On Computing All Suboptimal Alignments†

Kun-Mao Chao

Department of Computer Science and Information Management
Providence University
Shalu, Taichung, Taiwan 43309

Abstract. Naor and Brutlag [16] proposed a new compact representation for suboptimal alignments. The kernel of that representation is a minimal directed acyclic graph (DAG) containing all suboptimal alignments. In this paper, a flexible space-saving scheme for computing such a DAG is proposed. For two sequences of lengths M and N ($M \leq N$), the general scheme runs in $O(MN \log \frac{1}{\Upsilon})$ time and $O(M^{\Upsilon}(M+N))$ space for arbitrarily small $0 < \Upsilon < 1$. As a consequence, the worst-case running time is $O(MN \log\log M)$ using only $O(N)$ working space. A variant of the method restricts the $\log\log M$ factor to affect only grid points lying between suboptimal alignments. It is also shown that a running time of $O(MN)$ can be achieved by using only $O(M^{1+\varepsilon}+N)$ working space for arbitrarily small constant $\varepsilon > 0$. To exploit the computed DAG, a variant of Aho-Corasick pattern matching machine [1] is employed to locate all occurrences of specified patterns, and then a path is found in the DAG that maximizes the sum of the scores of the non-overlapping patterns occurring in it. An example illustrates the utility.

Key Words. Divide-and-conquer, Dynamic programming, Linear-space algorithm, Sequence comparison.

†This work was supported in part by grant R01 LM05110 from the National Library of Medicine, National Institutes of Health, USA, and grant NSC84-2213-E-126-002 from the National Science Council, Taiwan.

1. Introduction.

Molecular biology is rapidly becoming a data-rich science with extensive computational needs [12]. More and more computer scientists are working together on developing efficient software tools for molecular biologists. One major area of potential interaction between computer scientists and molecular biologists arises from the need for analyzing biological information. In particular, optimal alignments have been used to reveal similarities among biological sequences, study gene regulation, or even infer evolutionary trees.

However, biologically significant alignments are not necessarily mathematically optimized. It has been shown that sometimes the neighborhood of an optimal alignment reveals additional interesting biological features [17, 20]. Besides, the most strongly conserved regions can be effectively located by inspecting the range of variation of suboptimal alignments [6, 19, 21]. While rigorous statistical analysis for the mean and variance of an optimal alignment score is not yet available, suboptimal alignments have been successfully used to informally estimate the significance of an optimal alignment.

For most applications, it is essentially impractical to enumerate all suboptimal alignments since the number could be enormous. Therefore, a more compact representation of all suboptimal alignments is indispensable. A 0-1 matrix can be used to indicate if a pair of positions is in some suboptimal alignment or not [19, 21]. As pointed out by Naor and Brutlag [16], this approach misses some connectivity information among those pairs of positions. They then used a set of "canonical" suboptimal alignments to represent all suboptimal alignments. The kernel of that representation is a minimal directed acyclic graph (DAG) containing all suboptimal alignments.

Space, rather than time, is often the constraining factor when applying dynamic-programming techniques to biological sequences [14]. Traditional dynamic-programming algorithms for sequence comparison require quadratic space, and hence are infeasible for long protein or DNA sequences. For example, with a DNA sequence (string of the four letters A, C, G, T) of length 50,000, a quadratic-space method uses billions of computer memory locations.

In this paper, we propose several space-efficient methods for computing the DAG representing all suboptimal alignments. In spite of the need for storing the DAG, these methods require very little additional working space. Let A and B be two sequences of length M and N respectively, where without loss of generality $M \leq N$. Our general scheme runs in $O(MN \log \frac{1}{\Upsilon})$ time and $O(M^{\Upsilon}(M+N))$ space for arbitrarily small $0 < \Upsilon < 1$. As a consequence, using only $O(N)$ working space, one method runs in time $O(MN \log\log M)$. Let F be the area of the region of the dynamic-programming matrix bounded by the suboptimal alignments and W the maximum width of that region. A variant of the method achieves a running time of $O(MN + F \log\log W)$ by shrinking the subproblems at each recursion step. Another method runs in time $O(MN)$ using $O(M^{1+\varepsilon} + N)$ working space for arbitrarily small constant $\varepsilon > 0$.

To exploit the computed DAG, we employ a variant of Aho-Corasick pattern matching machine [1] to locate all occurrences of specified patterns, and then find a path in the DAG that maximizes the sum of the scores of the non-overlapping patterns occurring in it. This is useful in delivering a more "meaningful" alignment. For instance, if there is more than one optimal alignment, we would prefer the one revealing more motifs of interest.

The rest of the paper is organized as follows. Section 2 gives some key definitions and a relatively simple linear-space algorithm for computing the DAG in time $O(MN + F \log W)$. Section 3 describes a divide-and-conquer approach that runs in $O(MN \log \frac{1}{\Upsilon})$ time and $O(M^{\Upsilon}(M+N))$ space for arbitrarily small $0 < \Upsilon < 1$. Section 4 refines the approach used in Section 3. Section 5 discusses an algorithm that finds a path in the computed DAG with the maximum pattern score. Section 6 gives an example to illustrate the utility. Finally, Section 7 discusses some future research directions.

2. Preliminaries.

Given two sequences $A = a_1 a_2 \cdots a_M$ and $B = b_1 b_2 \cdots b_N$, an *alignment* of A and B is obtained by introducing dashes into the two sequences such that the lengths of the two resulting sequences are identical and no column contains two dashes. Let Σ denote the input symbol alphabet. A score $\sigma(a, b)$ is defined for each $(a, b) \in \Sigma \times \Sigma$. A gap of length k is penalized $\alpha + k \times \beta$. The score of an alignment is the sum of σ scores of all columns with no dashes minus the penalties of the gaps. Figure 1 gives an example of an alignment's score. An *optimal alignment* is an alignment that maximizes the score.

```
ATACG---TA
AC--GTTCAA
```

Figure 1. An alignment of ATACGTA and ACGTTCAA. If $\sigma(a, a) = 1$, $\sigma(a, b) = -1$ if $a \neq b$, and a gap of length k is penalized $3 + k$, then the alignment's score is -10. There exist higher scoring alignments of these two sequences.

It is helpful to think of an alignment as a path in the alignment graph, $G_{A,B}$, defined as follows. $G_{A,B}$ is a directed graph with $3(M+1)(N+1)$ nodes, denoted $(i, j)_D$, $(i, j)_I$ and $(i, j)_S$, where $i \in [0, M]$ and $j \in [0, N]$. Table 1 depicts all the edges in $G_{A,B}$:

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

edge	*weight*	*aligned pair*	*range*
$(i-1,j)_D \rightarrow (i,j)_D$	$-\beta$	$\begin{bmatrix} a_i \\ - \end{bmatrix}$	$i\in[1,M]$ and $j\in[0,N]$
$(i-1,j)_S \rightarrow (i,j)_D$	$-(\alpha+\beta)$	$\begin{bmatrix} a_i \\ - \end{bmatrix}$	$i\in[1,M]$ and $j\in[0,N]$
$(i,j-1)_I \rightarrow (i,j)_I$	$-\beta$	$\begin{bmatrix} - \\ b_j \end{bmatrix}$	$i\in[0,M]$ and $j\in[1,N]$
$(i,j-1)_S \rightarrow (i,j)_I$	$-(\alpha+\beta)$	$\begin{bmatrix} - \\ b_j \end{bmatrix}$	$i\in[0,M]$ and $j\in[1,N]$
$(i-1,j-1)_S \rightarrow (i,j)_S$	$\sigma(a_i,b_j)$	$\begin{bmatrix} a_i \\ b_j \end{bmatrix}$	$i\in[1,M]$ and $j\in[1,N]$
$(i,j)_D \rightarrow (i,j)_S$	0	none	$i\in[0,M]$ and $j\in[0,N]$
$(i,j)_I \rightarrow (i,j)_S$	0	none	$i\in[0,M]$ and $j\in[0,N]$

Let s denote $(0,0)_S$ and t denote $(M,N)_S$. A path is *normal* if and only if it does not contain subpaths of the form $(i-1,j)_D \rightarrow (i-1,j)_S \rightarrow (i,j)_D$ or $(i,j-1)_I \rightarrow (i,j-1)_S \rightarrow (i,j)_I$. It can be shown that alignments of A and B are in one-to-one correspondence with normal s-t paths [15]. Furthermore, define the score of an s-t path P, denoted as $Score(P)$, to be the sum over the weights of its edges. $Score(P)$ is the score of the alignment corresponding to P.

Suppose we are given a threshold score Δ that does not exceed the optimum score. A Δ-suboptimal path (or Δ-path) is an s-t path with score at least as large as Δ. A Δ-suboptimal grid point (or Δ-point) is a grid point where at least one of its nodes appears in some Δ-path. Obviously, both $(0,0)$ and (M,N) are Δ-points. A Δ-suboptimal edge (or Δ-edge) is an edge that appears in some Δ-path.

Our goal is to compute a directed acyclic graph, denoted by $DAG_\Delta = (V_\Delta, E_\Delta)$, where V_Δ is the set of nodes in all Δ-points and E_Δ is the set of all Δ-edges. In the following, we will show how to construct V_Δ. Once V_Δ is constructed, it is straightforward to construct E_Δ.

Let $Score^{-}(i, j)_X$ be the maximum score of any path from s to $(i, j)_X$, where $X \in \{D, I, S\}$. With proper initializations, these scores can be computed by the following recurrence relations [10, 14]:

$$Score^{-}(i, j)_D = \max\{Score^{-}(i-1, j)_D - \beta,\ Score^{-}(i-1, j)_S - \alpha - \beta\}$$

$$Score^{-}(i, j)_I = \max\{Score^{-}(i, j-1)_I - \beta,\ Score^{-}(i, j-1)_S - \alpha - \beta\}$$

$$Score^{-}(i, j)_S = \max\{Score^{-}(i-1, j-1)_S + sigma(a_i, b_j),\ Score^{-}(i, j)_D,\ Score^{-}(i, j)_I\}$$

Since the scores in row i depend only those in row $i-1$, these scores can be computed in linear space [14].

Similarly, let $Score^{+}(i, j)_X$ be the maximum score of any path from $(i, j)_X$ to t, where $X \in \{D, I, S\}$. With proper initializations, these scores can be computed by the following recurrence relation:

$$Score^{+}(i, j)_S = \max\{Score^{+}(i+1, j+1)_S + \sigma(a_{i+1}, b_{j+1}),\ Score^{+}(i+1, j)_D - \alpha - \beta,$$
$$Score^{+}(i, j+1)_I - \alpha - \beta\}$$

$$Score^{+}(i, j)_D = \max\{Score^{+}(i+1, j)_D - \beta,\ Score^{+}(i, j)_S\}$$

$$Score^{+}(i, j)_I = \max\{Score^{+}(i, j+1)_I - \beta,\ Score^{+}(i, j)_S\}$$

Define $Score(i, j) = \max\ \{Score^{-}(i, j)_X + Score^{+}(i, j)_X \mid X \in \{D, I, S\}\}$.

Lemma 1. A grid point (i, j) is a Δ-point if and only if $Score(i, j) \geq \Delta$.

Proof. By definition, $Score(i, j)$ is the best score of all s-t paths using some node at grid point (i, j). If $Score(i, j) \geq \Delta$, there exists an s-t path, using some node at grid point (i, j), with score at least as large as Δ. It follows that (i, j) is a Δ-point.

On the other hand, if (i, j) is a Δ-point, at least one of its nodes appears in some Δ-path. $Score(i, j)$ is at least as large as Δ. $\square$

Lemma 1 immediately suggests that if all $Score^{-}$ and $Score^{+}$ are stored in $O(MN)$ space, we can compute (V_Δ) in $O(MN)$ time. Another alternative is to compute the dynamic-programming matrix back and forth, which would require $O(M^2N)$ time.

However, none of these two approaches are applicable for comparing long sequences.

Let $[T, B] \times [L, R]$ denote the rectangle whose upper left corner is (T, L) and lower right corner is (B, R). We say that $[T, B] \times [L, R]$ contains (i, j) (or (i, j) is in $[T, B] \times [L, R]$) if $T \le i \le B$ and $L \le j \le R$.

Given a rectangle, denoted by Π, let π be the set of Δ-points on Π's boundaries. If π is not empty, let π_{i_1} and π_{i_2} be the minimum and maximum index, respectively, of the rows containing some of π's elements, and let π_{j_1} and π_{j_2} be the minimum and maximum index, respectively, of the columns containing some of π's elements.

Lemma 2. If π is empty, there is no Δ-point in Π. Otherwise, $[\pi_{i_1}, \pi_{i_2}] \times [\pi_{j_1}, \pi_{j_2}]$ contains all Δ-points in Π.

Proof. Suppose there are some Δ-points in Π, and π is empty. Take any such Δ-point. We can always trace back from that Δ-point to a boundary Δ-point. A contradiction with the assumption that π is empty.

If π is not empty, we claim $[\pi_{i_1}, \pi_{i_2}] \times [\pi_{j_1}, \pi_{j_2}]$ contains all Δ-points in Π. Indeed, suppose there exists a Δ-point in Π with a row index smaller than π_{i_1}. We can trace back from that Δ-point to a boundary Δ-point with a row index smaller than π_{i_1}, contradicting the assumption that π_{i_1} is the minimum index of the rows that contain some of π's points. Similar arguments apply to π_{i_2}, π_{j_1} and π_{j_2}. It follows that $[\pi_{i_1}, \pi_{i_2}] \times [\pi_{j_1}, \pi_{j_2}]$ contains all Δ-points in Π. $\square$

In fact, it can be shown that $[\pi_{i_1}, \pi_{i_2}] \times [\pi_{j_1}, \pi_{j_2}]$ is the smallest rectangle that contains all Δ-points in Π.

The algorithm for computing all Δ-points is outlined as follows. For each conducted subproblem, the invariant is that $Score^-$ are given for every grid point on the left and upper boundaries, and $Score^+$ are given for every grid point on the right and lower boundaries. With these scores, the $Score^-$ and $Score^+$ for grid points within the subproblem can be computed. Problems with one or two rows or columns, can be solved directly. In general, a larger subproblem is then divided into four non-overlapping subproblems by the middle row and middle column.

To do so, a linear-space forward pass is performed to compute $Score^{-}$. To maintain the invariant, $Score^{-}$ are stored in every grid point on the two middle rows and two middle columns. To decide a more accurate range of each subproblem, *Score* for each grid point on the right and lower boundaries is also determined and stored.

Similarly, a linear-space backward pass is performed to compute $Score^{+}$. To maintain the invariant, $Score^{+}$ are stored in every grid point on the two middle rows and two middle columns. *Score* for each grid point on the left and upper boundaries is also determined and stored.

At this point, the *Score* for each grid point on the boundaries of the four subrectangles, divided by the middle row and middle column, can be determined in constant time. Take one subrectangle for example, we determine the minimum and maximum indices of the rows and the minimum and maximum indices of the columns that contain at least one Δ-point on the subrectangle's boundaries. Lemma 2 says that the rectangle bounded by these rows and columns contains all Δ-points in the subrectangle. It is therefore enough to consider only the "shrunken" subrectangle. Figure 1 illustrates the approach.

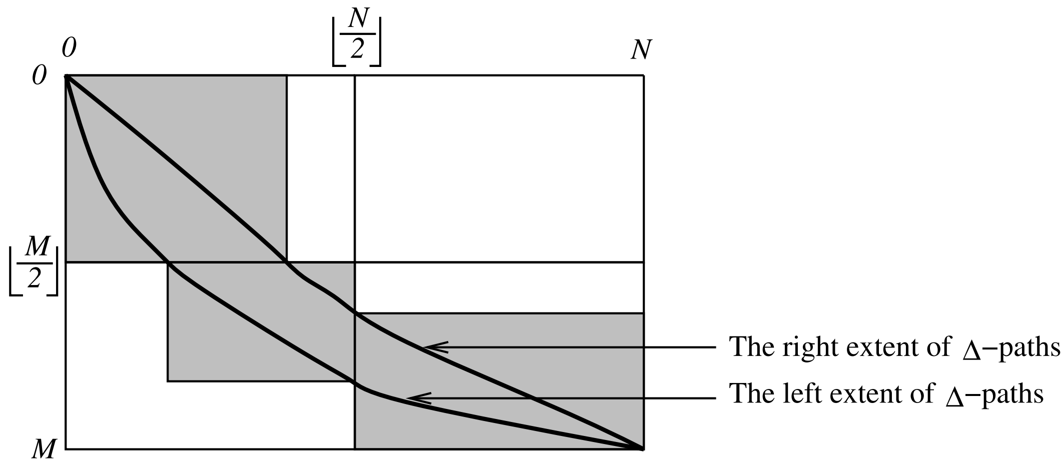

Figure 1. Splitting the problem into subproblems (shaded areas).

Figure 2 gives the pseudo code for constructing V_Δ in linear space. Let *Sub*[i] be a linked list to store all Δ-points in row i for $0 \le i \le M$. Initially, they are set to be empty. Each time when a Δ-point is found, the function *append* is called to add the point to its *Sub* list. We assume that $Score^-$ and $Score^+$ are stored in each Δ-point.

procedure $SUB\_OPT(M, N)$
{ Compute $Score^{-}$ for row 0 and column 0
Compute $Score^{+}$ for row M and column N
for $i \leftarrow 0$ **to** M **do** $Sub[i] \leftarrow \phi$
$sub\_opt(0, 0, M, N,$ initial boundary score vectors$)$
}

recursive procedure $sub\_opt(I_1, J_1, I_2, J_2,$ boundary score vectors$)$
/* *Compute all* Δ*-points in* $[I_1, I_2] \times [J_1, J_2]$ */
{ **if** $I_1 + 1 \geq I_2$ **or** $J_1 + 1 \geq J_2$ **then**
{ Compute and store $Score(i, j)$ for each (i, j) in $[I_1, I_2] \times [J_1, J_2]$.
for $i \leftarrow I_1$ **to** I_2 **do**
for $j \leftarrow J_1$ **to** J_2 **do** { **if** $Score(i, j) \geq \Delta$ **then** $append(Sub[i], POINT(i, j))$ }
return
}
$midI \leftarrow \lfloor (I_1 + I_2)/2 \rfloor$
$midJ \leftarrow \lfloor (J_1 + J_2)/2 \rfloor$
A linear-space forward computation is performed to compute $Score^{-}$:
store $Score^{-}(i, j)$ if $i = midI$ or $midI + 1$, or $j = midJ$ or $midJ + 1$;
store $Score(i, j)$ if $i = I_2$ or $j = J_2$.
A linear-space backward computation is performed to compute $Score^{+}$:
store $Score^{+}(i, j)$ if $i = midI$ or $midI + 1$, or $j = midJ$ or $midJ + 1$;
store $Score(i, j)$ if $i = I_1$ or $j = J_1$.
/* Divide the problem by row $midI$ and column $midJ$ */
$\Pi_1 \leftarrow$ the set of the grid points on the boundaries of $[I_1, midI] \times [J_1, midJ]$
$\Pi_2 \leftarrow$ the set of the grid points on the boundaries of $[I_1, midI] \times [midJ + 1, J_2]$
$\Pi_3 \leftarrow$ the set of the grid points on the boundaries of $[midI + 1, I_2] \times [J_1, midJ]$
$\Pi_4 \leftarrow$ the set of the grid points on the boundaries of $[midI + 1, I_2] \times [midJ + 1, J_2]$
for $k \leftarrow 1$ **to** 4 **do**
{ $\pi \leftarrow \{(i, j) \mid Score(i, j) \geq \Delta, (i, j) \in \Pi_k\}$
if $\pi \neq \phi$ **then**
{ $i_1 \leftarrow \min\{i \mid (i, j) \in \pi\}$
$j_1 \leftarrow \min\{j \mid (i, j) \in \pi\}$
$i_2 \leftarrow \max\{i \mid (i, j) \in \pi\}$
$j_2 \leftarrow \max\{j \mid (i, j) \in \pi\}$
Compute $Score^{-}$ for row i_1 and column j_1
Compute $Score^{+}$ for row i_2 and column j_2
$sub\_opt(i_1, j_1, i_2, j_2,$ new boundary score vectors$)$;
}
}
}

Figure 2. The algorithm for constructing V_Δ in linear space.

Space requirement

Theorem 3. The space for the boundary score vectors of all pending subproblems is $O(M+N)$.

Proof. Let $S(m,n)$ denote the worst-case space requirement for the boundary score vectors of all pending subproblems when applying *sub_opt* to a subproblem with m rows and n columns. Since each of its four possible subproblems is solved independently,

$$S(m,n) \le \begin{cases} c(m+n) & \text{for } m \le 2 \text{ or } n \le 2 \\ S(\lceil m/2 \rceil, \lceil n/2 \rceil) + c(m+n) & \text{for } m > 2 \text{ and } n > 2 \end{cases}$$

where c is a constant. It follows $S(M,N) = O(M+N)$. $\square$

Since $|V_\Delta|$ is $\Omega(\max\{M,N\})$, the space for the boundary score vectors and computed Δ-points is $O(|V_\Delta|)$. To see that this 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. The number of rows (and columns) of the problem at a recursive call to *sub_opt* is at most half that of the containing problem (rounded up), so the maximum stack depth is $O(\min\{\log M, \log N\})$.

Time analysis

For each row i, define $L[i]$ and $R[i]$ to be the minimum and maximum index, respectively, of the columns where a Δ-path intersects row i. The band width of row i, $R[i]-L[i]+1$, is denoted by $W_{row}[i]$. $W_{col}[j]$ is defined in a similar way. W is defined to be $\min\{\max\{W_{row}[i]\}, \max\{W_{col}[j]\}\}$. Let F denote the area of the region of the dynamic-programming matrix bounded by Δ-paths, i.e. $F = \sum_{i=0}^{M} W_{row}[i]$.

Lemma 4. If $R[i] \le midJ$ in the current subproblem, $(i, midJ+1), \cdots, (i, J_2)$ will not be included in any subsequent subproblem. Similarly, if $L[i] > midJ$ in the current subproblem, $(i, J_1), \cdots, (i, midJ)$ will not be included in any subsequent subproblem.

Proof. Since the right extent of Δ-paths is monotonically increasing, it is easy to see that if $R[i] \le midJ$, $[I_1, i] \times [midJ+1, J_2]$ does not contain any Δ-points. Either $[I_1, I_2] \times [midJ+1, J_2]$ does not contain any Δ-points, or the minimum index of the rows that contain some Δ-points in $[I_1, I_2] \times [midJ+1, J_2]$ is larger than i. In either case, $(i, midJ+1)$, $\cdots$, (i, J_2) will not be included in any subsequent subproblem. The case when $L[i] > midJ$ can be proved in a similar way. $\square$

Theorem 5. Let T be the total number of grid points in all the calls to *sub_opt*. $T = O(MN + F \log W)$.

Proof. Let subproblems with no more than two rows or two columns be trivial subproblems. Since each grid point can be included in at most one trivial subproblem, $O(MN)$ grid points are included in such subproblems.

Fix a row i, consider all nontrivial subproblems that include some row i's grid-points. Before reaching the first subproblem with the property $J_1 \le L[i] \le midJ \le R[i] \le J_2$, all its containing subproblems include in total $O(N)$ row i's grid points. This is because all its containing subproblems is either with the property $J_1 \le L[i] \le R[i] < midJ \le J_2$ or $J_1 \le midJ < L[i] \le R[i] \le J_2$ which will truncate half of row i's grid points in the subsequent call (Lemma 4).

The subproblem is further split into at most one subproblem with the property $J_1 \le L[i] \le J_2 \le R[i]$, and at most one with the property $L[i] < J_1 \le R[i] \le J_2$. Now we show that each of them will include $O(N + W_{row}[i] \log W_{row}[i])$ row i's grid points in its subsequent calls. Indeed, consider the subproblem with $J_1 \le L[i] \le J_2 \le R[i]$. If $midJ \ge L[i]$, it is easy to see that all its subsequent subproblems include in total $O(W_{row}[i] \log W_{row}[i])$ row i's grid points. If $midJ < L[i]$, (i, J_1), $\cdots$, $(i, midJ)$ will be truncated (Lemma 4). Before reaching the subproblem with $midJ \ge L[i]$, those containing subproblems include in total $O(N)$ row i's grid points. Similar arguments apply to the case when $L[i] < J_1 \le R[i] \le J_2$.

It follows that all subproblems include $O(N + W_{row}[i] \log W_{row}[i])$ row i's grid points. Therefore, we have

$$T = O(MN + \sum_{i=0}^{M} W_{row}[i] \log W_{row}[i]) = O(MN + F \log \max \{W_{row}[i]\})$$

In a similar way, we can derive $T = O(MN + F \log \max\{W_{col}[j]\})$. It follows $T = O(MN + F \log W)$.

$\square$

Since $F \leq MN$ and $W \leq \min \{M, N\}$, $T = O(MN \log \min \{M, N\})$. This remains even when DAG_Δ is sparse because the width of DAG_Δ could be independent of its density. On the other hand, Theorem 5 implies that if $F = O(MN / \log W)$, $T = O(MN)$.

To complete the construction of DAG_Δ, we need to build E_Δ. Let e be an edge from node u to node v. Define $Score(e)$ to be $Score^{-}(u)$ + $weight(e)$ + $Score^{+}(v)$. It can be shown that e is a Δ-edge if and only if $Score(e) \geq \Delta$. Obviously, if e is a Δ-edge, both u and v are at some Δ-point. Constructing all Δ-edges from the *Sub* lists takes $O(|V_\Delta|)$ time.

It should be noted that not every s-t path in DAG_Δ has score at least Δ. However, methods of Waterman and Byers [20] or Naor and Brutlag [16] can be applied to DAG_Δ to generate Δ-paths efficiently.

As defined by Naor and Brutlag [16], an s-t path P is called canonical if there exists an edge e in P such that $Score(e)$ = $Score(P)$. They further showed that canonical Δ-paths can represent all Δ-paths and their number is far less than the number of all Δ-paths. It can be shown that their theorems for canonical paths also hold for DAG_Δ. In the next section, we introduce a novel approach which dramatically reduces the space requirement while keeping the time complexity from growing radically.

3. An $O(MN \log \frac{1}{\Upsilon})$-Time, $O(M^{\Upsilon}(M + N))$-Space Algorithm for Computing V_Δ

For each conducted subproblem, the invariant is that $Score^{-}$ are given for every grid point on the left and upper boundaries, and $Score^{+}$ are given for every grid point on the right and lower boundaries. Instead of partitioning a subproblem into four subproblems, we partition it into a different number of subproblems, depending on the recursion depth of a subproblem. Let $L(i)$ and $W(i)$ be the number of rows and columns of a subproblem in recursion depth i, respectively. In general, an $L(i) \times W(i)$ subproblem at recursion depth i is divided into $T^2(i)$ non-overlapping $\frac{L(i)}{T(i)} \times \frac{W(i)}{T(i)}$ subproblems, where $T(i)$ is determined by the following recurrence relation.

$$T(i) = \begin{cases} M^{\Upsilon} & \text{for } i = 0 \\ cT^2(i-1) & \text{for } i > 0 \end{cases}$$

where $c > 0$ is a constant, $0 < \Upsilon < 1$, and $cM^{\Upsilon} > 1$.

Lemma 6. $T(i) = \frac{1}{c}(cM^{\Upsilon})^{2^i}$, for $i \geq 0$.

Proof. (Induction on i.) For $i = 0$, we have $T(0) = \frac{1}{c}(cM^{\Upsilon}) = M^{\Upsilon}$.

Now for $i > 1$, we can use the recurrence relation and the induction hypothesis to get $T(i) = cT^2(i-1) = c(\frac{1}{c}(cM^{\Upsilon})^{2^{i-1}})^2 = \frac{1}{c}(cM^{\Upsilon})^{2^i}$. $\square$

$L(i)$ and $W(i)$ are computed as follows.

$$L(i) = \begin{cases} M+1 & \text{for } i = 0 \\ L(i-1)/T(i-1) & \text{for } i > 0 \end{cases}$$

$$W(i) = \begin{cases} N+1 & \text{for } i = 0 \\ W(i-1)/T(i-1) & \text{for } i > 0 \end{cases}$$

With an analogous proof to Lemma 6, one can show that $L(i) = \frac{c^i}{(cM^{\Upsilon})^{2^i - 1}}(M+1)$ and $W(i) = \frac{c^i}{(cM^{\Upsilon})^{2^i - 1}}(N+1)$, for $i \geq 0$. Theorem 7 gives the upper bound of working space.

Theorem 7. The space for the boundary score vectors of all pending subproblems when the recursion depth is k, denoted as $S(k)$, is $O(M^{\Upsilon}(M+N)\sum_{i=0}^{k} c^i)$.

Proof. Let $s(n+m)$ be the space required to store boundary score vectors for an $n \times m$ subproblem, where s is a constant.

$$S(k) = \sum_{i=0}^{k} s(L(i) + W(i))T(i)$$

$$= s\sum_{i=0}^{k}\left(\frac{c^i}{(cM^{\Upsilon})^{2^i-1}}(M+1) + \frac{c^i}{(cM^{\Upsilon})^{2^i-1}}(N+1)\right)\frac{1}{c}(cM^{\Upsilon})^{2^i}$$

$$= s\sum_{i=0}^{k} c^i M^{\Upsilon}(M+N+2)$$

$$= O(M^{\Upsilon}(M+N)\sum_{i=0}^{k} c^i) \quad \square$$

Corollary 8. If $c < 1$, $S(k) = O(M^{\Upsilon}(M+N))$. Furthermore, if $c < 1$ and $\Upsilon = \Theta(\frac{1}{\log M})$, $S(k) = O(M+N)$.

Proof. It follows from the fact that $\sum_{i=0}^{k} c^i = O(1)$ for $c < 1$, and $M^{\Theta(\frac{1}{\log M})} = O(1)$. $\square$

Table 2 illustrates the growth rate of $T(i)$, $L(i)$ and $W(i)$.

Table 2. The growth rate of $T(i)$, $L(i)$ and $W(i)$.

Recursion depth i	0	1	2	3	4	...
$T(i)$	M^{Υ}	$cM^{2\Upsilon}$	$c^3M^{4\Upsilon}$	$c^7M^{8\Upsilon}$	$c^{15}M^{16\Upsilon}$	...
$L(i)/(M+1)$	1	$M^{-\Upsilon}$	$c^{-1}M^{-3\Upsilon}$	$c^{-4}M^{-7\Upsilon}$	$c^{-11}M^{-15\Upsilon}$	...
$W(i)/(N+1)$	1	$M^{-\Upsilon}$	$c^{-1}M^{-3\Upsilon}$	$c^{-4}M^{-7\Upsilon}$	$c^{-11}M^{-15\Upsilon}$	...
$\dfrac{T(i)\times(W(i)+L(i))}{M+N+2}$	M^{Υ}	cM^{Υ}	c^2M^{Υ}	c^3M^{Υ}	c^4M^{Υ}	...

The following theorems give the maximum recursion depth, which will in turn determine the time complexity of the algorithm.

Theorem 9. $L(i)\le 1$, for $i\ge \max\,\{3, 1+\log\dfrac{\log(M+1)}{\log cM^{\Upsilon}}\}$.

Proof. By definition, $L(i)=\dfrac{c^i}{(cM^{\Upsilon})^{2^i-1}}(M+1)$. It is easy to see that $L(i)$ is monotonically decreasing. Since $i\ge 3$, $2^i-i-1\ge 2^{i-1}$. If i satisfies $(cM^{\Upsilon})^{2^{i-1}}\ge M+1$, $L(i)\le 1$. This yields the bound $1+\log\dfrac{\log(M+1)}{\log cM^{\Upsilon}}$. $\square$

Corollary 10. $L(i)$ decreases to 1 in $O(\log\dfrac{1}{\Upsilon})$ steps.

Proof. It follows by observing that $\log\dfrac{\log(M+1)}{\log cM^{\Upsilon}}=\log\dfrac{\log(M+1)}{\log c+\Upsilon\log M}\le\log\dfrac{\log(M+1)}{\Upsilon(\log c+\log M)}\le\log\dfrac{1}{\Upsilon}+C$, where C is some constant. $\square$

Theorem 11. If $c<1$, the algorithm runs in $O(MN\log\dfrac{1}{\Upsilon})$ time and $O(M^{\Upsilon}(M+N))$ space.

Proof. Since subproblems at the same recursion depth are non-overlapping, the amortized time for each recursion depth is $O(MN)$. It follows that the worst-case running time is (MN) times the maximum recursion depth, which $O(MN \log \frac{1}{\Upsilon})$. If $c < 1$, Corollary 8 shows that the space for the boundary score vectors of all pending subproblems is $O(M^{\Upsilon}(M+N))$. This dominates the algorithm's working space requirement since the maximum size of the procedure activation stack is $O(\log \frac{1}{\Upsilon})$. $\square$

Theorem 12. If Υ is fixed to a small constant, say ε, the algorithm runs in $O(MN)$ time and $O(M^{\varepsilon}(M+N))$ space. If $c < 1$ and $\Upsilon = \Theta(\frac{1}{\log M})$, the algorithm runs in $O(MN \log\log M)$ time and $O(M+N)$ space.

Proof. If Υ is fixed to a small constant, say ε, Corollary 10 implies that the maximum recursion depth is $O(\log \frac{1}{\varepsilon})$, which is $O(1)$. Therefore, the time complexity is $O(MN)$. By Theorem 7, the size of working space is $O(M^{\varepsilon}(M+N) \sum_{i=0}^{O(1)} c^i)$, which is $O(M^{\varepsilon}(M+N))$.

If $c < 1$ and $\Upsilon = \Theta(\frac{1}{\log M})$, Corollary 10 implies that the maximum recursion depth is $O(\log\log M)$. Therefore, the time complexity is $O(MN \log\log M)$. By Corollary 8, the space for the boundary score vectors of all pending subproblems is $O(M+N)$. $\square$

Figure 3 gives the space-efficient pseudo code for constructing V_{Δ}. Let $Sub[i]$ be a linked list to store all Δ-points in row i for $0 \le i \le M$. Initially, they are set to be empty. Each time when a Δ-point is found, the function *append* is called to add the point to its *Sub* list. We assume that $Score^{-}$ and $Score^{+}$ are stored in each Δ-point.

```
procedure NEW_SUB_OPT(M, N, T)
  { Compute Score⁻ for row 0 and column 0
    Compute Score⁺ for row M and column N
    for i ← 0 to M do Sub[i] ← φ
    new_sub_opt(0, 0, M, N, T, initial boundary score vectors)
  }

recursive procedure new_sub_opt(I1, J1, I2, J2, T, boundary score vectors)
  /* Compute all Δ-points in [I1, I2] × [J1, J2] */
  { if I1 + 1 ≥ I2 or J1 + 1 ≥ J2 then
      { Compute and store Score(i, j) for each (i, j) in [I1, I2] × [J1, J2].
        for i ← I1 to I2 do
            for j ← J1 to J2 do { if Score(i, j) ≥ Δ then append(Sub[i], POINT(i, j)) }
        return
      }
    l ← ⌈(I2 − I1 + 1)/T⌉
    w ← ⌈(J2 − J1 + 1)/T⌉
    Compute Score⁻(i, j) for all (i, j) in [I1, I2] × [J1, J2]:
        Store Score⁻(i, j) if i − I1 is an integer multiple of l or j − J1 is an integer multiple of w.
    Compute Score⁺(i, j) for all (i, j) in [I1, I2] × [J1, J2]:
        Store Score⁺(i, j) if i − I1 − 1 is an integer multiple of l or j − J1 − 1 is an integer multiple of w.
    t ← c × T²
    i1 ← I1
    while i1 ≤ I2 do
      { i2 ← min{I2, i1 + l − 1}
        j1 ← J1
        while j1 ≤ J2 do
          { j2 ← min{J2, j1 + w − 1}
            new_sub_opt(i1, j1, i2, j2, t, boundary score vectors for [i1, i2] × [j1, j2])
            j1 ← j2 + 1
          }
        i1 ← i2 + 1
      }
  }
```

Figure 3. The algorithm for constructing V_Δ.

4. Refinements.

The basic algorithm can be embellished in a number of ways. Let F be the area of the region of the dynamic-programming matrix bounded by the suboptimal alignments and W the maximum width of that region. Our first variant achieves a running time of $O(MN + F \log\log W)$ by shrinking the subproblems at each recursion step. Second, the algorithm can be refined to use only $O(M^{1+\Upsilon} + N)$ space. This is especially useful for some applications where $M \ll N$.

4.1. An $O(MN + F \log\log W)$-Time, $O(N)$-Space Algorithm.

With the additional "shrinking" operation as defined in Lemma 2, we can achieve an algorithm that runs $O(MN + F \log\log W)$ time and $O(N)$ space [5]. Since $F = O(MN)$ and $W = O(M)$, the time complexity remains $O(MN \log\log M)$. However, the conducted experiments showed that with threshold score Δ close to the optimum score, the time complexity is reduced to $O(MN)$.

4.2. An $O(MN \log \frac{1}{\Upsilon})$-Time, $O(M^{1+\Upsilon} + N)$-Space Algorithm.

For some applications where $M \ll N$, an $O(M^{\Upsilon}(M + N))$-space method may not be practical. The space complexity can be reduced by pre-partitioning the subproblem (rectangle) into $\lceil (N+1)/(M+1) \rceil$ square-like subproblems. Specifically, a $[0, M] \times [0, N]$ subproblem is divided into $[0, M] \times [0, M]$, $[0, M] \times [M+1, 2M+1]$, $[0, M] \times [2M+2, 3M+2]$, $\cdots$, $[0, M] \times [kM + k, N]$ subproblems, where $k = \lceil (N+1)/(M+1) \rceil - 1$. This preprocessing phase can be done in $O(MN)$ time and $O(N)$ space because the total number of boundary grid points is $O(N)$.

For each square-like subproblem, the algorithm developed in Section 3 computes all its Δ-points in $O(M^2 \log \frac{1}{\Upsilon})$ time and $O(M^{1+\Upsilon})$ space. Since each of them is solved independently, the space can be recycled once the subproblem is solved. Therefore, the total space requirement is $O(M^{1+\Upsilon} + N)$. The total time complexity is $\lceil (N+1)/(M+1) \rceil \times O(M^2 \log \frac{1}{\Upsilon})$, which $O(MN \log \frac{1}{\Upsilon})$.

5. Finding an s-t Path in DAG_{Δ} with the Maximum Pattern Score

This section discusses one way of utilizing DAG_{Δ}. Given is a set of patterns, where each pattern ω is given a positive score ω_{score}. The pattern score of a path P is defined as the maximum sum of the scores of non-overlapping patterns occurring in P. The goal is to find an s-t path P_Δ in DAG_Δ such that the pattern score of P_Δ is maximum among all s-t paths in DAG_Δ. Furthermore, if there are more than one s-t paths maximizing the pattern score, $Score(P_\Delta)$ is maximum among all such paths.

A pattern ω is said to *occur* at Δ-point (i, j) if $a_{i-|\omega|+1} a_{i-|\omega|+2} \cdots a_i = b_{j-|\omega|+1} b_{j-|\omega|+2} \cdots b_j = \omega$, and $(i-|\omega|, j-|\omega|)_S \rightarrow (i-|\omega|+1, j-|\omega|+1)_S \rightarrow \cdots (i, j)_S$ is a path in DAG_Δ. An occurrence edge from $(i-|\omega|, j-|\omega|)_S$ to $(i, j)_S$, denoted by $(i-|\omega|, j-|\omega|)_S \rightarrow_\omega (i, j)_S$, is augmented to DAG_Δ if ω occurs at (i, j) for some pattern ω in the given pattern set.

In order to augment DAG_Δ with all such occurrence edges, a finite state pattern matching machine, following the scheme of Aho and Corasick [1], is constructed. It is operated by three functions: a goto function g, a failure function f, and an output function *output* (see [1]). Figure 4 outlines the algorithm for constructing all occurrence edges.

```
for each Δ-point (i, j) in topological order do
    if (i − 1, j − 1)_S → (i, j)_S is not a Δ-edge then
      { state ← 0
        k ← 0
        while (i + k, j + k)_S → (i + k + 1, j + k + 1)_S is a Δ-edge
          { if a_{i+k+1} = b_{j+k+1} then
              { while g(state, a_{i+k+1}) = fail do state ← f(state)
                for each pattern ω in output(state) do
                    Construct (i + k + 1 − |ω|, j + k + 1 − |ω|)_S →_ω (i + k + 1, j + k + 1)_S
              }
            else state ← 0
            k ← k + 1
          }
      }
```

Figure 4. The algorithm for constructing all occurrence edges.

Let l be the sum of the pattern lengths. Let Num_s be the number of the patterns recognized by state s. The time for constructing an Aho-Corasick pattern matching machine is $O(l|\Sigma| + \sum_s Num_s)$. It can be shown that the total number of state transitions made by the algorithm in Figure 4 is $O(|V_\Delta|)$. If a pattern ω occurs at a Δ-point (i, j), we have to construct an occurrence edge $(i - |\omega|, j - |\omega|)_S \rightarrow_\omega (i, j)_S$. A stack can be used to backtrack the starting location of the occurrences on the same diagonal. The time for constructing all occurrence edges is $O(|V_\Delta| + Occ)$, where Occ is the number of occurrences.

Let $Pat\_Score(u)$ be the maximum pattern score of any path from u to t in DAG_Δ. The following recurrence relation computes $Pat\_Score(u)$.

$$Pat\_Score(u) = \max \{ \max \{ Pat\_Score(v) \mid u \rightarrow v \text{ is a } \Delta\text{–edge.} \},$$

$$\max \{ Pat\_Score(v) + \omega_{score} \mid u \rightarrow_\omega v \text{ is an occurrence edge.} \} \}$$

It can be computed in $O(|V_\Delta| + Occ)$ time for all nodes in DAG_Δ. A simple backtracking method with the tie-breaking rules yields an s-t path P_Δ in DAG_Δ such that $Score(P_\Delta)$ is maximum among all s-t paths in DAG_Δ with the maximum pattern score. It should be noted that $Score(P_\Delta)$ may be worse than Δ.

In particular, when the threshold score Δ is the optimum score, it is easy to see that every s-t path in DAG_Δ is an optimal path. Therefore, the algorithm presented in this section can be used to deliver an optimal alignment with the maximum pattern score. It should be noted that the problem of finding optimal alignments containing patterns has been explored before. For example, Lawerence *et al*. [13] compute the alignment score as the score of the concatenated optimal local alignments which were extended from homologies exceeding or equal to a specified minimum length.

6. An Example

We have implemented the linear-space algorithms for computing the DAG. The conducted experiments showed that with threshold score Δ close to the optimum score, $T < 2(M+1)(N+1)$. Surprisingly, in that reasonable range, it even ran faster than the quadratic-time, linear-space *left_right* program [6] that locates merely the left and right extents of Δ-paths.

To illustrate the utility of the algorithm developed in Section 5, we aligned the ε-globin gene regions of human and rabbit. Identical matching nucleotides scored 1, mismatches scored -1 and k-symbol gaps were penalized $6 + 0.2k$. Figures 5 and 6 are a portion of two different optimal alignments. Figure 7 is a multiple alignment shown in Hardison *et al*. [9]. The human sequence is given in full, and periods denote a matching nucleotide in the other species.

The alternate optimal alignment in Figure 6 reveals three interesting features that are not in the optimal alignment in Figure 5. Box 1 contains the same gap revealed in the multiple alignment in Figure 7. Box 2 contains a gap followed by a matching block instead of two matching blocks split by a gap. Incidentally, this can be used to improve the multiple alignment in Figure 7. Finally, box 3 contains a matching block GAAGAG, a candidate for the phylogenetic footprint, which is defined as at least six consecutive invariant positions [8, 18]. Phylogenetic footprints have been demonstrated to be useful as a guide to identifying nuclear protein binding sites. In fact, GAAGAG also appears in the corresponding region of galago.

```
                  |         |         |         |         |
19091:   TTTGTCAACTGTCACCACCTTTAAGGCAAATGTTAAATGTGCTTTGGCTGAAACTTTTTT   human
 5544:   .......-------...A.C..G.CC.....A.G...C-.A....A...CT...AC...-   rabbit

                  |         |         |         |         |
19151:   TCCTATTTTGAGATTTGCTCCTTTATATGAGGCTTTCTTGGAAAAGGAGAATGGGAGAGA   human
 5595:   -..AG.C..AT.C..A...G..C.C.....AT...............G........AT..   rabbit

                  |         |         |         |         |
19211:   TGGATATCATTTTGGAAGATGATGA-----------AGAGGGTAAAAAAGGGGACAAATG   human
 5654:   .....GC...C.........T.CATGGAAAAAGAAG....T.A...C.T.ATA.TGT...   rabbit
```

Figure 5. An optimal alignment.

```
                        1                                                  2
19091:   TTTGTCAACTGTCACCACCTTTAAGGCAAATGTTAAATGTGCTTTGGCTGAAACTTTTTT   human
 5544:   ..-------........A.C..G.CC.....A.G...C-.A....A...CT...AC--..   rabbit

19151:   TCCTATTTTGAGATTTGCTCCTTTATATGAGGCTTTCTTGGAAAAGGAGAATGGGAGAGA   human
 5594:   ...AG.C..AT.C..A...G..C.C.....AT...............G........AT..   rabbit

                                                 3
19211:   TGGATATCATTTTGGAAGATGAT---------GAAGAGGGTAAAAAAGGGGACAA--ATG   human
 5654:   .....GC...C.........T.CATGGAAAAA......A..G....CAT.ATA.TGT...   rabbit
```

Figure 6. An alternate optimal alignment. `TGTCACCA`, `TTTCC` and `GAAGAG` are in the given pattern set.

```
              εNEBP
19091:   TTTGTCAACTGTCACCACCTTTAAGGCAAATGTTAAATGTGCTTTGG-CTGAAACTTTTT   human
 3960:   .......T-.......T.T.CA...TC.....T.....CTCAT.....-.....G..G...   galago
 5544:   ..-------........A.C..G.CC.....A.G...CT-A....A.-..CT...AC...   rabbit
   76:                                                             ..   goat
21352:   C..A.T.T-.A.AG...TTC..TCCCT...C.G....GT.AT...A.A...G.TT.....   mouse

                                                YRC          YY1
19150:   TTCCTATTTTGAGATTTGCTCCTTTATATGAGGCTTTCTTGGAAAAGGAG-AATGGGAGA   human
 4018:   .-....CC.GAG.C..A.T..A..---.....A..--...........G.G....A....   galago
 5595:   --..AG.C..AT.C..A...G..C.C.....AT...............G.-.......AT   rabbit
   78:   C-.TAG.C.GAT.C..A...A.AC.CC...GAT.A....G.A.G......-.........   goat
21411:   .-.....AGC.G.C.AAA.....C.T.C..T.C....G........A.T.--...A....   mouse

           YY1   GATA1        Ets
19209:   GATGGATATCATTTTGGAAGATGATGAAGAGGGTAAAAAAGGGGACAAATG            human
 4072:   A...TT.G.T......T...G.....TG.G.AAAG..G.GA.T..GT.GA.            galago
 5652:   .......GC...C.........T.CATG..AAAAG..G.GA.T..A..CAT            rabbit
  136:   .......GC...CG....G.....GAGG.GAA.A.C.G.G..T..A..TG.            goat
21468:   A....GCG.T..AC...G..T.T.CATGTG.A.AGT.G..AAT..A..GA.            mouse
```

Figure 7. Multiple alignment of the 5' flank of mammalian ε-globin genes.

7. Discussion.

It has been shown that Δ-points are useful in speeding up the computation for multiple sequence alignment problem [2, 4]. The $O(MN)$ space, which is required by a straightforward method for computing all Δ-points, may be the dominant space requirement for inputs consisting of a few long sequences [11]. The space-efficient algorithms presented here can be applied in this context.

The divide-and-conquer scheme proposed in this paper is quite general and works readily for parallelization. It remains to be investigated if this approach can be used to improve the time (or space) bound of some other divide-and-conquer algorithms for comparing sequences, e.g., the $O((N+M)^2 \log(N+M))$ algorithm for computing the distance table [3].

We close this paper by mentioning a few open problems. First, could we compute V_Δ in $O(MN)$ time and $O((M+N) polylog(M))$ space? Second, could this divide-and-approach be extended to the fragment alignment problems [7]. Finally, a model remains to be designed that does some pattern matching to extract more information from the DAG.

Acknowledgements

I would like to thank Dr. Webb Miller for valuable discussions and encouragement. I would also like to thank Drs. Eugene Myers, John Kececioglu and Gary Benson for helpful comments.

References

[1] Aho, A. V. and Corasick, M. J. (1975) Efficient string matching: an aid to bibliographic search. *Comm. ACM*, **18**, 333-340.

[2] Altschul, S. F. and Lipman, D. J. (1989) Trees, stars, and multiple biological sequence alignment. *SIAM J. Appl. Math.*, **49**, 197-209.

[3] Apostolico, A., Atallah, M. J., Larmore, L. L., and McFaddin, S. (1990) Efficient parallel algorithms for string editing and related problems. *SIAM J. Comput.*, **19**, 968-988.

[4] Carrillo, H., and Lipman, D. J. (1988) The multiple sequence alignment problem in biology. *SIAM J. Appl. Math.*, **48**, 1073-1082.

[5] Chao, K.-M. (1994) Computing all suboptimal alignments in linear space. In Proceedings of the 5th Symposium on *Combinatorial Pattern Matching*, Lecture Notes in

Computer Science, **807**, 31-42.

[6] Chao, K.-M., Hardison, R. C. and Miller, W. (1993) Locating well-conserved regions within a pairwise alignment. *CABIOS*, **9**, 387-396.

[7] Chao, K.-M. and W. Miller (1995) Linear-space algorithms that build local alignments from fragments. *Algorithmica*, **13**, 106-134.

[8] Gumucio, D. L., Shelton, D. A., Bailey, W. J., Slightom, J. L., and Goodman, M. (1993) Phylogenetic footprinting reveals unexpected complexity in trans factor binding upstream from the ε-globin gene. *Proc. Natl. Acad. Sci. USA*, **90**, 6018-6022.

[9] Hardison, R. C., Chao, K.-M., Adamkiewicz, M., Price, D., Jackson, J., Zeigler, T., Stojanovic, N., and Miller, W. (1993) Positive and negative regulatory elements of the rabbit embryonic ε-globin gene revealed by an improved multiple alignment program and functional analysis. *DNA Sequence*, **4**, 163-176.

[10] Hirschberg, D. S. (1975) A linear space algorithm for computing maximal common subsequences. *Comm. ACM*, **18**, 341-343.

[11] Kececioglu, J. D. (1989) Notes on a multiple sequence alignment cost bound of Carrillo and Lipman. Manuscript.

[12] Lander, E., Langridge, R., and Saccocio, D. (1989) Mapping and interpreting biological information. *Comm. ACM*, **34**, 33-39.

[13] Lawerence, C. B., Goldman, D. A., and Hood, R. T. (1986) Optimized homology searches of the gene and protein sequence data banks. *Bull. Math. Biol.*, **48**, 569-583.

[14] Myers, E. W. and Miller, W. (1988) Optimal alignments in linear space. *CABIOS*, **4**, 11-17.

[15] Myers, E. W. and Miller, W. (1989) Approximate matching of regular expressions. *Bull. Math. Biol.*, **51**, 5-37.

[16] Naor, D. and Brutlag, D. (1993) On suboptimal alignments of biological sequences. In Proceedings of the 4th Symposium on *Combinatorial Pattern Matching*, Lecture Notes in Computer Science, **684**, 179-196.

[17] Saqi, M. and Sternberg, M. (1991) A simple method to generate non-trivial alternative alignments of protein sequences. *J. Mol. Biol.*, **219**, 727-732.

[18] Tagle, D. A., Koop, B. F., Goodman, M., Slightom, J., Hess, D. L. and Jones, R. T. (1988) Embryonic ε and γ globin genes of a prosimian primate (Galago crassicaudatus): Nucleotide and amino acid sequences, developmental regulation and phylogenetic footprints. *J. Mol. Biol.*, **203**, 7469-7480.

[19] Vingron, M. and Argos, P. (1990) Determination of reliable regions in protein sequence alignment. *Protein Engineering*, **3**, 565-569.

[20] Waterman, M., and Byers, T. (1985) A dynamic programming algorithm to find all solutions in a neighborhood of the optimum. *Math. Biosciences*, **77**, 179-185.

[21] Zuker, M. (1991) Suboptimal sequence alignment in molecular biology: alignment with error analysis. *J. Mol. Biol.*, **221**, 403-420.