2022310853 박연우

알고리즘개론 Assignment 2

< k DNA Sequence Alignment >

1. Objective

The objective of this assignment was to implement an efficient algorithm to implement multiple sequence alignment (MSA) with k DNA sequences. The DNA sequence is a string formed with four alphabets – A, T, G, C – with length ranging from 1 to 120. The longest common subsequence (LCS) of the given DNA strings was to be found, and the DNA sequences were to be aligned with gaps in between to make all the sequences the same length. Asterisks were printed below the aligned DNA subsequences to show which macromolecule is part of the LCS.

2. Implementation

For an LCS problem, the solutions of the subproblems are part of the final solution. By taking advantage of its optimal substructure and overlapping subproblems, we can incorporate memoization (i.e. dynamic programming) to solve for the LCS. Therefore, for a given k-sequences, a k-dimensional table is created and stores the solution to the related sub-problems. By filling in the table, the length of the LCS is retrieved at the last element of the table. Using the table and the length of the LCS, we can backtrack and find the actual sequence of the LCS.

Two mainly different methods were used to solve this problem: first being the conventional method of dynamic programming using a tabulation (explained in our class's lecture), and second being a dynamic programming method using indexes and taking advantage of the limited set of characters given. For instances, in which $k = 2$ or $k = 3$, the former method was applied, and when k=4 and k=5, the latter method was applied. I will refer to the former method as the conventional method, and the latter method as the index method from now on.

Different methods were implemented for different number of input sequences because while the conventional dynamic programming method is an effective method to find the LCS, it encounters a problem when k or the length of the sequences (n) becomes too large. In order to implement tabulation, a k-dimensional array has to be created for k sequences, which can become exponentially large and thus, impossible to create. Hence, a second method – the index method – was applied.

The index method was implemented based on an article called "Computing a Longest Common Subsequence for a Set of Strings"[1]. In this method, a 4 dimensional array is created to store the index values at which each letter exists (refer to the article for a better explanation). First each sequence has its own set of arrays saving each of its letter's indexes. I will use the example sequences given in the assignment. For the three sequences, their indexes for each letter will be as shown in <Table 1> below.

[1] HSU, W.J. "Computing a Longest Common Subsequence for a Set of Strings - NCTU." *COMPUTING A LONGEST COMMON SUBSEQUENCE FOR A SET OF STRINGS* , 1984, ir.nctu.edu.tw/bitstream/11536/4866/1/A1984SR65400005.pdf.

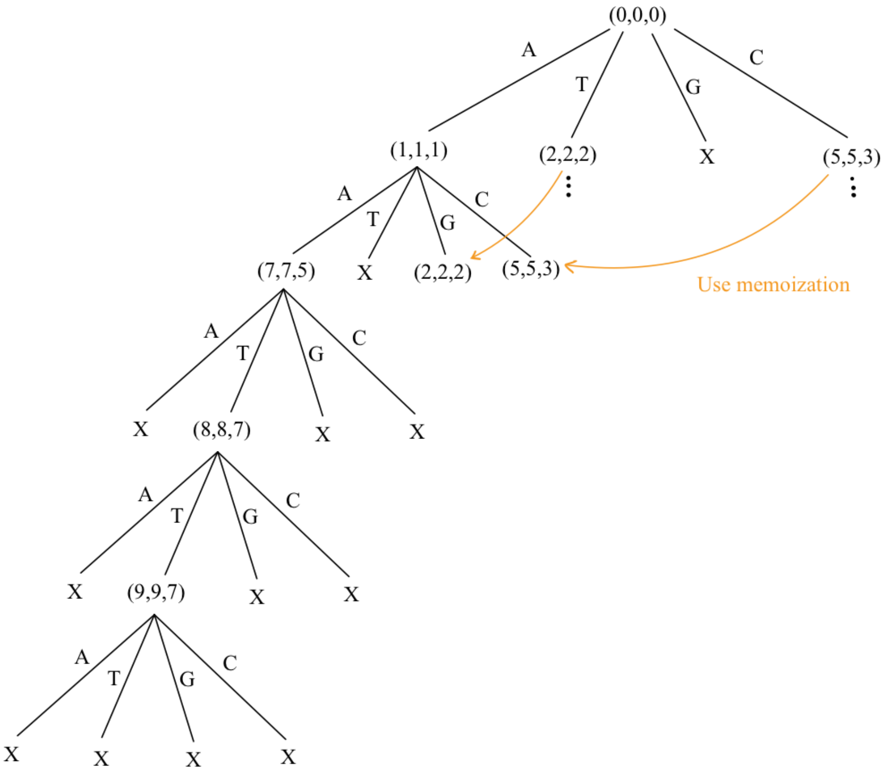
Sequence	Indexes
"ATTGCCATT"	**A**: [1, 7, 7, 7, 7, 7, 7, -1, -1] **T**: [2, 2, 3, 8, 8, 8, 8, 8, 9] **G**: [4, 4, 4, 4, -1. -1, -1, -1, -1] **C**: [5, 5, 5, 5, 5, 6, -1, -1, -1]
"ATGGCCATT"	**A**: [1, 7, 7, 7, 7, 7, 7, -1, -1] **T**: [2, 2, 3, 8, 8, 8, 8, 8, 9] **G**: [3, 3, 3, 4, -1. -1, -1, -1, -1] **C**: [5, 5, 5, 5, 5, 6, -1, -1, -1]
"ATCCAAT"	**A**: [1, 5, 5, 5, 5, 6, -1] **T**: [2, 2, 7, 7, 7, 7, 7] **G**: [ ] **C**: [3, 3, 3, 4, -1, -1, -1]

<Table 1: Indexing of Sequences>

For the first sequence ("ATTGCCATT"), A's index array is [1, 7, 7, 7, 7, 7, 7, -1, -1] because A first appears at the beginning of the sequence. The next instance where A appears is at index 7. Hence, the array is filled with 7 until the next "A" appears, which is at index 7. If the letter no longer appears in the sequence, the rest are filled with -1.

Using these index values, we are able to understand where the common character values are at each index. If three sequences are given, we start with each of their indexes at (0, 0, 0). At (0, 0, 0), all three sequences' A indexes are at 1. Hence, we move to (1, 1, 1). For T, all three sequences' indexes are at 2, and thus, we move to (2, 2, 2). Since there are no letter G's in the third sequence, there will not be a branch. Lastly, at (0, 0, 0), C's indexes are (5, 5, 3) for sequences 1, 2, and 3 respectively. For each index, we compare the indexes of each letter. If any sequences' index is equal to -1, it implies that the sequence has ended, ultimately ending the branch as well. The next index must always be larger than the current index to ensure that we are moving along the sequence. Follow <Image 1> on the next page to observe how the indexes can be compared with one another. (Branches that end are marked with X, and branches that are not fully drawn are marked by three dots).

The index values are stored in a 4th dimensional array, and the table is used for memoization. For when *k*=5, the memoization table further expands by having nodes append to this table in a linked list manner. This modification was put into place in order to avoid complications of running out of memory.



<Image 1: visualization of the index method>

As exemplified with the example sequences, the index method works perfectly fine for $k = 2$ and $k = 3$. However, the conventional method had a faster running time, and hence, I chose to implement different methods for different values of k. For $k = 4$ and 5, the index method had a better performance, and thus, the index method was chosen.

3. Performance Analysis

The running time of the conventional method is in polynomial time when given a constant number of sequences. Since k is fixed from a range of 2 to 5 and n is also fixed from 0 to 120, polynomial time is given[2].

The run time of the conventional method can be expressed as the product of the number of subproblems and the work per subproblem. Since the number of subproblems is equal to $\Pi(length\ of\ sequence\ +\ 1)$ and the work per subproblem is O(1), the total running time is $O(\Pi(length\ of\ sequence\ +\ 1))$.

Since the conventional method is only applied for cases in which $k = 2$ and 3, the time complexities for these two instances will be as shown in <Table 2>.

[2] Demaine, Erik. "6.006 Introduction to Algorithms, Lecture 16: Dynamic Programming Subproblems: Introduction to Algorithms: Electrical Engineering and Computer Science." *MIT OpenCourseWare*, MIT OpenCourseWare, 2020, ocw.mit.edu/courses/6-006-introduction-to-algorithms-spring-2020/resources/mit6_006s20_lec16/.

Situations, in which k = 4 or 5, the time complexity appears a little differently because a different algorithm is in place. (This can also be found in the article mentioned previously).

First, a pre-processing time is required in order to create the index arrays for each sequence. Since the array size will depend on the length of the sequence and there are only 4 letters, the pre-processing time can be given in O($L \cdot 4$), where L = $\Sigma(length\ of\ sequence)$.

Next, the time complexity of the processing time can be found by looking at the time complexity of each function:

1. **node_matches_table_val function:** This function is only called for $k = 5$. It has a while loop that traverses a linked list. In the worst case, this function has a time complexity of O(n), where n is the length of the linked list.

2. **calc_lcs_length function**: This function recursively finds the length of the LCS, which contributes most to the time complexity of the processing time. The time complexity can be expressed as O($k \cdot n \cdot 4 \cdot P$), where P denotes the number of matches, k denotes the number of strings given, and n denotes the maximum length of the sequences.

3. **traceback function:** This function contains a loop that runs for lcs_length iterations. Hence, the function is bound to O(n·m) time, where n is the length of the linked list and m is the lcs_length.

4. **find_lcs function:** In this function, the memoization table is created which will take O(n^3), where n is the maximum length of the sequences. However, this function mainly calls calc_lcs_length and traceback. Therefore, its time complexity is dominated by the larger of the two, which is O($k \cdot n \cdot 4 \cdot P$).

The overall time complexity of the index method takes O($L \cdot 4$) pre-processing time and O($K \cdot 4 \cdot P$). When $k = 5$, O(n) must be taken into account because of having to traverse the linked list. Hence, its time complexity will become O($k \cdot n^2 \cdot 4 \cdot P$).

To summarize, the time complexities for each value of k is as shown in <Table 2>. (Let the length of each sequence equal to A, B, C, D, and E respectively, P = number of matches, n = maximum length of the sequence, k = number of sequences given)

***k* =**	**2**	**3**	**4**	**5**
Time	O(A·B)	O(A·B·C)	O($k \cdot n \cdot 4 \cdot P$)	O($k \cdot n^2 \cdot 4 \cdot P$)

<Table 2: Time Complexity for each k>

4. Code Explanation

The basic outline of my code will be written in this section.

Define:

#define MAX_LENGTH 150

- Defines the maximum length of a sequence n. (Although n ≤ 120, it was set to 150 just in case).

#define MAX_SEQUENCES 5

- Defines the maximum number of sequences *k*

Structures:

This structure shown below stores the dna_sequences. The sequence stores the DNA sequence string, and the length stores the length of the given sequence. The int double array contains each letter's indexes. A one-dimensional array of DNA_SEQ will be used to store all sequences.

```
typedef struct dna_seq{
    char* sequence;
    int length;
    int** letter; // stores the letters of each letter's indexes (0 = A, 1 = T, 2 = G, 3 = C)
} DNA_SEQ;
```

This structure shown below is a node which composes the 4D array used in the index method. The str_index double array contains the indexes we are looking at as well as the indexes for the node after it. The lcs_length stores the length of the LCS and next_node indicates the next node connected to this node.

```
typedef struct node{
    int* str_indexes[2]; //stores indexes of each letter for each sequence and its next one
    int lcs_length;
    struct node* next_node;
}NODE;
```

Functions:

The functions implemented in this code can be largely divided into three parts: the main function, LCS functions, and helper functions.

1. **Main function:**
   The main function makes calls to the other functions, such as taking in the input file and writing the final output. Depending on the number of inputs, different functions to find the LCS are called.

2. **LCS functions:**

char* find_lcs2(DNA_SEQ* dna, int string_num);

- Calculates and retrieves the LCS when given two sequences. A 2 dimensional array is created and used for tabulation. Using the values in the table, the function backtracks to find the LCS.

char* find_lcs3(DNA_SEQ* dna, int string_num);

- Calculates and retrieves the LCS when given three sequences. A 3 dimensional array is created and used for tabulation. Using the values in the table, the function backtracks to find the LCS.

int calc_lcs_length4(DNA_SEQ* dna, int string_num, NODE*** table, int* indexes);**

- Recursively calculates the length of the LCS when given four sequences.
- If any of the values in the indexes array are equal to -1, it means that the string has ended, thus it returns a length of 0.
- If the index already exists in the table (which is used for memoization), then we simply use the lcs_length value from the table.
- If the index does not exist in the table, a new node is created and appended to the table. As the node is saved, its lcs_length will be the maximum of the four branches it can take plus 1 (the four branches are each branching out to A, G, T, C respectively). The index of the next node will also be saved since it will come handy when tracing back the table to find the actual sequence of the LCS.

char* traceback4(DNA_SEQ* dna, int string_num, int lcs_length, NODE*** table);**

- Using the table, it traces back the LCS to retrieve the actual sequence of the LCS.
- We begin at the root node, which is (0, 0, 0, 0) of the table. Then, we look at the node following the root by using the previously saved index values of the next node (specifically stored in str_indexes[1]). We follow each node in such a manner and append to the LCS, until all the LCS is filled. Finally, return the full LCS.

char* find_lcs4(DNA_SEQ* dna, int string_num);

- This function merely puts together the calc_lcs_length4 and traceback4 functions together. It also initializes the memoization table. This function is called in the main function when four sequences are given.

int node_matches_table_val(int indexes[5], NODE*** table, NODE** current_node);**

- Checks if the current_node exists in the table or not by comparing the str_indexes of the current_node and the indexes passed to the function.

int calc_lcs_length5(DNA_SEQ* dna, int string_num, NODE*** table, int* indexes);**

- Recursively calculates the length of the LCS when given five sequences.
- If any of the values in the indexes array are equal to -1, it means that the string has ended, thus it returns a length of 0.

- If the index already exists in the table (which is used for memoization), then we simply use the lcs_length value from the table.
- If the index does not exist in the table, a new node is created and appended to the table or linked to the table. As the node is saved, its lcs_length will be the maximum of the four branches it can take plus 1 (the four branches are each branching out to A, G, T, C respectively). The index of the next node will also be saved since it will come handy when tracing back the table to find the actual sequence of the LCS.

char* traceback5(DNA_SEQ* dna, int string_num, int lcs_length, NODE*** table);**

- Using the table, it traces back the LCS to retrieve the actual sequence of the LCS.
- We begin at the root node, which is (0, 0, 0, 0, 0) of the table. Then, we look at the node following the root by using the previously saved index values of the next node (specifically stored in str_indexes[1]). We follow each node in such a manner and append to the LCS, until all the LCS is filled. Finally, return the full LCS.

char* find_lcs5(DNA_SEQ* dna, int string_num);

- This function merely puts together the calc_lcs_length5 and traceback5 functions together. It also initializes the memoization table. This function is called in the main function when five sequences are given.

3. **Helper functions:**

void take_input(DNA_SEQ* dna, int* string_num)

- This function takes in the sequences of the given input file. It also initializes the letter array by counting all the indexes of each letter.

int is_common(DNA_SEQ* dna, int size, char* lcs, int lcs_index, int indexes[5])

- This is a helper function used to print out the final results. It helps check if at a given index, all the sequences have the same character.

void write_final_results(DNA_SEQ* dna, int size, char* lcs)

- By comparing the given sequences and the LCS found, the function aligns all the sequences and marks the LCS with asterisks. It writes the final output to the file named “h2_output.txt”.

5. Conclusion

The objective of this assignment was to implement an efficient algorithm to implement multiple sequence alignment with *k* DNA sequences. The DNA sequence is a string formed with four alphabets – A, T, G, C – with length ranging from 1 to 120. By using two different methods, the MSA problem was able to be solved in an efficient manner.

6. Work Cited

[1] HSU, W.J. “Computing a Longest Common Subsequence for a Set of Strings - NCTU.” *COMPUTING A LONGEST COMMON SUBSEQUENCE FOR A SET OF STRINGS* , 1984, ir.nctu.edu.tw/bitstream/11536/4866/1/A1984SR65400005.pdf.

[2] Demaine, Erik. “6.006 Introduction to Algorithms, Lecture 16: Dynamic Programming Subproblems: Introduction to Algorithms: Electrical Engineering and Computer Science.” MIT OpenCourseWare, MIT OpenCourseWare, 2020, ocw.mit.edu/courses/6-006-introduction-to-algorithms-spring-2020/resources/mit6_006s20_lec16/.