A Space-Efficient Algorithm for the Constrained Pairwise Sequence Alignment Problem

Dan He dhe@cs.uvm.edu
Abdullah N. Arslan aarslan@cs.uvm.edu

Department of Computer Science, University of Vermont, Burlington, VT 05405, USA

Abstract

The *constrained pairwise sequence alignment* ($CPSA$) problem aims to align two given sequences by aligning their similar subsequences in the same region under the guidance of a given pattern (constraint). Let the lengths of the sequences be m, and n where $n \leq m$, and let $r \leq n$ be the length of the given pattern. The optimum constrained pairwise alignment score can be computed using $O(rn)$ space by a naive dynamic programming solution. If an optimal alignment path is desired then the space requirement of the naive dynamic programming algorithm is $O(rnm)$. There is a divide-and-conquer algorithm that reduces the memory requirement of finding an optimal alignment for the $CPSA$ problem to $O(rn)$. In this paper, we present a space-efficient $CPSA$ algorithm that returns an optimal alignment. Our analysis on real protein sequences suggests that our algorithm requires only $O(n)$ space in practice. This algorithm is not only space efficient but also very fast. A generalization of the $CPSA$ problem for multiple sequences is called the constrained multiple sequence alignment ($CMSA$) problem. Our $CPSA$ algorithm also improves the space requirement of progressive $CMSA$ algorithms that use solutions of $CPSA$ problems.

Keywords: constrained sequence alignment, pairwise alignment, multiple alignment, dynamic programming, divide-and-conquer, space efficient

1 Introduction

The *constrained pairwise sequence alignment* ($CPSA$) problem [8] aims to find the optimum alignment score of two given sequences S_1 and S_2 such that the alignment contains a given pattern string P, i.e. in the alignment matrix there exists a sequence c 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 c, a column containing $P[i]$ appears before column containing $P[j]$ for all $i, j,\ i < j$. The generalization of this problem for multiple sequences is called the *constrained multiple sequence alignment* ($CMSA$) problem [8]. The motivation for these constrained sequence alignment problems are to incorporate the biologically meaningful prior knowledge of the structure or pattern of the input sequences into the alignment process. Patterns the optimal alignments are constrained to contain can lead to more biologically meaningful alignments other than alignments with mathematically high scores. For example, in all members of a certain protein family a particular cysteine residue enters into a disulfide bond, all those cysteines should be placed in the same column by an alignment of members of that family, even if some other configurations result in higher scores. Another application of 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. Constrained sequence alignment may also be used in comparing other long biological sequences which share a common pattern, and it may also

have applications outside computational biology such as text alignment problem. It is an interesting problem in computer science.

There are many dynamic programming algorithms for the $CMSA$ and $CPSA$ problems, and their variations [5, 1, 2, 3, 8, 9, 10]. In the $CPSA$ problem let m, and n be the length of the sequences aligned, and let r be the length of the pattern string P. We assume without loss of generality that $r \leq n \leq m$, and typically r is much smaller than n. The time complexity of the $CPSA$ problem is $O(rnm)$ (see for example Chin et al. [2]). In this paper we concentrate on the space complexity of the $CPSA$ problem. The space complexity in sequence alignment problems can be problematic for long sequences. Progressive $CMSA$ algorithms align the sequences into a multiple alignment by using a minimum spanning tree obtained from a constrained pairwise distance matrix of the sequences that is generated by solving $CPSA$ problems [2, 8]. Space requirement of the $CPSA$ algorithms are a bottleneck for these $CMSA$ algorithms. Therefore a space efficient solution for the $CPSA$ problem is important not only for the feasibility of constrained pairwise sequence alignment but also for the practicality of some $CMSA$ algorithms.

Space complexity of the naive $CPSA$ dynamic programming algorithm presented by Chin et al. [2] is $O(rnm)$. This can easily be reduced to $O(rn)$ by performing the calculations layer by layer where each layer is the alignment matrix for S_1 and P, and the layers are indexed by positions in S_2. In this case, we only need to store the previous layer and the current layer when we compute the current layer. Therefore we only need $O(rn)$ space. If an optimal alignment is also desired then finding an optimal alignment using the naive algorithm requires $O(rnm)$ space. Lu and Huang [7] propose a memory efficient algorithm based on the divide-and-conquer approach for linear-space pairwise sequence alignment [6]. Their algorithm can find an optimal alignment for the $CPSA$ problem in $O(rn)$ space. Arslan and Eğecioğlu [1] discuss similar space complexity results for the *constrained longest common subsequence*, and *edit-distance constrained longest common subsequence* problems which are the variations of the $CPSA$ problem.

He and Arslan [5] presented an algorithm for the $CPSA$ and $CMSA$ problems based on the dynamic programming solution devised by Chin et al. [2]. Even though the algorithm does not improve the worst-case time complexity of these problems, it avoids redundant computations in the original dynamic programming matrix. The algorithm uses the pattern string P to pre-compute the regions (which are usually much smaller than the entire matrix) at each layer (an alignment matrix for S_1 and S_2) where an optimal alignment can possibly pass. He and Arslan [5] showed that this algorithm is much faster in practice than a naive implementation of the algorithm of Chin et al. [2]. The worst-case space complexity of this algorithm for the $CPSA$ problem is $O(n^2)$ when we compute the optimum score, and $O(rn^2)$ if an optimal alignment is also desired.

In this paper, we propose a $CPSA$ algorithm that uses $O(n + \alpha)$ space where our experiments suggest that $\alpha \ll n$ when we use protein sequences. Our algorithm uses the $CPSA$ algorithm of He and Arslan [5] and the divide-and-conquer linear-space pairwise sequence alignment algorithm [6]. The algorithm considers the dynamic programming matrix for the $CPSA$ problem as a graph, and computes an optimal constrained path (an optimal alignment path that contains P) in this graph between the vertices $(0, 0, 0)$ and (m, n, r). An optimal path enters, spans, and exits layers indexed by the positions in P where each layer is a subregion of the pairwise alignment matrix of the two sequences S_1 and S_2 whose boundaries can be pre-computed at each layer. For each layer we pre-compute all possible points where an optimal constrained alignment can enter and exit. We apply the divide-and-conquer approach on the dynamic programming matrix to calculate a mid-point through which an optimal constrained alignment path passes. We calculate and store optimum scores obtained at these entry and exit points layer by layer, and within a layer in row by row manner. This requires $O(n + \alpha)$ space where α is the maximum total number of entry and exit vertices for any layer. Once a mid-point is found we recurse on the two halves to find the two halves of an optimal constrained alignment path which passes through this mid-point. The space complexity of the resulting algorithm is $O(n + \alpha)$.

The outline of this paper is as follows: In Section 2 we summarize the current algorithms for the $CPSA$ problem. In Section 3 we describe our algorithm for the $CPSA$ problem. In Section 4, we show evidence using real data that our algorithm is space efficient. We include our final remarks in Section 5.

2 Algorithms for the $CPSA$ Problem

Given two sequences S_1, S_2 with lengths $n \leq m$, respectively, the *pairwise sequence alignment* (PSA) problem is to find the optimum alignment score for these two sequences. The problem can be solved according to the following recurrence:

Theorem 1 *[11]* $D(i_1, i_2) = \infty$ *if* $i_1 = 0$, *or* $i_2 = 0$, *and* $D(0,0) = 0$, *and for all* i_1, i_2, $0 < i_1 \leq s_1, 0 < i_2 \leq s_2$,

$$D(i_1, i_2) = \min \begin{cases} D(i_1 - 1, i_2 - 1) + \delta(S_1[i_1], S_2[i_2]) \text{ if } (S_1[i_1] = S_2[i_2]) \\ D(i_1 - 1, i_2) + \delta(S_1[i_1], '-') \\ D(i_1, i_2 - 1) + +\delta('-', S_2[i_2]) \end{cases}$$

where $\delta(a, b)$ *is the score for the alignment of symbol* a *and* b.

Computing the optimum score $D(m, n)$ of the PSA problem takes $O(nm)$ time and $O(n)$ space since at one time we only need to record the current row, the preceding row, and after every iterations the current row is moved onto the preceding row. If we also desire an optimal alignment path then this can be done using $O(nm)$ space. Hirschberg [6] showed that an optimal solution for the PSA problem can be solved in only $O(n)$ space by binary-recursion. This approach is based on computing mid-point of an optimal alignment along one string whose length is m, and recursing on the two halves. An optimal alignment can be obtained by merging the series of mid-points. We show this approach in Figure 1. For sequence S_1 with length n and sequence S_2 with length m, Hirschberg's algorithm divides S_1 into two halves: $S_{11}[1...\frac{n}{2}]$ and $S_{12}[\frac{n}{2}...n]$. S_{11} is then aligned with S_2 using the linear-space algorithm, namely each time saving only two adjacent columns. The alignment proceeds from vertex A to the middle dividing line . Then S_{12} and S_2 are reversed and aligned using the same linear-space algorithm, which proceeds from vertex B to the middle dividing line. The mid-point can be determined by picking vertex C from the middle dividing line such that the sum of the alignment score from A to C and the alignment score from B to C is minimized. The original PSA problem then is reduced to two smaller PSA problems: the dynamic programming matrix from A to C, and the dynamic programming matrix from C to B. Each of these two subproblems can be solved recursively in the same manner. The recursion ends when the length of one of two sequences to be aligned is one. We can write recurrence relations for the time complexity, and space complexity. Solving these recurrences shows that the time to find an optimal alignment is $O(nm)$ (it is twice the time required to calculate the optimum score but not an optimal path), and the space used is $O(n)$.

For the $CPSA$ problem Chin et al. [2] present a dynamic programming formulation that modifies the solution of the PSA problem by considering the additional string P. We next show this solution.

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

Theorem 2 *[2]* $D(i_1, i_2, k) = \infty$ *if* $i_1 = 0$, *or* $i_2 = 0$, *for all* k, $1 \leq k \leq r$, *and* $D(\{0\}^2, 0) = 0$, *and for all* i_1, i_2, k, $0 < i_1 \leq s_1, 0 < i_2 \leq s_2$, $0 \leq k \leq r$,

$$D(i_1, i_2, k) = \min \begin{cases} D(i_1 - 1, i_2 - 1, k - 1) + \delta(S_1[i_1], S_2[i_2]) \\ \quad \text{if } (S_1[i_1] = S_2[i_2] = P[k]) \text{ and } k \geq 1 \\ \\ min_{e \in \{0,1\}^2} D(i_1 - e_1, i_2 - e_2, k) + \delta(e_1 * S_1[i_1], e_2 * S_2[i_2]) \\ \quad \text{for } i_j - e_j \geq 0 \text{ for all } j, 1 \leq j \leq 2 \end{cases}$$

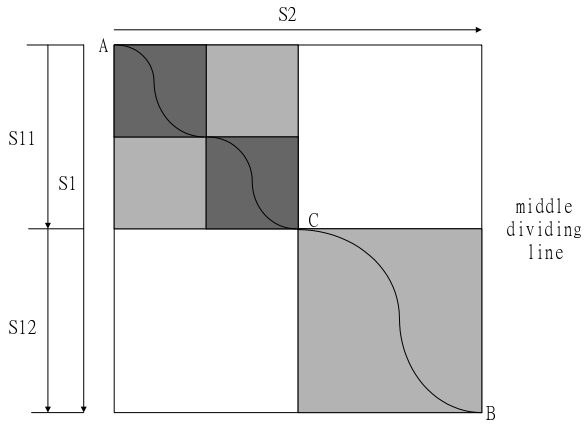

Figure 1: Hirschberg's Linear-space algorithm in layer by layer manner.

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* $e_j = 1$.

$D(m, n, r)$ is the optimum score of the $CPSA$ problem. We can visualize the dynamic programming computations in Theorem 1 for the $CPSA$ problem in $r+1$ layers, one layer at each iteration k, where each layer is a two dimensional dynamic programming matrix (or alignment matrix of S_1 and S_2). Figure 2 illustrates the layers during the computations of $CPSA$ for a pattern whose length is 2.

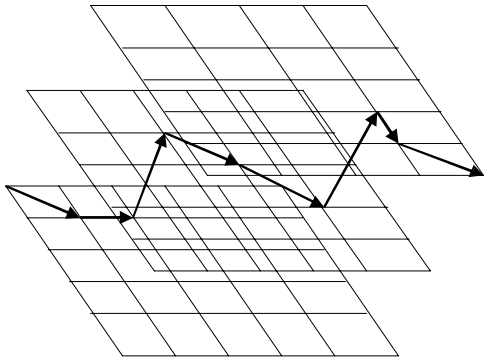

Figure 2: $CPSA$ computation for a pattern string of length 2.

The score $D(i_1, i_2, k)$ for each node (i_1, i_2, k) in the dynamic programming matrix is computed using the scores of all the neighbor nodes (nodes that have direct links to this node) on the same layer k, or on the preceding layer $k-1$ (if it exists).

He and Arslan [5] improved the naive dynamic programming algorithm of Chin et al. [2] using the observation that if the symbol $P[k]$ is aligned with a symbol of S_i then the region before this symbol $P[k]$ in S_i can never be aligned with the region after $P[k]$ in S_j. Thus redundant computations in the original dynamic programming matrix can be avoided by precomputing regions that need to be considered using the pattern string P. This saves both time and space. The same idea can be extended for the $CMSA$ problem. The speed-up achieved by this way is significant when the pattern string is long, or the number of sequences n is large. Although the algorithm does not improve the worst-case time complexity of the $CMSA$ problem (it is $O(2^n s_1 s_2 ... s_n r)$ where each s_i denotes the length of S_i), experiments show that for 5 RNA sequences, and a pattern of length 4 the algorithm is 60 times faster than a naive implementation of the dynamic programming solution in Theorem 1. An interesting result is that for the $CPSA$ problem the time requirement of the algorithm is observed to be $O(nm)$ when the pattern length r is sufficiently large with respect to n and m, and it may even decrease with increasing r.

Lu and Huang [7] proposed a memory efficient algorithm $CPSA\text{-}DC$ based on Hirschberg's divide-and-conquer approach [6]. The algorithm can solve the $CPSA$ problem in $O(rn)$ space. In the algorithm, the constrained alignment problem is divided into two smaller constrained alignment problems at a middle point. Then each of these smaller constrained alignment problems is recursively divided into two smaller subproblems in the same manner. The optimal constrained alignment is obtained by merging these series of mid-points. The dynamic programming matrix for the $CPSA$ problem can be visualized as a 3-dimensional matrix with each dimension indexed by positions of symbols in S_1, S_2 and P, respectively. This 3-dimensional matrix can be computed in a layer by layer manner, where the layers can be indexed by the positions of symbols in pattern string P or S_1 or S_2 and each layer is a 2-dimensional dynamic programming matrix. Figure 2 shows the case where layers are indexed by the positions of symbols in the pattern string P. According to Theorem 1, the optimum score of one vertex in the dynamic programming matrix can be obtained from the optimum scores of vertices on the same layer and the vertices on the preceding layer. Therefore the calculation of the optimum score at a mid-point in the divide-and-conquer approach needs to store the preceding layer, and the current layer when the current layer is processed. Since in practice $r \ll n$, the $CPSA\text{-}DC$ algorithm computes layers indexed by the positions of symbols in S_1 as shown in Figure 3. Therefore the memory requirement is $O(rn)$ (instead of $O(nm)$ space required if we compute layers indexed by the positions of symbols in pattern string).

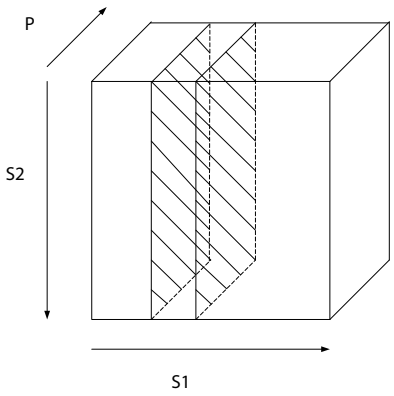

Figure 3: $CPSA\text{-}DC$ algorithm for $CPSA$ problem. The layers are indexed by the positions of symbols in S_1. Each time we need to record two adjacent layers. An example to such two layers is shown in areas with dashed lines.

3 Algorithm Layer-DC

The algorithm of Lu and Huang [7] applies the divide-and-conquer approach to the complete 3-dimensional dynamic programming matrix in a layer-by-layer manner and the algorithm needs to store two complete layers at any time. Therefore its space complexity is $O(rn)$. In our algorithm, we not only take the divide-and-conquer approach in a layer-by-layer manner, but also we apply the linear-space algorithm on each layer. With our approach at any given time we only need to store two rows on the current 2-dimensional layer using $O(n)$ space.

The dynamic programming solution for the $CPSA$ problem can be considered as finding a shortest path in the 3-dimensional dynamic programming matrix through layers. Here we index the layers with the positions of symbols in the pattern string, as shown in Figure 2, where each layer is a 2-dimensional matrix. We observe that a shortest path goes through each layer of the dynamic programming matrix beginning at layer 0 and ending at layer r, where r is the length of the pattern string. A shortest path enters each layer at a vertex (we call it an *entry vertex*), after traversing a number of vertices in

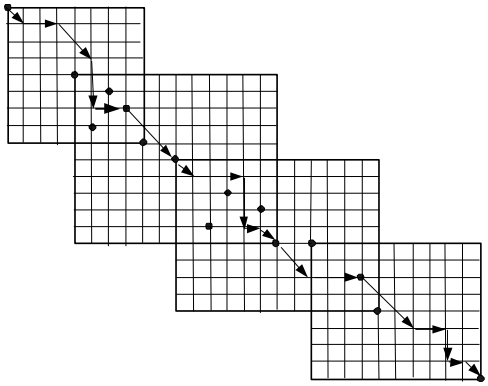

Figure 4: For the $CPSA$ problem with pattern string of length 3, a global shortest path passing through entry and exit vertices, and connecting sub-paths on each layer.

each layer exits the layer at a vertex (we call it an *exit vertex*) never to come back to this layer again as shown in Figure 4. In the dynamic programming algorithm the optimum scores for the vertices on layer k are obtained from the optimum scores of the vertices on the same layer and the vertices on the preceding layer, or more precisely the exit vertices on the preceding layer. Therefore the only information on layer k we need to store in order to compute the optimum scores on layer $k+1$ is the scores of the exit vertices on layer k.

We can precompute all of the candidate entry and exit vertices (we call *entry vertex set* and *exit vertex set*, respectively) on each layer. The entry vertices on layer k are all the vertices where the symbol is $P[k]$ for all sequences at these coordinates. These vertices are in the overlapping region of layer $k-1$ (for $k \geq 1$) and k. Similarly, the exit vertices on layer k (for $k < r$) are all the vertices where the symbol is $P[k+1]$ for all sequences at these coordinates. These vertices are in the overlapping region of layer k and $k+1$. Note that the entry vertices on layer k are also the exit vertices on layer $k-1$, and the exit vertices on layer k are also the entry vertices on layer $k+1$. Layer 0 has only one entry vertex $(0,0,0)$ and layer r has only one exit vertex (m,n,r). This is shown in Figure 4.

In our algorithm, we apply the divide-and-conquer approach through layers such that the original $CPSA$ problems can be divided into two smaller subproblems at a middle layer. While computing through layers, we use the linear-space algorithm on each layer k, and store all the exit vertices which are pre-calculated on layer k. Then we initialize the scores of the entry vertices on layer $k+1$ as those of their corresponding exit vertices on layer k so that the linear-space algorithm can be again applied to layer $k+1$. Therefore we do not need to store the complete 2-dimensional dynamic programming matrix of layer k. Let the forward computation starting at the top-left corner stop at a middle layer f and the backward computation starting at the bottom-right corner stop at layer $f+1$, we next find the mid-point T in the exit vertex set E_1 on layer f (also the entry vertex set E_2 on layer $f+1$) which minimizes the score of the global path through the whole dynamic programming matrix. After we find the mid-point T, we can divide the original problem into two smaller problems and then solve each of them recursively in the same manner. Note that for the first subproblem, the end vertex of the new smaller dynamic programming matrix is changed to T, and for the second subproblem the start vertex of the new smaller dynamic programming matrix is changed to T as shown in Figure 5. If the forward and backward computation fall on the same layer, it is just the ordinary case for Hirschberg's divide-and-conquer approach. We show our algorithm Layer-DC in Figure 6 and Figure 7.

The space complexity of our algorithm Layer-DC is $O(n+\alpha)$ where α is the maximum number of entry and exit vertices on any layer. This is because at each layer until a mid-point is reached the algorithm stores scores for $O(\alpha)$ entry and exit points, and the total area (in terms of vertices) computed is $O(n)$. Chin et al. [2] showed that the number of occurrences of constraints is very small compared with the size of the aligned sequences. Thus usually $\alpha \ll n$. Therefore our algorithm

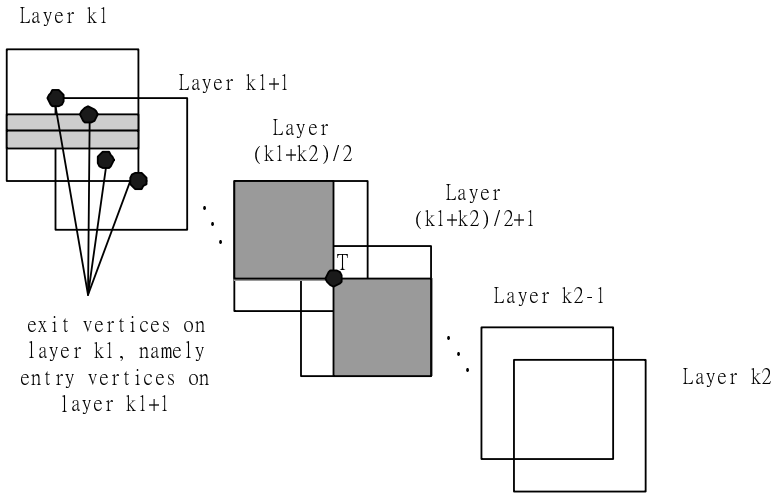

Figure 5: When we apply the divide-and-conquer approach through layers, we also apply linear-space algorithm on each layer. Layer $\frac{k_1+k_2}{2}$ is the middle-layer. T is the mid-point which minimizes the score of the global path through the whole dynamic programming matrix. On each layer each time we only need to record two adjacent rows and the exit vertices (or equivalently the entry vertices on the next layer) that have been computed already as shown on layer k_1. The original $CPSA$ problem is divided at T into two smaller subproblems. For the first subproblem, the end vertex of the new smaller dynamic programming matrix is changed to T, that is, layer $\frac{k_1+k_2}{2}$ is reduced to the top left gray area. For the second subproblem the start vertex of the new smaller dynamic programming matrix is changed to T, in other words, layer $\frac{k_1+k_2}{2}+1$ is reduced to the bottom right gray area. Each of these subproblems can be solved recursively in the same manner.

requires $O(n)$ space in practice.

We expect our algorithm is not only more space efficient but also fast in practice since we avoid the computation of redundant regions on each layer.

4 Space Requirement of Algorithm Layer-DC

In performance analysis of our algorithm Layer-DC, we use the first 4 $RNase$ sequences in Data Set 1 in Figure 7 and Data Set 2 in Figure 8 of Chin et al. [2] with the pattern strings $HKSH$ and $HKSTH$, respectively. These 4 sequences are:

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

We consider $CPSA$ problems for all distinct pairs of these sequences (we note that to solve the $CMSA$ problem for all these sequences one needs to solve a $CPSA$ problem for every distinct pair). In our analysis for the space requirement of our algorithm Layer-DC we use the maximum number of vertices the algorithm loads at any time. This way we measure the memory usage of our algorithm, and we compare it with that of the CPSA-DC algorithm of Lu and Huang [7]. As we show in Figure 8 and Figure 9, for both pattern strings our algorithm Layer-DC algorithm loads much fewer vertices into memory than that of Algorithm CPSA-DC. Therefore we expect that the memory usage of our algorithm Layer-DC is much less than that of Algorithm CPSA-DC. This implies that progressive $CMSA$ algorithms can reduce their space requirement by using our algorithm Layer-DC in solving

```
Layer-DC Algorithm for the CPSA problem:
1. Compute the boundary for each layer:
    For each k, find every pair of first and last possible positions that match P[k] in each
    sequence, S_1, S_2, then find pair of start point and end point
      (S_1begin[k], S_1end[k]), (S_2begin[k]), S_2end[k]) as the boundary for layer k
    for t = 1 to 2 do {
      for k = 0 to r − 1 do {
        set S_first[t][k] = the first possible position that can be aligned to p[k + 1] in S_t;
        set S_last[t][k] = the last possible position that can be aligned to p[k + 1] in S_t;
      }
    }
    for k = 0 to r do
      if(k == 0){
        S_1begin[0] = 0;
        S_2begin[0] = 0;
      } else {
        S_1begin[k] = S_first[0][k − 1] + 1;
        S_2begin[k] = S_first[1][k − 1] + 1;
      }
      if (k == r){
        S_1last[k] = r;
        S_2last[k] = r;
      }else{
        S_1last[k] = S_last[0][k] + 1;
        S_2last[k] = S_last[1][k] + 1;
      }
2. Compute the entry and exit vertices on each layer:
    For layer k, 1 ≤ k ≤ r, find all the candidate entry vertices (t_1, t_2, k) such that
    S_1[t_1] = S_2[t_2] = P[k], in the overlapping region of layer k − 1 and k
    For layer k, 0 ≤ k ≤ r − 1, find all the candidate exit vertices (m_1, m_2, k) such that
    S_1[m_1] = S_2[m_2] = P[k + 1], in the overlapping region of layer k and k + 1
3. Apply the divide-and-conquer approach:
    Find-Concat-Mids(0, r) {Concatenate the series of mid-points to get the optimal solution}

    Function Find-Concat-Mids(k_1,k_2){
      if (k_1! = k_2) {
        for k = k_1 to ⌊(k_1+k_2)/2⌋ do {
          Compute and record the scores for all the candidate exit vertices on layer k
          in row by row manner storing two rows at any time.
          Initialize the scores for all the entry vertices on layer k + 1.
        }
        for k = k_2 to ⌊(k_1+k_2)/2⌋ + 1 do (in reverse matrix) {
          Compute and record the scores for all the candidate exit vertices on layer k
          in row by row manner storing two rows at any time.
          Initialize the scores for all the entry vertices on layer k + 1.
        }
        { The algorithm continues in Figure 7 }
```

Figure 6: Algorithm Layer-DC for the $CPSA$ problem.

```
    Find the mid-point T in the exit vertex set E_1 on layer ⌊(k_1+k_2)/2⌋, namely in the entry
    vertex set E_2 on layer ⌊(k_1+k_2)/2⌋ + 1, which minimizes the score of the global path
    through the whole dynamic programming matrix.
    Change the end vertex of the first new dynamic programming matrix to vertex T, and
    change the start vertex of the second new dynamic programming matrix to vertex T.
    Find-Concat-Mids(k_1, ⌊(k_1+k_2)/2⌋);
    Find-Concat-Mids(⌊(k_1+k_2)/2⌋ + 1, k_2);
  } else { Compute the scores for all the candidate exit vertices on layer k
           in row by row manner storing two rows at any time }
}
```

Figure 7: Algorithm Layer-DC for the $CPSA$ problem (Continuation).

the $CPSA$ problems that arise. For the longer pattern string $HKSTH$, the reduction in memory usage of our algorithm Layer-DC over Algorithm CPSA-DC algorithm is more significant. The reason is that the amount of space required by Algorithm CPSA-DC linearly increases with the pattern length since this algorithm's space complexity is $O(rn)$. On the other hand, in our algorithm Layer-DC, as the pattern string length increases the size of the region necessary to consider in each layer decreases as well as the number of entry and exit vertices on each layer. Therefore the space requirement of our algorithm for the long pattern string $HKSTH$ is even less compared with that for the short pattern string $HKSH$ as shown in Figure 8 and Figure 9. Thus as the length of the pattern string increases our algorithm Layer-DC performs significantly better over Algorithm CPSA-DC. Since our algorithm Layer-DC avoids redundant computations on each layer, we expect that the execution time of our algorithm Layer-DC is faster than that of Algorithm CPSA-DC.

Sequences for CPSA	CPSA−DC	Layer−DC	CPSA−DC/Layer−DC
Seq1 and Seq2	2,340	264	8.86
Seq1 and Seq3	2,010	262	7.67
Seq1 and Seq4	1,980	174	11.37
Seq2 and Seq3	2,010	264	7.61
Seq2 and Seq4	1,980	307	6.45
Seq3 and Seq4	1,980	174	11.37

Figure 8: The numbers of vertices loaded into memory by Algorithm *CPSA-DC* and Algorithm *Layer-DC*, and the performance ratio of Algorithm Layer-DC over Algorithm CPSA-DC for all the $CPSA$ problems in the Data Set 1 in Figure 7 and Data Set 2 in Figure 8 in Chin et al. [2]. We use the first 4 sequences and string $HKSH$ as the pattern.

5 Concluding Remarks

We develop a space-efficient algorithm for the constrained pairwise sequence alignment problem based on our previous layer-by-layer strategy [5] and the Hirschberg's divide-and-conquer approach [6]. By applying the divide-and-conquer approach layer by layer and applying the linear-space pairwise sequence alignment algorithm on each layer simultaneously, we can obtain a $O(n + \alpha)$-space algorithm for the $CPSA$ problem where $\alpha \ll n$ in practice. Our analysis on real data shows that our algorithm requires less space than the algorithm proposed by Chin et al. [2], especially for long pattern strings. Our algorithm can also be used in progressive $CMSA$ algorithms to reduce the space requirement.

Sequences for CPSA	CPSA−DC	Layer−DC	CPSA−DC/Layer−DC
Seq1 and Seq2	2,808	189	14.86
Seq1 and Seq3	2,412	195	12.37
Seq1 and Seq4	2,376	139	17.09
Seq2 and Seq3	2,412	189	12.76
Seq2 and Seq4	2,376	193	12.31
Seq3 and Seq4	2,376	132	18.00

Figure 9: The numbers of vertices loaded into memory by Algorithm *CPSA-DC* and Algorithm *Layer-DC*, and the performance ratio of Algorithm Layer-DC over Algorithm CPSA-DC for all the $CPSA$ problems in the Data Set 1 in Figure 7 and Data Set 2 in Figure 8 in Chin et al. [2]. We use the first 4 sequences and string $HKSTH$ as the pattern.

References

[1] Arslan, A. N. and Eğecioğlu, Ö., Algorithms for the constrained common sequence problem, *Proc. Prague Stringology Conf. 2004*, 24–32, 2004.

[2] Chin, F. Y. L., Ho, N. L., Lam, T. W., Wong, P. W. H., and Chan, M. Y. A., Efficient constrained multiple sequence alignment with performance guarantee, *Proc. IEEE Comp. Systems Bioinform. (CSB 2003)*, 337–346, 2003.

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

[4] Hart, P., Nilsson, N., and Raphael, B., A formal basis for the heuristic determination of minimum cost paths, *IEEE Trans. Syst. Sci. Cybernet.*, 4(2):100–107, 1968.

[5] He, D. and Arslan, A. N., A fast algorithm for the Constrained Multiple Sequence Alignment problem, *Proc. 11th Int. Conf. Automata and Formal Languages (AFL 2005)*, (eds. Ésik, Z. and Fülöp, Z.), Institute of Informatics, University of Szeged, 131–143, 2005.

[6] Hirschberg, D. S., A linear space algorithm for computing maximal common subsequences, *Commun. ACM*, 18:341–343, 1975.

[7] Lu, C. L. and Huang, Y. P., A memory-efficient algorithm for multiple sequence alignment with constraints, *Bioinformatics*, 21(1):20–30, 2005.

[8] 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, *Proc. 1st IEEE Comp. Society Bioinform. Conf. (CSB 2002)*, 127–137, 2002.

[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 and Hall, 1995.