CONSTRAINED SEQUENCE ALIGNMENT

□ KUN-MAO CHAO[1]

□ ROSS C. HARDISON[2,3]

□ WEBB MILLER[1,3]

[1]Department of Computer Science,

[2]Department of Molecular and Cell Biology and

[3]Institute for Molecular Evolutionary Genetics

The Pennsylvania State University

University Park, PA 16802

This paper presents a dynamic programming algorithm for aligning two sequences when the alignment is constrained to lie between two arbitrary boundary lines in the dynamic programming matrix. For affine gap penalties, the algorithm requires only $O(F)$ computation time and $O(M+N)$ space, where F is the area of the feasible region and M and N are the sequences' lengths. The result extends to concave gap penalties, with somewhat increased time and space bounds.

1. Introduction. A number of authors have used dynamic programming methods to align sequences. One line of investigation has been to reduce the space requirements, which at first glance appear to be proportional to the product of the sequence lengths. In particular, for aligning two sequences, space proportional to the sum of the sequence lengths is adequate for affine gap penalties (Hirschberg, 1975; Myers and Miller, 1988; Huang and Miller, 1991), which are generally considered appropriate for aligning DNA and protein sequences. (''Affine'' means that a gap of length k is penalized $g + e \times k$, i.e., it costs g to open up a gap plus e for each symbol in the gap.) These ''linear-space'' algorithms

K.-M. C. and W. M. were supported in part by grant R01 LM05110 from the National Library of Medicine. R. C. H. was supported by PHS grant R01 DK27635.

require only a small constant factor more time than do the quadratic-space versions.

These earlier results apply only to "unconstrained" sequence alignment, i.e., an element of the first sequence can be aligned to any element of the second. However, there are situations where the ith entry of the first sequence can be aligned to the jth entry of the second sequence only if $L[i] \le j \le R[i]$, for given left and right bounds L and R. It is straightforward to modify the linear-space unconstrained alignment algorithms to handle these constraints, but the resulting algorithms are not as time-efficient as one might want. In particular, their running time may be worse than proportional to the area of the dynamic-programming matrix between the "constraint lines" L and R. In other words, the time to explicitly produce an optimal alignment may exceed the time to compute simply the alignment's score by more than a constant factor; the factor attained can be as large as $\log_2 N$ for sequences of length N (Chao *et al.*, 1991).

Chao *et al.* (1991) give a linear-space and time-efficient (i.e., using just "score-only" time) algorithm for aligning two sequences within a diagonal band, i.e., where $L[i] = L[0] + i$ for some constant $L[0]$ and similarly for R. The basic idea is to define a diagonal *partition line* that bisects the band into two nearly equal parts. Unlike the earlier approach, which recursively divides the problem into two subproblems, the new method determines a variable number of subproblems. To attain the score-only time bound, the total size of the subproblems (i.e., the area of the region of the dynamic-programming matrix to be evaluated) is restricted to at most α times the size of the containing problem, for some $\alpha < 1$. The algorithm of Chao *et al.* matches the two-subproblem methods in attaining the constant $\alpha = ½$.

In this paper, we extend the partition-line approach to general L and R. The basic result, for affine gap penalties, is couched in terms of finding a maximum-weight path through a certain graph. The correspondence between alignments and paths in graphs has been exploited by a number of workers; here we utilize the formulation of Myers and Miller (1989). Near the end of the paper we sketch how the result can be extended from affine gap penalties to the "concave" gap penalties first studied by Waterman (1984).

The C source code for our constrained global and local alignment algorithms for affine gap penalties can be obtained over the Internet via anonymous ftp from groucho.cs.psu.edu. The authors can be contacted by electronic mail at webb@cs.psu.edu.

2. The Basic Algorithm.

Definition 1. For sequences $A = a_1 a_2 \cdots a_M$ and $B = b_1 b_2 \cdots b_N$, the *alignment graph*, $G_{A,B}$, is a directed graph with weighted edges, defined as follows. G has $3(M+1)(N+1)$ nodes, denoted $(i, j)_S$, $(i, j)_D$, and $(i, j)_I$, where $i \in [0, M]$ and $j \in [0, N]$. (We use $[x, y]$ to denote the set of integers t such that $x \le t \le$ y.) Nodes $(i, j)_S$, $(i, j)_D$, and $(i, j)_I$ are said to *occur at grid point* (i, j). The following edges, and only these edges, are in $G_{A,B}$.

1. $(i-1, j)_D \rightarrow (i, j)_D$ for $i \in [1, M]$ and $j \in [0, N]$.
2. $(i-1, j)_S \rightarrow (i, j)_D$ for $i \in [1, M]$ and $j \in [0, N]$.
3. $(i, j-1)_I \rightarrow (i, j)_I$ for $i \in [0, M]$ and $j \in [1, N]$.
4. $(i, j-1)_S \rightarrow (i, j)_I$ for $i \in [0, M]$ and $j \in [1, N]$.
5. $(i-1, j-1)_S \rightarrow (i, j)_S$ for $i \in [1, M]$ and $j \in [1, N]$.
6. $(i, j)_D \rightarrow (i, j)_S$ for $i \in [0, M]$ and $j \in [0, N]$.
7. $(i, j)_I \rightarrow (i, j)_S$ for $i \in [0, M]$ and $j \in [0, N]$.

The edge weights are numbers; typically some of the type 5 (substitution) edges have positive weights and all other weights are negative or zero. (We defer details about the weights until they are needed; see Table 1, below.) The *score* of a path is the sum of the weights of its edges, and the problem of optimally aligning A and B is equivalent to the problem of determining a path in $G_{A,B}$ of maximum score. With the *constrained* alignment problem we are also given a left boundary point $(i, L[i])$ and a right boundary point $(i, R[i])$ in each row i, where $L[i] \le R[i]$; paths are limited to nodes at grid points (i,j) satisfying $L[i] \le j \le R[i]$ for $0 \le i \le M$, called the *feasible* grid points.

The goal of this section is to develop an algorithm for constrained *global* alignment, meaning that the path must begin at $(0,0)_S$ and end at $(M,N)_S$. Our algorithm is recursive, i.e., the algorithm performs subcomputations that consist of applying itself to smaller optimal-path subproblems. We require that the original problem and all subproblems that arise must satisfy the following conditions.

Definition 2. Consider the problem of constructing an optimal path from $(I_1, J_1)_{Type1}$ to $(I_2, J_2)_{Type2}$ in a given alignment graph, subject to the constraints $L[i] \le j \le R[i]$ for $I_1 \le i \le I_2$. The problem is *normal* if the following three conditions hold.

(1) $L[I_1] = J_1$ and $R[I_2] = J_2$.

(2) $L[i] \ge L[i-1]$ and $R[i] \ge R[i-1]$ for $I_1 < i \le I_2$.

(3) $L[i] \le R[i-1]+1$ for $I_1 < i \le I_2$.

Observe that if condition (3) is violated then no paths satisfy the constraints. An L satisfying $L[I_1] \le J_1$ but not conforming to conditions (1) and (2) can be replaced by $L'[i] = \max\{J_1, \max\{L[k]\colon I_1 \le k \le i\}\}$, and a nonconforming R can be replaced by $R'[i] = \min\{J_2, \min\{R[k]\colon i \le k \le I_2\}\}$. The set of possible paths is not affected because edges of G are always directed downward and/or to the right. Moreover, the resulting left and right boundaries satisfy (1) and (2) and determine a subset of the original feasible region.

For $I_1 \le i \le I_2$, set $mid(i) = \lceil \frac{1}{2}(L[i] + R[i]) \rceil$. (This notation means that midpoints are rounded *up*; this is done for technical reasons explained below.) A grid point (i, j), where $I_1 \le i \le I_2$ and $J_1 \le j \le J_2$, is called a *partition point* if either (1) $\max\{L[i], mid(i-1)\} \le j \le mid(i)$ or (2) $mid(i) < j \le \min\{L[i+1], R[i]\}$. For the purposes of this definition, we interpret $mid(I_1 - 1)$ to be J_1 and $L[I_2 + 1]$ to be J_2, so that (I_1, J_1) and (I_2, J_2) are partition points. Case (1) applies to all rows and guarantees that there is at least one partition point in each row, namely at $(i, mid(i))$. (Note that condition (2) of Definition 2 guarantees that $mid(i-1) \le mid(i)$.) Case (2) applies when the feasible region in one row begins after the midpoint of the previous row, and serves to eliminate gaps in the *partition line*, i.e., the set of partition points. This case is illustrated near the bottom of Fig. 1. The intuition behind the definition is to make the rightmost partition point of one row occur in the same column as the leftmost partition point of the next row.

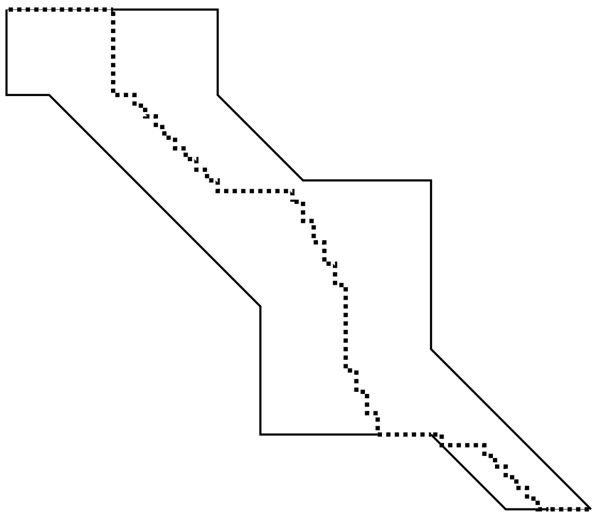

Figure 1. Partition points.

A number of minor observations about partition points will prove useful. When the grid points in row i are listed in left-to-right order, there are zero or more non-partition points (which we describe as lying "left of the partition line"), followed by one or more partition points, followed by zero or more non-partition points (lying "right of the partition line"). We classify a node of G as lying left of the partition line, being a partition node, or lying right of the partition line according to the status of its grid point. Moreover, each back-diagonal (i.e., the set of grid points where $i + j$ equals a constant) contains at most one partition point, so the number of partition points is at most $M + N + 1$. To be more precise, every back-diagonal between $I_1 + J_1$ and $I_2 + J_2$ contains exactly one partition point, except for back-diagonal $i + L[i] - 1$ when $L[i] = R[i - 1] + 1$ (in which case the back-diagonal contains no points of the feasible region). Also, of the $R[i] - L[i] + 1$ feasible grid points in row i, at most half lie left of the partition line and at most half lie to the right. The most critical property of the partition line is stated as the following obvious Lemma.

Lemma 1. Suppose that nodes r and s are connected by a path P in G, where one of r or s lies left of the partition line and one lies to the right. Then a partition node occurs on P between r and s.

To compute the score of an optimal path from $(I_1, J_1)_{Type1}$ to $(I_2, J_2)_{Type2}$, and to break the problem into subproblems, we employ a variant of the standard dynamic-programming alignment algorithm. For reasons that will later become clear, rows are processed from I_2 down to I_1, sweeping right to left within a row. At grid point (i, j), the following quantities are computed for each of the three nodes.

Score — score of some optimal path P from the node to $(I_2, J_2)_{Type2}$.

Succ — next partition node on P.

At a given node, these quantities can be determined from the corresponding quantities for nodes immediately below and/or to the right, provided that they are evaluated for the S node at the current grid point before the D and I nodes are treated.

For example, consider $(i, j)_D$. Any optimal path P must begin with one of two edges, $(i, j)_D \rightarrow (i, j)_S$ or $(i, j)_D \rightarrow (i+1, j)_D$. In the former case, the suffix of P starting at $(i, j)_S$ must constitute an optimal path from $(i, j)_S$ to $(I_2, J_2)_{Type2}$, so P's score is the sum of the weight of the edge $(i, j)_D \rightarrow (i, j)_S$ and $Score((i, j)_S)$. Similarly, we can compute the score of the best path that begins with $(i, j)_D \rightarrow (i+1, j)_D$, so by comparing two quantities we can determine both *Score* and the next node on an optimal path. To see how *Succ* is found, suppose that an optimal path passes through $(i, j)_S$ (the other case is analogous). If (i, j) is a partition point, then $Succ((i, j)_D)$ is $(i, j)_S$; otherwise $Succ((i, j)_D)$ equals $Succ((i, j)_S)$. This reasoning extends to the other types of nodes.

To describe this part of the computation somewhat more formally, let $g > 0$ denote the gap-open penalty and let $e > 0$ denote the gap-extension penalty. Also, let $\sigma(a, b)$ be the score for aligning a with b. As explained by Myers and Miller (1989), each edge of the graph $G_{A,B}$ of Definition 1 has an associated weight (needed for computing the score of an optimal alignment) and an associated aligned pair (needed for producing the alignment itself). The correspondence is as follows.

edge type	*weight*	*aligned pair*
1	$-e$	$\begin{bmatrix} a_i \\ - \end{bmatrix}$
2	$-(g+e)$	$\begin{bmatrix} a_i \\ - \end{bmatrix}$
3	$-e$	$\begin{bmatrix} - \\ b_j \end{bmatrix}$
4	$-(g+e)$	$\begin{bmatrix} - \\ b_j \end{bmatrix}$
5	$\sigma(a_i, b_j)$	$\begin{bmatrix} a_i \\ b_j \end{bmatrix}$
6 or 7	0	none

Table 1. The weights and aligned pairs associated with edges of $G_{A,B}$.

Fig. 2 formalizes the computation described above. In practice, values at $(i+1, j)_D$ where $L[i] \le j < L[i+1]$ must be treated appropriately (e.g., initialized to $Score = -\infty$). Also, observe that no values are assigned to *Succ* at grid point (I_2, J_2), which allows us to determine the end of the next subproblem after processing the current subproblem.

```
/* Initializations */
for t ∈ {'D','I','S'} do
  {  if t = Type2 or Type2 = 'S' then
           Score((I2, J2)t) ← 0
     else Score((I2, J2)t) ← − ∞
  }
for j ← J2 − 1 down to L[I2] do
     Compute Score and Succ at grid point (I2, j).

for i ← I2 − 1 down to I1 do
  {  Compute Score and Succ at grid point (i, R[i]).
     for j ← R[i] − 1 down to L[i] do
       {  Compute Score and Succ for (i, j)S.
          Compute Score and Succ for (i, j)I.
          if Score((i, j)S) ≥ Score((i + 1, j)D) − e then
            {  Score((i, j)D) ← Score((i, j)S)
               if (i, j) is a partition point then
                     Succ((i, j)D) ← (i, j)S
               else  Succ((i, j)D) ← Succ((i, j)S)
            }
          else
            {  Score((i, j)D) ← Score((i + 1, j)D) − e
               if (i + 1, j) is a partition point then
                     Succ((i, j)D) ← (i + 1, j)D
               else  Succ((i, j)D) ← Succ((i + 1, j)D)
            }
       }
  }
```

Figure 2. Sketch of backward computation of *Score* and *Succ*.

To attain linear-space performance, Fig. 2 can be implemented so that *Score* and *Succ* are saved only for partition nodes and for nodes in rows i and $i+1$. A more detailed discussion of space requirement is given in Sec. 4, below. When the process finally treats $(I_1, J_1)_{Type1}$, we can use *Succ* information to reconstruct the sequence of nodes where an optimal alignment crosses the partition line, as depicted in Fig. 3, thus breaking the problem into subproblems.

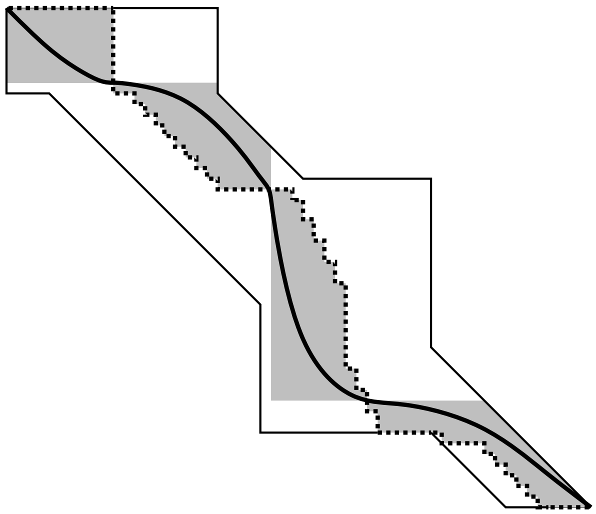

Figure 3. Splitting the problem into subproblems (shaded areas).

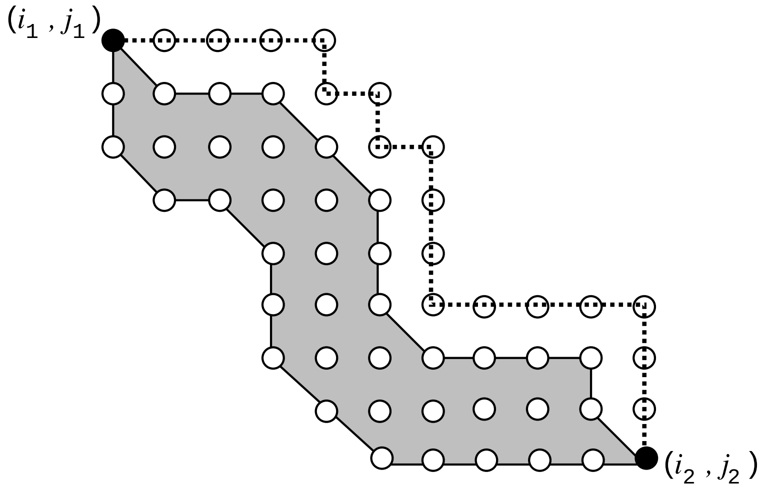

Figure 4. A closer view of a subproblem.

Fig. 4 shows a prototypical subproblem lying on the left side of the partition line. The left constraints $L[i]$ for the subproblem differ from those for the containing problem only in that they are "chopped off" at column j_1. The new right constraints are as follows. $R[i_1] = j_1$ and $R[i_2] = j_2$. For i satisfying $i_1 < i < i_2$, the new $R[i]$ is one less than the column index of row i's leftmost partition point in the containing problem. This leftmost partition point occurs in column $\max\{L[i], mid(i-1)\}$. However, if $L[i]$ exceeded

$mid(i-1)$ then $(i, L[i])$ would be a partition point and this could not be a *left* subproblem. Thus the new value of $R[i]$ is $mid(i-1)-1$. The following pseudo-code makes these changes.

```
for i ← i2 down to i1 do
  {  L[i] ← max{ j1, L[i]}
     R[i] ← mid(i − 1) − 1
  }
R[i1] ← j1
R[i2] ← j2
```

Observe that rows are treated in reverse order because the new $R[i]$ depends on $mid(i-1)$, which in turn depends on $R[i-1]$; this allows us to overwrite the old values of L and R with the new ones. The first and last values of R are computed in the loop, then later reset; this saves a couple of lines of code (and makes the value of $mid(i_1-1)$ irrelevant here). Finally, note that normality of the containing problem (see Definition 2) is transferred to the subproblem.

Similar code handles subproblems lying on the right side of the partition (see lines 27-33 of Fig. 5). For i satisfying $i_1 < i < i_2$, the new $L[i]$ is one more than the rightmost partition point in row i, which occurs in column $\max\{mid(i), \min\{L[i+1], R[i]\}\}$. However, because this is a *right* subproblem we can argue that $L[i+1] < R[i]$, which simplifies the formula for the new $L[i]$.

The next question is how to decide, given successive partition points along an optimal path, whether the subproblem is a left subproblem or a right subproblem. The following Lemma provides an answer.

Lemma 2. Suppose that p and q are partition nodes on a constrained path P in G, p occurs at (i, j), q follows p on P, and no partition nodes occur on P between p and q. Also, let Q denote the partition line. If $j < mid(i)$, then all nodes on P strictly between p and q lie left of Q. Otherwise, all nodes on P strictly between p and q lie right of Q.

Proof. Let s be the node immediately following p on P. If $s = q$, then the Lemma holds trivially because there are no nodes on P between p and q. In particular, this obtains if the edge of P that leaves p is of type 6 or 7, since such edges stay at the same grid point. Thus we can assume that s lies on one side of Q or the other and that the edge leaving p is of type 1-5. Lemma 1 implies that all nodes between p and q lie on this same side, so the problem is to locate s.

First suppose that $j < mid(i)$. The edge leaving p cannot be a horizontal edge since the node immediately right of (i, j) is a partition node. Thus s occurs at either $(i+1, j)$ or $(i+1, j+1)$. The partition point $(i+1, mid(i))$ must lie to the right of that node, i.e., s lies left of Q.

Conversely, suppose $j \geq mid(i)$. Since the leftmost partition point in row $i+1$ occurs at column $\min\{L[i+1], mid(i)\}$, s cannot lie left of Q. □

```
shared arrays A, B, L, R

procedure ALIGN(M, N)
  { align(0, 0, 'S', M, N, 'S') }

recursive procedure align(I_1, J_1, Type1, I_2, J_2, Type2)
  /* Construct optimal path from (I_1, J_1)_Type1 to (I_2, J_2)_Type2 */
  { Do the backward computation in Fig. 2.
    for each subproblem do
      { Let partition nodes (i_1, j_1)_t1 and (i_2, j_2)_t2 bound the subproblem
        if i_1 = i_2 then { if j_2 > j_1 then write [ - ; b_j2 ] }
        else if j_1 = j_2 then write [ a_i2 ; - ]
        else if i_1 + 1 = i_2 and j_1 + 1 = j_2 then
          { if t2 = 'I' then write [ a_i2 ; - ][ - ; b_j2 ]
            else if t2 = 'D' then write [ - ; b_j2 ][ a_i2 ; - ]
            else write [ a_i2 ; b_j2 ]
          }
        else
          { l_save <- L[i_2]
            r_save <- R[i_2]
            if j_1 < mid(i_1) then /* Form left subproblem. */
              { for i <- i_2 down to i_1 do
                  { L[i] <- max{ j_1, L[i] }
                    R[i] <- mid(i - 1) - 1
                  }
                R[i_1] <- j_1
                R[i_2] <- j_2
              }
            else              /* Form right subproblem. */
              { for i <- i_1 to i_2 do
                  { L[i] <- max{ mid(i), L[i + 1] } + 1
                    R[i] <- min{ j_2, R[i] }
                  }
                L[i_1] <- j_1
                L[i_2] <- j_2
              }
            align(i_1, j_1, t1, i_2, j_2, t2)
            L[i_2] <- l_save
            R[i_2] <- r_save
          }
      }
  }
```

Figure 5. The algorithm for constrained global alignment with affine gap penalties.

Lemma 3. The procedure *ALIGN* of Fig. 5 terminates execution and correctly computes an optimal alignment of A and B.

Proof. First we prove termination by showing that if *align* calls *align*, then the recursive call involves a subproblem with strictly fewer grid points. Suppose that there is only one subproblem, since the situation is clear with multiple subproblems. If the call to *align* at line 34 is executed, then the subproblem is nontrivial in that $i_2 > i_1$, $j_2 > j_1$, and either $i_2 > i_1 + 1$ or $j_2 > j_1 + 1$. In other words, the partition nodes bounding the problem do not occur at immediate neighbors in the grid, hence there is at least one point on the partition line lying between them. This point is discarded before the subproblem is solved, which completes verification of the algorithm's termination.

It is important to observe that execution of lines 7-38 (including recursive calls to *align*) does not affect values of $L[i]$ and $R[i]$ (and hence $mid(i)$) for rows i that appear in later subproblems. The only row that overlaps later subproblems is row i_2, and lines 16-17 and 35-36 of Fig. 5 guarantee that $L[i_2]$ and $R[i_2]$ are preserved.

Lemma 2 shows that line 18 correctly determines the location relative to the partition line of the subproblem. As mentioned in the discussion of Fig. 4, lines 19-25 correctly form a left subproblem, and similar reasoning verifies that lines 27-33 properly create a right subproblem. Thus, confirming correctness now focuses on the trivial cases (lines 8-14). First consider a subproblem where $i_1 = i_2$. If we also have $j_1 = j_2$, then the subproblem's optimal path is a single edge of type 6 or 7. Such edges have no associated aligned pair, and Fig. 5 processes such a subproblem by doing nothing. Otherwise, $j_1 > j_2$. Since all grid points between (i_1, j_1) and (i_1, j_2) must be partition points, it follows that $j_2 = j_1 + 1$. Thus the optimal path for this subproblem consists of a single edge of type 3 or 4 labeled $\begin{bmatrix} - \\ b_{j_2} \end{bmatrix}$, verifying line 8. If $j_1 = j_2$ and $i_1 < i_2$, then we have a run of partition points in the same column, so $i_2 = i_1 + 1$. Thus the optimal path for this subproblem consists of a single edge of type 1 or 2 labeled $\begin{bmatrix} a_{i_2} \\ - \end{bmatrix}$, verifying line 9.

The remaining case is where $i_2 = i_1 + 1$ and $j_2 = j_1 + 1$. If $t2$ = 'I', then the path has an edge to $(i_1 + 1, j_1)_D$, then a type-6 edge to $(i_1 + 1, j_1)_S$, and finally a type-4 edge to $(i_1 + 1, j_1 + 1)_I$. Line 11 writes the labels on the first and third edge (the second is unlabeled). Line 12 works similarly. The remaining alternative is that $t2$ = 'S', in which case the path consists of a single edge labeled $\begin{bmatrix} a_{i_2} \\ b_{j_2} \end{bmatrix}$, verifying line 13. (Lines 11 and 12 would be made unnecessary by the requirement that substituting one symbol for another has a better score than the alternative of deleting one symbol and inserting the other.) $\square$

3. Time Analysis. The intuitive idea behind the algorithm's "score-only" time performance is that the total sizes of the subproblems (where "size" means the number of grid points) is at most roughly half the size of the containing problem. Letting F denote the number of points in the original feasible region, the total number of grid points considered (including the subproblems and the subsubproblems, etc.) is $F + ½F + ¼F + \cdots < 2F$. Reasoning that the overhead for retaining *Succ* should at most double the cost of computing just *Score*, the time for the original (outermost) execution of Fig. 2 should be at most twice the score-only time, and the total time for delivering the alignment should be at most four times the score-only time.

A closer look reveals some complications. Consider an arbitrary call to *align*, say to compute an optimal path from $(I_1, J_1)_{Type1}$ to $(I_2, J_2)_{Type2}$, and let i be a row of that problem. How many of the grid points in row i are passed to a subproblem (including multiplicities when a point occurs in more than one subproblem)? If row i is not the bounding row for a subproblem, then row i occurs in at most one subproblem, after at least half the grid points in row i have been discarded by lines 20-21 or 28-29. However, suppose that row i is both the last row of a left subproblem (see Fig. 4) and the first row of a right subproblem. Then in general every point in row i is passed to one subproblem or the other, and one point is passed to both. Moreover, the first row of the problem might consist of a single node that is passed to a subproblem. These complications can be handled, but getting a tight bound seems to require a tedious analysis. We will content ourselves with proving a rather loose result.

In outline, our approach is as follows. All F of the grid points in the original problem are considered "charged". For any other problem arising from a recursive call to *align*, all points are "charged" except for the problem's two bounding grid points. We show that the total number of charged points passed to subproblems is at most half the number of charged points of the current problem. This guarantees that T, the total number of points in all the problems, is less than $2F$ plus the total number of uncharged points in all the subproblems. Since all uncharged points occur at the bounding points of subproblems, the constructed optimal path passes through all of them, and a detailed accounting shows that each point on the optimal path is uncharged in at most four subproblems. It follows immediately that $T \leq 6F$ and that the algorithm's running time is $O(F)$. Theorem 1 contains a formal statement of the result.

Theorem 1. Fix A, B, L and R specifying a constrained alignment problem. Let F be the number of grid points in the feasible region, let P be number of aligned pairs in the computed optimal alignment, and let T be the total number of grid points in all the calls to *align*. Then $T \leq 2F + 4P$.

Proof. We first show that the number of points that are charged in any subproblem is at most half of the number of charged points of the current problem. Consider any problem that arises in the computation, i.e., corresponding to a call to *align*. Thus, the problem is determined by nodes $(I_1, J_1)_{Type1}$ and $(I_2, J_2)_{Type2}$ and by the current values of $L[i]$ and $R[i]$ for $I_1 \leq i \leq I_2$. The idea is that each nonbounding point in each subproblem is "charged to" a charged point of the current problem; the charging is done so that (1) no two points in subproblems are charged to the same point (including occurrences of the same point in two subproblems), (2) within any row of the problem, at most half of the points will have points in subproblems charged to them and (3) a point of a subproblem can be charged to (I_1, J_1) or (I_2, J_2) only in the case of the original problem.

With one exception, discussed below, a point is charged to itself. If row i does not occur as the first or last row of a subproblem, this works fine since, as mentioned above, at least half of the row is discarded when forming the subproblem. No conflict can occur at the first row of a left subproblem or the last row of a right subproblem, because the row contains only a single uncharged point. If row i is the first row of a right subproblem, then the new first row begins with the rightmost partition point of the current row i, so it can contain one point more than half of the current row. However, the first point of this new row is uncharged in the subproblem. There is a slightly sticky point concerning the first row of a right subsubproblem of a right subproblem. The first entry of the first row of the subproblem is uncharged and hence cannot be used for charging in the subsubproblem. To make this work properly, we defined *mid* by *rounding up*.

The only apparent conflict is when row i is both the last row of a left subproblem and the first row of a right subproblem. This is the case where we deviate from the obvious charging scheme. For the last row of a left subproblem, charge each node except the last to the closest partition point in the same column; that point lies strictly left of the *mid* of its row and no other point has been charged to it. See Fig. 6. A question arises when the leftmost entry in the last row occurs in column j_1; it appears that (i_1, j_1) could be the initial point of the current problem and hence itself uncharged. However, if the current problem is the original (i.e., highest-level) problem, then the initial point is charged. If the current problem is a left subproblem, then row i_1 contains just one point, so $(i_1 + 1, j_1)$ is a partition point that can be used for charging. If the current problem is a right subproblem, then (i_1, j_1) is the only feasible point in column j_1, so no point in the last row will be charged to it.

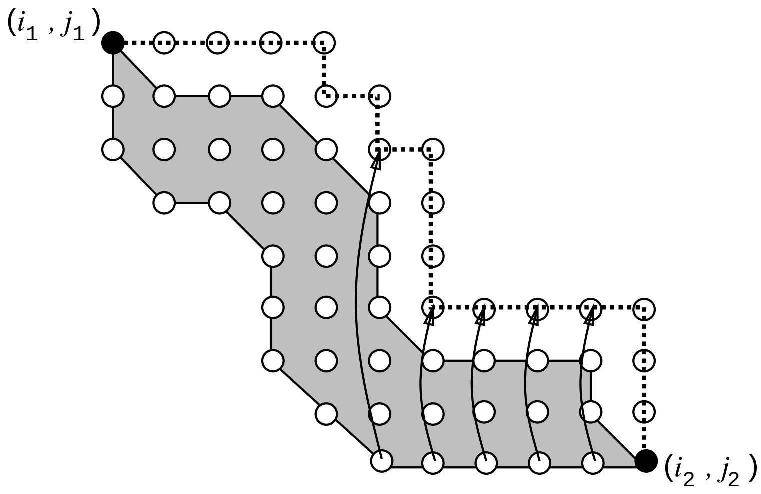

Figure 6. Charging scheme for last row of a left subproblem.

Fix a grid point (i, j) on the optimal path (i.e., alignment) that is found by the algorithm. We now bound the number of times that (i, j) can appear ‘‘uncharged’’ in a subproblem. Suppose that the point is neither $(0, 0)$ nor (M, N) (those cases are simpler). As the algorithm runs, the point will be passed as a nonbounding point to smaller and smaller problems until a problem is reached where (i, j) occurs as a bounding point to a subproblem. In the worst case, it is the terminal point of one subproblem and the initial point of another.

Suppose the upper subproblem is a left subproblem and the lower problem is a right subproblem. (There are three other cases, which are handled similarly.) Then (i, j) must be simultaneously the leftmost partition point in row i and the rightmost partition point. It follows that the partition point $(i - 1, j)$ is excluded from the upper subproblem and $(i + 1, j)$ is excluded from the lower subproblem, as in Fig. 7.

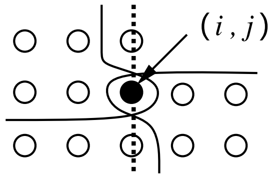

Figure 7. A left subproblem followed by a right subproblem.

When the upper subproblem is solved, there are two cases. First suppose that $(i-1, j-1)$ is a partition point for the subproblem. Then $(i, j-1)$ is also a partition point, the edge entering (i, j) must originate at a partition node, and (i, j) will not be passed to a subsubproblem. If $(i-1, j-1)$ is not a partition point, then (i, j) can be passed to a right subsubproblem of the left subproblem. However, the partition point $(i, j-1)$ will have been eliminated from the subsubproblem, so (i, j) will be passed no further. Similarly, the initial point (i, j) for the lower subproblem, can be passed to at most a left subsubproblem. Thus, in the worst case an interior point of the constructed optimal path is uncharged in four problems; the end points can be uncharged in at most two problems. Thus the total number of uncharged points is at most $4(P-1) + 2\times 2 = 4P$.

□

4. Implementation and space requirements. We earlier implied that values *Score* and *Succ* are stored at three nodes of each grid point (see Fig. 2). In practice, we need to save *Succ* only at partition nodes, and *Score* and *Succ* at nodes in the current row and the immediately following row. In fact, the critical information in those two rows requires only space for one row plus an additional grid point — see Myers and Miller (1988) for the basic idea.

Another issue concerns the representation of partition points. Our implementation uses the fact that each back-diagonal (the grid points where $i + j$ equals a constant) contains at most one partition point; we store the information for partition points in arrays of length $M + N + 1$ and use a subscript into these arrays to refer to a partition point. The space for data at partition points can be allocated once and shared by the subproblems. With this strategy, our implementation uses only array storage for $4M + 8N + 12$ integers and $3M + 5N + 7$ characters, plus $2M + 2$ integers and $M + N$ characters for the data. (The original L and R are overwritten.)

To see that the space for these shared arrays dominates the algorithm's space requirements, we need to consider the maximum size of the procedure activation stack, which depends on the maximum recursion depth. Define the *width* of the subproblem from $(I_1, J_1)_{Type1}$ to $(I_2, J_2)_{Type2}$ as $\max\{R[i] - L[i] + 1 \colon I_1 \le i \le I_2\}$, where L and R refer to the constraint values at the time of the call (i.e., after reduction at lines 19-33 of Fig. 5 in higher-level calls). The width of the problem at a recursive call to *align* is at most half that of the containing problem (rounded up), so the maximum stack depth is at most $\log_2 Y$ where Y is the width of the original problem. Thus, only $O(\log Y)$ space is needed for the procedure activation stack, where $Y \le N + 1$.

Certain economies are possible if the program is customized for alignment within a diagonal band. Our previous band-aligning algorithm (Chao *et al*., 1991) uses $4W + 3M + 11$ integers and $3M + 3$ characters of array storage, where W is the band width, plus 2 integers and $M + N$ characters for the data. Moreover, experiments indicated that for band-aligning problems the running time of the general constrained alignment program is roughly twice that of our previous band-aligning program.

5. An Example. The algorithm described in this paper was developed as part of our ongoing project to build a comprehensive software environment for aligning DNA or protein sequences (Schwartz *et al*., 1991; Boguski *et al*., 1992). Pairwise alignment programs are a major component of this software package. Currently we distribute only two pairwise alignment programs; one is a variant of the *blast* database-searching program (Altschul *et al*., 1990) and the other, called *sim*, uses dynamic programming (Huang and Miller, 1991). These two programs lie near the opposite extremes of the speed-versussensitivity spectrum; *blast* is several orders of magnitude faster than *sim*, but it can only produce gap-free alignments and sometimes fails to detect conserved regions found by *sim*.

The algorithm discussed here is used in a new alignment program, which is currently undergoing evaluation for possible distribution. Preliminary indications are that it has nearly the same sensitivity as *sim* and much greater efficiency. The algorithm begins by determining a large number of (possibly very short) gap-free alignments, as was done by Wilbur and Lipman (1984). For example, with DNA sequences we might start with all exact matches involving eight or more consecutive nucleotides from each sequence, or with proteins we might use "word hits" as in the *blast* algorithm (Altschul *et al*., 1990). Although there can be many of these tiny alignments, the number of them is far smaller than the product of the sequence lengths (e.g., smaller by a factor of around $4^8 = 65536$ for 8-nucleotide matches). A new technique developed by Galil and coworkers (Eppstein *et al*., 1992) has been adapted to efficiently determine an optimal alignment consisting of a chain of these fragmentary alignments. The adaptations (Chao and Miller, 1992) permit space-efficient construction of the k best nonintersecting chains, following the general approach of Huang and Miller (1991). Given an alignment constructed by chaining together fragmentary alignments, we then refine the alignment (e.g., to permit aligning two nucleotides that are not part of a run of eight consecutive exact matches) using this paper's algorithm. For region bounds we sometimes use the upper and lower envelopes of all upright rectangles extending from the end of the ith fragmentary alignment to the start of the $(i+2)$th, for relevant i, as pictured in Fig. 8.

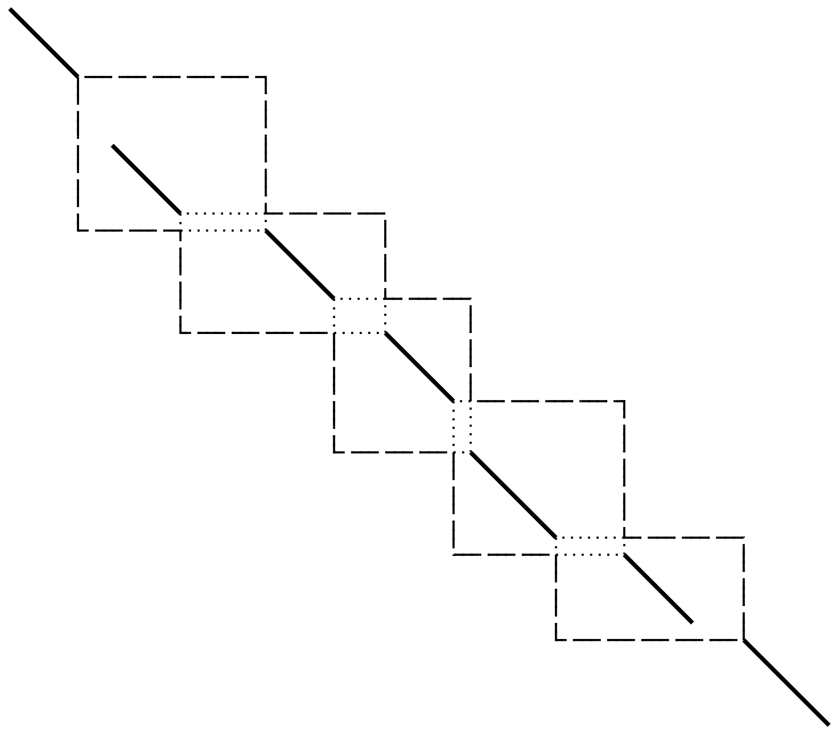

Figure 8. Alignment constraints derived from a chain of fragmentary alignments.

To illustrate the power and utility of this approach to sequence alignment, the entire 155844 base pair chloroplast genome from tobacco (Shinozaki *et al*., 1986) was aligned with the 121024 base pair chloroplast genome of the liverwort *Marchantia* (Ohyama *et al*., 1986). This is a challenging computational problem because these are two of the largest complete genome sequences known. From a biological perspective, they represent the two major groups of plants, vascular (tobacco) and nonvascular (liverwort), whose ancestors are thought to have diverged about 400 million years ago. The chloroplasts in both species, however, are believed to have arisen by endosymbiosis of a purple bacterium in the ancestor to all modern plants. Despite their prolonged separation, these two chloroplast genomes have retained substantial similarity in the content and order of genes (Palmer, 1991).

The process outlined above generated a single alignment covering well over half of the chloroplast genome, which is plotted by the *laps* program (Schwartz *et al*., 1991; Boguski *et al*., 1992) as the longer, downward-sloping line in Fig. 9. This pattern of matching sequence is broken in two places. First, the liverwort chloroplast genome has undergone an inversion of the region between *rpoB* and *psbA*; the matches in this region can be obtained by aligning the sequence of tobacco chloroplast DNA with the reverse complement of the liverwort chloroplast sequence. This alignment is plotted as the shorter, upward-sloping line in Fig. 9. The two analyses (aligning one sequence with both the "forward orientation" and reverse complement of the other) align about 72% of the tobacco sequence. Second, the homolog to ORF 2280 in tobacco (i.e. ORF 2136) is in a different position in liverwort, perhaps resulting from an inversion followed by an expansion of the inverted repeat region (Zhou *et al*., 1988). At the gap penalties used in this alignment, the region preceding the rRNA genes in liverwort generates a series of short matches with the ORF2280 region in tobacco, producing a horizontal broken line in Fig. 9, but these latter short matches are not meaningful (and were not counted in the 72% figure quoted above).

Computing the longer alignment in Fig. 9 took about 8 minutes on a Sun SparcStation II to chain together 1952 fragmentary alignments (from 678112 available fragmentary alignments between the two sequences), then 170 seconds to run our implementation of this paper's algorithm. Alignment of the same sequences by *blast* takes less than a minute, but the result is a large collection of gap-free alignments, many of which are very short and/or overlap other alignments, and which must be chained together to obtain a useful extended alignment. In contrast, the program described here generates very long alignments with appropriate gaps, which permits easier access to the biologically interesting information. The *sim* program, which applies dynamic programming to the full rectangular grid, would run for about a week on a SparcStation to generate the comparable alignment.

Investigation of the variability of sequence conservation illustrates the sorts of biologically useful information that can be readily obtained from long sequence alignments. The differential conservation of the chloroplast genomes, tabulated previously by Wolfe and Sharp (1988) and by Shimada and Sugiura (1991), is depicted in Fig. 10, which was generated automatically from the alignments in Fig. 9. The strongest matches are in the rRNA genes. The genes encoding the proteins of photosystems I (*psa*) and II (*psb*) are the most conserved protein-coding genes. Genes encoding other proteins involved in photosynthesis, such as the cytochrome b/f complex (*pet*) and ATPase (*atp*) are also well conserved, as is the gene encoding one subunit of an enzyme required for fixation of carbon dioxide (*rbcL*). The genes proposed to encode subunits of NADH dehydrogenase (*ndh*) are also conserved, but less than are the *psa* and *psb* genes. Less highly conserved genes include those whose products are involved in gene expression, such as the RNA polymerase subunits (*rpo*) or ribosomal proteins (*rps* and *rpl*). Two unidentified, long open reading frames (ORF1244 and ORF228 in tobacco, ORF1068 and ORF464 in liverwort) in the small single copy region show only short matches of about 50-60% identity. Although the reading frames are preserved in these regions, many insertions and/or deletions have accumulated since the divergence between the ancestors to tobacco and liverwort, leading to the differences in length of the two open reading frames in the two species. This is not the pattern of conservation observed in protein-coding genes, which suggests that perhaps this region is involved in some other function (or no function). DNA segments between the genes still align, but often in short segments and with a lower percent identity. Thus Fig. 10 provides a snapshot of the amount of sequence variation allowed between these two chloroplast genomes, presumably with the more functionally constrained sequences showing the highest percent identity. The constraints on the genes for photosystems I and II must be particularly severe, resulting in a segment of 2444 base pairs (essentially the *psbD,C* sequence) that align without interruption and others of 2225 base pairs (*psaB*) and 2262 base pairs (*psaA*), all of which show nearly 85%

identity.

Figure 9 goes near here.

Figure 9. Dot-plot-like representation of an alignment between the tobacco and liverwort chloroplast genomes (negative slope) and an alignment between tobacco sequence and the reverse complement of liverwort sequence (positive slope). Each alignment was computed by chaining together exact matches of length at least eight, followed by application of this paper's algorithm in the region depicted in Fig. 8, scoring matches as $\sigma(a, a) = 1$, mismatches as $\sigma(a, b) = -1$ when $a \neq b$, a gap-open penalty of $g = 3.0$ and a gap-extension penalty of $e = 0.5$. The plot was prepared by the *laps* program (Schwartz *et al*., 1991; Boguski *et al*., 1992) directly from the computed alignments and hand-edited files giving positions of sequence features.

Figure 10 goes near here.

Figure 10. *Laps* plot of the percentage matches for the two alignments of Fig. 9. The distance along the x-axis of a horizontal line depicts the extent of ten consecutive gap-free segments of the alignment (the rightmost line is a special case). The position of a line along the y-axis represents the number of matching aligned pairs in those ten segments, divided by the sum of the segments' lengths and the total of the lengths of the gaps that follow each of those segments, quoted as a percentage.

6. Extensions and Open Problems. This section sketches extensions of the basic algorithm of Sec. 2 to the local alignment problem and to concave gap penalties. A few open problems are posed, including that of aligning within a region containing holes.

6.1. Local alignment. In the *local alignment problem*, one seeks a highest-scoring alignment where the end-nodes can be arbitrary, i.e., they are not restricted to $(0,0)_S$ and $(M,N)_S$. A constrained local alignment problem can be reduced to a constrained *global* alignment problem by performing a preliminary pass over the feasible region to locate the first and last nodes of an optimal local alignment, then delivering a global alignment within a restricted region. This can be done in "score-only" space by any of several approaches (Huang *et at*., 1990). Given the starting point (I_s, J_s) and end point (I_e, J_e), the constraints are adjusted to $L'[i] = \max\{J_s, \max\{L[k]: I_s \le k \le i\}\}$, and $R'[i] = \min\{J_e, \min\{R[k]: i \le k \le I_e\}\}$ for $I_s \le i \le I_e$, and the global procedure is applied. Fig. 11 depicts a new feasible region.

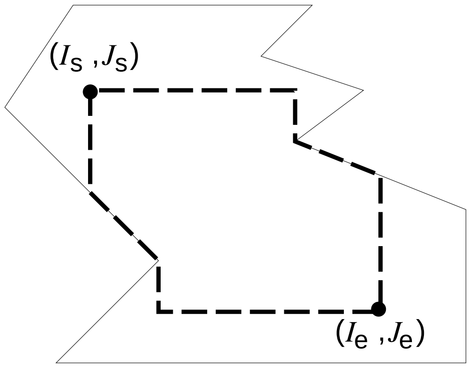

Figure 11. Reduction of a local constrained alignment problem to a global one.

6.2. Concave gap penalties. Dynamic-programming algorithms have been developed to optimally align two sequences when gaps are penalized by an arbitrary "concave" function, i.e., where the additional penalty for extending a gap by one position is a decreasing function of the gap length. (Waterman, 1984; Miller and Myers, 1988; Galil and Giancarlo, 1989). As described by Miller and Myers (1988), the idea is to compute, for each node of the dynamic-programming matrix, two "candidate lists" that record all potentially desirable gaps (vertical or horizontal edges) to that node. Because each of these lists (one for insertions and one for deletions) can require $O(\log(M + N))$ time per update, the alignment algorithm's running time is $O(MN \log(M + N))$.

Miller and Myers apply a variant of Hirschberg's strategy to produce an algorithm that requires linear space on average. However, the length of a candidate list can be as large as $O(M + N)$, which means that on some sets of data the method takes $O((M + N)^2)$ space. Indeed, a result of Rabani and Galil (1992) implies that any alignment algorithm for concave gap penalties must require quadratic space in the worst case.

The method for constrained alignment described in this paper can be modified for use with concave gap penalties. A backward pass analogous to Fig. 2 computes *Score*, *Succ* and the two candidate lists for each partition node. *Succ* gives both the next partition node on an optimal path and a specific entry in one of the node's candidate lists. This breaks the problem into subproblems bounded by candidate-list entries, as pictured in Fig. 12.

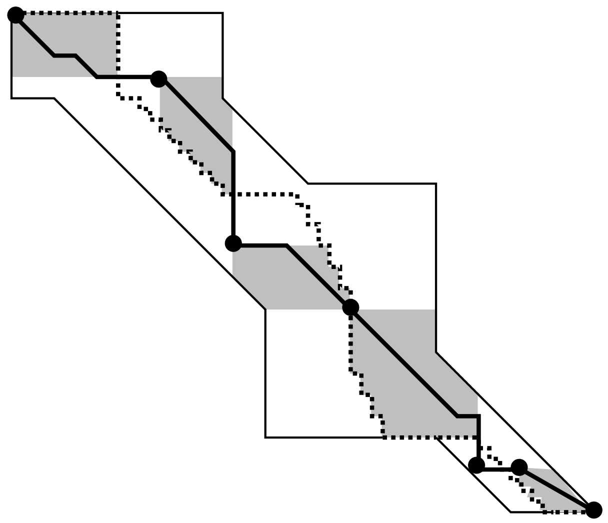

Figure 12. Subproblems when there are concave gap penalties.

Miller and Myers (1988) report the results of extensive experiments suggesting that the average size, $\bar{T}$, of a candidate list is constant in practice. This observation implies that in practice our method handles constrained alignment with concave gap penalties in linear space.

6.3. Open Problems. The bound stated as Theorem 1 is clearly pessimistic. There are ways to modify the algorithm so that a $2F$ bound is easy to prove. One approach is to increase the width of the partition line so that subproblems cannot overlap and/or the charging scheme depicted in Fig. 6 can be applied to right subproblems. However, we preferred to favor algorithm simplicity and program performance above simplicity of analysis. We conjecture that an equally simple algorithm with a provable $2F$ bound exists.

Another problem that we leave unsolved concerns feasible regions with holes. If there are only a few holes in the feasible region, then one can define partition lines along those rows where holes first appear and perform a preliminary pass to find where an optimal path crosses the lines. This splits the problem into several ‘‘hole-free’’ problems, as pictured in Fig. 13.

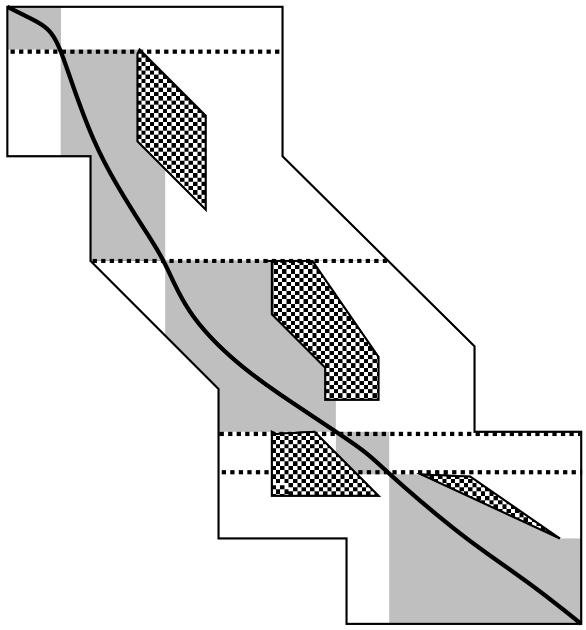

Figure 13. Reduction of a region with holes to several hole-free problems.

We leave as an open problem the description of an algorithm that handles an ‘‘arbitrary’’ feasible region in score-only time and space proportional to the input size.

7. Acknowledgement. We thank Scott Schwartz for assistance in producing Fig. 10.

LITERATURE

Altschul, S., W. Gish, W. Miller, E. W. Myers and D. Lipman. 1990. ‘‘Basic local alignment search tool.’’ *J. Mol. Biol.* **215**, 403-410.

Boguski, M., R. C. Hardison, S. Schwartz and W. Miller. 1992. ‘‘Analysis of conserved domains and sequence motifs in cellular regulatory proteins and locus control regions using new software tools for multiple alignment and visualization.’’ *The New Biologist* **4**, 247-260.

Chao, K.-M., W. R. Pearson and W. Miller. 1992. ‘‘Aligning two sequences within a specified diagonal band.’’ To appear in *CABIOS*.

Chao, K.-M., and W. Miller. 1992. A sparse dynamic programming algorithm. Submitted.

Eppstein, D., Z. Galil, R. Giancarlo and G. F. Italiano. 1992. ‘‘Sparse dynamic programming. I. Linear cost functions.’’ To appear in *Jour. Assoc. Comput. Mach.*.

Galil, Z., and R. Giancarlo. 1989. ‘‘Speeding up dynamic programming with applications to molecular biology.’’ *Theoretical Computer Science* **64**, 107-118.

Hirschberg, D. S. 1975. ‘‘A linear space algorithm for computing maximal common subsequences.’’ *Comm. ACM* **18**, 341-343.

Huang, X., R. Hardison and W. Miller. 1990. ‘‘A space-efficient algorithm for local similarities.’’ *CABIOS* **6**, 373-381.

Huang, X., and W. Miller. 1991. ‘‘A time-efficient, linear-space local similarity algorithm.’’ *Adv. Appl. Math.* **12**, 337-357.

Miller, W., and E. W. Myers. 1988. "Sequence comparison with concave weighting functions." *Bull. Math. Biol.* **50**, 97-120.

Myers, E. W., and W. Miller. 1988. "Optimal alignments in linear space." *CABIOS* **4**, 11-17.

— and —. 1989. "Approximate matching of regular expressions." *Bull. Math. Biol.* **51**, 5-37.

Ohyama, K., H. Fukuzawa, T. Kohchi, H. Shirai, T. Sano, S. Sano, K. Umesono, Y. Shiki, M. Takeuchi, Z. Chang, S.-I. Aota, H. Inokuchi and H. Ozeki. 1986. "Chloroplast gene organization deduced from complete sequence of liverwort *Marchantia polymorpha* chloroplast DNA." *Nature* **322**, 572-574.

Palmer, J. 1991. "Plastid chromosomes: Structure and evolution." In Bogorad, L. and Vasil, I. K. (eds) *The Molecular Biology of Plastids*, vol. 7 in *Cell Culture and Somatic Cell Genetics in Plants* (Vasil, I., ed.-in-chief).

Rabani, Y., and Z. Galil. 1992. "On the space complexity of some algorithms for sequence comparison." *Theoretical Computer Science* **95**, 231-244.

Schwartz, S., W. Miller, C.-M. Yang and R. C. Hardison. 1991. "Software tools for analyzing pairwise alignments of long sequences." *Nucleic Acids Research* **19**, 4663-4667.

Shinozaki, K., M. Ohme, M. Tanaka, T. Wakasugi, N. Hayashida, T. Matsubayashi, N. Zaida, J. Chunwongse, J. Obokata, K. Yamaguchi-Shinozaki, C. Ohto, K. Torazawa, B. Y. Meng, M. Sugita, H. Deno, T. Kamogashira, K. Yamada, J. Kusida, F. Takaiwa, A. Kato, N. Tohdoh, H. Shimada, M. Sugiura. 1986. "The complete nucleotide sequence of the tobacco chloroplast genome: its gene organization and expression." *EMBO J.* **5**, 2043-2049.

Shimada, H., and M. Sugiura. 1991. "Fine structural features of the chloroplast genome: comparison of the sequenced chloroplast genomes." *Nucleic Acids Research* **19**, 983-995.

Waterman, M. S. 1984. "Efficient sequence alignment algorithms." *J. theor. Biol.* **108**, 333-337.

Wilbur, W. J., and D. J. Lipman. 1984. "The context dependent comparison of biological sequences." *SIAM J. Appl. Math.* **44**, 557-567.

Wolfe, K. H., and P. M. Sharp. 1988. "Identification of functional open reading frames in chloroplast genomes." *Gene* **66**, 215-222.

Zhou, D., O. Massenet, F. Quigley, M. Marion, P. Huber and R. Mache. 1988. "Characterization of a large inversion in the spinach chloroplast genome relative to Marchantia: a possible transposon-mediated origin." *Curr. Genet.*, **13**, 433-439.