

## Local sequence alignments with monotonic gap penalties

Richard Mott\*

SmithKline-Beecham Pharmaceuticals R & D, New Frontiers Science Park (North),  
3rd Avenue, Harlow CM19 5AW, UK

Received on December 22, 1998; revised on March 9, 1999; accepted on March 11, 1999

### Abstract

**Motivation:** Sequence alignments obtained using affine gap penalties are not always biologically correct, because the insertion of long gaps is over-penalised. There is a need for an efficient algorithm which can find local alignments using non-linear gap penalties.

**Results:** A dynamic programming algorithm is described which computes optimal local sequence alignments for arbitrary, monotonically increasing gap penalties, i.e. where the cost  $g(k)$  of inserting a gap of  $k$  symbols is such that  $g(k) \geq g(k-1)$ . The running time of the algorithm is dependent on the scoring scheme; if the expected score of an alignment between random, unrelated sequences of lengths  $m$ ,  $n$  is proportional to  $\log mn$ , then with one exception, the algorithm has expected running time  $O(mn)$ . Elsewhere, the running time is no greater than  $O(mn(m+n))$ . Optimisations are described which appear to reduce the worst-case run-time to  $O(mn)$  in many cases. We show how using a non-affine gap penalty can dramatically increase the probability of detecting a similarity containing a long gap.

**Availability:** The source code is available to academic collaborators under licence.

**Contact:** Richard.Mott@well.ox.ac.uk

### Introduction

Gapped local sequence alignments are almost universally calculated using the Smith–Waterman algorithm (Smith and Waterman, 1981) with an affine gap penalty, where the cost of inserting a run of  $k$  letters is  $g(k) = A + Bk$ , for positive constants  $A, B$ . These alignments may be computed in  $O(mn)$  time, where  $m, n$  are the sequence lengths (Gotoh, 1982). The results are generally satisfactory, but there is evidence that other types of gap penalty may be more appropriate in some circumstances.

For example Benner *et al.* (1993) and Gu and Li (1995) suggest using a logarithmic gap penalty of the form  $A + B$

$\log k$ . The logarithmic penalty is a special case of the class of convex gap penalties (also called concave by some authors), where  $g(k+1) - g(k) \leq g(k) - g(k-1)$ , and which have been suggested as being generally more suited than the affine for alignments where long gaps are expected. Variations on the standard affine gap penalty expressly designed to allow long gaps include Gotoh (1990) and Mott (1997), for example for aligning cDNA sequences to genomic DNA, where the insertion of very long gaps is desirable. Altschul (1998) describes a generalised affine penalty which appears to be more sensitive than the standard. Despite these advances, non-affine gap penalties are not in general use, partly because it has been difficult to implement fast algorithms for them.

The first practical algorithm to find optimal global alignments using convex gap penalties was by Waterman (1984), who introduced the concept of the *candidate list*, described later. It was conjectured that, depending on the precise details of the algorithm, the algorithm's time complexity would be  $O(mn \log m)$  or even  $O(mn)$ . Miller and Myers (1988) gave rigorous treatments of two algorithms for convex gap penalties, also using candidate lists, with worst-case time complexity of  $O(mn \log m)$ . For certain types of gap penalty these algorithms are faster; for example for a piece-wise linear convex function made from  $K$  straight lines, the complexity is  $O(mn \log K)$ , and for logarithmic gap penalties  $A + B \log k$  it is  $O(mn)$ . Miller and Myers (1988) also compared the relative performance of Waterman's and their methods, and found that in some cases, depending on the scoring scheme and the degree of similarity of the sequences, the former had complexity  $O(mn(m+n))$ . See also Galil and Giancarlo (1989) and Gusfield (1997, pp. 293–302).

All the preceding methods deal with global alignments and convex gap penalties. This paper is mainly concerned with local alignments using the class of positive, *monotonically increasing* gap penalties, i.e. where  $g(k) \geq g(k-1)$ . The monotonic and convex classes overlap — for example the affine and logarithmic penalties are common to both. However, the monotonic class also includes the power law  $A + Bn^C$ , where  $C > 0$ , which is convex only for  $0 < C \leq 1$ . The convex class includes penalty functions which can decrease, and which are therefore non-monotonic, although it is diffi-

\*Present address: The Wellcome Trust Centre for Human Genetics, Roosevelt Drive, Oxford OX3 7BN, UK

cult to see any biological justification for using them. Probably the set of monotonic convex penalties is the most useful in practice. By contrast gap penalties which increase faster than linear (e.g. concave penalties) are unsuitable for biological applications, as they favour the use of many small gaps over fewer but longer insertions.

We describe an algorithm to find an optimal local alignment using any monotonic gap function. In order to understand the time complexity of the algorithm, it is necessary first to recall some facts about the statistical properties of alignments.

Depending on the severity of the scoring scheme (i.e. substitution matrix and gap penalty function) and the sequences' compositions, the expected score  $E(m, n)$  of the maximal local similarity between random, unrelated sequences of lengths  $m \geq n$  is either  $O(\log mn)$ , the *logarithmic domain*, or  $O(n)$ , the *linear domain*. See for example Waterman (1995). There is a phase transition between these two modes, where small changes in the scoring scheme can produce widely different alignments (Waterman *et al.*, 1998). The algorithm presented here has expected running time  $O(mn)$  in the logarithmic domain, with one exception described later, and  $O(mn(m+n))$  in the linear. We also show how to predict in which domain a given scoring scheme resides.

## Algorithm

### Definition

Our algorithm has some similarities with the candidate list methods of Waterman (1984), Miller and Myers (1988) and Galil and Giancarlo (1989), but also certain key differences, which exploit certain statistical properties of local alignment scores. Suppose we are comparing two sequences  $X_i, Y_j$  of lengths  $m, n$ . Let  $S(x, y)$  be the score for aligning the letters  $x, y$  and  $g(k)$  the cost of inserting a gap of  $k$  letters. We assume  $g(k) \geq g(k-1)$  for all  $k$ .

Let  $W_{ij}$  be the score of an optimal local alignment ending at  $(i, j)$ . Let  $D_{ij}$  be the score of the optimal local alignment ending at  $(i, j)$  such that  $X_i$  and  $Y_j$  are forced to align, or zero if this alignment has negative score. Similarly let  $H_{ij}$  be the score when a gap is inserted in the first sequence opposite  $Y_j$ , or zero, and  $V_{ij}$  when a gap is inserted in the second opposite  $X_i$ , or zero. In the familiar dot-matrix representation of dynamic programming, these quantities correspond to an optimal path to the cell  $(i, j)$ , entering it diagonally, horizontally and vertically respectively. Alignments where a gap in one sequence abuts a gap in the other are forbidden. Then the basic Smith–Waterman recursion is

$$W_{ij} = \max(D_{ij}, H_{ij}, V_{ij})$$

$W$  is computed for the entire  $m \times n$  dot-matrix. The maximal local similarity terminates at the coordinates  $(I, J)$  for which  $W_{ij}$  is a maximum, and the optimal alignment can be

reconstructed by back-tracking from this cell. The definitions of  $D, V, H$  are

$$D_{ij} = \max(0, W_{i-1, j-1} + S(X_i, Y_j))$$

$$V_{ij} = \max(0, \max_{k < j} (D_{ik} - g(j-k)))$$

$$H_{ij} = \max(0, \max_{k < i} (D_{kj} - g(i-k)))$$

The naive recursions for  $V, H$  have time complexity  $O(j)$  or  $O(i)$  respectively, resulting in an overall running time of  $O(mn(m+n))$ . However, it is possible to recast them in a much more efficient form. We will give the argument for  $V$ ; that for  $H$  is similar. For clarity, we drop the subscript  $i$  and write

$$V_j = \max(0, \max_{k < j} (D_k - g(j-k)))$$

The main idea is to maintain a candidate list  $L_j$  of those coordinates  $k$  for which  $D_k - g(j-k) > 0$ .  $V_j$  can be found by taking the maximum over these candidates only. For stringent gap penalties we will show that the expected size of the list is bounded, and hence the expected time taken to evaluate  $V_j$  is independent of the sequence lengths.

First we show how to compute the candidate list efficiently. Note that candidate list  $L_1$  is just the empty set. We then proceed by induction: suppose that we already know the contents of  $L_j$ . This list can contain any of the numbers in the range  $1 \dots j-1$ . It follows from the monotonicity of the gap penalty that  $L_{j+1}$  is a subset of  $j \cup L_j$ , i.e. any candidate for  $j+1$  must also be a candidate for  $j$ , or must be  $j$ :

$$\begin{aligned} L_{j+1} &= \{k < j+1 : D_k - g(j+1-k) > 0\} \\ &\subseteq \{j\} \cup \{k < j : D_k - g(j+1-k) > 0\} \\ &\subseteq \{j\} \cup \{k < j : D_k - g(j-k) > 0\} \\ &= \{j\} \cup L_j \end{aligned}$$

(1)

The monotonicity of  $g$  is used to get from the second to the third line above. Pseudocode for a function, `update_list`, that computes  $L_{j+1}$  given  $L_j$  is shown in Figure 1. The algorithm also computes  $V_j$  and the length of the gap, which is required for back-tracking. The running time for the update algorithm is  $O(|L_j| + 1)$  because each of the elements in the list must be examined, plus the potential new element  $j$ . The code fragment also includes two optimisations described later.

In contrast with local similarities considered here, global alignment algorithms do not restrict scores to be positive, and consequently our definition of the candidate list is new, although the update property (1) is common to the global alignment methods too. Instead, these rely on the convexity of the gap penalty to eliminate elements from the list, and one of the

```

update_list( next[], start, j, D[], g[], gap, is_convex ) {

    k = previous = start /* initialise */
    gap = max = 0

    while( k > 0 ) { /* go through the list */
        x = d[k]-g[j-k]
        if ( x > 0 ) { /* preserve this element */
            if ( max < x ) { /* it is the best so far */
                max = x
                gap = k
            }
            previous = k
            k = next[k]
        }
        else { /* delete this element */
            if ( k == start ) { /* start of the list */
                k = previous = start = next[k]
            }
            else { /* middle of the list */
                k = next[previous] = next[k]
            }
        }
    }

    /* Check if j should be added to the list */

    x = d[j]-g[1]
    if ( x > 0 ) {
        if ( max < x ) { /* the best so far */
            max = x
            gap = j
        }
        next[previous] = j
        next[j] = 0 /* end of the list */
        if ( start == 0 ) start = j
    }

    /* Optimisations of the list:
    (a) Remove all k < 1, where d[1] is max of d[] */

    if ( start > 0 ) {
        max2 = 0
        k = start
        while( k > 0 ) {
            if ( d[k] >= max2 ) {
                max2 = d[k];
                start = k; /* prunes the list */
            }
            k = next[k];
        }
    }

    /* (b) If the gap penalty is convex remove all
    k > 1' where d[l']-g[j-l'] is max */

    if ( is_convex && gap > 0 ) next[gap] = 0

    return max
}

```

**Fig. 1.** Pseudocode of the function `update_list` which computes the list  $L_{j+1}$  given the list  $L_j$ . `start` is the smallest member of the list  $L_j$ , or 0 if the list is empty. The list is represented by the array `next[ ]`, in which `k2 = next[start]` is the second member of the list, `k3 = next[k2]` the third member and so on, until `next[kn] = 0` indicates `kn` is the last element. The algorithm examines each member `k` in the list and tests whether  $D_k - g(j+1-k) > 0$ , in which case the member is retained, otherwise it is deleted. Finally, `j` is added to the list if  $D_j - g(1) > 0$ . The algorithm also returns `max = V_j`, and `gap`, the coordinate of the start of the gap.

optimisations described below re-uses the relatively simple method of Waterman (1984) to prune our candidate list still further. The more efficient algorithms of Miller and Myers (1988) and Galil and Giancarlo (1989) are rather complicated to describe here in detail, but essentially rely on convexity to determine efficiently whether an element in the list will always ‘dominate’ over another, in which case the latter may be deleted.

### Complexity

We now estimate  $E(|L_j|)$ , the expected size of the list  $L_j$ . It is known that provided the expected match score of any two symbols is negative and the gap penalties are severe enough to give alignments in the logarithmic domain there exist strictly positive constants  $R, \lambda$  such that, for alignments between random, unrelated sequences

$$\Pr (D_j > t) \leq Re^{-\lambda t} \quad (2)$$

It is important to note that this probability is independent of  $j$ . This result follows from the extreme-value theory for sequence matching in, for example, Arratia *et al.* (1988), Karlin and Altschul (1990), Mott (1992), Karlin and Dembo (1992), Waterman and Vingron (1994a) and Mott and Tribe (1999).

Therefore

$$\begin{aligned}
 E(|L_j|) &= \sum_{k < j} \Pr (D_k - g(j-k) > 0) \\
 &\leq \sum_{k < j} Re^{-\lambda g(j-k)} \\
 &\leq R \sum_{k < \infty} e^{-\lambda g(k)}
 \end{aligned} \quad (3)$$

(we ignore edge effects which do not affect the answer significantly). So if (2) holds *and* (3) converges then the expected size of the list is bounded, and consequently so is the time taken to compute  $V_{ij}$  (and  $H_{ij}$ ). Therefore, since the algorithm requires  $mn$  updates of the lists for  $H$  and  $V$ , the overall running time will be  $O(mn)$ .

Mott and Tribe (1999) showed the statistical behaviour of gapped alignments is well-characterised by a quantity  $\alpha$  which is a function of the sequences’ compositions, the substitution matrix and gap penalty. A formula for  $\alpha$  is given in the Appendix. When  $\alpha = 0$  no gaps are permitted (i.e. the gap penalties are infinite), and alignments will be in the logarithmic domain. As  $\alpha$  increases the gap penalties relax and gaps will begin to appear in alignments, and the parameter  $\lambda$  in (2,3) decreases, until the phase transition to linear behaviour occurs, at which point  $\lambda = 0$ . The precise  $\alpha_{\text{crit}}$  at which this occurs is unknown, but probably lies between 0.3 and 0.4. Values of  $\alpha > 0.25$  should be avoided in any case, in

order to maintain high sensitivity — database searches with affine penalties typically use  $\alpha$  in the range [0.05,0.2].

The time complexity of the algorithm therefore hinges on the convergence or otherwise of  $H(\lambda, g, n) = \sum_{k=1}^n e^{-\lambda g(k)}$  for different gap penalties and values of  $\lambda$ . This depends on the asymptotic behaviour of  $g(k)/\log(k)$ . The important cases are summarized in Table 1.

**Table 1.** Summary of the behaviour the series  $\sum_{k=1}^n e^{-\lambda g(k)}$  for different regimes of parameter  $\lambda$  and gap penalty  $g$ , and of the corresponding time-complexity of the monotone algorithm

Limit of $\frac{g(k)}{\log(k)}$ $K \rightarrow \infty$	$\lambda$	Behaviour of $\sum_{k=1}^n e^{-\lambda g(k)}$	Algorithm's average complexity
$\infty$	$\lambda > 0$	converges	$O(mn)$
$B < \infty$	$\lambda B > 1$	converges	$O(mn)$
$B < \infty$	$\lambda B = 1$	$O(\log(n))$	$O(mn(\log(m) + \log(n)))$
$B < \infty$	$\lambda B < 1$	$O(n^{1-\lambda B})$	$O(mn(m^{1-\lambda B} + n^{1-\lambda B}))$

The table indicates that  $H$  converges for any  $\lambda > 0$  if the gap penalty grows faster than  $\log(k)$ . For example, this is true for the affine and power law functions. Therefore, all gap penalties stronger than the logarithmic will have  $O(mn)$  average behaviour whenever the scoring scheme is in the logarithmic domain; in these cases one can determine the complexity of the algorithm simply by calculating  $\alpha$ .

The critical case is the logarithmic penalty  $g(k) = A + B \log(k)$ .  $H$  converges when  $\lambda B > 1$ , so the algorithm's complexity is  $O(mn)$  in this region as well. For strong gap penalties this is definitely the case, but as  $B$  decreases so does  $\lambda$ , and so  $\lambda B$  must eventually equal 1 before  $\lambda = 0$ .  $H$  grows like  $\log n$  when  $\lambda B = 1$ , and like  $n^{1-\lambda B}$  when  $\lambda B < 1$ . Consequently, for weak logarithmic gap penalties, but where the scoring scheme is in the logarithmic domain, the complexity is  $O(mn(\log m + \log n))$  or  $O(mn(m^{1-\lambda B} + n^{1-\lambda B}))$  respectively. In practice, fairly stringent gap penalties are the norm, so this behaviour is of mainly academic interest. Also, the series  $H(\lambda, A + B \log(\cdot), n)$  converges extremely slowly when  $\lambda B$  is only slightly greater than 1, so that asymptotic behaviour may not be applicable for typical sequence lengths in the low hundreds.

### Optimisation

These arguments apply only to average-case complexity, not worst-case. Indeed, when aligning a sequence against itself, the candidate lists can grow very large, and the algorithm becomes unacceptably slow, even when the expected complexity is  $O(mn)$ . However, some simple pruning of the candidate lists can be done which reduces running time dramatically in these cases:

(a) Suppose  $l$  is the member of the candidate list  $L_j$  for which  $D_l$  is a maximum. Then monotonicity implies that we can delete all members of  $L_j$  prior to  $l$  to form a reduced candidate list, because whenever  $k < l < j$

$$D_k - g(j - k) \leq D_l - g(j - l) \\ \text{and } D_k - g(j + 1 - k) \leq D_l - g(j + 1 - l)$$

so the update property (1) still holds.

(b) Suppose that in addition to being monotonic, the gap penalty is also convex. Then we can apply a variant of Waterman's argument (Waterman, 1984). Let  $l'$  be the least member of  $L$  for which  $D_k - g(j - k)$  is a maximum. By construction  $l' \geq l$ . Then we can delete all  $k > l'$  from  $L_j$  but still retain the update property (1), because if  $k > l'$

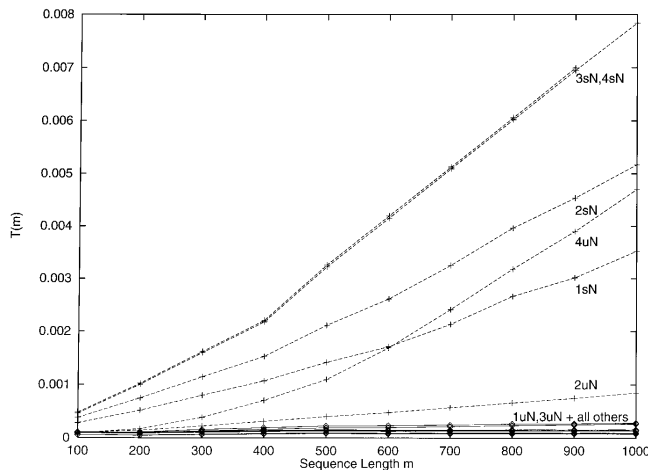
$$D_{l'} - g(j - l') \geq D_k - g(j - k) \\ D_{l'} - g(j + 1 - l') + g(j + 1 - l') - g(j - l') \geq \\ D_k - g(j + 1 - k) + g(j + 1 - k) - g(j - k) \\ \text{so } D_{l'} - g(j + 1 - l') \geq D_k - g(j + 1 - k)$$

by convexity. Thus if  $l'$  dominates  $k$  in  $L_j$  then it does so in  $L_{j+1}$  too.

Thus for monotonic convex gap penalties the candidate list can be truncated just to those values  $k \in L_j: l \leq k \leq l'$ . In practice we find that usually  $l = l'$ , so the candidate list is of length 1, or is empty. The effects of these optimisations on the list size are best illustrated by an example. For a comparison between two independent, randomly generated protein sequences of length 300 using the affine gap penalty  $10 + n$  and the BLOSUM62 matrix ( $\alpha = 0.1046$ ), the average length of a candidate list is about 0.50 without the optimisations 0.36 and with them. For a comparison between a random sequence and itself, the average list sizes were 98.5 (no optimisations) 34.4 (just (a)) and 0.94 with both optimisations, which is only marginally worse than average-case behaviour.

The effects of the optimisations on self-comparisons may be understood as follows: the main diagonal contains the top  $D$ -scores, but on both sides there will be positive  $D$ 's shadowing the main diagonal. The optimisation (a) removes those sub-optimal elements before the main diagonal while (b) deletes those following it.

As presented, the space complexity of the algorithm is  $O(mn)$  because it is necessary to maintain a list  $L_j$  for each of the  $m$  columns  $j$  in the dot-matrix (if the matrix is processed row-by-row then only the current row list is needed). Since the maximum size of a list is  $n$ , and other data structures require not more than  $O(mn)$  space, the overall requirement is still  $O(mn)$ . However, for scoring schemes in the logarithmic domain, it is possible to represent each list in  $O(1)$  space on average, at the cost of some extra processing time. In this case the lists can be stored in  $O(m + n)$  space, and it should be possible to compute the alignments in linear space and quadratic time, although we have not attempted to do so here.



**Fig. 2.** Scaled running time behaviour  $T(m)$  of the **monotone** algorithm, as a function of sequence length  $m$ .  $T(m)$  is the total time for comparisons between 100 pairs of random sequences, divided by  $m^2$ . Each graph corresponds to a different experiment, coded by: 1, 2, 3, 4 = which scoring scheme (see text), s = self-comparisons, u = comparisons between unrelated sequences, O = with optimisations, N = with none. Quadratic ( $O(m^2)$ ) behaviour is indicated by near-horizontal  $T(m)$ .

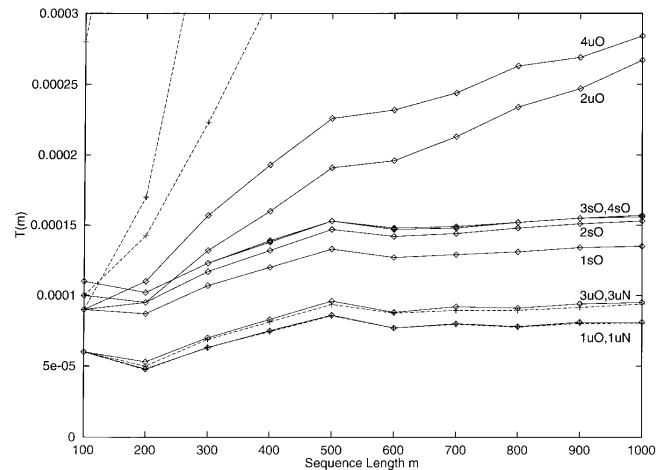
## Implementation and evaluation

The algorithm is implemented in a program called **monotone**, written in ANSI-C and compiled and run under Digital OSF1. The program accepts as input two FASTA-format sequences and a BLAST-format symbol comparison matrix. The following gap-penalties are supported: (a) logarithmic  $A + B \log n$ , (b) affine  $A + Bn$ , (c) power-law  $A + Bn^C$ , and (d) user-defined via an input file. The program calculates  $\alpha$  and so can usually determine the running time behaviour.

Because the affine penalty is a member of the class of monotonic penalties, we were able to check that **monotone** gave the same results as the standard Smith–Waterman in this instance.

The running time of the program is illustrated in Figures 2 and 3. Sets of 100 pairs of random protein sequences were generated, for lengths  $n = m = 100, 200, \dots, 900, 1000$ . Each pair was compared on a 400 MHz Digital ALPHAstation 500, using **monotone**, with the BLOSUM62 matrix (Henikoff and Henikoff, 1992) and

- (1) the affine penalty  $g(n) = 10 + 2n$  ( $\alpha = 0.0442$ , logarithmic domain)
- (2) the affine penalty  $g(n) = 5 + n$  ( $\alpha = 0.5072$ , linear domain)
- (3) the logarithmic penalty  $9 + 5 \log n$  ( $\alpha = 0.1217$ ,  $\lambda B = 1.208$  logarithmic domain)



**Fig. 3.** Magnified section of Figure 2, corresponding to the region  $T(m) < 0.0003$ .

- (4) the logarithmic penalty  $5 + 5 \log n$  ( $\alpha = 0.4299$ , linear domain).

$T(m)$ , the total CPU time divided by  $m^2$ , is plotted against  $m$ . Thus if the time complexity is  $O(m^2)$  then  $T(m)$  should be constant, subject to fluctuations caused by computer hardware, e.g. cache size. If the complexity is  $O(m^3)$  then the curves should increase linearly.

The timings were repeated with the optimisations (labelled ‘O’) and without them (labelled ‘N’), for comparisons between pairs of unrelated sequences (labelled ‘u’) and for self-comparisons (labelled ‘s’). For example, the time graph for self-comparisons without optimisations for case (1) are labelled 1sN.

The theoretical arguments presented above imply that, when the optimisations are not used, and the scoring scheme is in the logarithmic domain, i.e. cases 1uN and 3uN, comparisons between random unrelated sequences should have quadratic time complexity, but all other cases should be cubic. This is confirmed by Figure 2. The speeds, expressed in millions of cells processed per second, of 1uN and 3uN are 1.24 and 1.05.

There are interesting and unexplained phenomena in Figure 2: the times for the cases 3sN, 4sN are virtually identical, and 4uN increases at a steeper rate after  $m = 500$ .

When the optimisations are added, all comparisons are speeded up significantly, and the resulting curves occupy the bottom of Figure 2. This region is magnified in Figure 3. All cases except 2uO and 4uO (comparisons between unrelated sequences, with scoring scheme in the linear domain) appear to have quadratic behaviour, and even for these two cases the run times are reduced by factors of 32 and 180 respectively.

Thus provided the scoring scheme is in the logarithmic domain the worst-case complexity of the algorithm can be con-

trolled by adding the optimisations, and strongly suggest the complexity is  $O(mn)$  in the worst case, except for comparisons between unrelated sequences in the linear scoring domain. Interestingly, self-comparisons are *faster* than unrelated comparisons in this case. Further examination showed this was because for self-comparisons, scores along the main diagonal dominate random scores, so the candidate lists usually contain only one element, whereas for unrelated sequences the lists are not dominated.

### Sensitivity of the algorithm

As mentioned in the Introduction, the primary motivation for using non-affine gap penalties is to improve sensitivity. We now discuss how to predict the likelihood that a given scoring scheme will let us detect a similarity. Recall that a dynamic programming algorithm will find an optimal alignment, i.e. one that maximises the scoring scheme. If there is a genuine similarity present between two sequences, it will only be found if its score  $s$  exceeds  $r$ , that of the best random similarity that happens to be present as well. In other words the probability of detection, i.e. the statistical power, is just the probability  $\Pr(r \leq s)$  that a random similarity has a score less than  $s$ .

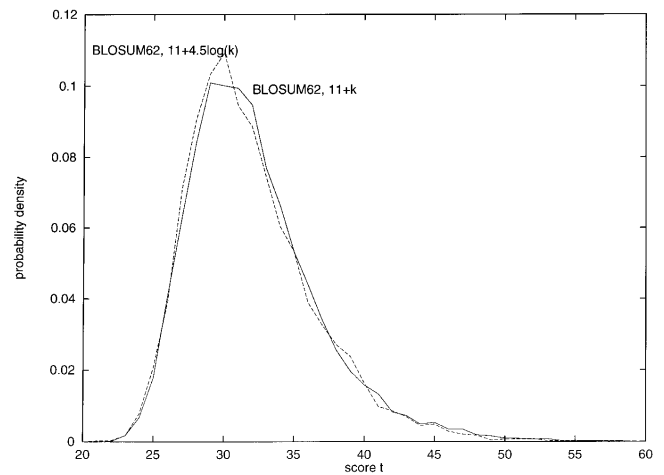
Now we know that provided the scoring scheme is in the logarithmic domain, to a good approximation the distribution of the score between random, unrelated sequences follows an extreme-value distribution

$$\Pr(r \leq t) = \exp(-K m n e^{-\lambda t}) \quad (4)$$

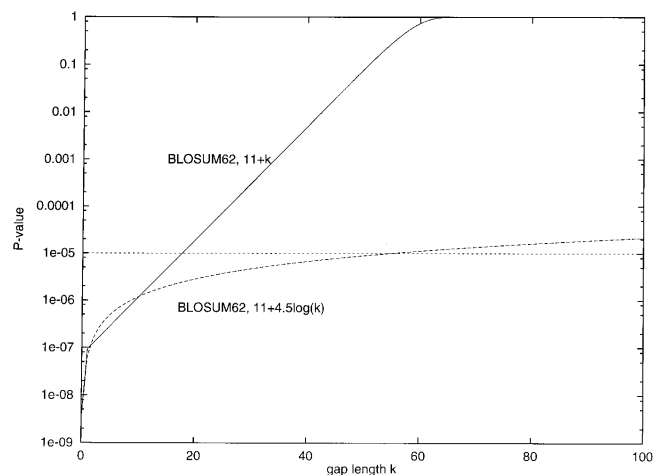
for constants  $K, \lambda$  depending on the scoring scheme, which can be estimated by simulation (Waterman and Vingron, 1994b; Altschul and Gish, 1996) or (for affine gap penalties) by the approximate formulas in Mott and Tribe (1999). Therefore we need only substitute  $s$  for  $t$  in (4) to obtain the sensitivity.

For the sake of concreteness, consider the following important example: suppose the real similarity comprises two ungapped domains, with total score  $D$ , separated by a gap of  $k$  residues. Then the correct alignment has score  $s(k) = D - g(k)$ . For a fixed scoring scheme, as the separation  $k$  increases,  $s(k)$ , and hence the detectability of the similarity, will decrease. However, the more slowly  $g(k)$  increases, the more likely it is to remain above the random background.

This is illustrated in Figures 4 and 5 where the sensitivity of the affine and logarithmic penalties are compared. The BLOSUM62 matrix was used in both cases. The affine penalty was  $g_1(k) = 11 + k$  (the defaults for BLASTP (Altschul *et al.*, 1990, 1997)), and the logarithmic penalty was  $g_2(k) = 11 + 4.5 \log(k)$ . These two scoring schemes are equivalent, in the sense that they have similar values for  $\alpha, K, \lambda$  of (0.0764, 0.0488, 0.27937) and (0.0749, 0.04618, 0.27958) respectively (all parameters estimated by fitting Eqn (4) to 10 000 comparisons between pairs of random, unrelated sequences



**Fig. 4.** Empirical distributions of two statistically equivalent scoring schemes. Solid line: BLOSUM62,  $g(k) = 11 + k$ . Dashed line: BLOSUM62,  $g(k) = 11 + 4.5 \log(k)$ . Each distribution was derived from 10 000 comparisons between randomly-generated independent pairs of sequences of length 300.



**Fig. 5.** Comparison of the sensitivities of the affine and logarithmic gap penalties, expressed as the log p-value of a similarity comprising two ungapped domains with total score 100 separated by a gap of  $k$  residues. Solid line: BLOSUM62,  $g(k) = 11 + k$ . Dashed line: BLOSUM62,  $g(k) = 11 + 4.5 \log(k)$ . Horizontal line, p-value =  $10^{-5}$ .

by maximum-likelihood). Consequently they share similar random score distributions, as is confirmed in Figure 4, which compares their empirical distribution functions.

Figure 5 compares the probabilities of detection, expressed as log p-values, i.e.  $\log(1 - \Pr(s < t))$ , as a function of the gap length  $k$ , when  $D = 100$  and the sequence lengths  $m = n = 300$ . When  $k = 0$  the similarity contains no gaps and both scoring

schemes return the same p-value of about  $10^{-9}$ . As  $k$  increases the p-value increases. For  $k < 10$  the affine penalty is marginally superior, but for larger values of  $k$  the logarithmic penalty is far more sensitive than the affine. For example, the largest gap length for which the affine penalty returns a p-value  $< 10^{-5}$  is  $k = 17$ , but with the log penalty, the corresponding threshold is  $k = 53$ . [In practice, the behaviour of the affine penalty is likely to flip from detecting the complete alignment to finding the stronger of the two domains.]

In general, we can *guarantee* improving the sensitivity with respect to finding alignments with gaps longer than  $k'$ , say, relative to a gap penalty  $g_1(k)$  if we can find another penalty  $g_2(k)$  such that (i) both penalties have the same random score distribution in the logarithmic domain and (ii)  $g_2(k) \leq g_1(k)$  for  $k > k'$ . In our example, the log penalty was superior to the affine for large  $k$  because  $11 + 4.5 \log(k) > 11 + k$  when  $k > 10$ .

## Discussion

The **monotone** algorithm presented here should be very useful in situations where long gaps are expected, and provided the scoring scheme is such that alignments between random sequences are in the logarithmic domain. In practice, this is no real constraint, because penalties are nearly always chosen to ensure logarithmic behaviour. One may select suitable scoring schemes, and assess statistical significance of the resulting scores by using the results in Mott and Tribe (1998).

The novel features of this algorithm, over the earlier methods are (i) the use of general monotonically increasing gap penalties, as opposed to convex, (ii) the definition and construction of the candidate lists, and (iii) the use of local alignment statistics to show these lists are bounded. The proof that the candidate lists are bounded on average cannot be extended to the case of global alignments because the distribution of the scores  $D_{ij}$  would then depend on  $i, j$ .

Although we have not proved that the additional list-pruning reduces the algorithm's worst-case complexity to quadratic, the simulations indicate that for all practical purposes the method is efficient, provided the scoring scheme is in the logarithmic domain.

A complete comparative study of the sensitivity and accuracy of different gap penalties is beyond the scope of this paper, but the example discussed above illustrates the basic principles and the close relationship between sensitivity and the statistical properties of a scoring scheme. It should persuade the reader that the use of non-affine penalties is worthy of consideration.

## Appendix

### Formula for $\alpha$

Suppose the substitution matrix  $S(x, y)$  is such that the expected score between two letters is negative, and let  $h(x)$  be the probability that two letters drawn at random will have match score  $x$ . Let  $\lambda_0$  be the positive root of the equation

$$1 = \sum_x h(x)e^{\lambda_0 x}$$

$\lambda_0$  is always larger than  $\lambda$  in Eqs (2) and (4). Then the parameter  $\alpha$  is given by the formula

$$\alpha = 2s \sum_{k>0} e^{-\lambda_0 g(k)} \quad (5)$$

Equation (5) is similar to (3), except that  $\lambda$  is substituted for  $\lambda_0$ .  $s$  is computed as follows: let  $Y_n$  be the sum of  $n$  iid random variables with mass function  $h(\cdot)$ . Let  $E(X; X < 0)$  mean the expectation of the random variable  $X$  restricted to negative values, i.e.  $\sum_{x < 0} x \Pr(X = x)$  and  $\delta$  be the smallest span of score values. Let

$$E = \exp \left\{ - \sum_{k>0} \frac{1}{k} E(e^{\lambda_0 Y_k}; Y_k < 0) \right\}$$

$$P = \exp \left\{ - \sum_{k>0} \frac{1}{k} \Pr(Y_k \geq 0) \right\}$$

These quantities may be calculated numerically by successive auto-convolution of the mass function  $h(\cdot)$ . Then

$$s = \frac{\delta}{e^{\lambda_0 \delta - 1}} \frac{PE}{E(Y_1 e^{\lambda_0 Y_1})}$$

See Iglehart (1972), Karlin and Altschul (1990), Karlin and Dembo (1992) and Mott and Tribe (1999).

## References

- Altschul,S.F. (1998) Generalized affine gap costs for protein sequence alignment. *Proteins*, **32**, 88–96.
- Altschul,S.F. and Gish,W. (1996) Local alignment statistics. *Meth. Enzymol.*, **266**, 460–480.
- Altschul,S.F., Gish,W., Miller,W., Myers,E.W. and Lipman,D.J. (1990) Basic tool alignment search tool. *J. Mol. Biol.*, **215**, 403–410.
- Altschul,S.F., Madden,T.L., Schaffer,A.A., Zhang,J., Zhang,Z., Miller,W. and Lipman,D.J. (1997) Gapped blast and psi-blast: a new generation of protein database search programs. *Nucleic Acids Res.*, **25**, 3389–3402.
- Arratia,R.A., Morris,P. and Waterman,M.S. (1988) Stochastic scrabble: large deviations for sequences with scores. *J. Appl. Prob.*, **25**, 106–119.

- Benner,S.A., Cohen,M.A. and Gonnet,G.H. (1993) Empirical and structural models for insertions and deletions in the divergent evolution of proteins. *J. Mol. Biol.*, **229**, 1065–1082.
- Galil,Z. and Giancarlo,R. (1989) Speeding up dynamic programming with applications to molecular biology. *Theor. Comp. Sci.*, **64**, 107–118.
- Gotoh,O. (1982) An improved algorithm for matching biological sequences. *J. Mol. Biol.*, **162**, 705–708.
- Gotoh,O. (1990) Optimal sequence alignment allowing for long gaps. *Bull. Math. Biol.*, 359–373.
- Gu,X. and Li,W.-H. (1995) The size distribution of insertions and deletions in human and rodent pseudogenes suggests the logarithmic gap penalty for sequence alignment. *J. Mol. Evol.*, **40**, 464–473.
- Gusfield,D. (1997) *Algorithms on Strings, Trees and Sequences*. Cambridge University Press, Cambridge.
- Henikoff,S. and Henikoff,J.G. (1992) Amino acid substitution matrices from protein blocks. *Proc. Natl Acad. Sci. USA*, **89**, 10915–10919.
- Iglehart,D.L. (1972) Extreme values in the  $g_i/g_1$  queue. *Ann. Math. Stat.*, **43**, 627–635.
- Karlin,S. and Altschul,S.F. (1990) Methods for assessing the statistical significance of molecular sequence features by using general scoring schemes. *Proc. Natl Acad. Sci. USA*, **87**, 2264–2268.
- Karlin,S. and Dembo,A. (1992) Limit distributions of maximal segmental score among makrov-dependent partial sums. *Adv. Appl. Prob.*, **24**, 113–140.
- Miller,W. and Myers,E.W. (1988) Sequence comparisons with concave weighting functions. *Bull. Math. Biol.*, **50**, 97–120.
- Mott,R. and Tribe,R. (1999) Approximate statistics of gapped alignments. *J. Comp. Biol.*, **6**, 91–112.
- Mott,R.F. (1992) Maximum-likelihood estimation of the statistical distribution of Smith-Waterman local sequence similarity scores. *Bull. Math. Biol.*, **54**, 59–75.
- Mott,R.F. (1997) Est\_genome: a program to align spliced DNA sequences to unspliced genomic DNA. *Comput. Appl. Biosci.*, **13**, 477–478.
- Smith,T.F. and Waterman,M.S. (1981) Identification of common molecular subsequences. *J. Mol. Biol.*, **147**, 195–197.
- Waterman,M.S. (1984) Efficient sequence alignment algorithms. *J. Theor. Biol.*, **108**, 333–337.
- Waterman,M.S. (1995) *Introduction to Computational Biology. Maps, Sequences and Genomes*. Chapman and Hall, London.
- Waterman,M.S., Gordon,L. and Arratia,R. (1987) Phase transitions in sequence matches and nucleic acid structure. *Proc. Natl Acad. Sci. USA*, **84**, 239–243.
- Waterman,M.S. and Vingron,M. (1994a) Rapid and accurate estimates of statistical significance for sequence database searches. *Proc. Natl Acad. Sci. USA*, **91**, 4635–4628.
- Waterman,M.S. and Vingron,M. (1994b) Sequence comparison significance and poisson approximation. *Stat. Sci.*, **9**, 367–381.