CMPUT 657 – Heuristics Search Project Report

Evaluation of the search algorithms in Multiple Sequence Alignment

Students: Tommy Chu & Xiang Wan

Professor: Dr. Jonathan Schaeffer

Contents

List of Figures

List of Tables

1. Introduction

Multiple sequence alignment (MSA) is a controversial problem in computational biology. This particular problem computes the similarity based on the biological properties of nuclei acid (or amino acid) among the DNA strands (or protein sequences). When this biological problem is mapped to a computing science problem, the formulation becomes finding the similarity between multiple strings. The similarity of two aligned characters relies on the cost function, which will return a distance (or score value). The similarity of the alignment, then, is the sum of all pair aligned characters' distances (or all pair scores). The optimal pair-wise alignment is referred to align two strings and spaces could be inserted into each string to obtain the optimal similarity. MSA is a generalized version, which aligns multiple strings (we refer the number of strings is d) simultaneously. In this project, we will use the distance as similarity measurement like in [5]. That is, when two characters match, mismatch, or aligning with space, its distance is 0, 1, and 2 respectively. Then the optimal alignment should have the minimal distance.

The report is organized as follows. Next section, we will present the previous work and techniques from computational biology to artificial intelligence. Then, section 3 reveals our objective of this project, and section 4 will explain the implementation issues of three implemented algorithms. After that, section 5 will discuss the results we obtain and evaluate each implemented algorithm. At last, we will conclude the comparison study we have learnt from this project.

2. Literature review

2.1 Dynamic Programming Approach

Needleman and Wunsch's algorithm [1] is the first one that formulates the optimal pair-wise sequence alignment problem. Given two length-N sequences, say S1 and S2, the algorithm initializes a matrix of size (N+1) x (N+1). Each entry will store the optimal alignment score for two sub-strings. For instance, entry with column i and row j, will store the optimal score for the alignment between S1[0…i-1] and S2[0…j-1]. Therefore, the first row and first column are initialized the distance of total number of spaces that can be inserted in front if aligning to another string. Then, computing from left to right and from top to bottom, each entry can be optimized by looking at the current character of each string. There exist three possibilities of looking at current entry, which are aligning S1[i] to a space, S2[j] to a space, or S1[j] to S2[j].

Multiple sequence alignment, moreover, is a similar problem. MSA requires aligning d sequences simultaneously. With same dynamic programming approach, the algorithm increases the dimension of the matrix to d, which increases the space complexity. Then, for each entry, there will be 2^d

possibilities to align the last character of each sting. Therefore, the dynamic programming approach has time and space complexity of O(n^d).

Since the DNA sequence can be thousands to millions bytes long and the number of strings we like to align simultaneously increases day by day, the time and space complexity of dynamic programming are not sufficient, and researchers improve the algorithm by reducing the search space of the matrix. Hischberg's algorithm [2] reduces the space complexity from O(n^d) to O(n^{d-1}). The idea is to use divide and conquer approach on top of the dynamic programming. To divide the problem into two sub problems, it finds a point k that separates the sequences into two parts, and then recursively build the path from upper left corner of the matrix to k, and the path from k to the lower right corner of the matrix. Although D.S. Hirschberg introduced an O(n^{d-1}) space algorithm in [2], the time necessitated to solve this problem requires twice amount efforts as the original dynamic programming approach.

2.2 Heuristic Search Approach

Turning the dynamic programming approach to a path-finding problem in a d-dimensional matrix, we may apply the existing heuristic search techniques in artificial intelligence. In the formulation of dynamic programming, a d-dimensional matrix is created and to find the optimal alignment is to find the optimal path from a initial node (located at the first entry where its value is always initialized to 0) to a goal node (located at the last entry where its stores the optimal alignment score when the computation is completed). For instance, aligning 3 length-N strings, we will find the optimal path from dimensional the matrix entry M[0][0][0] to M[N][N][N].

Path finding problem can be solved using greedy algorithm, such as Dijkstra's shortest path algorithm and A* algorithm. However, because of the huge search space due to large number of short cycle and the requirement of storing distinct path in each node, the memory limitation turns out to be a major problem when solve the general problem using these techniques. Take a simple pair-wise optimal alignment as an example. If two strings S1 and S2 are given and the length of both strings is N, it will initial a 2 dimensional matrix with size (N+1) x (N+1). Between the source node and goal node, if we ignore the diagonal moves, there is $N!/(N!)^2$ number of distinct paths.

2.2.1 A* Algorithm

A* eventually is a best first search algorithm. Each node consists of an *f-value* defined as f = g + h, in which g-value is the known distance for aligned between each sub-string from the first to its current aligned character while h value is the estimated distance for the alignment for each sub-string from the current aligned character to its last character. The g-value builds up recursively where h-value is generated from a heuristic function. A popular admissible heuristic function used in MSA problem is the sum of optimal pair-wise alignments. The heuristic value is a lower bound since the cost of the actual alignment of each pair is at least as good as the cost of the optimal pair-wise alignment.

The algorithm will first put the initial node into the OPEN list, which stores the nodes that are not fully considered. Then, at each step, it will select the best f-value node from the OPEN list to explore, and the algorithm terminates when the goal node is found (or no solution when OPEN list becomes empty). If the solution cannot be found after such node is explored, we will put it into the CLOSE list, which prevents the repeated search

2.2.2 Divide-and-Conquer Bi-directional Search (DCBDA*)

In 1999, Korf introduced a technique, *forbidden operator*, and applied *divide and conquer* and *bi-directional search* techniques as A* enhancement [8], namely Divide and Conquer Bi-directional Search (or DCBDA*). The space requirement reduces to O(n^{d-1}), and the most significant result is that the time complexity will not be twice as original dynamic programming (like Hischberg's algorithm does), but increase by constant amount of time [8]. Indeed, DCBDA* is a variant of A* algorithm; however, applying it to MSA problem, the author claims only storing the OPEN list is enough when forbidden operator is introduced into each node. The forbidden operator indicates which direction(s) of this node has been searched before. With the forbidden operator, DCBDA* will not repeatedly expand those nodes that have been explored.

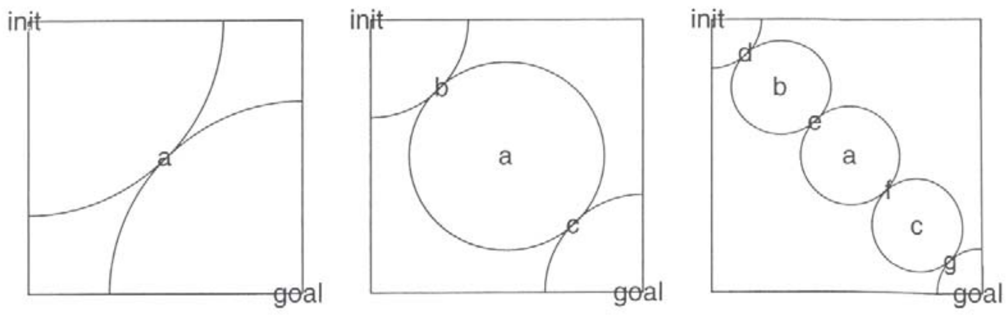


Figure 1: Divide-and-Conquer path construction [5].

Furthermore, divide-and-conquer approach is similar to Hischberg's idea, and it divides the problem into two halves at first. When the search finishes, we can get a roughly intermediate point in the optimal path. Then, we can use this point to divide the whole problem into two sub-problems and recursively apply the same algorithm to solve sub problems until the whole optimal path is constructed. Figure 1 illustrates the divide-and-conquer approach. In general problem, we believe that the computed area in each search is roughly half of the computed area in the previous search. So, the extra computation cost for path constructions is approximately equal to the computation cost in the first iteration. Moreover, bi-directional search allows simultaneously searching both forward from the initial state node and from the goal node. Nonetheless, two conditions must be satisfied if this algorithm is applied in general case. One is that the heuristic function must be consistent and the other is that the problem space should be polynomial. The details analysis is described in [5].

2.2.3 Divide-and-Conquer Frontier Search (DCFA*)

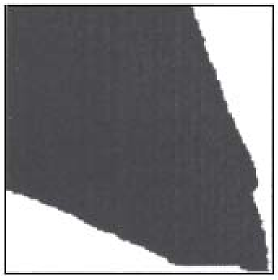


Figure 2: Computed area of frontier search [6].

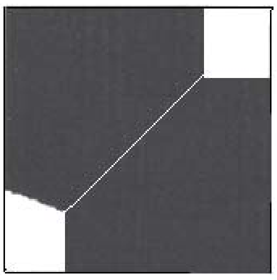


Figure 3: Computed area of bi-directional search.

A year later, Korf and Zhang, extended DCBDA* work and showed that unidirectional search with forbidden operator can be successfully applied to this problem. The new designed algorithm called Divide and Conquer Frontier Search (DCFA*), which employs the same idea as DCBDA*, i.e forbidden operator and ignorance of CLOSE list, but use unidirectional search instead (from initial state node to goal node). Their analysis indicated that DCFA is better than DCBDA* in terms among number of node search, time and memory usage. Our general understanding to this result is that the closer the search to the goal node, the more accurate the heuristic function and the smaller the search space. Therefore, more extra search work is done at the beginning than that at the end. Figure 2 is borrowed from Aaron Davidson's paper, which displays the computed area of frontier search in two-sequence alignment. It can be seen that most search work is done at the beginning. If the search starts from two directions, as shown in Figure 3, the work will be double. This explain why the performance of DCFA* is better than the DCBA* in MSA problem.

2.2.4 A* with partial expansion

In the same year, 2000, Yoshizumi and et al proposed A* with partial expansion to solve MSA (or perhaps general problem with large branching factor) [7]. It is also a variant of A*, which proposed new enhancement to solve the huge branching problem like MSA. The idea is to store only *promising nodes* for finding an optimal solution. To determine a promising node, authors introduce a parameter C, which is a predefined and nonnegative cutoff value, and F-value, which is the priority of node expansion, to A* algorithm. When expanding the best node in the OPEN list, the corresponding children are referred as promising nodes and will be stored in the OPEN list where the f-value of the children node is less than the C value plus the F-value of the expanding node. The F-value of a node is equal to f-value at the beginning. If the expanding node contains any unpromising child, it will put back to the OPEN list after expansion and set the F-value to the lowest f-value among its unpromising child nodes. Therefore, when C is equal to infinite the algorithm becomes A*, but unlike A* expanding in the order of f-value, the introduced parameter C and F-value decides the priority of nodes for expansion.

3. Objectives

The conflict accusation from [5] and [7] attracts our interest. In 1999, [5] demonstrates that the forbidden operator allows A* only storing the OPEN list. Korf believed that using this technique could overcome the memory problem of A* because he concluded that the size of CLOSE list was much larger than open list when applying to the MSA problem. Nevertheless, [7] argued that the size of open list was much larger than the CLOSE list and forbidden operator could not prevent search from leaking back into CLOSE region.

Because both believes do not follow with any reasoning, the goal of our project is to justify the following two questions.

(1) Is there any node revisiting in the CLOSE region?

(2) Is the size of OPEN list larger than the CLOSE list?

The next objective of our project is to evaluate the existing heuristic search algorithms that target on MSA problem. We want to see which one is much better than the other in the general case.

4. Implementation Issues

We have implemented two main algorithms, DCFA* and PEA*, which are variants of A*. When we set the C value of PEA* to infinity, it eventually becomes A*. Because A* is the basic building block of these two algorithms, we will discuss the main implementation issues in A* in this section.

4.1 A Heuristic Search (A*)*

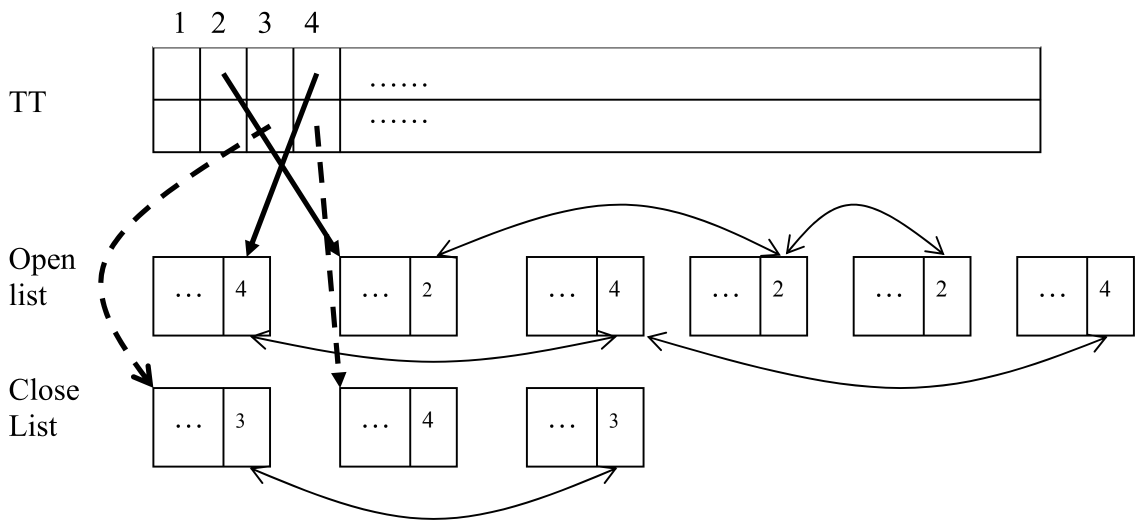


Figure 4: Transposition Table in A*

The most critical issue in the implementation of A* is how to efficiently design the data structure of OPEN list and CLOSE list. An efficient way is to use the heap structure since the complexity of insertion and deletion in a heap is O(log n). However, only using the heap is not efficient enough. When a node is expanding, all its successors ensure that they do not exist in both OPEN list and CLSOE list, which in turn search for both lists. The branching factor of MSA is very large, as explained in the second section, so the search efficiency in list becomes a bottleneck in the MSA program. In our implementation, we incorporated the Transposition Table (TT) into A* algorithm. Each entry of TT saves two pointers. One pointer stores the address of a node in the open list and the other pointer saves the address of a node in the close list in which both pointers point at the first visited node with the same hash key. If there are collisions in a TT entry, nodes with same hash key will be chained together and this is done by adding an extra pointer in each node. Figure 3 illustrates our design. Even though the TT allocates a significant amount of memory, the efficiency of our implementation is improved by order of magnitudes (shown in next section).

Another issue for A* is to determine the heuristics value. As described in section 2.2.1, the h-value estimates the distance from current node to the goal node. If we compute all h-values in real time, there are d(d-1)/2 combination of pair sequences and the computational complexity to compute the optimal distance between two sequences is, causing O(d^2 N^2) time for each node and resulting in high granularity. Hence, we introduce the pre-computation tables. Each pre-computation table stores the h-value of each pair-wise sequence at any given position, and hence it consists of O(d^2) number of tables.

		A	G	T	C
	0	2	4	6	8
C	2	1	3	5	6
G	4	3	1	3	5
A	6	4	3	2	4
G	8	6	4	4	3

(a)

		C	T	G	A
	0	2	4	6	8
G	2	1	3	4	6
A	4	3	3	4	4
G	6	5	4	3	5
C	8	6	6	5	4

(b)

	A	G	T	C	
C	4	5	6	6	8
G	5	3	4	5	6
A	4	3	3	4	4
G	6	4	3	1	2
	8	6	4	2	0

(c)

Figure 5: the evolution of finding the h-value for each node from (a) dynamic approach to compute pair-wise sequence alignment to (b) reversed pair-wise sequence alignment and then to (c) h-value table.

Figure 5(a) shows a simple example to compute the pair-wise optimal sequence alignment using dynamic programming approach. The first impression is that each cell, saying that the position is i-th column and j-th row, computes the optimal alignment distance between two sub-strings, S1[0…i – 1] and S2[0…j – 1]. In other words, it computes the g value of the corresponding node. In order to compute the h-value, then, each cell should compute the optimal alignment distance between S1 [i…N–1] and S2[j…N-1]. In other words, as shown in Figure 5(b), we can compute the optimal sequence alignment for two reversed sequences. Then, by swapping the value of the i-th row and j-th column to (|S1| - i)-th row and (|S2| - j)-th column entry, we may obtain the matrix is shown in Figure 5(c), which gives the h-value. To simplified this process, we implement it using he dynamic programming approach with the initial node locate at |S1|-th row and |S2|-th column entry to the goal node which is 0-th row and column entry. Using this method, the pre-computation stage requires O(d^2N^2) extra memory; however, once the collection of tables is computed, the time needed to calculate the h-value for each node is reduced from O(d^2N^2) to O(d^2).

4.2 Divide-and-Conquer Frontier Search (DCFA)*

The implementation of this algorithm is similar to A*, which also uses transposition table. Since forbidden operator is introduced, we use one bit to represent an operator. All right operators and wrong operators should be saved. Therefore, for a 5-sequences problem, we need 8 bytes for forbidden operators in each node.

4.3 A with Partial Expansion (PEA*)*

The algorithm [7] is published in AAAI-2000, and it also attempts to solve the memory problem of A*. The solution of this algorithm is to save only promising nodes in open list. When a node is being expanded, only promising nodes are generated and this node is still in open list if it has unpromising nodes. The node in open list is ordered by its F value, which is the least value of unpromising children nodes.

4.4 Improved A with Partial Expansion (Improve PEA*)*

There is a drawback of PEA*, which is regarding to use the single C value to differentiate promising nodes and unpromising nodes. In the second expansion of a node, the nodes generated in the first expansion will be generated again since these nodes are still promising nodes. Therefore, if the re-expansion happens frequently, the performance of this algorithm will be decreased largely.

One straight-ward solution is to use a window to separate the children nodes into three sets, which are expanded, promising and unpromising node sets. However, this solution can't get much improvement because it just prevent the duplicate search but doesn't prevent duplicate node generation and duplicate computation of heuristic value. An efficient solution is an efficient design to remember the expanded nodes. Our implementation uses one bit to represent one child node and for a 5-sequences problem, 4 bytes in each node are needed to remember the expanded children nodes.

4.5 Statistical Measurements:

In order to measure the performance, the implementation is incorporated with some statistical variables or functions. Search_Nodes is a variable storing the number of nodes has searched in TT table, and Search_Time is another variable storing the number of time we initiate the TT search. A function Check_Node_Existence in list.C allows us to check whether the fully explore node leaking back to the CLOSE region (comment out for the actual time performance). Similar to [8], we also check the size of OPEN and CLOSE list and the time used to solve each problem.

5. Experiment Results

Our evaluation is base on the experiments done in [8]. We use the same heuristic function as stated in [8]. Each experiment computes the optimal alignment for three strings, and each string is uniformly randomly generated from 20 characters. The length of each string is ranged from 500 to 3000 characters long, and it increases by 500 characters each string for each experiment.

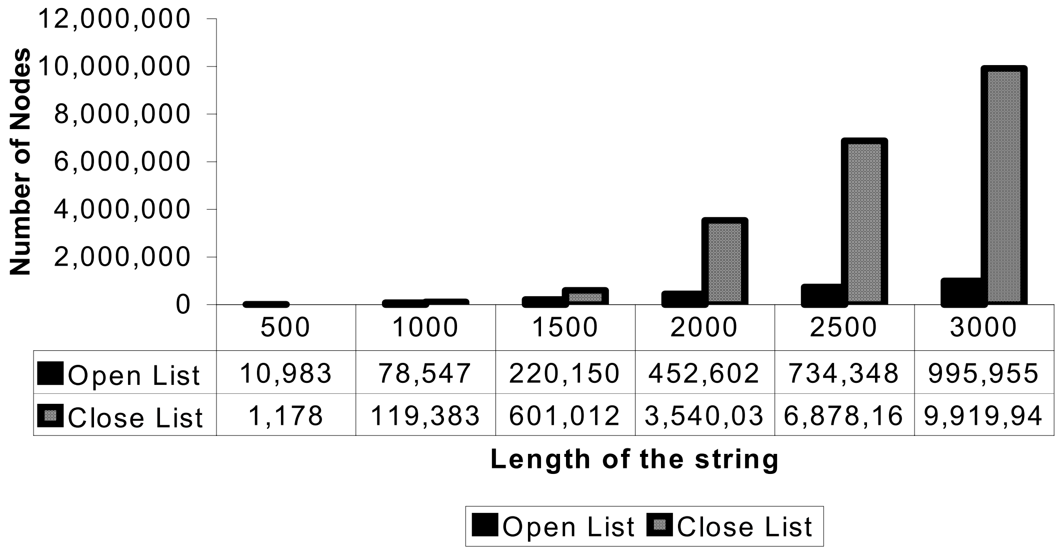
In this section, we will first present the results that follow up the questions we ask in section 3. Then, we will discuss the general comparison of each algorithm, and last but not least we will explain how much improvement we made for the improve PEA* algorithm,

5.1 Is there any node revisiting in the CLOSE region?

From our experiment result, it indicates that the idea of forbidden operator is correct and it successfully prevents the search leaking back into close region. Our test program set the CLOSE list aside and checks the existence of the node to be expanded; however, we do not find any existence of such node leaking back to the CLOSE region.

5.2 Is the size of OPEN list larger than the CLOSE list?

Figure 6 demonstrates the result obtain from A*. Although the size of OPEN list is larger than the CLOSE list when the length of each sequence is 500, when we linear increase the length of sequences, the size of CLOSE list is actually larger than the OPEN list in general. Hence, general speaking, Korf's conclusion in [5] is proved.



	500	1000	1500	2000	2500	3000
■Open List	10,983	78,547	220,150	452,602	734,348	995,955
▨Close List	1,178	119,383	601,012	3,540,03	6,878,16	9,919,94

Figure 6 Size of OPEN list vs. CLOSE List solving by A*

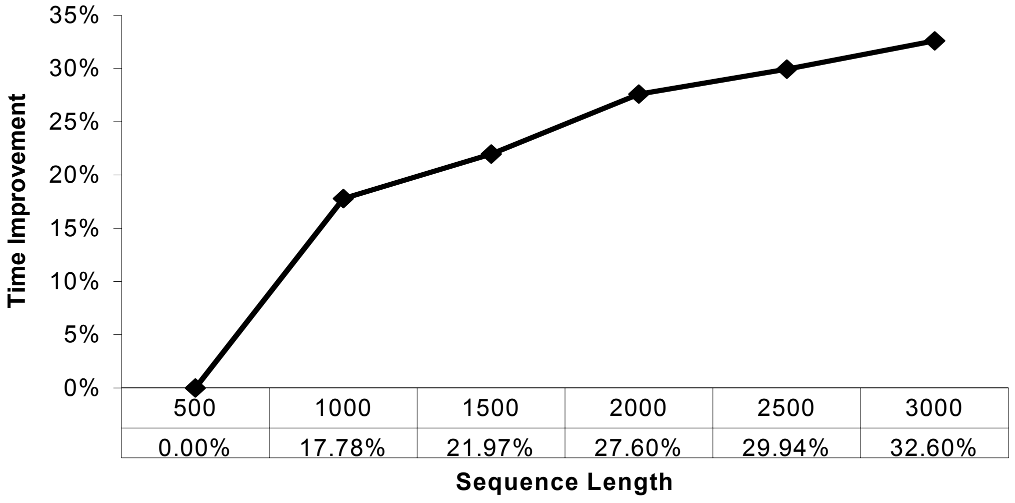
5.3 Evaluation of Different Algorithms:

	A*		DCFA*		PEA*(C=0)		Improved PEA*(C=0)	
Sequence Length	Nodes	Running Time (Sec)	Nodes	Running Time (Sec)	Nodes	Runnin g Time (Secs)	Nodes	Running Time (Secs)
500	19,476	0.36	71,509	0.58	8,495	0.40	8,495	0.40
1000	292,006	3.30	488,719	3.44	213,461	7.22	213,461	6.13
1500	1,110,200	14.67	1,682,644	12.95	890,050	36.48	890,050	29.91
2000	4,632,028	80.50	6,595,645	65.03	4,180,098	227.03	4,180,098	177.93
2500	8,662,163	168.00	12,411,315	129.35	7,932,095	478.32	7,932,095	368.10
3000	12,414,071	262.00	30,711,483	372.69	11,419,491	737.61	11,419,491	556.25

Table 1: Performance Comparison for 3-sequence alignment.

Table 1 reports the result obtaining from A* DCFA* PEA* and PEA*'s improvement. The nodes columns is actually the size of OPEN list plus CLOSE list, and from all of these tests, the data looks A* is the best algorithm of all in terms of searching time. Nevertheless, our implemented A* can't solve the problem when the sequence length is larger than 3000. DCFA*, on the other hand, spends more time on its A* enhancement; in turn, it results in solving much harder problem. In other words, the current version of our DCFA* program can solve up to 3 strings and each string can be at most 5000 characters. Another observation is that PEA* doesn't search much less nodes than those of A*. Therefore, our implemented PEA* also can't solve the problem with sequence length longer than 3000.The reason is that the data is randomly generated and three strings have little similarity. Of course, we believe that the performance of PEA* will be much better if actual data and distance matrix are tested. But this result shows that the performance of PEA* is not as good as DCFA* in the general domain. DCFA* then is the one that can solve a general path finding problem with large amount of short cycle in the grid.

5.4 PEA Improvement*



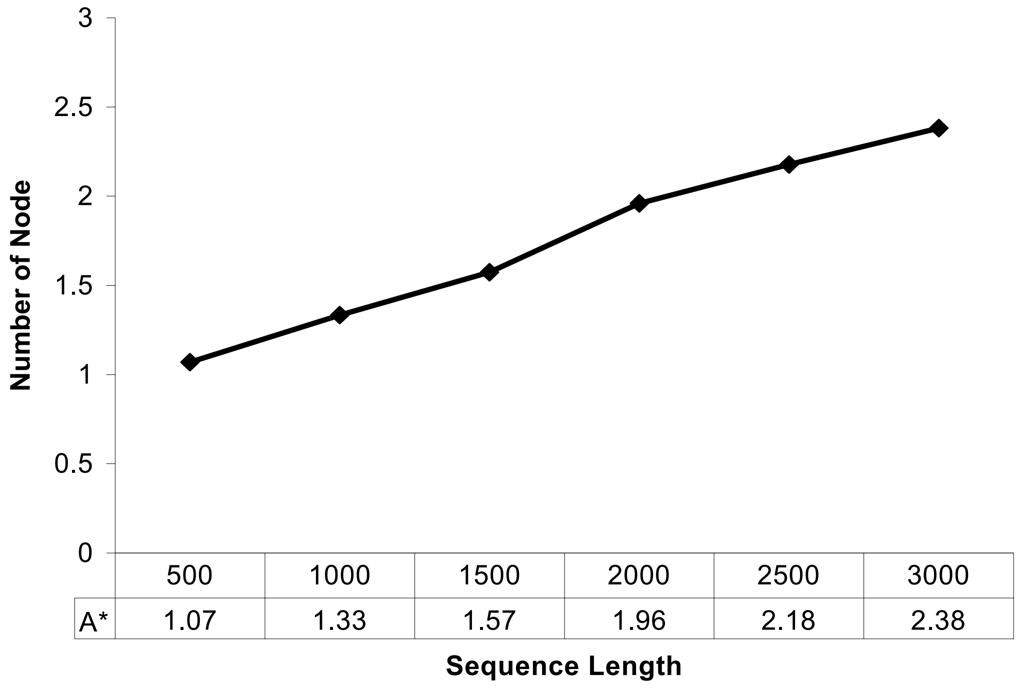
	500	1000	1500	2000	2500	3000
	0.00%	17.78%	21.97%	27.60%	29.94%	32.60%

Figure 7: Improvement of our version

After looking at three different algorithms, we will focus on our improvement in partial A*. From Table 1, we may look at the actual value. The actual improvement is referred to the time efficiency. Looking at Figure 7, it compares the PEA* with our improved version. It indicates that when the sequence length increases the computation time reduction can reach 32.60%. If the real score matrix is used, the time reduction can be more significant because the re-expansion would happen more frequently.

5.5 Transposition Table Performance

The last result we obtain is the performance of transposition table. Figure 8 indicates the average number of node need to be accessed in order to check the existence of a new generated node in the list. As the length increase by 500, the average number of accessed node in one search is less than 3 attempts. Thus, the performance of the transposition table is acceptable to as an enhancement in A* for this problem.



	500	1000	1500	2000	2500	3000
A*	1.07	1.33	1.57	1.96	2.18	2.38

Figure 8: Average Number of nodes need to search in the TT in order to obtain the result.

6. Conclusion

In this project, we study several algorithms in artificial intelligence to solve the multiple sequence alignment, a problem from the computational biology, and we implement two interesting algorithms published in 2000, divide and conquer frontier search and A* with partial expansion. Our final conclusion is that DCFA* can solve the problem in a general area while PEA* may perform better when similar data are used.

7. Future Works

Current algorithm solving hard MSA problem requires exponential time and space. This project summarizes some techniques to reduce the time complexity as well as memory usage. However, to solve the problem with exponentially increase number of branching factor is also a challenging work in our future.

8. Acknowledge

We gratefully give our thanks to Matthew McNaughton and Dr. Jonathan Schaeffer for their creative comments.

9. References:

[1] S.B. Needleman and C.D. Wunsch, “A General Method Applicable to the Search for Similarities in the Amino Acid Sequences of Two Proteins”, Journal of Molecular Biology, Vol. 48, 1970, pp. 443-453.

[2] D.S. Hirschberg, “A Linear Space Algorithm for Computing Maximal Common Subsequences”, Communication of the ACM, Vol. 18, No. 6, June, 1975, pp. 341-343.

[3] J.L. Spogue, “Speeding up Dynamic Programming Algorithms for Finding Optimal Lattice Paths”, SIAM Journal of Applied Math, Vol. 49, No. 5, 1989, pp. 1552-1566.

[4] E. Ukkonen, “ Algorithms for approximate string matching”, Information and Control, Vol. 64, 1985, pp. 100-118.

[5] R.E. Korf , “Divide and Conquer Bidirectional Search: First Results”, Proc. of IJCAI-99, Stockholm, Sweden, August 1999, pp. 1184-1189.

[6] A. Davidson, “A Fast Pruning Algorithm for Optimal Sequence Alignment”, Proc. BIBE’2001, November 2001.

[7] T. Yoshizumi, T. Miura, and T. Ishida, “A* with partial expansion for large branching factor problems”, AAAI-00, pp.923-929.

[8] R.E., Korf, W. Zhang, “Divide-and-Conquer Frontier Search Applied to Optimal Sequence Alignment”, AAAI National Conference, 910-916, 2000.