

# A\* Algorithms for the Constrained Multiple Sequence Alignment Problem

Dan He and Abdullah N. Arslan  
Department of Computer Science  
University of Vermont  
Burlington, VT 05405, USA  
{dhe, aarslan}@cs.uvm.edu

## Abstract

It is well known that  $A^*$  algorithm reduces the search space in many applications. For shortest path computations, the algorithm uses a heuristic estimator which can better guide the search to the destination. It allocates memory dynamically to store only the necessary vertices, which are those vertices in its open list and close list. Therefore, an  $A^*$  algorithm for the shortest paths search problem is much more space efficient than ordinary search algorithm such as the dynamic programming algorithm and the *Dijkstra* algorithm. The constrained multiple sequence alignment problem (*CMSA*) aims to align similar subsequences in the same region under the guidance of a given pattern (constraint). The *CMSA* problem can be considered as an optimal path search problem in the dynamic programming matrix. In this paper, we propose two  $A^*$ -based algorithms which we experimentally show that in practice they solve the *CMSA* problem using much less memory than does the ordinary dynamic programming algorithm.

**Keywords:** constrained sequence, pairwise alignment, multiple alignment, dynamic programming,  $A^*$  algorithm.

## 1 Introduction

*Multiple sequence alignment (MSA)* is one of the most important problems in computational biology. It is used for extracting biologically homogeneous sequences from a set of DNA or protein sequences, and for the prediction of protein structure or the phylogenetic tree construction. Given  $n \geq 2$  sequences, the *MSA* problem aims to align similar subsequences in the same region such that the score for the resulting alignment is minimized (or maximized according to the scoring function). When  $n = 2$ , namely the sequence set has only two sequences  $S_1, S_2$ , this problem is the classical pairwise sequence alignment, which has an  $O(s_1 s_2)$ -time

dynamic programming algorithm [20]. The same two-dimensional dynamic programming algorithm can be extended into a multi-dimensional dynamic programming algorithm with  $O(s_1 s_2 \dots s_n)$  time complexity. There are many heuristic algorithms to approximate optimal solution. These include Clustal W [14], T-Coffee [11]. Recent progress in multiple sequence alignment is summarized in [12].

The *constrained multiple sequence alignment (CMSA)* problem has been introduced by Tang et al. [13]. The *CMSA* problem is the *MSA* problem where an optimal alignment is constrained to contain a given pattern  $P$ . The problem aims to incorporate the biologically meaningful prior knowledge of the structure or pattern into the alignment process.

There are many dynamic programming algorithms for the *CMSA* and *CPSA* (the *CMSA* problem for two sequences), and their variations [13, 4, 15, 16, 2, 5, 1]. The time complexity of each of these algorithms is  $O(s_1 s_2 \dots s_n r)$  where  $s_1, s_2, \dots, s_n$  are, respectively, the lengths of the strings  $S_1, S_2, \dots, S_n$ , and  $r$  is the length of the pattern  $P$ . For the *MSA* and *CMSA* problems, space complexity is a major bottleneck. For the ordinary dynamic programming algorithm, the space requirements for *MSA* and *CMSA* problems are  $O(s_1 s_2 \dots s_n)$  and  $O(s_1 s_2 \dots s_n r)$ , respectively.

It is well known that  $A^*$  algorithm for the shortest paths search problem is much more space efficient than ordinary search algorithm such as the dynamic programming algorithm and the *Dijkstra* algorithm.  $A^*$  algorithm has been used for the *MSA* problem [8, 17, 9, 18, 19, 10].

In this paper, we propose two  $A^*$ -based *CMSA* algorithms by using some observations made by He and Arslan [1]. Our experiments on *RNase* sequences show that our algorithms are much more space efficient than the existing dynamic programming algorithms [4, 1].

The outline of this paper is as follows: In Section 2 we summarize previous results for the multiple sequence

alignment problems *MSA* and *CMSA*, and briefly describe an  $A^*$  algorithm for shortest paths search. In sections 3 and 4 we present our  $A^*$  algorithms for the *CMSA* problem. We summarize the results of our experiments in Section 5. We include our final remarks in Section 6.

## 2 Background

### 2.1 Algorithms for the *MSA* and *CMSA* Problems

The *multiple sequence alignment* problem is defined as follows: given a set of  $n \geq 2$  sequences  $S_1, S_2, \dots, S_n$ , the global sequence alignment allows insertions of the gap symbol '-' into this set of sequences, resulting in, respectively, equal length strings  $S_1^*, S_2^*, \dots, S_n^*$  such that the global similarity score, which is defined as the sum of pair-wise similarity scores  $\sum_{1 \leq i < j \leq n} \text{score}(S_i^*, S_j^*)$ , is optimized. When  $n = 2$ , i.e. the sequence set has only two sequences  $S_1, S_2$ , this problem is the classical pairwise sequence alignment which has an  $O(s_1 s_2)$ -time dynamic programming algorithm [20]. This dynamic programming solution can be extended for the multiple sequence alignment case in a simple way using a multi-dimensional matrix, and requiring  $O(s_1 s_2 \dots s_n)$  time.

For the *CMSA* problem Chin et. al [4] present the following dynamic programming formulation: Let  $D(i_1, i_2, \dots, i_n, k)$  be the optimum constrained sequence alignment score of sequences  $S_1[1..i_1], S_2[1..i_2], \dots, S_n[1..i_n]$  such that optimal alignment contains  $P[1..r]$ . Then this score can be computed by the following recurrence:

**Theorem 1** [4]  $D(i_1, \dots, i_n, k) = \infty$  if  $i_1 = 0$ , or  $i_2 = 0$ , or  $\dots$ , or  $i_n = 0$  for all  $k$ ,  $1 \leq k \leq r$ , and  $D(\{0\}^n, 0) = 0$ , and for all  $i_1, i_2, \dots, i_n, k$ ,  $0 < i_1 \leq s_1, 0 < i_2 \leq s_2, \dots, 0 < i_n \leq s_n, 0 \leq k \leq r$ ,

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

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  $\delta(x_1, x_2, \dots, x_k) = \sum_{1 \leq i < j \leq n} \delta(x_i, x_j)$ .

### 2.2 $A^*$ Algorithm

The  $A^*$  algorithm [7] is a very popular heuristic search algorithm which is the extension of Dijkstra's single source shortest path algorithm [6]. The  $A^*$  algorithm keeps track of the explored vertices on the search frontier in an open list, and store the already expanded nodes in a close list. It uses a heuristic estimator as the lower bound on the distance of shortest path from each vertex to the destination. The score for each vertex is the sum of the heuristic value and the shortest found distance from the source to the vertex. The algorithm always expands the vertex with the minimum score. For the shortest paths search problem, the  $A^*$  algorithm is much more space efficient than ordinary search algorithms such as the dynamic programming algorithm and the *Dijkstra* algorithm.

Given  $n$  sequences  $S_1, S_2, \dots, S_n$  with lengths, respectively,  $s_1, s_2, \dots, s_n$ , the *multiple sequence alignment (MSA)* problem can be considered as a path problem on a graph whose vertices are placed at entries in an  $n$ -dimensional matrix of size  $s_1 \times s_2 \times \dots \times s_n$ , and to each vertex  $v$  there are vertices from each of its neighbors, i.e. vertices whose coordinates are either the same or one less than that of  $v$  in each dimension. In this graph the shortest path (or the longest path depending on the similarity model) between vertices  $(0, 0, \dots, 0)$  and  $(s_1, s_2, \dots, s_n)$  is an optimal solution of the *MSA* problem.

The  $A^*$  algorithm was first used in solving the *MSA* problem by Ikeda and Imai [8]. They successfully applied  $A^*$  algorithm on the dynamic programming matrix to improve the efficiency of the shortest path search. The heuristic function used in their algorithm is the sum of the pairwise distances in sequence alignment:

$$h(v) = \sum_{1 \leq i < j \leq d} h_{ij}^*(v_{ij})$$

where  $h_{ij}^*(v_{ij})$  represents the shortest path length from vertex  $v_{ij}$  to destination  $t_{ij}$  in the two-dimensional matrix for  $S_i$  and  $S_j$ .

### 3 Algorithm *Dijkstra-A\*-CMSA*

The dynamic programming solution for the *CMSA* problem can be considered as finding a shortest path in the dynamic programming matrix which we can visualize in layers indexed by its last dimension (the positions in the pattern string) as shown in Figure 1 where each layer is an  $n$ -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

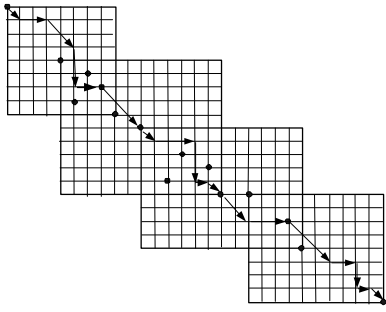


Figure 1: For the *CPSA* (*CMSA* with  $n = 2$ ) with pattern string of length 3, a global shortest path passing through entry and exit vertices, and connecting sub-paths on each layer.

shortest path enters each layer at a vertex (we call it an *entry vertex*), after traversing a number of vertices in 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 1. The length of a shortest path is the sum of the length of the sub-paths on each layer, and each sub-path between entry vertex  $u$  and exit vertex  $v$  on layer  $k$  of the shortest path is the shortest path between  $u$  and  $v$  on layer  $k$ . We call an optimal solution of the *CMSA* problem as *global shortest path* to distinguish it from the shortest paths on individual layers.

Given all possible entry and exit vertices on each layer, we consider the computation of shortest paths between all entry-exit vertex-pairs. We observe the following:

- The number of entry vertices and the number of exit vertices on each layer are comparatively much smaller than the total number of vertices on the same layer. We show this in Figure 2.
- For each exit vertex, we only need to store its shortest path from the destination. Once we compute the shortest paths we can delete from the memory the vertices that are not on these paths.

layer	#entry	#exit	#total
0	1	10	66,048
1	10	2	86,400
2	2	96	605,160
3	96	32	392,000
4	32	1	385,836

Figure 2: The number of entry, exit and total vertices on each layer for the Data Set 1 in Figure 7 and Data Set 2 in Figure 8 in Chin et. al (2003). We use the first 3 sequences and string *HKSH* as the pattern.

We present two  $A^*$  algorithms for the *CMSA* problem which are identical except for the last step. The first of these is Algorithm *Dijkstra-A\*-CMSA*:

1. Find a boundary of the region that needs to be considered in computations for each layer  $k$  in the dynamic programming matrix.
2. Find all entry and exit vertices for each layer  $k$ ,  $0 \leq k \leq r$ .
3. Use the local  $A^*$  algorithm to find shortest paths among entry and exit vertices on each layer, and use the *Dijkstra* algorithm to compute a shortest path from the vertex  $(0, 0, \dots, 0, 0)$  to vertex  $(s_1, s_2, \dots, s_n, r)$ .

In Step 1 of Algorithm *Dijkstra-A\*-CMSA* we use part (Steps 2 and 3) of Algorithm *FastCMSA* of He and Arslan [1] to find the boundary for each layer  $k$  in the dynamic programming matrix. The execution of these steps determines at each layer  $k$  a rectangular region whose two diagonal corners respectively, are  $(S_{1begin}[k], S_{2begin}[k], \dots, S_{nbegin}[k])$ , and  $(S_{1last}[k], S_{2last}[k], \dots, S_{nlast}[k])$ .

In Step 2, we find entry and exit vertices 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  $k+1$ . 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,  $(0, 0, \dots, 0)$ , and layer  $r$  has only one exit vertex  $(s_1, s_2, \dots, s_n)$  (see Figure 1).

After we find all the entry vertices and exit vertices on each layer, we can construct a graph  $G = (V, E)$  where  $V$  is the set of entry and exit vertices for all layers combined, and  $E$  is the set of edges among those vertices. Only the entry vertices and exit vertices on the same layer have edges among them, and the weight of an edge  $e(u, v)$  on layer  $k$  is the shortest distance between vertices  $u$  and  $v$  on layer  $k$ .

In Step 3, we use the *Dijkstra* algorithm on graph  $G$  to find a global shortest path from the vertex  $(0, 0, \dots, 0, 0)$  to vertex  $(s_1, s_2, \dots, s_n, r)$ .

For each entry vertex  $u$  we only need to find the shortest paths to the exit vertices whose coordinates on all dimensions are no less than the coordinates of vertex  $u$  since we consider the dynamic programming matrix as an acyclic directed graph. Therefore, an entry vertex  $u$  has edges in graph  $G$  only to exit vertices on the same layer whose coordinates on all dimensions are no less

than those of  $u$  (we call these exit vertices an *exit vertex set* of  $u$ ).

The weight of the edge  $e(u, v)$  can be computed by using  $A^*$  algorithm. When we expand a vertex  $u$  in graph  $G$  and update the score of the vertices in its exit vertex set, we can use the  $A^*$  algorithm to compute the shortest paths from this vertex  $u$  to all the vertices in its exit vertex set (We call this algorithm the *local  $A^*$  algorithm*). We use in the local  $A^*$  algorithm the following heuristic estimator introduced by Ikeda [8]:

$$h(v) = \sum_{1 \leq i < j \leq d} h_{ij}^{*local}(v_{ij}, u_{ij})$$

where  $h_{ij}^{*local}(v_{ij}, u_{ij})$  represents the shortest path length from vertex  $v_{ij}$  to local destination  $u_{ij}$ , which is decided by the boundary, in the two-dimensional matrix for  $S_i$  and  $S_j$ .

Once we compute the shortest path from  $u$  to some vertex  $v$  in  $u$ 's exit vertex set, we just need to trace back from  $v$  and store only the shortest path from  $u$  to  $v$ . Therefore, we can close (delete) other loaded vertices from the memory.

Computing the heuristic estimators of the  $A^*$  algorithm for the *CMSA* problem would take impractically long time since for each vertex on our search paths we need to compute the sum of the pairwise sequence alignment scores to the local destination of the search. We propose three strategies to improve *Dijkstra- $A^*$ -CMSA*:

1. Since we use the *Dijkstra* algorithm for the global shortest path, for the vertices on each layer, we record not only their local scores in the  $A^*$  search on that layer, but also their distances (which we call the *global distance*) from the source. We expand the exit vertices in their ascending order of their scores starting with an exit vertex with the minimum score. Our local  $A^*$  algorithm only expands the un-expanded exit vertices in any exit vertex set. Therefore, an exit vertex is never re-expanded.
2. Since we expand vertices in ascending order of their scores, the computation of the local shortest paths on each layer should be done backwards from the exit vertices to entry vertices. When we expand an entry vertex, we compute the shortest paths backwards from all the exit vertices in its exit vertex set. We do the computations backwards so that we can store the heuristic values for the already explored vertices. These values can be reused in the search from every exit vertex to this entry vertex on that layer. Therefore, we can avoid the re-computation for the heuristic estimators among the entry vertex and the exit vertices in its exit vertex

set. This strategy is a tradeoff between memory requirement and computation time since we need to store all the explored vertices in the computation of shortest paths from the vertices in the exit vertex set to their corresponding entry vertex. Given that in practice the number of exit vertices and entry vertices are much smaller than the total number of vertices on a layer, this strategy does not increase memory requirement significantly. After we finish the computation of a set of shortest paths, we can delete all of the loaded vertices in the search process from the memory, and store only the shortest paths.

3. After the global distance for one exit vertex is computed, it can be used as an upper bound for any other shortest paths search from this vertex to all the other entry vertices on the same layer. Therefore, when we compute the shortest path from the exit vertex to another entry vertex, we calculate the sum of the local score of the newly expanded vertex and the global score of the corresponding entry vertex. If the sum is not smaller than the upper bound, the search can stop immediately. Otherwise, if the new global distance for the exit vertex is smaller than its previous global distance then we update its global distance to the new one, which will also be the new upper bound, and we also update the corresponding local shortest path for this exit vertex and remove the old path. Our experiments show that this strategy reduces the search space and thus the computation time.

## 4 Algorithm *Dual- $A^*$ -CMSA*

Since the heuristic estimators for the local  $A^*$  algorithm are expensive to compute, the order to expand the exit vertices on each layer is of vital importance. The order of expansion changes the time and space requirements of the algorithm significantly.

In our algorithm *Dijkstra- $A^*$ -CMSA* we use the results of the *Dijkstra* algorithm to decide the order of expansion of the exit vertices. Since, in general, the  $A^*$  algorithm is more efficient on the shortest paths search problem than the *Dijkstra* algorithm, we propose to use  $A^*$  algorithm for the computation of a global shortest path between vertices  $(0, 0, \dots, 0, 0)$  and  $(s_1, s_2, \dots, s_n, r)$  on graph  $G$ . We call the resulting algorithm the *global  $A^*$  algorithm* to distinguish it from the local  $A^*$  algorithm. Our experiments show that the global  $A^*$  algorithm is more efficient than the *Dijkstra* algorithm on  $G$ .

**Proposition 1** *The following heuristic estimator  $h(v)$*

we use in our global  $A^*$  algorithm for the computation of a global shortest path is monotonic:

$$h(v) = \sum_{1 \leq i < j \leq d} h_{ij}^{*global}(v_{ij})$$

where  $h_{ij}^{*global}(v_{ij})$  represents the shortest path length from vertex  $v_{ij}$  to the global destination  $t_{ij}$  (namely the exit vertex on the last layer) in the two-dimensional matrix for  $S_i$  and  $S_j$ .

**Proof** For the same given sequences, the optimum score of the  $CMSA$  problem can be no better than the optimum score of the  $MSA$  problem since the  $CMSA$  problem is subject to some pattern string.

For vertices  $u$  and  $v$  in graph  $G$ , we say a heuristic estimator  $h()$  is monotonic if for any nodes  $u, v$ ,  $l(u, v) + h(v) \geq h(u)$ , where  $l(u, v)$  is the length of the edge  $e(u, v)$ . A monotonic heuristic estimator can guarantee an optimal solution. In graph  $G$  we construct after the first two steps of our  $A^*$ - $CMSA$  algorithms, for vertices  $u$  and  $v$ :

$$l(u, v) + h(v) = \sum_{1 \leq i < j \leq d} h_{ij}^{*local}(u_{ij}, v_{ij}) + \sum_{1 \leq i < j \leq d} h_{ij}^{*global}(v_{ij})$$

Without loss of generality, let  $u$  and  $v$  be two vertices on layer  $k$  which contains one symbol  $P[k+1]$  as the constraint. We can see that  $\sum_{1 \leq i < j \leq d} h_{ij}^{*local}(u_{ij}, v_{ij}) + \sum_{1 \leq i < j \leq d} h_{ij}^{*global}(v_{ij})$  = the optimum score of an ordinary  $MSA$  beginning from vertex  $u$  ending at the global destination with the constraint of symbol  $P[k+1]$  at vertex  $v$ . The constrained score can be no better than the optimum score of an ordinary  $MSA$  beginning from vertex  $u$  ending at the global destination, i.e.  $\sum_{1 \leq i < j \leq d} h_{ij}^{*global}(u_{ij})$ . Therefore, we have:

$$\begin{aligned} l(u, v) + h(v) &= \sum_{1 \leq i < j \leq d} h_{ij}^{*local}(u_{ij}, v_{ij}) \\ &+ \sum_{1 \leq i < j \leq d} h_{ij}^{*global}(v_{ij}) \\ &\geq \sum_{1 \leq i < j \leq d} h_{ij}^{*global}(u_{ij}) \\ &= h(u) \end{aligned}$$

This concludes that the heuristic estimator  $h(v) = \sum_{1 \leq i < j \leq d} h_{ij}^{*global}(v_{ij})$  is monotonic and can guarantee an optimal solution. •

We could also use the global  $A^*$  algorithm with the heuristic estimator  $h(v) = \sum_{1 \leq i < j \leq d} h_{ij}^{*global}(v_{ij})$  directly to compute an optimal solution for the  $CMSA$  problem. We do not follow this strategy because the following two reasons make the resulting algorithm less space efficient: (1) The number of exit vertices on each layer is too small for the global  $A^*$  search to meet and to enter next layer. (2) If we use the global  $A^*$  algorithm directly, then we need to store all the vertices visited during the search whose number may be very large.

## 5 Experiments

In our experiments we use the first 3 *RNase* sequences in Data Set 1 in Figure 7 and Data Set 2 in Figure 8 of Chin et. al [4], with the pattern string *HKSH*. These 3 sequences are:

*Seq1* :  $gi|119124|sp|p12724|ecp\_human$   
*Seq2* :  $gi|2500564|sp|p70709|ecp\_rat$   
*Seq3* :  $gi|13400006|pdb|ldyt|$

All experiments are done on an Intel Xeon 2.4GHZ processor with 2GB memory.

In Figure 3, we show the number of loaded and stored vertices on each layer by our algorithms *Dijkstra-A\*-CMSA* and *Dual-A\*-CMSA* respectively, and compare them with the total number of vertices in the same layer. When we finish the backward search from one exit vertex set to its entry vertex, we just need to record the shortest paths and delete all the other loaded vertices from the memory. The memory requirement for the computation on each layer is decided by one of the backward searches on that layer which explored the largest number of vertices. Therefore, we only show this number as #loaded. The #stored is the number of vertices appearing on the shortest paths, and each exit vertex corresponds to one shortest path, other redundant paths will be deleted from memory. The #total is the number of all the vertices in the region computed by Algorithm *FastCMSA* of He and Arslan [1].

In Figure 4, we compare the total number of vertices stored until layer  $k$ ,  $0 \leq k \leq 4$  for our  $A^*$ - $CMSA$  algorithms, Algorithm *FastCMSA* [1] and the naive  $CMSA$  algorithm which we call *NaiveCMSA* [4]. Algorithm *NaiveCMSA* [4], for each layer  $k$ , needs to store all the vertices from layer 0 to layer  $k-1$ . Although Algorithm *FastCMSA* [1] avoids redundant regions, when processing the current layer, it still keeps all the vertices in the preceding layers in memory. In our  $A^*$ - $CMSA$  algorithms when we process the current layer we only need the local shortest paths in the preceding layers. Given that the number of exit vertices on each layer is much smaller than the total number of vertices on the same layer, and since for each exit vertex we only need to store one local shortest path, our algorithms keep track of a small number of vertices. This is a significant improvement in space requirement over the other two algorithms *NaiveCMSA*, and *FastCMSA*.

We show in Figure 5 the real memory usages by algorithms *FCMSA*, *Dijkstra-A\*-CMSA*, *Dual-A\*-CMSA*. Obviously our two  $A^*$  algorithms require much less memory than *FCMSA* algorithm. In addition, techniques like dynamic memory allocation, which we do not use here, could make the  $A^*$  algorithms more space efficient.

In Figure 6, we show that our *Dijkstra-A\*-CMSA*

layer	#total	<i>Dijkstra-A*-CMSA</i> #loaded	<i>A*-CMSA</i> #stored	<i>Dual-A*-CMSA</i> #loaded	<i>A*-CMSA</i> #stored
0	66,048	7,927	681	7,927	681
1	86,400	1,960	117	525	117
2	605,160	16,205	6,465	16,205	6,465
3	392,000	11,357	1,514	1,247	221
4	385,836	483	7	35	7

Figure 3: The number of loaded, stored and total vertices on each layer for the Data Set 1 in Figure 7 and Data Set 2 in Figure 8 of Chin et. al (2003) by our algorithms *Dijkstra-A\*-CMSA* with upper bound, and *Dual-A\*-CMSA* with upper bound. We use the first 3 sequences and use string *HKSH* as the pattern.

layer	<i>Dijkstra-A*-CMSA/Dual-A*-CMSA</i>	<i>Fast-CMSA</i>	<i>Naive-CMSA</i>
0	681/681	66,048	3,798,795
1	858/858	152,448	7,597,590
2	7,323/7,323	757,608	11,396,385
3	8,837/7,544	1,149,608	15,195,180
4	8,844/7,551	1,535,444	18,993,875

Figure 4: The total number of vertices stored until layer  $k$ ,  $0 \leq k \leq 4$ , by our *A\*-CMSA* algorithms, and algorithms *FastCMSA* (Dan and Arslan (2005)), *NaiveCMSA* (Chin et al. (2003)), for Data Set 1 in Figure 7 and Data Set 2 in Figure 8 of Chin et. al (2003). We use the first 3 sequences and string *HKSH* as the pattern.

	<i>FCMSA</i>	<i>Dijkstra-A*</i>	<i>Dual-A*</i>
<i>Mem(MB)</i>	46	29	11

Figure 5: The memory usages by algorithms *FCMSA*, *Dijkstra-A\*-CMSA*, *Dual-A\*-CMSA*, for Data Set 1 in Figure 7 and Data Set 2 in Figure 8 of Chin et. al (2003). We use the first 3 sequences and string *HKSH* as the pattern.

algorithm that uses dynamically updated global distances as upper bounds reduce the number of explored vertices by more than 25%. When the sum of the pairwise sequence alignment scores is used as the heuristic estimators for the local *A\** algorithm, the computation of the estimators is very expensive, and the use of an upper bound can reduce the computation time significantly. While this is true for Algorithm *Dijkstra-A\*-CMSA*, Algorithm *Dual-A\*-CMSA* makes slight improvement with the use of the upper bound. The reason is that in our experiments, Algorithm *Dual-A\*-CMSA* always expands optimal exit vertices. That is, the global *A\** algorithm in our experiments never re-expand any vertex. Therefore, the search space is nearly the same with or without the upper bound. If there are re-expansions as in Algorithm *Dijkstra-A\*-CMSA*, the use of the upper bound can often significantly reduce the search space, and also improve the time complexity.

	without UB	with UB
<i>Dijkstra-A*-CMSA</i>	467,309	387,656
<i>Dual-A*-CMSA</i>	66,523	66,516

Figure 6: The number of explored vertices by algorithms *Dijkstra-A\*-CMSA*, and *Dual-A\*-CMSA* for Data Set 1 in Figure 7 and Data Set 2 in Figure 8 of Chin et. al (2003) with upper bound (UB), and without upper bound separately. We use the first 3 sequences and string *HKSH* as the pattern.

## 6 Conclusion and Future Work

We have developed two *A\** algorithms for the constrained multiple sequence alignment problem. We experimentally show that our algorithms are much more space efficient than the existing algorithms.

Instead of the ordinary *A\** algorithm, the variants of the *A\** algorithm [17, 9, 18, 19, 10] can also be applied to the constrained multiple sequence alignment problem. This can improve the space requirement even further. Techniques such as dynamic memory allocation can make our *A\** algorithms much more space efficient. These are topics of future research.

We expect that our *A\** algorithms are very efficient when the sequences aligned are similar because in this case the local and global shortest paths are less costly to

find, and keep track. These are the main steps in our  $A^*$  algorithms.

## References

- [1] D. He and A. N. Arslan. A fast algorithm for the Constrained Multiple Sequence Alignment problem *Proc. 11th International Conference on Automata and Formal Languages (AFL 2005)* (Eds. Zoltan Ésik, Zoltan Fülöp), Institute of Informatics, University of Szeged, pp. 131-143, Dobogoko, Hungary, May 2005.
- [2] A. N. Arslan and Ö. Eğecioğlu. Algorithms for the constrained common sequence problem. *Proc. Prague Stringology Conference 2004*, (Eds. M. Simanek and J. Holub), pp. 24-32, Prague, August 2004.
- [3] D. Champeaus. Bidirectional Heuristic Search Again *J. ACM*, vol. 30, pp.22-32, 1983.
- [4] F. Y. L. Chin, N. L. Ho, T. W. Lam, P. W. H. Wong, M. Y. Chan. A. Efficient constrained multiple sequence alignment with performance guarantee. *Proc. IEEE Computational Systems Bioinformatics (CSB 2003)*, pp. 337-346, 2003.
- [5] F. Y.L. Chin, A. D. Santis, A. L. Ferrara, N. L. Ho, S. K. Kim. A simple algorithm for the constrained sequence problems. *Information Processing Letters* Vol. 90, pp. 175-179, 2004.
- [6] E. Dijkstra. A note on two problems in connection with graphs. *Numerical Mathematics*, 1, 395-142, 1959.
- [7] P. Hart, N. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths *IEEE Trans. Syst. Sci. Cybernet.*, 4(2):100-107, 1968.
- [8] T. Ikeda and H. Imai. Fast  $A^*$  algorithm for multiple sequence alignment. *Genome Informatics Workshop 94*, 90-99, 1994.
- [9] R. Korf and W. Zhang. Divide-and-conquer frontier search applied to optimal sequence alignment. *Proceedings of the 17th National Conference on Artificial Intelligence (AAAI-00)*, pp. 910-916, 2000.
- [10] M. McNaughton, P. Lu, J. Schaeffer, and D. Szafron. Memory-efficient  $A^*$  heuristics for multiple sequence alignment. *Proc. of 18th National Conference on Artificial Intelligence (AAAI-02)*, 737-743, 2004
- [11] C. Notredame, D. G. Higgins, J. Heringa. T-Coffee: A novel algorithm for multiple sequence alignment. *J.Mol.Biol.*, 302,205-217, 2000.
- [12] C. Notredame. Recent progresses in multiple sequence alignment: a survey. Ashley Publications Ltd, ISSN 1462-2416, 2001.
- [13] C. Y. Tang, C. L. Lu, M. D.-T. Chang, Y.-T. Tsai, Y.-J. Sun, K.-M. Chao, J.-M. Chang, Y.-H. Chiou, C.-M. Wu, H.-T. Chang, and W.-I. Chou. Constrained multiple sequence alignment tool development and its applications to rna family alignment. *Proceeding of the 1st IEEE Computer Society Bioinformatics Conference (CSB 2002)*, pp. 127-137, 2002.
- [14] J. Thompson, D. Higgins, T. Gibson. 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.
- [15] Y.-T. Tsai. The constrained common sequence problem. *Information Processing Letters*, 88:173-176, 2003.
- [16] Y.-T. Tsai, C. L. Lu, C. T. Yu, and Y. P. Huang. MuSiC: A tool for multiple sequence alignment with constraint. *Bioinformatics*, 20(14):2309-2311, 2004.
- [17] T. Yoshizumi, T. Miura and T.Ishida  $A^*$  with partial expansion for large branching factor problems. *Proceedings of the 17th National Conference on Artificial Intelligence (AAAI 2000)*, pp. 923-929, 2000.
- [18] R. Zhou and E. Hansen. Sweep  $A^*$ : Space-efficient heuristic search in partially ordered graph. *Proceedings of the 15th IEEE International Conf. on Tools with Artificial Intelligence*, pp. 427-434, 2003.
- [19] R. Zhou and E. Hansen. Space-Efficient Memory-Based Heuristics *19th National Conference on Artificial Intelligence (AAAI-04)*, 2004.
- [20] M. S. Waterman. *Introduction to computational biology*. Chapman & Hall, 1995.