Acta Cybernetica **17** (2006) 701–717.

A Fast Algorithm for the Constrained Multiple Sequence Alignment Problem*

Dan He† Abdullah N. Arslan† and Alan C. H. Ling†

Abstract

Given n strings $S_1, S_2, ..., S_n$, and a pattern string P, the *constrained multiple sequence alignment* ($CMSA$) problem is to find an optimal multiple alignment of $S_1, S_2, \ldots, S_n$ such that the alignment contains P, i.e. in the alignment matrix there exists a sequence of columns each entirely composed of symbol $P[k]$ for every k, where $P[k]$ is the kth symbol in P, $1 \leq k \leq |P|$, and in the sequence, a column containing $P[i]$ appears before the column containing $P[j]$ for all $i, j,\ i < j$. The problem is motivated from the problem of comparing multiple sequences that share a common structure, or sequence pattern. There are $O(2^n s_1 s_2 ... s_n r)$-time dynamic programming algorithms for the problem, where $s_1, s_2, ..., s_n$ and r are, respectively, the lengths of the input strings and the pattern string. Feasibility of these algorithms in practice is limited when the number of sequences is large, or the sequences are long because of the impractically long time required by these algorithms. We present a new algorithm with worst-case time complexity also $O(2^n s_1 s_2 ... s_n r)$, but the algorithm avoids redundant computations in existing dynamic programming solutions. Experiments on both randomly generated strings and real data show that this algorithm is much faster than the existing algorithms. We present an analysis that explains the speed-up obtained in our experiments by our algorithm over the naive dynamic programming algorithm for constrained multiple sequence alignment of protein sequences. The speed-up is more significant when pattern is long, or n is large. For example in the case of constrained pairwise sequence alignment (the $CMSA$ problem with $n = 2$) when the pattern is sufficiently long for strings S_1 and S_2, the asymptotic time complexity is observed to be $O(s_1 s_2)$ instead of $O(s_1 s_2 r)$. Main ideas in our algorithm can also be used in other constrained sequence alignment problems.

Keywords: constrained sequence alignment, pairwise alignment, multiple alignment, dynamic programming

*This work was supported in part by NSF Award No. CCF-0514819.

†Department of Computer Science, University of Vermont, Burlington, VT 05405, USA, E-mail: {dhe,aarslan,aling}@cs.uvm.edu

1 Introduction

Multiple sequence alignment [2] is one of the most important problems in computational biology. Detecting similarities in DNA sequences gives clues about the evolutionary relatedness of different species, and similarities in protein sequences point out similar functionality. The multiple sequence alignment problem can be defined in various ways depending on the objective function used for measuring the similarity. When *sum of pairs (SP)* scoring is used, the problem is defined as follows: Given a set of $n \geq 2$ sequences $S_1, S_2, ..., S_n$, insert gap symbols $'-'$ into these sequences to obtain equal length strings, respectively, S_1^*, S_2^*, ..., S_n^* so that the global similarity score $\sum_{1 \leq i < j \leq n} score(S_i^*, S_j^*)$ is optimized where $score(S_i^*, S_j^*)$ is the similarity between S_i^* and S_j^* computed under a given scoring scheme. When $n = 2$, namely the sequence set has only two sequences S_1, S_2, the problem is the classical pairwise sequence alignment problem for which there is an $O(s_1 s_2)$-time dynamic programming algorithm [11]. This dynamic programming solution is extended to multiple sequence alignment problem with the resulting time complexity $O(2^n s_1 s_2 ... s_n)$. However, there are many heuristic algorithms to approximate the optimal solution (e.g. Clustal W [8], T-Coffee [5]). Recent progress in multiple sequence alignment is summarized in [6].

Given strings $S_1, S_2, ..., S_n$, and pattern string P, the *constrained multiple sequence alignment* ($CMSA$) problem is to find an optimal multiple alignment of $S_1, S_2, ..., S_n$ such that the alignment contains P, i.e. in the alignment matrix there exists a sequence of columns each entirely composed of symbol $P[k]$ for every k, where $P[k]$ is the kth symbol in P, $1 \leq k \leq |P|$, and in the sequence, a column containing $P[i]$ appears before column containing $P[j]$ for all $i, j,\ i < j$. A motivation for the problem is the alignment of RNase sequences. Such sequences are all known to contain three active residues `His(H), Lyn(K), His(H)` that are essential for RNA degrading. Therefore, it is natural to expect that in an alignment of RNA sequences, each of these residues should be aligned in the same column. The $CMSA$ problem when $n = 2$ is called the *constrained pairwise sequence alignment (CPSA) problem*.

For example, for $S_1 =$ `bbaba`, $S_2 =$ `abbaa`, and $P =$ `ab`, an optimal alignment that maximizes the number of matches with the constraint is shown in Figure 1.

Solutions for $CPSA$ can also be used to solve the $CMSA$ problem. One idea is to progressively align the sequences into a multiple alignment by using a mini-

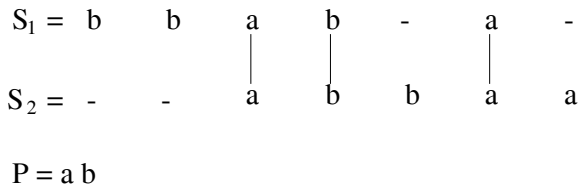

Figure 1: For $S_1 =$ `bbaba`, $S_2 =$ `abbaa`, and $P =$ `ab`, an optimal alignment which maximizes the number of matches with the constraint.

mum spanning tree obtained from a pairwise distance matrix of the sequences [7, 3]. There are many dynamic programming algorithms for the $CMSA$ and $CPSA$ problems, and their variations [7, 3, 9, 10, 1, 4]. The best known time complexity for the $CMSA$ problem is $O(2^n s_1 s_2 \dots s_n r)$ (see for example Chin et al. [3], or Tsai et al. [10]).

In this paper, we present a new dynamic programming algorithm for $CMSA$ based on the dynamic programming formulation given by Chin et al. [3], and the observation that we can use the pattern string P to avoid redundant computations in the dynamic programming matrix.

We have implemented our algorithm, and performed tests on both randomly generated data and real protein sequences. Experiments show that our algorithm is much more efficient in both time and space than a naive implementation of the algorithm presented by Chin et al. [3]. For the $CPSA$ problem the time requirement of our algorithm we observe in experiments is $O(s_1 s_2)$ when the pattern length r is large for given strings S_1 and S_2. For the $CMSA$ problem when $n > 2$, efficiency with respect to the naive algorithm we achieve with our algorithm increases significantly as the pattern length of P, or the number n of the set of sequences, $S_1, S_2, ..., S_n$ increases. The speed-up we obtain by our algorithm over the original naive dynamic programming algorithm proposed in [3] for the case of real protein sequences indicates that our algorithm is more feasible for solving the constrained multiple sequence alignment problem in practice.

The outline of this paper is as follows: in Section 2 we present our algorithm for the $CMSA$ problem. We summarize the results of our experiments in Section 3, and present mathematical analysis in Section 4 to explain the speed-up we observe in these tests using our algorithm. We include our final remarks in Section 5.

2 An Algorithm for the Constrained Multiple Sequence Alignment Problem

Our algorithm uses the dynamic programming formulation given by Chin et al. [3].

Let $D(i_1, i_2, ..., i_n, k)$ be the optimal constrained pairwise sequence alignment score of sequences $S_1[1..i_1]$,$S_2[1..i_2]$,...,$S_n[1..i_n]$ with constrained pattern sequence $P[1..r]$. Then this score can be computed by the following recurrence:

Theorem 1 ([3])**.** *For all k, $1 \le k \le r$, $D(i_1, \dots, i_n, k) = \infty$ if $i_1 = 0$ or $i_2 = 0$ or ... or $i_n = 0$. $D(\{0\}^n, 0) = 0$. For all $i_1, i_2, \dots, i_n, k$, $0 < i_1 \le s_1, 0 < i_2 \le s_2, \dots, 0 < i_n \le s_n$, $0 \le k \le r$,*

$$D(i_1, i_2, ..., i_n, k) = min \begin{cases} D(i_1 - 1, i_2 - 1, ..., i_n - 1, k - 1) \\ \quad +\delta(S_1[i_1], S_2[i_2], ..., S_n[i_n]) \\ \quad \text{if } (S_1[i_1] = S_2[i_2] = ... = S_n[i_n] = P[k]) \text{ and } k \ge 1 \\ \\ min_{e \in \{0,1\}^n} D(i_1 - e_1, i_2 - e_2, ..., i_n - e_n, k) \\ \quad +\delta(e_1 * S_1[i_1], e_2 * S_2[i_2], ..., e_n * S_n[i_n]) \end{cases}$$

where $e_j = 0$ *or* 1, $e_j * S_j[i_j]$ *with* $e_j = 0$ *represents a space character* $'-'$, *and* $S_j[i_j]$ *when* $e_j = 1$, *and* $\delta(x_1, x_2, ..., x_k) = \sum_{1 \le i < j \le n} \delta(x_i, x_j)$ *(when sum-of-pairs distance is used) where* $\delta(x_i, x_j)$ *is the given minimum distance between the symbols* x_i *and* x_j.

A naive $CMSA$ algorithm for the dynamic programming solution in Theorem 1 is shown in Algorithm 1. The algorithm returns the optimal $CMSA$ score, $D(s_1, s_2, ..., s_n, r)$, in time $O(2^n s_1 s_2 \dots s_n r)$ where $s_1, s_2, ..., s_n, r$ are the lengths of the sequences S_1, S_2, ..., S_n, and P, respectively. The reason for factor 2^n is that computing $D(i_1, i_2, ..., i_n, k)$ uses $\Theta(2^n)$ neighboring entries of $(i_1, i_2, ..., i_n, k)$ in the dynamic programming matrix. When $n = 2$, the solution in Theorem 1 is a solution for the $CPSA$ problem.

Algorithm 1 The dynamic programming algorithm for the $CMSA$ problem proposed by Chin et al. [3].

`Algorithm` $NaiveCMSA$

1. `Initialize` $D(0, 0, ..., 0) = 0, D(i_1, i_2, ..., i_n, k) = \infty$, `for all` $i_1 * i_2 * \dots * i_n = 0$, $0 \le i_1 \le s_1$, $0 \le i_2 \le s_2, ...,$ $0 \le i_n \le s_n$, $1 \le k \le r$
2. `for` $k = 0$ `to` r `do`
   `for` $i_1 = 0$ `to` s_1 `do`
   `for` $i_2 = 0$ `to` s_2 `do`
   $\vdots$
   `for` $i_n = 0$ `to` s_n `do`
   `If` $D(i_1, i_2, ..., i_n, k)$ `is not initialized, compute` $D(i_1, i_2, ..., i_n, k)$
   `according to Theorem 1`
3. `return` $D(s_1, s_2, ..., s_n, k)$

This algorithm computes the complete dynamic programming matrix parts of which are redundant in many cases. We observe that in an alignment matrix for S_1, S_2, ..., S_n, each $P[k]$ in P is required to appear in an entire column (we call such a column a *constraint-column* for $P[k]$) for the constraint to be satisfied. If $S_i[j_i]$ is aligned to $P[k]$ for the satisfaction of the constraint (i.e. if $S_i[j_i]$ appears in a constraint-column for $P[k]$ together with $S_1[j_1], S_2[j_2], \dots, S_{i-1}[j_{i-1}]$, $S_{i+1}[j_{i+1}], \dots, S_n[j_n]$) then $S_i[1..(j_i - 1)]$ can never be aligned with $S_p[(j_p + 1)..s_p]$ for all p, $1 \le p \le n$ and $p \ne i$. This means that we can save time by avoiding calculations in redundant regions in the dynamic programming matrix.

Our algorithm is based on the same dynamic programming formulation for computing $D(i_1, i_2, ..., i_n, k)$ given in Theorem 1. It is shown in Algorithm 2.

We first analyze Algorithm $FastCMSA$ for $CPSA$ computations. The analysis, and the results can be generalized for $CMSA$ computations which involve more than two sequences (i.e. $n > 2$). The dynamic programming algorithm here can be seen as computing $r + 1$ layers, one layer at each iteration k, where each layer is an n dimensional dynamic programming matrix. Figure 2 illustrates layers during the computations of $CPSA$ for a pattern whose length is 2.

Algorithm 2 Our algorithm for the $CMSA$ problem.

`Algorithm` $FastCMSA$

1. `Initialize` $D(0,0,...,0)=0, D(i_1,i_2,...,i_n,k)=\infty$, `for all` $i_1 * i_2 * i_3 * ... * i_n = 0,\ 0 \le i_1 \le s_1, 0 \le i_2 \le s_2, ..., 0 \le i_n \le s_n, 1 \le k \le r$
2. `For each` k, `find every pair of first and last possible positions that match` $P[k]$ `in each string` $S_1, S_2, \ldots, S_n$ `in a constrained alignment:`
   `for` $t = 1$ `to` n `do`
    `for` $k = 0$ `to` $r-1$ `do`
     `set` $S_{first}[t][k] =$ `the first position` f `in` S_t
       `such that` $P[1..(k+1)]$ `is a subsequence of` $S_t[1..f]$
     `set` $S_{last}[t][k] =$ `the last position` l `in` S_t
       `such that` $P[(k+1)..r]$ `is a subsequence of` $S_t[l..s_t]$
3. `For each` k, `find a pair of start point and end point:` $(S_{1begin}[k], S_{1last}[k]), (S_{2begin}[k], S_{2last}[k]), ..., (S_{nbegin}[k], S_{nlast}[k])$
   `for k=0 to r do`
    `if`$(k == 0)\{$
     $S_{1begin}[0] = 0;$
     $S_{2begin}[0] = 0;$
     $\vdots$
     $S_{nbegin}[0] = 0;$
    `}` `else` `{`
     $S_{1begin}[k] = S_{first}[1][k-1] + 1;$
     $S_{2begin}[k] = S_{first}[2][k-1] + 1;$
     $\vdots$
     $S_{nbegin}[k] = S_{first}[n][k-1] + 1;$
    `}`
    `if` $(k == r)\{$
     $S_{1last}[k] = s_1;$
     $S_{2last}[k] = s_2;$
     $\vdots$
     $S_{nlast}[k] = s_k;$
    `}else{`
     $S_{1last}[k] = S_{last}[1][k] + 1;$
     $S_{2last}[k] = S_{last}[2][k] + 1;$
     $\vdots$
     $S_{nlast}[k] = S_{last}[n][k] + 1;$
    `}`
4. `for` $k = 0$ `to` r `do`
   `for` $i_1 = S_{1begin}[k]$ `to` $S_{1last}[k]$
    `for` $i_2 = S_{2begin}[k]$ `to` $S_{2last}[k]$
     $\vdots$
     `for` $i_n = S_{nbegin}[k]$ `to` $S_{nlast}[k]$
      `compute` $D(i_1, i_2, ..., i_n, k)$ `using the expression in Theorem 1`
5. `return` $D(s_1, s_2, ..., s_n, r)$

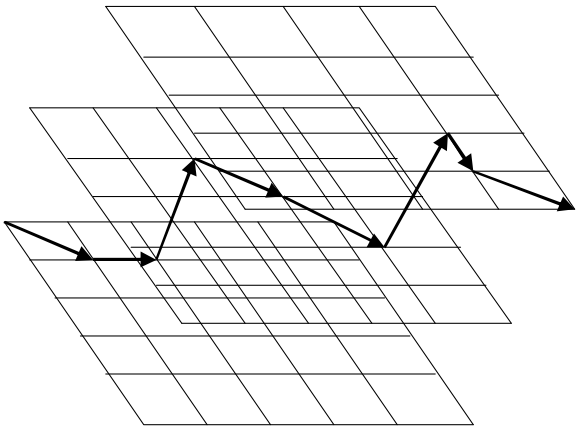

Figure 2: $CPSA$ computation for pattern string of length 2.

In the naive solution in Algorithm 1, at every iteration k (staring at $k = 0$) the whole layer is computed. On the other hand in Algorithm $FastCMSA$, when we process Layer k we compute only the subregion of the n dimensional dynamic programming matrix whose two diagonal corners, respectively, are $(S_{1begin}[k], S_{2begin}[k], ..., S_{nbegin}[k])$, $(S_{1last}[k], S_{2last}[k], ..., S_{nlast}[k])$. This is based on our observation that the area outside this region is not needed in later iterations because an optimal constrained alignment path does not pass there. For illustrative purposes, we only give an example for $CPSA$ computations in Figure 3. We only show the first two layers, and the last layer in the figure. Layers for $CMSA$ when $n > 2$ are similar, but have more dimensions. In Layer 0 we only need to compute the region whose two diagonal corners are $((S_{1begin}[0], S_{2begin}[0])$, $(S_{1last}[0], S_{2last}[0]))$. This is the only region required in the computations in the next layer, Layer 1. Similarly, at Layer 1, we only need to consider the region identified by two diagonal corners $((S_{1begin}[1], S_{2begin}[1])$, and $(S_{1last}[1], S_{2last}[1]))$. Computations in our algorithm proceed layer by layer in this manner.

Compared to the naive algorithm, our algorithm performs fewer operations on average for the points in the computed region of the dynamic programming matrix. For simplicity, we show this in the pairwise alignment case in Figure 4. On layer 0, we need to compute the rectangular region identified by its two diagonal corners $(S_1begin[0], S_2begin[0])$, $(S_1last[0], S_2last[0])$. In this region, the number of operations per point is the same in both algorithms. The differences are on Layer 1 and higher. For Layer 1, we need to compute the rectangular region of $(S_1begin[1], S_2begin[1])$, $(S_1last[1], S_2last[1])$. In the rectangular region of $(S_1begin[1]$, $S_2begin[1])$, $(\ S_1last\ [0]$, $S_2last\ [0])$ (in Figure 4 the rectangular region shaded with backward diagonal lines) the number of operations per point considered is still the same in both algorithms, but for the region elsewhere on Layer 1 (non-rectangular region shaded with forward diagonal lines in the figure), we do not need to consider the entries from the previous layer, Layer 0 in this case, since

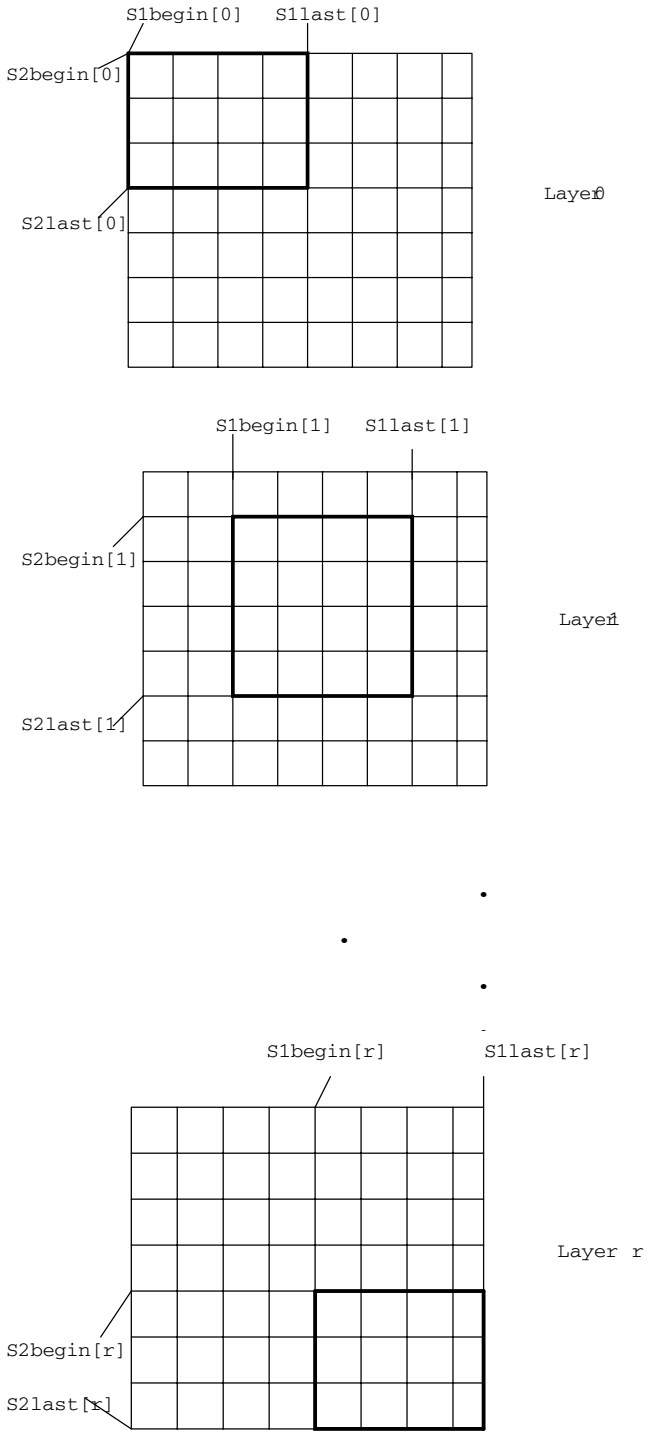

Figure 3: Regions in each layer considered in the computation of $CPSA$ with pattern string length $r > 2$.

on Layer 0, this region is not computed at all since there are no entries from last layer in this region.

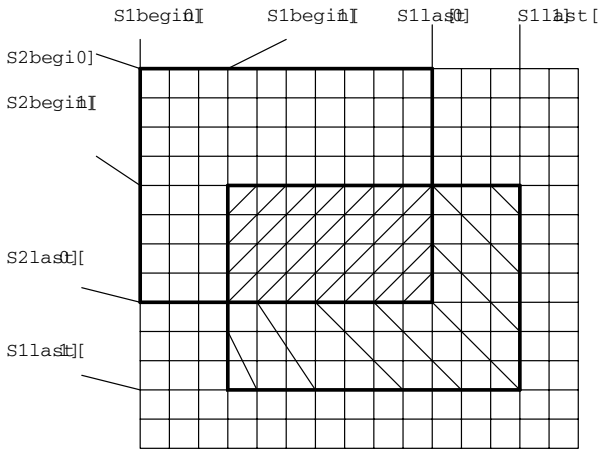

Figure 4: Illustration of the computation efficiency of our algorithm $FastCMSA$ over the naive dynamic programming algorithm.

Clearly the time complexity of our solution in Algorithm 2 is $O(s_1 s_2 r)$ for $CPSA$ computations. In our algorithm, for each layer, we only compute the region identified by $(S_{1begin}[k], S_{2begin}[k], ..., S_{nbegin}[k])$, $(S_{1last}[k], S_{2last}[k], ..., S_{nlast}[k])$. The larger the area, the longer our algorithm runs. We can create a worst case scenario as follows: For Layer 0, we try to move the last possible position which matches $P[1]$ as far as possible and the most backward position for S_i is $s_i - r$ since the length of the pattern string is r, there must be at least r symbols from this position. For the first layer, the area we need to compute is $\Omega((s_1 - r)(s_2 - r)...(s_n - r))$. For simplicity we only consider the pairwise sequence alignment case in Figure 5. For Layer 1, we try to move the first possible position which matches $P[1]$ to the beginning as much as possible, and move the last possible position which matches $P[2]$ to the end as much as possible. For similar reasons we discuss for the case of Layer 0, the smallest and largest positions, that determine the region we need to consider, in S_1, respectively, are 1 and $s_1 - r + 1$. Then we can see that the computations for Layer 1 takes $\Omega((s_1 - r)(s_2 - r))$ time. We can conclude that there is a case in which our algorithm requires $\Omega((s_1 - r)(s_2 - r)r)$ time for $CPSA$ computation. For $n > 2$ case, we can create a similar worst-case scenario for $S_1, S_2 \ldots S_n$, and P, and therefore, the worst-case computation time for $CMSA$ is $\Omega(2^n(s_1 - r)(s_2 - r)...(s_n - r)r)$. From the analysis of the worst-case scenarios, we can see that the longer the pattern string, or the higher the dimension, the better the speed-up we achieve relative to the naive $CMSA$ algorithm. We verify this by the results of our experiments.

Our discussions about the application of Algorithm $FastCMSA$ for the $CPSA$ computations can be extended to $CMSA$ computations that involve more than two

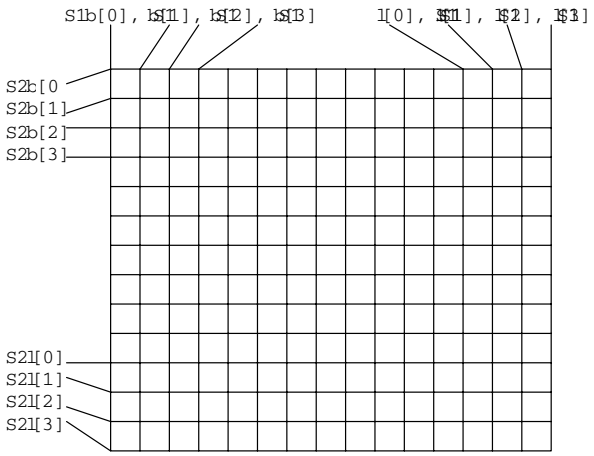

Figure 5: A worst case scenario for our algorithm $FastCMSA$ for a $CPSA$ computation with pattern string length 3. We use $S_{ib}[j]$ for $S_{ibegin}[j]$, and $S_{il}[j]$ for $S_{ilast}[j]$ to save space in the figure.

dimensions. Compared to the naive solution in Algorithm 1, our algorithm does computations for fewer points, and spends less time at each point.

3 Experiments

We first tested the performance of our algorithm $FastCMSA$ (which we call $FastCPSA$ when $n = 2$, i.e. when it is used for solving the $CPSA$ problem). We compare its performance with that of Algorithm $NaiveCMSA$ (which we call $NaiveCPSA$ when it is used for solving the $CPSA$ problem). In our tests, we randomly generate, over the alphabet of amino acids that contains 20 symbols, strings S_1 and S_2 with equal length, and pattern string P. We use 10 consecutive seeds to generate the sequences and the pattern each time, and show only the average performance. To measure time we count in the dynamic programming matrix the number of points for which the algorithms perform computations. Our algorithm is consistently faster than the naive solution in Algorithm 1. We note that when sequences S_1 and S_2 are fixed, the time requirement of our algorithm does not increase linearly with the increasing length of P. Figure 6 illustrates this. We plot pattern length *plength* versus time in the figure. In this test, we fix the sequence lengths *seqlength* as $1,000$ and increase the pattern length *plength* from 4 to 35. The time requirement of the naive algorithm linearly increases with the pattern length, and for our algorithm, it increases at slower pace first, and it starts to decrease permanently after certain level of *plength*. This is because as the *plength* increases, the matching regions in the matrix on average is confined to smaller parts in the matrix and the volume computed by our algorithm is expected to be smaller

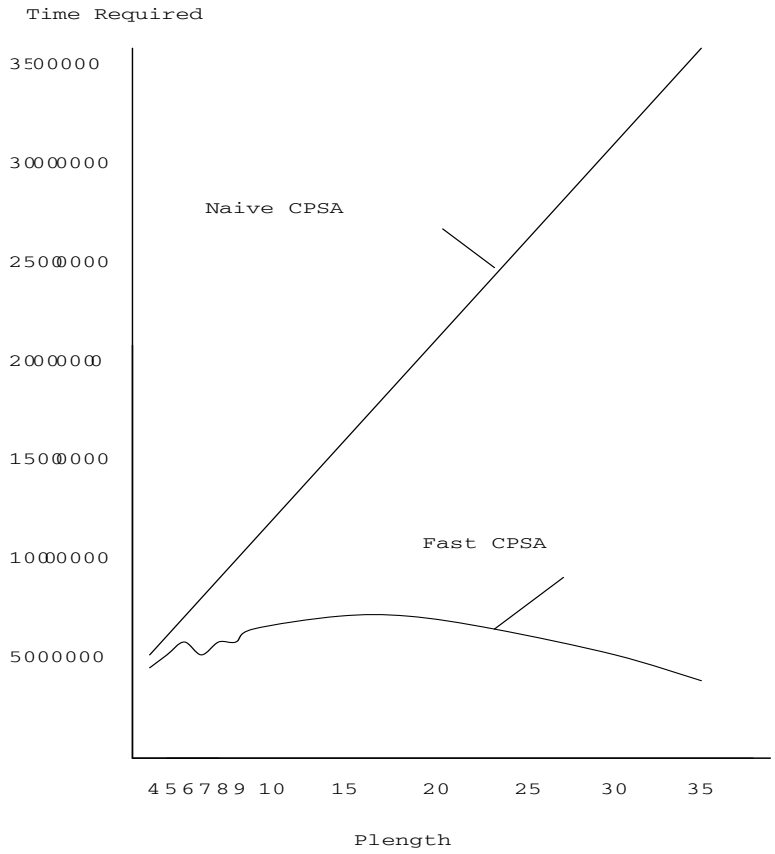

Figure 6: Time requirement of $CPSA$ computation when *seqlength* is fixed as $1,000$, and *plength* is increased from 4 to 35. For each pattern length we use 10 consecutive seeds to generate the sequences and the pattern, and show only the average performance.

in ratio on average to the size of the entire matrix. We will discuss this in more detail in Section 4.

We next tested the performance of Algorithm $FastCMSA$ on randomly generated sets of 4 protein sequences with equal length, and pattern string with length 1, 2, 3, 4 separately, over alphabet of 20 amino acid symbols. For each pattern length we use 10 consecutive seeds to generate the sequences and the pattern, and show only the average performance.

We compare the number of points in the dynamic programming matrix Algorithm $FastCMSA$ needs to compute with the number of points the naive dynamic programming algorithm computes. Table 1 shows that our algorithm is consistently faster than the naive $CMSA$ algorithm, and the performance of our algorithm over

Table 1: Average number of points the two algorithms need to compute for the alignment of 4 sequences when we fix *seqlength* as 100 and increase *plength* from 1 to 4 at increments of 1, and use 10 consecutive seeds to generate the sequences and the pattern for each pattern length, and show only the average performance.

plength	FastCMSA	NaiveCMSA	$Naive/Fast$
1	$8.09e+007$	$2.12e+008$	2.62
2	$6.30e+007$	$3.18e+008$	5.05
3	$4.77e+007$	$4.25e+008$	8.90
4	$2.10e+007$	$5.31e+008$	25.28

$seqlength = 100$

Table 2: Number of points both algorithms need to compute when we fix *seqlength* as 200, *plength* as 4 and increase the number of sequences from 3 to 6. For each case, we use 10 consecutive seeds to generate the sequences and the pattern, and show only the average performance.

dimension	FastCMSA	NaiveCMSA	$Naive/Fast$
3	$9.60e+006$	$4.06e+007$	4.22
4	$1.10e+008$	$8.16e+009$	7.42
5	$1.19e+011$	$1.64e+012$	13.78
6	$1.30e+013$	$3.30e+014$	25.38

$seqlength = 200,\ plength = 4$

the naive $CMSA$ algorithm increases quickly with the increasing pattern length. This is because the larger the *plength*, the less chances there are for the worst-case scenario. Therefore, for the same sequence set, the longer the pattern string is, the more significantly our algorithm outperforms the naive $CMSA$ algorithm.

In another set of tests, we fixed the sequence lengths *seqlength* as 200 and the pattern length *plength* as 4. Then we solved $CMSA$ problems for $n = 3, 4, 5, 6$. For each n, we also show the average performance of 10 tests by 10 consecutive seeds. We summarize the results in Table 2. We observe that the performance of Algorithm $FastCMSA$ over the naive $CMSA$ algorithm nearly doubles every time we add one more sequence (increase n by one). This is because with new sequences being involved in the alignment, a larger region in the original dynamic programming matrix is avoided.

Another advantage of our algorithm is that it first computes the possible pattern occurrence positions in each sequence, if there are no such positions then our algorithm stops immediately while $NaiveCPSA$ computes the entire dynamic programming matrix.

Table 3: Experiments on constrained alignment of 5 $RNase$ sequences with pattern string HKH and $HKSH$, separately.

pattern	FastCMSA	NaiveCMSA	$Naive/Fast$
HKH	$7.343e+009$	$2.737e+011$	37.3
$HKSH$	$5.053e+009$	$3.421e+011$	67.7

number of computation points

We have also done experiments on real protein sequences. We used the set of sequences with references given in [3](Data Set 1, and Data Set 2):

$Seq1 : gi|119124|sp|p12724|ecp\_human$,
$Seq2 : gi|2500564|sp|p70709|ecp\_rat$,
$Seq3 : gi|13400006|pdb|ldyt|$,
$Seq4 : gi|20930966|ref|xp\_142859.1$,
$Seq5 : gi|20930966|ref|xp\_142859.1$

The results of the experiments are shown in Table 3. Clearly, our algorithm is much faster than the naive $CMSA$ algorithm on RNase sequences.

4 Performance analysis of our algorithm

The performance of our algorithm depends on the total size of the layers from Layer 0 to Layer r.

We note that our algorithm does not perform computations for all the points considered by the naive algorithm implementing Theorem 1, and for the points it does it spends less time than the naive algorithm. Therefore, we compare the total volume (number of points) at which our algorithm performs computations with the total size of the $(n+1)$-dimensional dynamic programming matrix the naive algorithm uses.

Size of each layer in our algorithm is determined by the first and last matches of the given pattern P in each dimension (i.e. on each sequence). Let $b_{i,k}$ be the position of $P[k]$ in the first occurrence of $P[1..k]$ in S_i, and let $e_{i,k}$ be the position of $P[k]$ in the last occurrence of $P[k..r]$ in S_i.

We assume that pattern P occurs at least once in each sequence S_i. Otherwise, our algorithm does not do any computations in the dynamic programming matrix.

Throughout our analysis we also assume that each symbol in alphabet Σ over which sequences $S_1, S_2, \ldots, S_n$ are defined appears with equal probability in each position in these sequences.

Layer 0 is identified by two extreme points $(0, 0, \ldots, 0)$ and $(e_{1,r}, e_{2,r}, \ldots, e_{n,r})$, and its size is

$$\prod_{i=1}^{n} e_{i,1} \tag{1}$$

Each Layer k, $1 < k < r$, has two extreme points $(b_{1,k}, b_{2,k}, \ldots, b_{r,k})$ and $(e_{1,k+1}, b_{2,k+1}, \ldots, b_{r,k+1})$, and its size is

$$\sum_{k=1}^{k=r-1} \prod_{i=1}^{n} (e_{i,k+1} - b_{i,k}) \tag{2}$$

Two extreme points on Layer r are $(b_{1,1}, b_{2,1}, \ldots, b_{n,1})$ and $(s_1, s_2, \ldots, s_n)$, and the size of this layer is

$$\prod_{i=1}^{n} (s_i - b_{i,r}) \tag{3}$$

We study the expected sizes of these layers and their sum.

Lemma 2. *Suppose $P = a_1 a_2 \ldots a_r$ is a pattern of length r. Let S be a sequence of length s that contains P as a subsequence. Let Σ be the alphabet for P and S. The expected position of $P[r]$ in the first occurrence of $P[1..r]$ in S is $|\Sigma|r$.*

Proof. Let $\bar{a_i} = Q \setminus \{a_i\}$ be the set of alphabet except a_i. Then, all strings contain the first occurrence of P as a subsequence must have a unique representation of the form $A = \bar{a_1}^{\star} a_1 \bar{a_2}^{\star} \ldots \bar{a_r}^{\star} a_r$. One can see this because when we scan the sequence from left to right, we first seek for a_1, then a_2, and so on until we find a_r eventually. We next compute a generating function $f(x)$ that counts the number of strings in A. Here, we mean $f(x) = \sum_{a \in A} x^{len(a)}$ where $len(a)$ denotes the length of a. Based on the decomposition of A, we can easily deduce that $f(x) = (\frac{x}{1-(|\Sigma|-1)x})^r$ [12]. In order to compute the expected length of such sequences, we need to determine $\sum_{i=0}^{\infty} i f_i$ where f_i is the coefficient of x^i in the function $f(x)$. It is evident that the expected length is equal to $x f^{'}(x)|_{x=\frac{1}{|\Sigma|}}$. Simple calculus shows that the expected length of such strings is $|\Sigma|r$. We can also calculate the expected length when the sequence length is finite. This gives us the expected position of $P[r]$ in the first occurrence of P in S given that P occurs in S at least once. In this case, for a given sequence length s, the expected length is $\sum_{i=0}^{s} i f_i = \sum_{n=0}^{s} n \frac{\binom{n-1}{r-1}(|\Sigma|-1)^{n-r}}{|\Sigma|^n}$. We calculate expected lengths for $s = 10, \ldots, 200$ in increments of 10, and in Figure 7 we plot them versus sequence length s for varying pattern lengths $r = 1, \ldots, 5$, and for a fixed alphabet size $|\Sigma| = 20$. We see that they converge to $|\Sigma|r$ quickly (before the sequence length s approaches to 200). We note that length of a protein sequence used in constrained multiple sequence alignments is typically 150 [3, 7]. □

By using Lemma 2, and observing that the expected position of the last occurrence of pattern P is the same as the expected first occurrence of the pattern P^R where P^R means the reverse of the pattern, we can reach the following corollaries:

Corollary 3. *For a given pattern P of length r, and a string S of length s that contains P as a subsequence, the expected position of $P[1]$ in the last occurrence of $P[1..r]$ approaches quickly to $s - |\Sigma|r$ if S is sufficiently long for r and $|\Sigma|$ where Σ is the alphabet for S and P.*

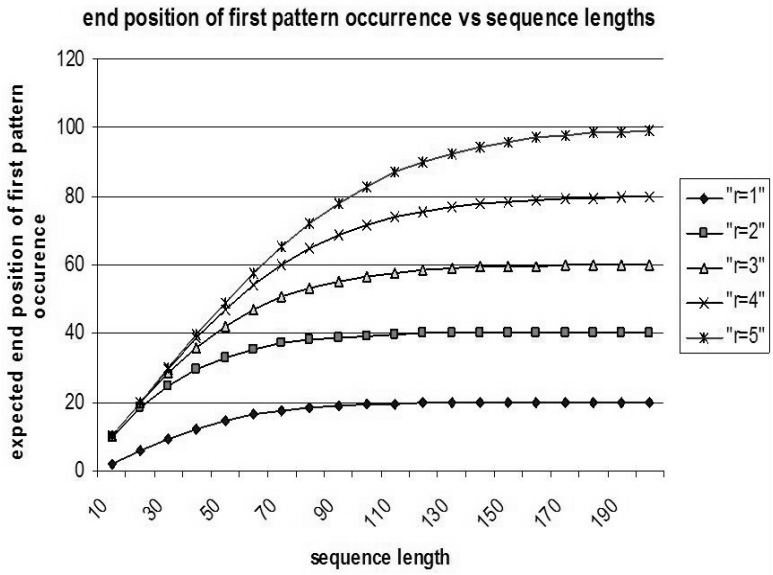

Figure 7: Expected position of $P[r]$ in the first occurrence of pattern $P[1..r]$ in string S that contains P as a subsequence versus the length s of S. Pattern length r varies from 1 to 5. The alphabet size is $|\Sigma| = 20$. The convergence is observed when s approaches to 200.

We use $x \sim V$ to denote that the value of x approaches to V.

Corollary 4. *For all i, $1 \leq i \leq n$, $E(b_{i,r}) = |\Sigma|r$, and if S_i is sufficiently long for r and $|\Sigma|$ then $E(e_{i,1}) \sim s_i - |\Sigma|r$.*

Corollary 5. *For a given pattern P of length r, and a string S of length s that contains P as a subsequence where P and S are defined over alphabet Σ, for all k, $1 < k < r$, let b_k be the position of $P[k]$ in the first occurrence of $P[1..k]$ in S, and let e_{k+1} be the position of $P[k+1]$ in the last occurrence of $P[(k+1)..r]$ in S. The expected position $E(b_k) = |\Sigma|k$, and if S is sufficiently long for r and $|\Sigma|$ then the expected position $E(e_{k+1}) \sim s - |\Sigma|(r-k)$, and therefore, the expected difference $E(e_{k+1} - b_k) = E(e_{k+1}) - E(b_k) \sim s - |\Sigma|r$.*

Corollary 6. *For all i, $1 \leq i \leq n$, and k, $1 \leq k \leq r$, if S_i is sufficiently long for r and $|\Sigma|$ then $E(e_{i,k+1} - b_{i,k}) \sim s_i - |\Sigma|r$.*

It is easy to see that $e_{i,1}$ for different S_i's are independent, and by the product rule of expectation for independent random variables, and using Equation (1) the expected size of Layer 0 is

$$E(\prod_{i=1}^{n} e_{i,1}) = \prod_{i=1}^{n} E(e_{i,1}) \tag{4}$$

If we consider $e_{i,k+1}$ and $b_{i,k}$ as random variables then $e_{i,k+1} - b_{i,k}$ are independent for different S_i's. We note that $e_{i,k+1} - b_{i,k}$ are not independent for different layer k's for the same S_i but the linearity of expectation does not require this property, and therefore, using Equation (2) the expected size of Layer k, for all $1 \leq k \leq r-1$, is

$$E(\prod_{i=1}^{n} e_{i,k+1} - b_{i,k}) = \prod_{i=1}^{n} E(e_{i,k+1} - b_{i,k}) \tag{5}$$

Since $(s_i - b_{i,r})$ are independent for different S_i's, and if we use Equation (3) we can see that the expected size of Layer r is

$$E(\prod_{i=1}^{n} (s_i - b_{i,r}) = \prod_{i=1}^{n} E(s_i - b_{i,r}) \tag{6}$$

Adding equations (4), (5), and (6), and using corollaries 4 and 6, if S_i is sufficiently long for r and $|\Sigma|$ for all i, $1 \leq i \leq n$, then the expected total volume of layers from 0 to r approaches to

$$(r+1) \prod_{i=1}^{n} (s_i - |\Sigma| r) \tag{7}$$

If we compare this volume with the total size $(r+1)\prod_{i=1}^{n} s_i$ of the dynamic programming matrix used by the naive algorithm we can see that the expected speed-up achieved by our algorithm over the naive algorithm approaches to

$$\prod_{i=1}^{n} \frac{s_i}{s_i - |\Sigma| r} \ .$$

Given a pattern of length r, and n sequences of lengths $s_1, s_2, \ldots, s_n$ over alphabet Σ where each S_i contains P as a subsequence, and S_i sufficiently long for r and $|\Sigma|$, and $s_i > |\Sigma| r$, let $C_i = \frac{s_i}{|\Sigma| r}$ for all i, $1 \leq i \leq n$, then we can see that the expected speed-up of our algorithm over the naive algorithm approaches to

$$\prod_{i=1}^{n} \frac{s_i}{s_i - |\Sigma| r} \geq \prod_{i=1}^{n} \frac{C_i}{C_i - 1} \ .$$

This expression for the speed-up explains the results we have shown in Figure 6, and tables 1, 2, and 3. The speed-up is more significant if $C_i = \frac{s_i}{|\Sigma| r} > 1$ is a small number close to 1. For example, for the $CPSA$ problem with fixed sequence lengths $s_1 = s_2 = 1000$ and with pattern length r increasing from 4 to 35, and alphabet size is 20, the speed-up accelerates with increasing r as shown in Figure 6.

The target application of this paper is the constrained multiple sequence alignment of protein sequences where the alphabet is composed of 20 amino acids, a typical protein sequence length is 150 [3, 7], and a pattern used as a constraint is typically $3-4$ character-long. In these cases all $C_i \leq 2.5$, and the expected speed-up $\sim (5/3)^n$ where n is the number of sequences compared.

5 Concluding Remarks

We present an algorithm for the constrained multiple sequence alignment problem based on the dynamic programming formulation given by Chin et al. [3]. We observe that it is redundant to compute the entire dynamic programming matrix because the alignments are constrained to include pattern string P. We can precompute a set of points that breaks the dynamic programming matrix into parts some of which are redundant for solving the problem. Although our algorithm does not improve the worst-case time-complexity of the problem, the experiments we have conducted on both syntectic data and real RNase sequences show that our algorithm is significantly faster than the original naive dynamic programming algorithm proposed by Chin et al. [3]. The speed-up we achieve is more significant when the pattern is long, and the number of sequences is large. We present mathematical analysis for the expected speed-up achieved by our algorithm. The speed-up is expected to be significant if the product of the alphabet size and the pattern length is a relatively large fraction of the sequences aligned. This is in general true in practice in constrained multiple sequence alignment of protein sequences [3, 7].

An interesting behavior of our algorithm is observed when it is applied to the constrained pairwise sequence alignment. In this case, our algorithm's observed asymptotic time complexity is quadratic instead of cubic when the pattern is sufficiently long for given sequences.

Our ideas on the $CMSA$ can also be used in the algorithms for the constrained longest common subsequence problems [1, 4], and similar speed-up can be achieved.

Other kinds of existing techniques for multiple sequence alignment, both heuristic and exact, can be combined with the main steps of our algorithm to increase the feasibility of the $CMSA$ problem in real-life applications.

References

[1] Arslan, A. N. and Eğecioğlu., Ö. Algorithms for the constrained longest common subsequence problems. *International Journal of Foundations of Computer Science*, (16)6:1099-1111, December 2005.

[2] Carrillo, H. and Lipman, D. J. The multiple sequence alignment problem in biology. *SIAM J. Appl. Math*, 48(5):1073-1082, 1988.

[3] Chin, F. Y. L., Ho, N. L., Lam, T. W., Wong, P. W. H., and Chan, M. Y. Efficient constrained multiple sequence alignment with performance guarantee. *Proc. IEEE Computational Systems Bioinformatics (CSB 2003)*, pp. 337-346, 2003.

[4] Chin, F. Y. L., Santis, A. D., Ferrara, A. L., Ho, N. L., and Kim, S. K. A simple algorithm for the constrained sequence problems. *Information Processing Letters* Vol. 90, pp. 175-179, 2004.

[5] Notredame, C., Higgins, D. G., and Heringa, J. T-Coffee: A novel algorithm for multiple sequence alignment. *J.Mol.Biol.*, 302,205-217, 2000.

[6] Notredame, C. Recent progresses in multiple sequence alignment: a survey. Ashley Publications Ltd, ISSN 1462-2416, 2001.

[7] Tang, C. Y., Lu, C. L., Chang, M. D.-T., Tsai, Y.-T., Sun, Y.-J., Chao, K.-M., Chang, J.-M., Chiou, Y.-H., Wu, C.-M., Chang, H.-T., and Chou, W.-I. Constrained multiple sequence alignment tool development and its applications to RNase family alignment. *Proceeding of the 1st IEEE Computer Society Bioinformatics Conference (CSB 2002)*, pp. 127-137, 2002.

[8] Thompson, J., Higgins, D., and Gibson, T. CLUSTAL W: Improving the sensitivity of progressive multiple sequence alignment through sequence weighting position specific gap penalties and weight matrix choice. *Nucleic Acids Res*, 22,4673-4690, 1994.

[9] Tsai, Y.-T. The constrained common sequence problem. *Information Processing Letters*, 88:173–176, 2003.

[10] Tsai, Y.-T., Lu, C. L., Yu, C. T., and Huang, Y. P. MuSiC: A tool for multiple sequence alignment with constraint. *Bioinformatics*, 20(14):2309-2311, 2004.

[11] Waterman, M. S. Introduction to computational biology. *Chapman & Hall*, 1995.

[12] Wilf, H. S. Generating functionology. *Academic Press, Inc*, 1994.