Bulletin of Mathematical Biology Vol. 54, No. 4, pp. 599–618, 1992.
Printed in Great Britain.

0092-8240/92$5.00 + 0.00
Pergamon Press plc

AN $O(N^2 \log N)$ RESTRICTION MAP COMPARISON AND SEARCH ALGORITHM

■ EUGENE W. MYERS*
Department of Computer Science,
University of Arizona,
Tucson, AZ 85721, U.S.A.
■ XIAOQIU HUANG†
Department of Computer Science,
The Pennsylvania State University,
University Park, PA 16802, U.S.A.

We present an $O(R \log P)$ time, $O(M + P^2)$ space algorithm for searching a restriction map with M sites for the best matches to a shorter map with P sites, where R, the number of matching site pairs, is bounded by MP. As first proposed by Waterman *et al.* (1984, *Nucl. Acids Res.* **12**, 237–242) the objective function used to score matches is additive in the number of unaligned sites and the discrepancies in the distances between adjacent aligned sites. Our algorithm is basically a sparse dynamic programming computation in which "candidate lists" are used to model the future contribution of all previously computed entries to those yet to be computed. A simple modification to the algorithm computes the distance between two restriction maps with M and N sites, respectively, in $O(MN (\log M + \log N))$ time.

1. Introduction. A *restriction map* of a DNA strand is an ubiquitous tool of molecular biology. For each of a finite and small number of *restriction enzymes*, a "map" gives the position along the strand of the *recognition sites* at which each enzyme cleaves the DNA. For example, Kohara *et al.* (1987) recently produced a map of the DNA of the bacteria *E. coli* with respect to eight restriction enzymes. The map of the first 25 kilobases of this 4.72 million nucleotide sequence is shown in Fig. 1 as originally presented by Kohara. There is an open bar for each restriction enzyme (its name is to the left of the bar) and each vertical line within a bar indicates the location of a recognition site for the given enzyme. Experimental errors inherent in the techniques used to produce such maps result in inaccurate site locations, and missing and spurious sites. Consequently one cannot expect perfect alignment and correlation between maps arising from different experimental sources. This fact alone necessitates software that can compute optimal alignments between two potentially similar maps. But there are many other uses for such a comparison algorithm, among them: assembling complete physical maps from partial ones, locating potentially conserved regions between and within species, finding the

* This author's work was supported in part by National Library of Medicine Grant R01-LM4960.
† This author's work was supported in part by National Library of Medicine Grant R01-LM5110.

physical location of sequenced segments of DNA, and diagnosing genetic disorders by analysing polymorphisms.

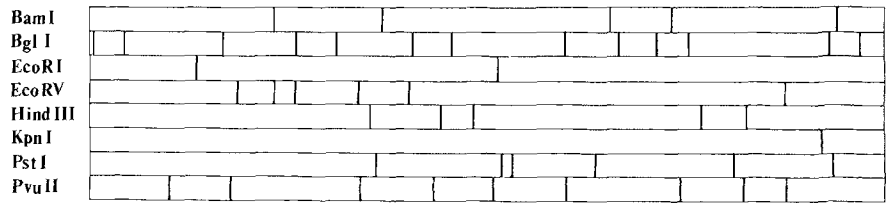

Figure 1. The first 25 kb of the Kohara map.

This paper focuses on the problem first considered by Waterman *et al.* (1984), of aligning two restriction maps under a measure of similarity that is additive in the number of unaligned sites and the discrepancies in the distances between adjacent aligned sites. This similarity measure admits a dynamic programming formulation that leads directly to an $O(M^2P^2)$ worst-case algorithm where M and P are the number of sites in the two maps. Recently, Miller and Huang (1988) considered the variation of finding subregions of a long map of M sites that are highly similar to a short probe map of P sites, where M is orders of magnitude larger than P. For this problem they designed an algorithm whose worst-case performance is $O(MP^3)$ but empirical evidence suggests that its expected case performance is $o(MP^2)$. Concurrently, Huang (1988) gave an $O(MP^2)$ worst-case algorithm.

This paper presents an $O(MP \log P)$ algorithm for comparing restriction maps or searching a long map for an approximate match to a short probe. The basic idea for the algorithm is an outgrowth of recent work on fast sequence comparison algorithms in the case where gap costs are concave (Hirschberg and Larmore, 1987; Miller and Myers, 1988; Eppstein *et al.*, 1988). While the objective function for aligning restriction maps appears at first to be affine, its dependence on map distances implies a dynamic programming recurrence for which multiple indels or gaps must be treated as a unit. Eppstein *et al.* (1990), have considered such recurrences for the problems of determining first-order RNA secondary structures and sparse sequence comparisons. Indeed, we discovered that their work encompasses that of this paper when slightly altered and specialized appropriate to the problem of comparing maps. Nonetheless, we proceed in this paper to give an algorithm development concretely tied to the map comparison problem. Our approach also differs in its conception, being based on the *candidate-list* paradigm introduced in our earlier work.

The remainder of the paper is organized as follows. In the next section preliminary concepts and notations are introduced. In Section 3, we give the outline of our algorithm based on a *candidate-list paradigm*. In Section 4,

profiles are introduced and their essential properties exposed. In Sections 5 and 6, sub-procedures for performing the necessary updates to candidate lists based on their profiles are presented in detail. In Section 7, refinements that use only $O(M+P^2)$ space, and that compare, as opposed to search, maps are discussed.

2. Preliminaries. A *restriction map* (or simply *map*) of length M is a sequence of M *sites*, where each site is an *enzyme-distance* pair. Any given mapping effort involves a small and finite number, U, of restriction enzymes, so we model *enzymes* as a small integer in the range $[1, U]$. The *distance* or location of a site is the number of base pairs to the site with respect to the origin of the map and so is modeled as an arbitrary integer. Thus we represent a map as an array of *site* records:

```
type site = record
                enz: [1..U];
                dist: integer
            end;
map = array [1..M] of site.
```

We assume all distances are non-negative and that each map is sorted according to distance, i.e. $map[i].dist \geqslant 0$ and $map[i-1].dist < map[i].dist$ for all $i > 1$. A *probe* is a map of length P where, in practice, P is much smaller than M. Throughout the remainder of the paper, it is assumed without loss of generality that $P \leqslant M$. While distances are modeled as integers, this is not critical to the algorithm of this paper which also applies in the case where distances are modeled as floating point numbers.

Introducing the problem formally requires some notation. Given *probe* of length P and *map* of length M, let:

$$a_i \equiv map[i].enz, \qquad b_j \equiv probe[j].enz,$$
$$m_i \equiv map[i].dist, \qquad p_j \equiv probe[j].dist,$$

for $1 \leqslant i \leqslant M$, and $1 \leqslant j \leqslant P$. Define the set of *match points*, *Matchpts*, to be $\{(i,j) \mid a_i = b_j\}$ and let $R \leqslant MP$ be the number of match points. An alignment or trace of *map* and *probe* is a sequence of match points $(i_1, j_1)\,(i_2, j_2) \cdots (i_L, j_L)$, such that for each k, $i_k < i_{k+1}$ and $j_k < j_{k+1}$, i.e. trace lines do not cross as illustrated in Fig. 2.

Let μ and λ be positive real constants. The score of an alignment $\alpha \equiv <(i_1, j_1)\,(i_2, j_2) \cdots (i_L, j_L)>$ is defined to be:

$$score(\alpha) = \lambda[(P-L)+(i_L - i_1 + 1 - L)] + \mu \sum_{k=2}^{L} \left|(m_{i_k} - m_{i_{k-1}}) - (p_{j_k} - p_{j_{k-1}})\right|$$

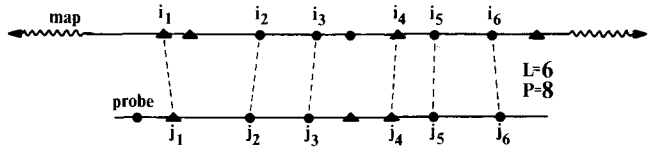

Figure 2. A restriction map alignment.

where $P-L$ is the number of sites in *probe* not in the alignment, $i_L - i_1 + 1 - L$ is the number of sites in *map* between sites i_1 and i_L not in the alignment, and $(m_{i_k} - m_{i_{k-1}}) - (p_{j_k} - p_{j_{k-1}})$ is the discrepancy in distance between aligned sites (i_k, j_k) and (i_{k-1}, j_{k-1}). In this formulation each unaligned site in *probe* and each "internal" unaligned site of *map* is charged λ. The constant μ determines the relative cost of distance discrepancies between adjacent aligned sites. Given *probe*, *map*, μ, λ and a threshold $T \geqslant 0$, our problem is to search for all alignments between *map* and *probe* that have scores greater than T.

For $(i, j) \in Matchpts$, let $D(i, j)$ be the score of the best alignment whose rightmost match points is (i, j). The following Lemma (Miller and Huang, 1988) gives a basic recurrence for $D(i, j)$.

LEMMA 1. $\mathrm{D(i,j)} = \min\{\lambda(\mathrm{P}-1), \min_{\substack{(\mathrm{I,J})\in\mathrm{Matchpts} \\ \mathrm{I}<\mathrm{i},\ \mathrm{J}<\mathrm{j}}} \mathrm{contrib_{I,J}(i,j)}\}$, *where*

$$\mathrm{contrib_{I,J}(i,j) = D(I,J) + \lambda(i-I-2) + \mu\left|(m_i - m_I) - (p_j - p_J)\right|}$$

A straightforward induction verifies the correctness of the lemma. Any algorithm that computes the R D-values suffices to solve our searching problem: we seek those D-values that are not greater than the threshold T.

The computation of $D(i, j)$ requires determining a point, (I, J), $I<i$, $J<j$, such that $contrib_{I,J}(i, j)$ is minimal. Simply computing each minimum of the recurrence requires $O(R)$ time per match point and thus takes $O(R^2) = O(M^2P^2)$ total time. Much greater efficiency is possible by organizing the points into a data structure that allows each minimum to be economically computed, e.g. in time proportional to the logarithm of the number of the points. In the next section we begin the development of an algorithm based on this idea.

3. Algorithm Outline. In order to exploit the recurrence of Lemma 1, the algorithm computes the R D-values in increasing order of i, and for a given i, in increasing order of j. The computation is restricted to the ordered pairs in *Matchpts*, so it is sparse. In order to only spend $O(R)$ time, as opposed to $O(MP)$ time locating the match points, *U match lists* are precomputed as follows. For each $a \in [1, U]$, let the match list of a-sites in *probe* be the ordered list,

$Match[a] = <j_1, j_2, \ldots, j_l>$ where $\{j_k\} = \{j | b_j = a\}$ and $j_1 < j_2 < \cdots < j_l$. It is a basic exercise to compute these match lists in $O(P)$ time and space using a bucket sort (Aho *et al.*, 1974, pp. 77–78). Note that it is assumed that the "names" of the enzymes in the maps have already been assigned to unique integers in the range $[1, U]$. If this is not the case, doing so requires $O(M \log P)$ time in the worst case with the aid of a comparison-based dictionary (Aho *et al.*, 1974, pp. 145–152). Any names in *map* not in *probe* can be mapped to the same integer, and thus we can assume $U \leqslant P+1$. With this preprocessing accomplished, the algorithm outline below spends only $O(R)$ time indexing the match points.

Our algorithm is based on the *candidate-list paradigm* first introduced by Waterman (1984), and later refined by Miller and Myers (1988). As the algorithm computes each "column" of the "D-matrix" in increasing order of i, a list of match points called candidates is updated and used to compute subsequent D-values. The candidate list is *conservative* in that a match point whose D-value is known is removed from the list if it cannot possibly contribute to the D-values of match points yet to be considered. At the time such a candidate is found to be unimportant to future match points it becomes *dead*; those that remain are *live*. In addition to being conservative, the list for the current column i represents the possible contribution of its candidates to future points and is a partitioning of the interval $[0, p_P]$, one candidate per partition interval. The candidate associated with an interval is called the *owner* of that interval.

An outline of the algorithm is given in Fig. 3. The function *Find_min*(i, j) returns $contrib_{I,J}(i, j)$, where (I, J) is the owner of the current candidate list interval containing p_j. After the D-values of the points of column i are computed, they are inserted into the list by the procedure *Insert* if not immediately found to be dead. Conversely, their insertion may cause older candidates to be removed because they are dead as a consequence. Finally *Update* shifts the candidate list to prepare it for the next column and may remove candidates that die as a result.

```
Initialize_candidate_list ()
for i←1 to M do
  {  for j∈Match[a_i] do
        D(i, j)→min(λ(P−1), Find_min(i, j))
     if i<M then
       { for j∈Match[a_i] and j<P in increasing order do
            Insert(i, j)
         Update(i)
       }
  }
```

Figure 3. An outline of the algorithm.

In Sections 5 and 6, the sub-procedures *Find_min*, *Insert* and *Update* will be presented in detail. It will be shown that the total time spent in these procedures is bounded by $O(R(\log L + \log M))$, where L is the maximal size of the candidate list. Because L is $O(R)$ it follows that the algorithm takes $O(R(\log M + \log P))$ time and $O(R)$ space in the worst case. The superior bounds claimed in the abstract are obtained by a refinement deferred to Section 7.

4. Profiles. In this section, the exact nature of the future contributions of previously computed match points is introduced via the concept of a *profile*. In previous dynamic programming algorithms one can imagine computing an $(M+1)$-by-$(P+1)$ grid or matrix of values $D(i,j)$ where i indexes columns and j indexes rows. As noted previously our problem is sparse in that we only require the value of D at the R match points. A key insight for the problem at hand is to consider the computation as proceeding over the rectangle $[0, m_M] \times [0, p_P]$ of the Cartesian plane where we require the value $D(i,j)$ of the R points (m_i, p_j). Over this sparse *map grid*, the contribution of a point (m_I, p_J) to future points can be conveniently and analytically described in the treatment that follows.

For $(I, J) \in Matchpts$ and $i \geqslant I$, define the function $f^i_{I,J}(x)$ on the interval $(p_J, p_P]$ as:

$$f^i_{I,J}(x) = \mu \left| C^i_{I,J} - x \right| + B^i_{I,J} \quad \text{where} \quad C^i_{I,J} = m_i + \Delta_{I,J}, \quad \Delta_{I,J} = p_J - m_I,$$
$$B^i_{I,J} = \lambda i + E_{I,J}, \quad E_{I,J} = D(I, J) - \lambda(I+2).$$

Assume $f^i_{I,J}(x) = \infty$ for x not in $(p_J, p_P]$ and notice that $\Delta_{I,J}$ and $E_{I,J}$ are independent of i. The function $f^i_{I,J}(x)$ gives the contribution of match point (I, J) to any point in "column" m_i of the map grid. Lemma 2 shows how f^i is used to compute D-values, and that f^{i+1} can be obtained by shifting f^i right by $\Delta m_i = m_{i+1} - m_i$ and up by λ as illustrated in Fig. 4. The proof of Lemma 2 follows directly from the definitions above.

LEMMA 2. *For match points* (I, J), (i, j) *such that* $I < i$ *and* $J < j$, $\mathrm{contrib}_{I,J}(i,j) = f^i_{I,J}(p_j)$. *Moreover, if* $i < M$ *then* $f^{i+1}_{I,J}(x) = f^i_{I,J}(x - \Delta m_i) + \lambda$.

The i-profile is the minimum envelope of the f-curves of all match points strictly left of column i and is defined formally by:

$$P^i(x) = \min_{\substack{(I,J) \in Matchpts \\ I < i,\ J < P}} f^i_{I,J}(x)$$

The i-profile gives the contribution of all match points left of column m_i to any point in column m_i of the map grid. From Lemma 2 it easily follows that $D(i,j) = \min(\lambda(P-1), P^i(p_j))$. Figure 5 illustrates a hypothetical i-profile. Note that it is not continuous in x.

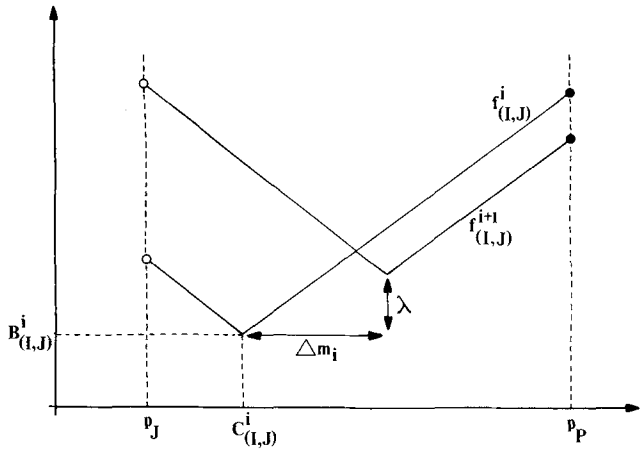

Figure 4. An illustration of f^i and f^{i+1}.

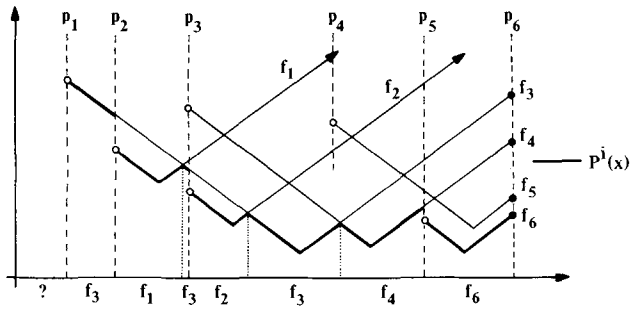

Figure 5. An i-profile.

From Fig. 5, observe that the domain $[0, p_P]$ of the i-profile is naturally partitioned into intervals for which there is some (I, J) such that $f^i_{I,J}(x)$ coincides with $P^i(x)$ over the entirety of the interval. For such an interval, the match point (I, J) is its owner in that it represents the minimum envelope or profile P^i over the interval. Note that a given match point may be the owner of several intervals. If one could maintain a list of these representative match points in order of the partition of $[0, p_P]$, then computing $P^i(x)$ is simply a matter of locating which subinterval x lies in and then evaluating the function $f^i(x)$ of its owner. The task of maintaining such a candidate list is considerably simplified by decomposing the f-curves into the left and right halves of their V-shape, and maintaining separate minimum envelopes and associated candidate lists for the two halves. Formally split $f^i_{I,J}(x)$ into the two simple functions, $L^i_{I,J}(x)$ and $R^i_{I,J}(x)$ defined as follows:

$$f^i_{I,J}(x) \equiv \begin{cases} L^i_{I,J}(x) = \mu(C^i_{I,J} - x) + B^i_{I,J} & \text{if } x \in (p_J, C^i_{I,J}] \\ R^i_{I,J}(x) = \mu(x - C^i_{I,J}) + B^i_{I,J} & \text{if } x \in (C^i_{I,J}, p_P] \end{cases}$$

Once again we assume both $L^i_{I,J}(x)$ and $R^i_{I,J}(x)$ have the value ∞ for x not in the intervals of the above definition. Define the left i-profile $L^i(x)$ and right i-profile $R^i(x)$ by:

$$L^i(x) = \min_{\substack{(I,J) \in Matchpts \\ I<i,\ J<P}} L^i_{I,J}(x) \qquad R^i(x) = \min_{\substack{(I,J) \in Matchpts \\ I<i,\ J<P}} R^i_{I,J}(x).$$

It follows directly from these definitions that $P^i(x) = \min(L^i(x), R^i(x))$. Thus computing $P^i(x)$ is equivalent to separately computing $L^i(x)$ and $R^i(x)$ and taking the minimum of the two.

Like the function P^i, the domains of each of L^i and R^i are naturally partitioned into intervals and can be represented by ordered candidate lists of owners (see Figs 6 and 8) Thus our algorithm actually maintains two

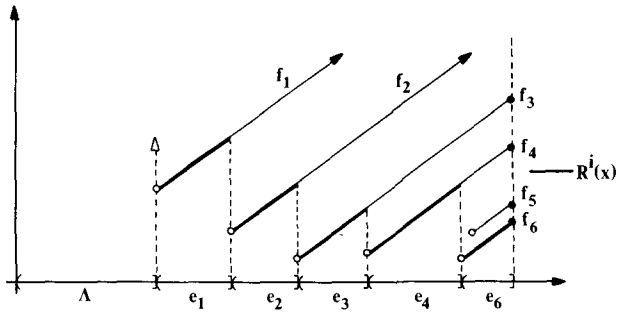

Figure 6. The profile R^i and list *Cand_R* for the profile of Fig. 5.

candidate lists, one for each "half" of the desired profile, that together effectively represent it. So in the outline of Fig. 3, the call to *Find_*min(i,j) is in fact the expression min (*Find_Left*(i,j), *Find_Right*(i,j)), the call to *Insert*(i,j) is really a call to *Insert_Left*(i,j) immediately followed by a call to *Insert_Right*(i, j), and so on. The procedures for the right candidate list are treated first because they are simpler than the ones for the left candidate list.

5. The Right Profile. Proceeding formally, an element e of the right candidate list is a four tuple $\langle I, J, \Delta_{I,J}, E_{I,J} \rangle$ where (I, J) is the match point from which the candidate arose. The notations $e.I$, $e.J$, $e.\Delta$, and $e.E$ index the four values in the candidate e. Further let the notation $e.C^i$ denote $e.\Delta + m_i$, let $e.B^i$ denote $e.E + \lambda i$, and let $e.R^i(x)$ denote $\mu(x - e.C^i) + e.B^i$. It is easy to see that $e.C^i = C^i_{I,J}$, $e.B^i = B^i_{I,J}$, and $e.R^i(x) = R^i_{I,J}(x)$ for $x \in (e.C^i, p_P]$. Thus a

candidate e is independent of i, but can deliver any of the relevant i-dependent quantities or functions in $O(1)$ time. This point is important since one cannot afford to update the record of each candidate in the list as the algorithm progresses from column to column. Further note, that for algorithmic convenience we have chosen to let $e.R^i(x)$ be defined for all values of x, whereas $R^i_{I,J}(x)$ is ∞ outside of its relevant interval.

The right candidate list, *Cand_R*, is a linear list of the tuples or records introduced above. A linear list data type that supports logarithmic insertion, deletion and search primitives is required and any height-balanced tree structure will do, e.g. AVL trees (Aho *et al.*, 1974, pp. 145–152) or splay trees (Sleator and Tarjan, 1985). Specifically, the following operations are assumed:

last: Return the last element of the list.
pred(e), *succ*(e): Return the predecessor and successor of list element e, respectively. If no such element exists, return Λ.
find(x): Return the element e such that $e.C^i < x$ and ($e = last$ or $x \leqslant succ(e).C^i$. If no such element exists, return Λ.
delete(e): Remove the element e from the list.
insert(e, f): Insert f as the successor of e.

By simply threading the tree, *last*, *succ*, and *pred* require only $O(1)$ time. The remaining primitives require $O(\log S)$ time where S is the current size of *Cand_R*. *Find* is possible because, as will be seen momentarily, *Cand_R* is always ordered so that $e.C^i < succ(e).C^i$ for all $e \neq last$.

To simplify matters, Λ is realized as an actual record whose tuple is $\langle 0, 0, -m_M, \lambda P\rangle$. This choice of values guarantees that for all $i \geqslant 1$ and all $x \geqslant 0$, $\Lambda.R^i(x) = \mu(x-(m_i-m_M)) + \lambda(P+i) \geqslant \mu x + \lambda P > \lambda(P-1)$. Thus by the recurrence of Lemma 1, Λ's curve is high enough to effectively represent ∞ over the entire map grid. In Fig. 7, *Initialize_Right* sets *Cand_R* to consist of this single record. Because of the way Λ is chosen, it will never be removed from *Cand_R* and the primitives above avoid having to treat the special case where *Cand_R* is empty. Moreover, the problematic case of intervals where $R^i(x) = \infty$ is also avoided: Λ will own these intervals and effectively represent their value.

Consider two candidates e and f and let $i = \max(e.I, f.I)$. If $e.R^i(0) \leqslant f.R^i(0)$ then $e.R^i(x) \leqslant f.R^i(x)$ for all x as both functions are straight lines. Moreover, $e.R^k(x) \leqslant f.R^k(x)$ for all $k \geqslant i$ because as k is increased each line is shifted right and up by the same amount by Lemma 2. The same reasoning implies that if $e.C^i \leqslant f.C^i$ then this remains true as i is increased. Thus if $e.R^i(0) \leqslant f.R^i(0)$ and $e.C^i \leqslant f.C^i$ then e's contribution to the right profiles R^k for any $k \geqslant i$ is always smaller than f's contribution over the domain $(f.C^k, p_P]$ of $R^k_{f.I,f.J}$. In this case we say f is *dominated* by e, and f is dead since it will never contribute to a subsequent right profile.

With this observation, it follows that R^i is represented by an ordered

candidate list $\langle e_1, e_2, \ldots, e_L\rangle$, where (1) $e_h.C^i < e_{h+1}.C^i$, (2) $e_h.R^i(0) > e_{h+1}.R^i(0)$, and (3) $R^i(x) = e_h.R^i(x)$ for $x \in (e_h.C^i, e_{h+1}.C^i]$ provided we define $e_{l+1}.C^i = p_P$. The illustration in Fig. 6 helps visualize these properties. Moreover, l is always nonzero and e_1 is always Λ as $\Lambda.C^i \leqslant 0$. Given such a candidate list, *Find_Right*(i, j) in Fig. 7 computes $D(i, j)$ by simply returning *find* $(p_j).R^i(p_j)$, i.e. the R^i-value of the owner of the interval containing p_j.

Next consider the procedures *Insert_Right* and *Update_Right* that are used to compute the candidate list for successive columns. Suppose that we start with a candidate list, *Cand_*R^i, properly representing R^i. To produce *Cand* R^{i+1}, the match points in column i are first added by *Insert_Right* to produce a candidate list representing $\bar{R}^i(x) = \bar{R}^{i/(R^i}(x)\min(R^i_{i,j}(x) \mid (i, j) \in Matchpts$ and $j < P))$. The resulting candidate list is then modified by *Update_ Right* to reflect a shift of $\bar{R}^i$ right Δm_i and up λ map units to produce *Cand_*R^{i+1} which represents $R^{i+1}(x) = \bar{R}^i(x - \Delta m_i) + \lambda$.

```
Procedure Initialize_Right()
    Cand_R ← ⟨0, 0, −m_M, λP⟩

Function Find_Right(i, j)
    return find(p_j).R^i(p_j)

Procedure Insert_Right(i, j)
    c ← R^i_{i,j}(0)
    e ← find (p_j)

    if c < e.R^i(0) then
      { f → succ(e)
        while f ≠ Λ and c ≤ f.R^i(0) do
          { delete(f)
            f ← succ(e)
          }
        insert(e, ⟨i, j, p_j − m_i, D(i, j) − λ(i+2)⟩)
      }

Procedure Update_Right(i)
    while last.C^{i+1} ≥ p_P do
        delete(last)
```

Figure 7. The right candidate list procedures.

The procedure *Insert_Right*(i, j) takes a candidate list for a right profile $\hat{R}^i$ at column i and adds the effect of match point (i, j) to produce a candidate list representing $\min(\hat{R}^i(x), R^i_{i,j}(x))$. Let $g = \langle i, j, p_j - m_i, D(i, j) - \lambda(i+2)\rangle$ be the candidate for match point (i, j) recalling that $D(i, j)$ is known at this time. First *find*(p_j) returns the candidate e in the current list that owns the interval containing $g.C^i = p_j$. If $e.R^i(0) \leqslant g.R^i(0)$ then e dominates g and *Insert_Right*

correctly exits without adding g to the list. Otherwise, g must be added to the candidate list immediately after e since it contributes to the desired profile on at least the interval $(g \,.\, C^i, succ(e) \,.\, C^i]$. The insertion of g may cause successors of e to become dead. Suppose $e \neq last$ and let $f = succ(e)$. If $g \,.\, R^i(0) \leqslant f \,.\, R^i(0)$ then f is dominated by g since $g \,.\, C^i \leqslant f \,.\, C^i$, and consequently is removed. Such successors of e are removed until either e becomes *last* or a successor is reached that is not dominated by g, whereupon g is inserted into the list immediately after e. The candidate list now reflects the contribution of (i, j) and *Insert_Right* returns.

The procedure *Update_Right*(i) takes the candidate list for $\bar{R}^i$ and updates it to represent the shift giving R^{i+1}. Because all candidates are independent of i, the shift requires no change to the elements. However, the shift does imply that candidate e near the end of the list may no longer be relevant because $e \,.\, C^{i+1} = e \,.\, C^i + \Delta m_i \geqslant p_P$, i.e. the interval owned by e no longer intersects $[0, p_P]$. Thus *Update_Right* need only remove the suffix of the candidate list containing such elements.

Consider the total time spent in the procedures of Fig. 7 over the course of the algorithm outline of Fig. 3. Let L be the maximum size of *Cand_R* at any point in the computation. *Initialize_Right*() is quickly dismissed since it is called once and takes constant time. *Find_Right* is called exactly R times at a cost of $O(\log L)$ time per call for a total of $O(R \log L)$ time. *Insert_Right* is called less than R times, and each call inserts at most one element to *Cand_R*. Each of these elements can only be removed once, thus the total number of iterations of both **while** loops in *Insert_Right* and *Update_Right* is bounded by R. Thus a total of $O(R \log L)$ time is spent in the **while** loops of both procedures. Apart from these loops, $O(\log L)$ time is consumed by each call to *Insert_Right*, and $O(1)$ time for *Update_Right*. Thus the total time spent in the procedures of Fig. 7 is $O(R \log L)$. Certainly L is $O(R)$ and later we will show how to guarantee that L is $O(P^2)$.

6. The Left Profile. Attention is now turned to the structure and manipulation of the candidate list, *Cand_L*, that represents the left profiles for successive columns. Like the right candidate list, *Cand_L* is a linear list of candidates, each giving the value of the left profile over some subinterval of $[0, p_P]$. For an element $e = \langle I, J, \Delta_{I,J}, E_{I,J} \rangle$ of this list, the notations, $e \,.\, I$, $e \,.\, J$, $e \,.\, \Delta$, $e \,.\, E$, $e \,.\, C^i$ and $e \,.\, B^i$ are as before. Analogous to $e \,.\, R^i(x)$, let $e \,.\, L^i(x)$ denote $\mu(e \,.\, C^i - x) + e \,.\, B^i$ and note that $e \,.\, L^i(x) = L^i_{I,J}(x)$ for $x \in (p_J, e \,.\, C^i]$.

Consider two candidates e and f and let $i = \max(e \,.\, I, f \,.\, I)$. If $e \,.\, L^i(0) \leqslant f \,.\, L^i(0)$ then as argued for the right case $e \,.\, L^k(x) \leqslant f \,.\, L^k(x)$ for all x and all $k \geqslant i$. Also, if $e \,.\, C^i \geqslant f \,.\, C^i$ then this remains true as i is increased. But for a left profile L^k, e and f potentially contribute over the intervals $(p_{e \,.\, J}, e \,.\, C^k]$ and $(p_{f \,.\, J}, f \,.\, C^k]$, respectively. Thus even if $e \,.\, L^i(0) \leqslant f \,.\, L^i(0)$ and $e \,.\, C^i \geqslant f \,.\, C^i$, e does not

dominate f when $f.J < e.J$ because f's curve is still exposed on the interval $(p_{f.J}, p_{e.J}]$. It is this feature of the left case that makes it difficult. For while the curves and their right ends (e.g. $e.C_i$) all shift the same relative amount with i, their left ends (e.g. $p_{e.J}$) are stationary and hence "moving" relative to right ends. For example, consider a third candidate g for which $g.J < f.J < e.J$, $g.I = i$, and $g.L^i(0) \leqslant f.L^i(0)$. As the algorithm progresses from column i forward, g's right end is less than $p_{f.J}$, then between it and $p_{e.J}$, and finally greater than $p_{e.J}$, at which instant f dies because g now covers that part of f not covered by e.

Figure 8 gives an example of a left profile and its candidate list. First note, that a given match point's curve may contribute at a number of distinct intervals and hence have several candidates in the list. Moreover, the right ends of some intervals are the right end of the owner's curve, while others are the left end of their successor's curve. Thus, to determine the interval endpoints from the candidate records in the list, we must add a fifth attribute to a candidate's record, *.mobile*, that is true in the former cae, and false otherwise. Formally, if e is a candidate, then its interval right end, $right^i(e)$, is $e.C^i$ if $e.mobile$ is true, and $p_{succ(e).J}$ otherwise. Note that the *.mobile* attribute is also independent of i but the right end may easily be determined in constant time. Of course, the left end, $left^i(e)$, is easily recovered as the right end of e's predecessor, or 0 if e has no predecessor.

The left candidate list, *Cand_L*, is a linear list of candidate tuples and is realized with the same implementation and primitive repertoire as for the right

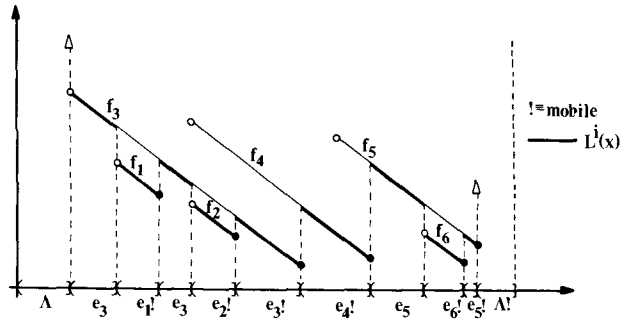

Figure 8. The profile L^i and list *Cand_L* for the profile of Fig. 5.

candidate list. We need only note that *find*(x) returns the element e such that $left^i(e) < x$ and $x \leqslant right^i(e)$, and this is possible because *Cand_L* is always ordered so that $left^i(e) < left^i(succ(e))$ for all $e \neq last$. As for the right profile, matters are simplified by realizing Λ as an actual record whose tuple in this case is $\langle 0, 0, p_P, \lambda P, true(=.mobile)\rangle$. As before, this choice of values guarantees

that Λ effectively represents ∞ where necessary (i.e. for all $i \geqslant 1$ and $x \leqslant p_P$, $\Lambda . L^i(x) > \lambda P - 1)$), and guarantees that *Cand_L* is never empty. In Figure 9, *Initialize_Left* sets *Cand_L* to initially consist of this single record.

With these remarks, it follows that L^i is represented by an ordered candidate list $\langle e_1, e_2, \ldots, e_l \rangle$, where (1) $left^i(e_h) < left^i(e_{h+1})$, and (2) $L^i(x) = e_h . L^i(x)$ for $x \in (left^i(e_h),\ left^i(e_{h+1})]$ provided we define $left^i(e_{l+1}) = p_P$. Moreover, *last . mobile* is always true and since such candidates are never removed, l is always nonzero. Unlike *Cand_R*, *Cand_L* may contain several candidates for a given match point, but each represents its curve over a different subinterval. Note in Fig. 8, that only the rightmost (if any) of these candidates is mobile (i.e. has *. mobile* set to true). Given such a candidate list, *Find_Left*(i, j) in Fig. 9 computes $D(i, j)$ by simply returning $find(p_j) . L^i(p_j)$, i.e. the L^i-value of the owner of the interval containing p_j.

Next we examine what happens when the left candidate list is shifted from column to column. The situation is more complex than for right profiles, because now the right ends of mobile candidates shift Δm_i to the right in going from column i to $i+1$, whereas the right ends of non-mobile candidates remain stationary. Thus if e is mobile and $f = succ(e)$ is not, then after enough column shifts the right end of e will *overrun* the right end of f. At this time f ceases to contribute to the profile as its left end (e's right end) becomes greater than its right end. Let $overrun\ (e, f) = \max\{k \mid right^k(e) \leqslant right^k(f)\} = \max\{k \mid e . C^k \leqslant p_{succ(f) . J}\} = \max\{k \mid m_k \leqslant p_{succ(f) . J} - e . \Delta\}$. A simple binary search over *map* suffices to compute $k = overrun(e, f)$ in $O(\log M)$ time.* Now observe that it is exactly when shifting from column k to $k+1$ that f's interval disappears, or equivalently, is overrun by e.

An array of sets, *Bucket*, is maintained that records all potential future overrun events for *Cand_L*. Specifically, for a given state of *Cand_L* during the computation, *Bucket*$[1 .. M]$ satisfies the following invariant:

$$Bucket[k] = \{e \mid e . mobile \text{ and } not\ succ(e) . mobile \text{ and } overrun(e, succ(e)) = k\}.$$

In essence, *Bucket* is an event queue of those overrun events that will take place if the adjacency relationships in the current candidate list remained unchanged as it is shifted through successive columns. Of course, these adjacency relationships will change due to the introduction of new candidates and overrun events themselves, thus necessitating continual updating of the *Bucket* structure to maintain the invariant.

Bucket is realized as an M-element array of pointers to doubly-linked lists representing each set. Each element in a list points at its candidate record in *Cand_L* and each candidate record in turn has a pointer to its *Bucket* list-

* In Section 7 a refinement limits the search to a map of size $2P$ and hence guarantees that only $O(\log P)$ time is spent on this operation.

element if it exists (i.e. the candidate overruns its successor). With this structure, the following operations are all performable in constant time:

pop(i): Return and remove the first element of *Bucket*[i]'s list. If the list is empty, return Λ.

push(e): If e is mobile and its successor is not then append an element for candidate e to the front of *Bucket*[*overrun*(e, *succ*(e))]'s list.

remove(e): Remove the element for candidate e from whatever *Bucket* list it is in, if any.

Note that both *push*(e) and *remove*(e) have no effect on *Bucket* unless the mobility of e and its successor are such that an overrun event can occur.

Now consider the procedures *Insert_Left* and *Update_Left* that compute the left candidate list for successive columns. Analogous to the right candidate list, given a list $Cand\_L^i$ properly representing L^i, the match points in column i are added by *Insert_Left* to produce a candidate list representing $\bar{L}^i(x)$. This list is then modified by *Update_Left* to produce $Cand\_L^{i+1}$.

The procedure *Insert*(i,j) takes a candidate list for a left profile $\hat{L}^i$ at column i and adds the contribution of match point (i, j) to produce a candidate list representing $\min(\hat{L}^i(x), L^i_{i,j}(x))$. Note first that the domain of $L^i_{i,j}(x)$ is $(p_j, C^i_{i,j}] = (p_j, p_j]$, i.e. the *point* p_j. It is useful to think of the curve's initial domain interval as being infinitesimally small. This feature is intentional, it is much easier to add such curves to the profile than those that span a measurable interval. As soon as the shift from column i to $i+1$ takes place the interval will be proper and of width Δm_i. First *find*(p_j) returns the candidate e in the current list that owns the interval containing p_j, i.e. $left^i(e) < p_j \leqslant right^i(e)$. Candidate e's interval contains p_j and thus contains (i,j)'s infinitesimally small interval. In the discussion that follows, let $d = pred(e)$, $f = succ(e)$, and let $g = \langle i, j, p_j - m_i, D(i,j) - \lambda(i+2), true\rangle$ be the candidate for match point (i, j).

First consider the case where $p_j < right^i(e)$. If $e.L^i(0) \leqslant g.L^i(0)$ then e's match point curve dominates g's match point curve and *Insert_Left* quits without adding g to the list. Otherwise, g represents the new profile over its infinitesimal interval $(p_j, g.C^i]$, and $e.L^i(x)$ continues to represent the profile over the intervals $(left^i(e), p_j]$ and $(g.C^i, right^i(e)]$. To reflect this, h, a non-mobile copy of e, and g are inserted between d and e. In order to maintain the *Bucket* invariant the new adjacencies between d and h, and between g and e, are checked for overrun. The new adjacency between h and g need not be checked because g is mobile and h is not.

In the case where $p_j = right^i(e) = left^i(f)$, we have two subcases depending on the mobility of e. If e is mobile then $left^i(f) = e.C^i$ and $e.L^i$ is below $f.L^i$. Thus g represents a part of the profile if and only if $g.L^i(0) < e.L^i(0)$. If so, then e becomes non-mobile as its new right end is p_j, and thus e's overrun event (if

```
Procedure Initialize_Left()
    Cand_L←⟨0, 0, p_P, λP, true⟩

Function Find_Left(i, j)
    return find(p_j).L^i(p_j)

Procedure Insert_Left(i, j)
    c←L^i_{i,j}(0)
    e←Find(p_j)
    if right^i(e)≠p_j then
      { if e.L^i(0)>c then
          { d←pred(e)
            remove(d)
            insert(d, ⟨i, j, p_j−m_i, D(i, j)−λ(i+2), true⟩)
            insert(d, ⟨e.I, e.J, e.Δ, e.E, false⟩)
            push(pred(e))
            push(d)
          }
      }
    else if e.mobile then
      { if e.L^i(0)>c then
          { remove(e)
            insert(e, ⟨i, j, p_j−m_i, D(i, j)−λ(i+2), true⟩)
            e.mobile←false
            push(pred(e))
            push(succ(e))
          }
      }
    else if succ(e).L^i(0)>c then
      { insert(e, ⟨i, j, p_j−m_i, D(i, j)−λ(i+2), true⟩)
        push(succ(e))
      }

Procedure Update_Left(i)
    while (e←pop(i))≠Λ do
      { delete (succ(e))
        if succ(e)≠Λ then
            if e.L^i(0)<succ(e).L^i(0) then
                push(e)
            else
              { e.mobile←false
                push(pred(e))
              }
      }
    while left^{i+1}(last)≥p_P do
        delete (last)
```

Figure 9. The left candidate list procedures.

any) must be removed and the adjacency between $pred(e)$ and e must be checked for overrun. In addition, g is inserted between e and f, and the adjacency between g and f is checked for overrun. It remains to treat the subcase where e is not mobile. In this event $left^i(f) = p_{f.J}$ and $f.L^i$ is below $e.L^i$. Thus g represents a part of the profile if and only if $g.L^i(0) < f.L^i(0)$. If so then g

is inserted between e and f, and the adjacency between g and f is checked. The new adjacency between e and g need not be checked as e is non-mobile and g is mobile.

The procedure *Update_Left*(i) takes the candidate list for $\bar{L}^i$ and updates it to represent the shift giving L^{i+1}. As for the right candidate list, the shift requires no change in the elements as they are independent of i. But some number of candidates intervals may become empty due to overrun events during the shift. Recall that these events are exactly those recorded in *Bucket*[i]'s list. So *Update_Left* pops these events off *Bucket*[i] giving the overrunning candidate e and deletes the overrun candidate, *succ*(e). Now the deletion creates a new adjacency between e and its new successor, say f. If $e.L^i(0) < f.L^i(0)$ then e is still mobile and it may overrun f (possibly during this shift) and must be so checked. Otherwise e becomes non-mobile (it passes "behind" f) and the adjacency between *pred*(e) and e must be checked for overrun to maintain the *Bucket* invariant. Elements are popped from *Bucket*[i] and so processed until its list becomes empty. The shift is completed like that of the right candidate list by removing the suffix of elements of *Cand_L* whose left-ends are greater than p_P.

Consider the total time spent in the procedures of Fig. 9 over the course of the algorithm outline of Fig. 3. Let L be the maximum size of *Cand_L* at any point in the computation. *Initialize_Left*() is quickly dismissed since it is called once and takes constant time. *Find_Left* is called exactly R times at a cost of $O(\log L)$ time per call for a total of $O(R \log L)$ time. *Insert_Left* is called less than R times, and each call inserts at most two elements to *Cand_L*. Thus the total number of elements ever entering *Cand_L* is bounded by $2R$. Because each iteration of either **while** loop of *Update_Left* deletes a candidate, it follows that the bodies of these loops are executed at most $O(R)$ times over the course of the algorithm. The straight-line portion of these routines involve $O(\log L)$ time candidate list routines, $O(\log M)$ time calls to *overrun*, and $O(1)$ calls to the bucket routines. Thus the total time spent in the procedures of Fig. 9 is $O(R(\log L + \log M))$. At most half the candidates in *Cand_L* can overrun their successors, so the total space consumed by the *Bucket* lists and *Cand_L* is $O(L)$. The *Bucket* array requires $O(M)$ space but this will be reduced to $O(P)$ in the next section where L is also reduced to $O(P^2)$.

7. Refinements

Using only $O(M + P^2)$ *space and* $O(R \log P)$ *time.* The analyses of the previous two sections only show that *Cand_R*, *Cand_L*, and the *Bucket* lists contain at most $O(R) = O(MP)$ elements. Indeed on certain inputs it is possible for this many elements to be in the lists at a particular instant, although in practice the lists are usually no worse than $O(P^2)$. For problems in which M is very large, one might wish to guarantee that only $O(P^2)$ space is used in the

worst case. Arranging this is quite easy to do with the initial observation that if $i-I>P$ then all j and J, $contrib_{I,J}(i,j)\geq\lambda(P-1)\geq D(i,j)$ by Lemma 1. Thus a candidate in column i cannot possibly affect the D-value of a match point in a column numbered greater than $i+P$.

This observation is levered by dividing the "$M\times P$" problem into roughly M/P "$2P\times P$" consecutive problems each of which overlaps its successor in P columns. Let $map_k=map[(k-1)P+1\ \ldots\ min(kP,M)]$ for $k=1,2,\ldots\lceil M/P\rceil$. Apply the central algorithm to search for P in map_1 and report the results. Then for $k=2,3,\ldots$, search for P in $map_{k-1}\bullet map_k$ (the concatenation of the two maps) and report the results found in the columns of map_k. From the observation above the results reported in each column must be correct since all candidates that can affect that column are considered in the given subproblem. Moreover, the number of match points in each subproblem is less than $2P^2$. Thus the list structures for any subproblem never require more than $O(P^2)$ space in the worst case.

This approach has two additional benefits. First, the time for candidate list primitives is reduced to $O(\log P)$ in the worst case because of their reduced size. Second, the $O(\log M)$ binary search for *overrun* is also reduced to $O(\log P)$ because the map of an individual subproblem contains at most $2P$ sites. So in summary, the algorithm requires at most $O(R\log P)$ time and $O(M+P^2)$ space where the factor of M is present solely to account for the space for *map*.

Comparing as opposed to searching maps. Thus far the focus has been on an algorithm for searching a large map for a good approximate match to a short probe, and therefore the score of alignments did not reflect any discrepancies in the large map to the left and right of the first and last aligned sites. Consider now the related problem in which *map* and *probe* are of roughly the same size and we wish to compare the two maps to see if they are similar. First we modify the score of an alignment $\alpha\equiv\langle(i_1,j_1)\ (i_2,j_2)\ \ldots\ (i_L,j_L)\rangle$ to be:

$$score(\alpha)=\lambda(M+P-2L)+\mu|m_{i_1}-p_{j_1}|+$$
$$\mu\sum_{k=2}^{L}|(m_{i_k}-m_{i_{k-1}})-(p_{j_k}-p_{j_{k-1}})|+\mu|(m_M-m_{i_L})-(p_P-p_{j_L})|$$

where $M+P-2L$ is the number of sites in *map* and *probe* not in the alignment, and $(m_{i_k}-m_{i_{k-1}})-(p_{j_k}-p_{j_{k-1}})$ is the discrepancy in distance between aligned sites (i_k,j_k) and (i_{k-1},j_{k-1}). Note that *all* unaligned sites in *map* are penalized, as are the distance discrepancies to the first and last aligned pairs (i.e. $m_{i_1}-p_{j_1}$ and $(m_M-m_{i_L})-(p_P-p_{j_L})$). An *optimal* alignment of *map* and *probe* is one with the minimum score. The distance between *map* and *probe* is defined as the score of their optimal alignment.

For $(i,j)\in Matchpts$, let $D(i,j)$ be the score of the best alignment whose

rightmost pair is (i, j), *excluding* the penalty for the distance discrepancy between aligned sites (i, j) and the end of the maps. Then we have the following recurrence for computing D-values.

$$D(i,j)=\min\{(\lambda(M+P-2)+\mu|m_i-p_j|, \min_{\substack{(I,J)\in Matchpts \\ I<i,J<j}} contrib_{I,J}(i,j)\}$$

$$\text{where } contrib_{I,J}(i,j)=D(I,J)-2\lambda+\mu|(m_i-m_I)-(p_j-p_J)|.$$

Any algorithm that computes the D-values can then deliver the cost of an optimal alignment between *map* and *probe* in $O(R)$ additional time by computing the minimum:

$$\min\{\lambda(M+P), \min_{(i,j)\in Matchpts} D(i,j)+\mu|(m_M-m_i)-(p_P-p_j)|\}.$$

Our search algorithm is easily modified to compute the D-values for this slight variation to the recurrence of Lemma 1. In fact, we need only redefine $E_{I,J}=D(I, J)+2\lambda$ and $B^i_{I,J}=E_{I,J}$. With these simple modifications, the treatment of profiles in Sections 4 through 6 is correct for this variation. In essence, curves in this problem are shifted right by Δm_i and *up by 0* (as opposed to λ) in advancing from column i to $i+1$. As shown earlier the basic algorithm takes $O(R(\log R+\log M))$ time, where R is bounded by MP. Thus computing the distance between two maps requires no more than $O(MP(\log M+\log P))$ time and $O(MP)$ space.

Empirical experience. Table 1 reports the running times, in seconds, of the Miller–Huang algorithm (1988) and the basic version of the new algorithm for various values of P. Both algorithms were coded in C and the times were obtained on a SUN 4/260 workstation running SunOS Unix (Release 4.0.3.). For the new algorithm the candidate lists were realized with AVL trees. As seen in Table 1, for $P\leqslant 400$, the Miller–Huang algorithm is faster than the new algorithm. However, while the running time of the new algorithm is increasing roughly as $P \log P$, the Miller–Huang algorithm is increasing faster than P^2.

Table 1. Running time

M	P	$O(MP^3)$	$O(MP \log P)$
7000	100	30.3	103.6
7000	200	93.5	222.7
7000	300	209.0	350.4
7000	400	414.9	501.4
7000	500	721.7	635.8
7000	600	1236.9	784.9
7000	700	2090.6	939.4

8. Conclusion. We have presented a sparse dynamic programming algorithm for comparing and searching restriction maps under an objective function that is a linear combination of mismatch and distance discrepancy penalties (Waterman *et al.*, 1984). Our algorithm is based on a candidate list paradigm and can either search or compare maps in $O(R(\log M + \log P))$ time and $O(R)$ space in the worst case. For the searching variation where P is much smaller than M, as simple refinement gives $O(M + P^2)$ space and $O(R \log P)$ time.

The notion of alignment and objective scoring function for a particular class of objects is usually chosen to reflect an underlying model for the origin of the differences. Unlike the case of sequences where differences are usually evolutionary, a principle reason for the need for approximate matching in comparing restriction maps is the introduction of experimental errors in producing the maps. The nature of these experimental inaccuracies has not been studied but it is clear from preliminary experience that: (1) close, adjacent sites may be ordered incorrectly; and (2) close, adjacent sites for the same enzyme may be detected as only one. Thus one may need to consider alignment models that permit transpositions and/or the alignment of a site with two or more sites on the other map. Miller *et al.* (1990) presents an $O(MP^2)$ algorithm that accommodates transpositions under an alternate scoring scheme. In addition to these "topological" considerations, the objective function itself is in question. For a linear distance discrepancy penalty, an alignment with one "unreasonably" large discrepancy can score better than an alignment with four or five "reasonable" ones. Perhaps one needs to penalize according to the square or some other power of the distance discrepancy? This leads to the problem of comparing maps under more general scoring schemes than the one considered in this paper, e.g. convex discrepancy penalties. Finally, there is the problem of designing an efficient algorithm that accommodates both more general alignment topologies *and* objective functions.

LITERATURE

Aho, A. V., J. E. Hopcroft and J. D. Ullman. 1974. *The Design and Analysis of Computer Algorithms*. Reading, MA: Addison-Wesley.

Eppstein, D., Z. Galil and R. Giancarlo, 1988. Speeding up dynamic programming. *29th Symposium on the Foundation of Computer Science*, pp. 488–496.

Eppstein, D., Z. Galil, R. Giancarlo and G. Italaino. 1990. Sparse dynamic programming. *First ACM-SIAM Symposium on Discrete Algorithms*, pp. 513–522.

Hirschberg, D. S. and L. L. Larmore. 1987. The least weight subsequence problem. *SIAM J. Comput.* **16**, 628–638.

Huang, X. 1988. *A fast algorithm for computing the distance between restriction maps*. TR 88–42 Dept. of Computer Science, The Pennsylvania State University Park, PA 16802.

Kohara, Y., K. Akiyama and K. Isono. 1987. The physical map of the whole *E. Coli* chromosome: application of a new strategy for rapid analysis and sorting of a large genomic library. *Cell* **50**, 495–508.

Miller, W. and X. Huang. 1988. *An algorithm for searching restriction maps.* TR 88–41 Dept. of Computer Science, The Pennsylvania State University, University Park, PA 16802.
Miller, W. and E. W. Myers. 1988. Sequence comparison with concave weighting functions. *Bull. math. Biol.* **50**, 97–120.
Miller, W., J. Ostell and K. E. Rudd. 1990. An algorithm for searching restrictions maps. *CABIOS* **6**, 247–252.
Sleator, D. D. and R. E. Tarjan. 1985. Self-adjusting binary search trees. *J. ACM* **32**, 652–686.
Waterman, M. S. 1984. Efficient sequence alignment algorithms. *J. theor. Biol.* **108**, 333–337.
Waterman, M. S., T. F. Smith and H. L. Katcher. 1984. Algorithms for restriction map comparisons. *Nucl. Acids Res.* **12**, 237–242.

Revised 5 March 1991