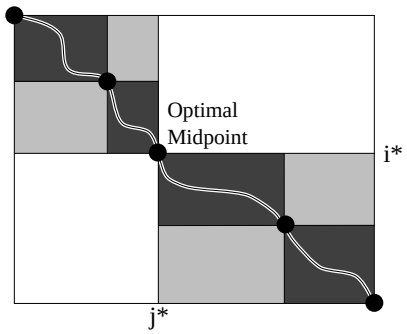

Figure 2: Splitting the problem into subproblems

Given an optimal midpoint $(i*, j*)$, an optimal conversion can then be delivered by (1) recursively finding an optimal conversion of A_{i*} to B_{j*}, (2) recursively finding an optimal conversion of A_{i*}^T to B_{j*}^T, and (3) concatenating these two partial conversions, being sure to coalesce bracketing deletes in the type 2 case. The splitting of the comparison problem for A and B into two smaller problems is pictured in Figure 2. The outer rectangle is the $M \times N$ C-matrix for A and B. The singly-hatched rectangles depict the sub-problems whose solutions are

to be concatenated, and the doubly-hatched rectangles depict sub-sub-problems. The dashed line indicates the eventual optimal alignment.

With type 2 midpoints, one must further constrain the first recursive call to conversions that end with a delete, and the second to conversions that begin with a delete. For example, the second sub-problem may have a script not beginning with a delete that is better when considered in isolation. However, since an initial delete is not charged the gap initialization penalty g for type 2 midpoints, a conversion beginning with a delete is actually superior when concatenated with the conversion for the first sub-problem. Considering sub-sub-problems, it becomes apparent that, in general, a recursive call may be required to consider only conversions that begin with, end with, or both begin and end with a delete.

The most elegant solution for type 2 midpoints is to split the problem into three parts: (1) an optimal conversion of A_{i^*-1} to B_{j^*}, where final deletes are not charged the gap initialization penalty g, (2) deletion of $a_{i^*}a_{i^*+1}$, and (3) an optimal conversion of A_{i^*+1} to B_{j^*}, where initial deletes are not charged g. For a cost-only problem where initial deletes are not charged for gap initialization, it suffices to simply subtract g from the starred line in the recurrence for C given in the previous sub-section. This is equivalent to setting t to 0, as opposed to g, in the starred lines of Figures 1A and 1B. Thus, in the forward phase, *CC* and *DD* are computed with this slight alteration if initial gaps are not to be penalized g. Because the latter half of a conversion is computed in the reverse phase, it suffices to use the same alteration when computing *RR* and *SS* on the reversed sequences in order to not penalize final gaps. To implement these conditional alterations, the algorithm *diff* in Figure 3 has parameters *tb* and *te* that are used to initialize t in the starred lines for the forward and reverse phases. The caller passes g if initial/final deletes are to be charged for gap initialization, and 0 otherwise.

The recursion's boundary cases, $N = 0$ and $M \leq 1$, are handled by exhaustive examination of all possible optimal conversions. When $N = 0$, the only possibility is to delete A. When $M = 0$, the only possibility is to insert B. When $M = 1$, an optimal conversion is the least costly of (1) inserting B and deleting $A = a_1$ or (2) inserting B_{j-1}, replacing a_1 by b_j, and inserting B_j^T, for some $j \in [1, N]$. Conversion (1) costs $gap(1) + gap(N)$ if initial and final deletes are charged a gap initialization penalty, but costs only $h + gap(N)$ otherwise. Also, the order of the insertion and deletion must be reversed if only initial deletes are not charged a gap penalty.

Figure 3 outlines a linear space alignment algorithm that writes an optimal conversion. To simplify the presentation, delete operations bracketing a type 2 midpoint are not coalesced. Our software package rectifies this deficiency by buffering the last operation to be written and coalescing it with the next as necessary.

```
shared vectors CC[0..M], DD[0..M], RR[0..N], SS[0..N]

procedure DIFF(A, B, M, N)
  { diff(A, B, M, N, g, g) }

recursive procedure diff(A, B, M, N, tb, te)
  { if N = 0 then
      { if M > 0 then write ''delete A'' }
    else if M = 0 then
        write ''insert B''
    else if M = 1 then
        write conversion of cost min{(min(tb, te)+h)+gap(N), min_{j∈[1, N]} (gap(j−1)+w(a_1, b_j)+gap(N−j))}
    else
      { i* ← ⌊M/2⌋
        Compute CC and DD in a forward phase, replacing [*] of Fig. 1B with ''t ← tb''.
        Compute RR and SS in a reverse phase, replacing [*] of Fig. 1B with ''t ← te''.
        Find j*∈[0, N] minimizing min(CC(j) + RR(N−j), DD(j) + SS(N−j) − g)
        if (i*, j*) is type 1 then
          { diff(A_{i*}, B_{j*}, i*, j*, tb, g)
            diff(A^T_{i*}, B^T_{j*}, M − i*, N − j*, g, te)
          }
        else
          { diff(A_{i*−1}, B_{j*}, i* − 1, j*, tb, 0)
            write ''delete a_{i*} a_{i*+1}''
            diff(A^T_{i*+1}, B^T_{j*}, M − i* − 1, N − j*, 0, te)
          }
      }
  }
```

Figure 3: Skeleton of Gotoh's algorithm in O(N) space

Example 2. Given the costs and sequences of Example 1, the algorithm of Figure 3 first applies *diff* to sequences *agtac* and *aag*, where $M = 4$, $N = 3$, and $i^* = 2$. The computed vectors are

CC:	3.0	2.5	1.0	2.5		*DD:*	3.0	2.5	5.0	5.5
RR:	3.5	4.0	3.5	2.0		*SS:*	3.5	4.0	3.5	6.0

There are eight possible ways to divide the problem, i.e., two types of midpoints for each $j \in [0, N]$. The corresponding costs are

	$j = 0$	$j = 1$	$j = 2$	$j = 3$
type 1 midpoint:	5.0	6.0	5.0	6.0
type 2 midpoint:	7.0	4.0	7.0	7.0

The optimum choice from among the eight possibilities is a type 2 midpoint at $j^* = 1$. This corresponds to combining (1) a minimum-cost conversion of $A_{i^*} = ag$ to $B_1 = a$ that ends with a delete and (2) a minimum-cost conversion of $A^T_{i^*} = tac$ to $B^T_j = ag$ that begins with a delete.

Combining the two scripts and adjusting the sum of their costs to account for the fact that only one gap initialization penalty is required, gives the cost $DD(1) + SS(2) - 2 = 4$.

The problem is thus decomposed into the problems of optimally converting $A_1 = a$ to $B_1 = a$, deleting $a_2 a_3 = gt$, and optimally converting $A_3^T = ac$ to $B_1^T = ag$. These two required optimal subconversions are generated by recursive calls to *diff*. For the first call, $M = N = 1$ and the final ''boundary case'' of *diff* produces the script of cost $gap(0) + w(a, a) + gap(0) = 0$, i.e., replacement of *a* by *a*. The second call divides the conversion of *ac* to *ag* into a conversion of *a* to *a* followed by a conversion of *a* to *g*, each of which is generated by a third-level call to *diff* with $M = N = 1$.

Performance. The algorithm uses $O(N + lg\ M)$ space: $O(N)$ for the globally shared vectors and $O(lg\ M)$ for the implicit activation stack needed for no more than $\lceil lg\ M \rceil + 1$ levels of recursion. The time required is approximately twice that for the cost-only version. There exist constants π and τ such that the time taken in the body of *diff* for an $M \times N$ problem is not more than $\pi(M + N)$ for the boundary cases and τMN for the recursive cases. It follows by induction that the total time taken in the worst case, including recursive calls, is not more than $(2 - 1/M)\tau MN + \pi(M + N)$. This result can be understood informally by examining Figure 2. The body of the top-level call takes τMN time, the total time spent in the bodies of the two sub-problems is $\frac{1}{2}\tau MN$, the total time spent in the bodies of the four sub-sub-problems is $\frac{1}{4}\tau MN$, and so on. Thus, the cumulative time is $(1 + \frac{1}{2} + \frac{1}{4} + \cdots)\tau MN \leq 2\tau MN$. A similar induction shows that the total time taken *in expectation* is $(2 - 2/M)\tau MN + \pi(M + N)$. The small difference between expected and worst-case time explains the surprisingly uniform time performance observed in practice. That is, for sequences of a given length, the algorithm's running time is virtually independent of the specific characters in the sequences.

Implementation

Our software package's dominant storage requirements are (1) $4N$ words for the vectors *CC*, *DD*, *RR*, and *SS*, (2) $M + N$ words for an optimal conversion, (3) 16 kilowords for the table, *w*, of replacement costs, and (4) $M + N$ bytes for the sequences *A* and *B*. Only the storage for the vectors is part of the package, *per se*. The other three storage components are declared in the user program and are largely avoidable. Operations converting *A* to *B* could be printed immediately, as in Figure 3. We store them to provide a more flexible user interface. The 128×128 table *w* need only be $\alpha \times \alpha$, where α is the alphabet size. *A* and *B* could be compressed; with DNA sequences, for example, only $2(M + N)$ bits are necessary.

The following table gives maximum lengths for sequences that can be aligned in a given amount of memory. The linear-space algorithm is compared with the $7MN$-bit approach of Altschul and Erickson (1986). Values are tabulated both without and with $M + N$ words for storing the generated operations, and we assume that 1 word = 4 bytes = 32 bits. In practice, figures for the linear-space algorithm are lowered slightly by the $O(lg\ M)$ space for the recursion stack.

Available Memory (in bytes)	Linear Space (ops not stored)	Linear Space (ops stored)	Altschul & Erickson
64K	4,000	2,666	270
128K	8,000	5,333	382
256K	16,000	10,666	540
1,000K	62,500	41,666	1,069

The software's time requirement is modest. Our Vax 11/780 running 4.3 BSD Unix needs an average of 153*MN* micro-seconds to align sequences of lengths *M* and *N*. Statistics that are easier to interpret, and relatively machine-insensitive, can be determined by comparison with a ''standard'' program. We chose the following straightforward implementation of the classic dynamic programming algorithm for the case where $gap(k) = hk$ (Wagner and Fischer, 1974). To facilitate comparative testing, the procedure arguments match the software interface described below. Our linear-space software's execution time exceeds that of the simple program by the factor 1.84.

```
#define NMAX 400

float C[NMAX+1][NMAX+1];

float DIFF(A, B, M, N, W, G, H, S)
char A[], B[]; int M, N; float W[][128], G, H; int S[];
{ register int i, j;
  register float c, d, e;

  C[0][0] = 0.;
  for (j = 1; j <= N; j++)
    C[0][j] = C[0][j-1] + H;
  for (i = 1; i <= M; i++)
    { C[i][0] = C[i-1][0] + H;
      for (j = 1; j <= N; j++)
        { c = C[i-1][j-1] + W[A[i]][B[j]];
          d = C[i-1][j] + H;
          e = C[i][j-1] + H;
          if (d < c) c = d;
          if (e < c) c = e;
          C[i][j] = c;
        }
    }
  return C[M][N];
}
```

The software package consists of three C files. The file *linear.h* gives complete interface information and several global definitions, *linear.c* contains the implementation, *per se*, together with a procedure to display alignments, and *sample.c* provides a sample user program. Following standard C conventions, the user program should include *linear.h* and should be compiled and linked with *linear.c*. The remainder of this section describes the interface.

#define NMAX <*integer*>

NMAX is a compilation constant giving the maximum input sequence length. It is to be adjusted according to available memory.

float DIFF(A, B, M, N, W, G, H, S) **int** M, N; **char** A[], B[]; **float** W[][128], G, H; **int** S[];

DIFF compares sequence $A[1..M]$ with sequence $B[1..N]$ and returns the minimum conversion cost. Costs are determined by the parameters *W*, *G*, and *H*. $W[128][128]$ is an array giving replacement costs for each pair of ASCII characters, e.g. $W['a']['b']$ is the cost of replacing $'a'$ by $'b'$. Be sure to set $W['a']['a']$ to zero if exact matches are to accrue no cost. The cost of a *k*-symbol indel is the affine function $G + Hk$.

DIFF also has the side-effect of placing an encoding of an optimal conversion in an integer array $S[0..M+N-1]$ supplied by the caller. The sequence of integers $S[0]$, $S[1]$, $S[2]$, $\cdots$ gives the editing operations in a left-to-right conversion where integers encode operations as follows:

0 => replace
$-k$ => delete *k* symbols
$+k$ => insert *k* symbols.

The script is guaranteed to have the properties:

(1) Inserts are never followed by inserts.
(2) Deletes are never followed by deletes or inserts.
(3) A replacement followed by a *k*-indel is always preferred to a *k*-indel followed by a replacement if both have the same cost.

DIFF returns −1.0 if *NMAX* isn't large enough.

int DISPLAY(A, B, M, N, S) **int** M, N; **char** A[], B[]; **int** S[];

DISPLAY places on the standard output a display of the alignment implied by the conversion *S* computed in the call *DIFF*(*A*, *B*, *M*, *N*, ?, ?, ?, *S*). For example:

```
 0    .    :    .    :    .    :    .    :    .    :
  ggcgtttcataccggcgagga  ctagagatcccagatgcagcctcgata
  !-!!!!|||!!!!!!!!!!!!|--!!!!!|!!!|!!!!!!!-!!!!!!!!!
  g cgttcataaccggcgaggtacctagacattcccagagc gcctcgata

50    .    :    .    :    .
  taggaagaa tc agcaacgatcggcatg
  !|!||!!!!-!!-!!!!!!!!-!!!|!-!!
  tggacagaaatcgagcaacga cgac tg
```

Discussion

This paper develops a linear-space algorithm for producing optimal sequence alignments with affine gap costs. It is superior in theory and practice to other approaches. By avoiding the use of

secondary storage and bit operations, it yields fast and portable software.

The underlying divide-and-conquer strategy, taken from a 1975 paper of Hirschberg, is quite general. Many, perhaps all, cost-only alignment algorithms yield an alignment-delivering variation with identical asymptotic time and space complexities. When applied to certain other alignment algorithms, the space requirement becomes sublinear (Myers, 1986), linear (Wagner and Fischer, 1974; Masek and Paterson, 1980), or linear in expectation (Miller and Myers, 1988). Occasionally, the variation is not space-efficient, as with the method of Waterman, Smith, and Beyer (1976), whose cost-only version needs $O(MN)$ space.

In practice, employing the strategy at most doubles the time and space requirements of the cost-only version. Indeed, with greedy methods (Fickett, 1984; Ukkonen, 1985, Section 3; Miller and Myers, 1985; Myers, 1986), the midpoint computation is twice as efficient as a one-pass cost-only computation, implying that the time overhead of the divide-and-conquer approach is negligible.

Acknowledgment

Stephen Altschul, David Lipman, and the referee made suggestions that improved the presentation of this paper.

References

Altschul, S. and B. W. Erickson (1986) Optimal sequence alignments using affine gap costs, *Bull. Math. Biol.*, **48**, 603-616.

Fickett, J. W. (1984) Fast optimal alignment, *Nucleic Acids Research*, **12**, 175-179.

Gotoh, O. (1982) An improved algorithm for matching biological sequences, *J. Molec. Biol.*, **162**, 705-708.

Gotoh, O. (1986) Alignment of three biological sequences with an efficient traceback procedure, *J. Theor. Biol.*, **121**, 327-337.

Gotoh, O. (1987) Pattern matching of biological sequences with limited storage, *CABIOS*, **3**, 17-20.

Hirschberg, D. S. (1975) A linear space algorithm for computing longest common subsequences, *Commun. Assoc. Comput. Mach.*, **18**, 341-343.

Masek, W. J. and M. S. Paterson (1980) A faster algorithm for computing string-edit distances, *J. Comput. System Sci.*, **20**, 18-31.

Miller, W. and E. W. Myers (1985) A file comparison program, *Software — Practice and Experience*, **15**, 1025-1040.

Miller, W. and E. W. Myers (1988) Sequence comparison with concave weighting functions, *Bull. Math. Biol.*, to appear.

Myers, E. W. (1986) An O(ND) difference algorithm and its variations, *Algorithmica*, **1**, 251-266.

Smith, T. F., M. S. Waterman, and W. M. Fitch (1981) Comparative biosequence metrics, *J. Molec. Evol.*, **18**, 38-46.

Taylor, P. (1984) A fast homology program for aligning biological sequences, *Nucleic Acids Research* **12**, 447-455.

Ukkonen, E. (1985) Algorithms for approximate string matching, *Information and Control*, **64**, 100-118.

Wagner, R. A. and M. J. Fischer (1974) The string-to-string correction problem, *J. ACM*, **21**, 168-173.

Watanabe, K., Y. Urano, and T. Tamaoki (1985) Optimal alignments of biological sequences on a microcomputer, *CABIOS*, **1**, 83-87.

Waterman, M. S. (1983) Sequence alignment in the neighborhood of the optimum, *Proc. Natl. Acad. Sci. USA*, **80**, 3123-3124.

Waterman, M. S. and T. H. Byers (1985) A dynamic programming algorithm to find all solutions in a neighborhood of the optimum, *Math. Biosciences*, **77**, 179-188.

Waterman, M. S., T. F. Smith and W. A. Beyer (1976) Some biological sequence metrics, *Advances in Mathematics*, **20**, 367-387.