An Approach for Improving Complexity of Longest Common Subsequence Problems using Queue and Divide-and-Conquer Method

Ratul Bhowmick, Md Ibrahim Sadek Bhuiyan, Md. Sabir Hossain, Muhammad Kamal Hossen
Department of Computer Science and Engineering
Chittagong University of Engineering and Technology (CUET)
Chittagong-4349, Bangladesh
Email: u1604038@student.cuet.ac.bd, u1604036@student.cuet.ac.bd, sabir.cse@cuet.ac.bd, kamalcsecuet@gmail.com

Ahsan Sadee Tanim
Department of Computer Science and Engineering
The International University of Scholars
Dhaka-1212, Bangladesh
Email: ahsansadeecuet@gmail.com

***Abstract*—The general algorithms which are followed to solve the Longest Common Subsequence (LCS) problems have both time complexity and space complexity of $O(m * n)$. To reduce this complexity, two ways are proposed in this work. One is the use of Divide and Conquer approach, and another one is changing the data structure from two-dimensional array to queue. By using these approaches, an algorithm has been developed and implemented for the long length string in this paper. The time complexity after using these approaches becomes a function of logarithm $O(n * log(n))$ which has been shown with proper explanation and the space complexity after using queue data structure becomes linear $O(n)$.**

Keywords—Longest Common Subsequence, Queue, Divide and conquer, Dynamic Programming, Recursion

I. Introduction

Longest Common Subsequence (LCS) is a popular algorithm to find the longest subsequence which is a common group of sequences. Normally, two sequences are given and from that, LCS has been found out. A subsequence is gained from another sequence by removing some elements but can't change the series of the other elements. There are many other algorithms to find common subsequence. Most of them follow the greedy approach. But LCS follows the dynamic approach. It can be decided in polynomial time.

The general LCS algorithm takes O(m*n) time. In this paper, there has been proposed an algorithm which finds the solution by following an optimal infrastructure. The problem which is given is broken into subproblems which is smaller and simple, which will be broken into yet further smaller subproblems until we find the final answer. For this approach, time and space complexity is optimal. The time complexity becomes $O(n * logn)$ for the best case and $O(n * \log(n)^2)$ for the average case. But the space complexity becomes linear by using data structure queue instead of two-dimensional array and it is $O(n)$in both best and worst cases. Although this algorithm is not unique, it can become really complex for a very bigger length. But for most of the cases of finding common subsequence, it finds the solution in optimum time.

Many researchers have researched about LCS over a period of time. They have given many results. But still it is very complex and there is a huge area left to research on this topic. The time and space needed to solve a problem by using LCS are still high. The main objectives of this research are:

1. To reduce the time complexity and find an optimum a bigger length string.
2. To reduce the space complexity by changing the data structure.
3. To use a new approach (Divide and Conquer) and to see the effects of using this approach.

The remainder of this paper is organized as follows. Section II provides a brief review of related work. In section III, we discuss in detail the proposed algorithm. Section IV contains experimental results and analysis. Finally, we conclude and sketch future research directions in Section V.

II. Related Research

J. Liu and S. Wu show that it is possible to find out the Longest Common Subsequence in linear time using a 2D array [1]. The result lies between in the diagonal line of the two-dimensional array. They used for them theorems provided in [2], [3]. Paper [3] has also shown that the space complexity can be linear in the longest common subsequence. This paper merges two algorithms to reduce the space complexity.

K. Kwarciak and P. Formanowicz propose a DNA sequencing technique using a greedy approach. In their algorithm, the greedy method is applied to two different sequences at the same time. That's why this greedy algorithm is much faster than the dynamic approach. This approach can be used as the longest common subsequence. J. F. Myoupo and D. Semé used a parallel algorithm to minimize time complexity [5]. This algorithm runs in O(logN*logM) which is much smaller than the existing LCS algorithm. In [6], this problem is solved in O(log m*log n) with O(n*m/log m) processors. The CRCW bound is O(log(n(log*log m)2)) time where the processors are O(m*n/log* log m).

In paper [7], X. Tang et al. used a block system by which the process become faster than before. The idea of a B+ tree and O tree is used in this algorithm which has less run time than other algorithms. For that reason, they get a result which takes only 36.5 mm^s. Paper [8] proposes a data-structure that helps to improve the competence of suffix array in the algorithm by storing in external memory model. The authors of this research paper also propose and implement two parallel algorithms (Index Partitioning and

Data Partitioning) on a PC cluster. Here the ASM problem is solved in DNA sequences with the help of suffix array algorithms.

R. L. Goldfeder et al. proposes a system where the DNA sequence/String is divided into some small pieces, and it will be reordered programmatically such that it can enable fast and efficient sequencing for human genomes or for a string [9]. In this paper, they divided the DNA or String sequence in a small portion and get all data and then merged.

Paper [10] presents real-time sequencing and single molecule data that is obtained from a DNA polymerase. DNA polymerase is performed in a continuous template directed fusion using four different fluorescently categorized dNTPs. The data is reported directly used to reveal the distinct polymerization states corresponding to DNA structure. The paper shows 90% accuracy in medical science. F. Chin and C.K. Poon demanded that there has L-matrix on the dynamic approach in LCS [11]. They showed the time complexity becomes O(ns+min(ds, lm)) and the space complexity becomes O(ns+d). They used different data structures to obtain variations of the basic algorithm.

III. The Proposed Algorithm

In the proposed algorithm, the existing Longest Common Subsequence algorithm has just modified by using the queue data structure and the divide-and-conquer method. The existing general LCS algorithm works on $O(MN)$ time. It can be made O(1) if the size of those strings is minimized. Here this idea is used. Let, two string A(A1, A2,….., An) and B(B1, B2,….Bn) are given and C(C1, C2,….., C k) is the sequence of two string. If A >= B, then this algorithm will divide the A string into small parts until the size of the string becomes less than 4. Then these small string and the B string will use in the second process. From the algorithm in Fig. 2the largest string is divided into several small strings (as shown in Fig.3). The small string and the second string are applied in the second algorithm which is given in Fig. 3. The whole algorithm runs in O(n*log(n)) time in the best case. This algorithm takes linear space because of using the queue instead of a two-dimensional array.

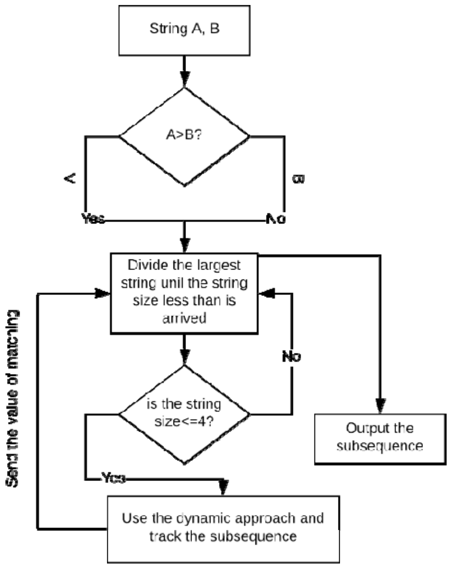
String A, B
A>B?
<
B
Yes
No
Divide the largest string unil the string size less than is arrived
No
is the string size<=4?
Yes
Output the subsequence
Use the dynamic approach and track the subsequence
Send the value of matching

Fig.1 The proposed methodology

From Fig.1, it is clear that there have two steps. The first step is the divide-and-conquer method and the second step is DP.

A. Algorithm

In this algorithm, we have two phases. In the first phase, the large string will be made into a small string by the divide and conquer method. In the algorithm in Fig. 2, the algorithm of the first phase is given.

```
Function making_a_short_string( start , end)
1.  If(start – end <= 4)
2.      return LCS( A[ start … end] , B [ 0 …m])
3.  end
4.  mid = (start+end)/2
5.      return making_a_short_string(start, mid) +
    making_a_short_string(mid+1, end);
End
```

Fig.2 Pseudo code describing the divide-and-conquer method to short the large string

From string A, when the base case arrives in the first phase, it will execute the second algorithm which is given in Fig. 3. From this algorithm, we can get the sequence.

```
Function LCS(s1[ 0…n] , s2[0 …m])
                    Bool Array[0………m)
1.  Declare a queue data structure Q;
2.  matching_number <- 0
3.  for i:= 0 to n
4.    for j:= temporary to m
5.       if i == 0 or j == 0
6.          if i == 0
7.    Q<- 0
8.          End
9.          if j == 0
10.             Q <- 0
11.         End
12.      End
13.      Else
14.         if s1[i-1] == s2[j-1]
15.             diag <- Q.front and pop the front value
16.             previous_value <- diag+1
17.             Q <- previous_value
18.                If prevous_value> matching_number
19.                    temporary <- j
20.                    Array[j]<- true
21.                    Matching_number = temporary
22.                    If j == n-1 return Matching_number
23.                End
24.         End
25.         Else
26.       diag <- Q.front
27.              Pop the front value of Q
28.             Vertical_value <- Q.front
29.             Temp = max( previous_value,
    Vertical_value)
30.             Previous_value <- temp
31.             Q <- previous_value
32.         End
```

```
33.         If j == m
34.            Pop the front value of the queue
35.         end
36.         end
37.      end
38.    end
39.    Return matching_number
40.  End
```

Fig. 3 Pseudo Code describing the algorithm of LCS

From this algorithm, we can get an array "Array" which track the sequence. If we iterate through the "Array", we will able to get the longest common subsequence C [0….m] which is "Array".

B. Example

Let Two string A= "abcadbcadadcaaddaddcdcddcdadccdaa ....aaddaada" and B = "abdcad" and C is the sequence where C is equal the size of the min (|A|, |B|). Here B is small so the size of |C| = |B|Here |A|>|B|, so we will divide A into many small strings by algorithm which is shown in the Fig. 4. From the best case of the string, the Strings will go to the next algorithm, where the small string and the new string will go through the algorithm in Fig. 3.

Let the best case string from the algorithm is "abca" and the B string is "abdcad". Then the process will get an LCS between those two strings where the two strings are small so it will run in constant time. Then the subsequence can get by Fig. 3, when i is zero means at the first row of the algorithm, then the queue will be 0 0 0 0 0, then the second loop the first element will be zero then the queue will be 0 0 0 0 0 0, then if the i-th element and the j-th element is similar then the front value of queue will be incremented and it will be popped out and pushed in the queue, then the queue will be 0 0 0 0 0 1. Then if the i-th and j-th are not similar then it will get from the previous value and the vertical value.

To get the vertical value firstly popped out the first value of the queue then take the new front value. Then compare which is maximum and push it to the queue. Then the queue will be, 0 0 0 0 1 1 by continuing this process after the second-row finishes then the queue will look like 0 1 1 1 1. Here the queue is using, again and again, the first value is popped and the new value is inserted. Lastly, the queue will be 0 1 2 3 4. Then it will return the value 4. During this method, a temporary "Array" is declared which will track the elements which are the subsequence of both strings.

Strings will go to the next algorithm, where the small string and the new string will go through the algorithm in Fig. 3. Let the best case string from the algorithm is "abca" and the B string is "abdcad". Then the process will get an LCS between those two strings where the two strings are small so it will run in constant time. When i is zero means at the first row of the algorithm then the queue will be 0 0 0 0 0, then the second loop the first element will be zero then the queue will be 0 0 0 0 0 0, then if the i-th element and the j-th element is similar then the front value of queue will be incremented and it will be popped out and pushed in the queue, then the queue will be 0 0 0 0 0 1. Then if the i-th and j-th are not similar then it will get from the previous value and the vertical value. To get the vertical value firstly popped out the first value of the queue then take the new front value. Then compare which is maximum and push it to the queue. Then the queue will be 0 0 0 0 1 1. By continuing this process, after the second-row finish then the queue will look like 0 1 1 1 1.

Here the queue is using again and again, the first value is popped and the new value is inserted. Lastly, the queue will be 0 1 2 3 4. Then it will return the value 4. During this method, a temporary "Array" is declared which will track the elements which are the subsequence of both strings.

IV. Experimental Result & Discussion

The time complexity of this proposed method is n*log(n) and the space complexity is O(n). The process is given in the next sections.

A. Time Complexity

Let, two strings are A[0,….k] and B [0,…..m] and n is the size of min(|A|, |B|). The algorithm has two parts. One is for the divide and conquers method which is recursive and one is a dynamic approach method. Let the time complexity of the dynamic approach is T1(c) and the time complexity of the total algorithm is T(n).

$$
\begin{aligned}
\text{Then, } T(n) &= T1(c) + 2T(n/2) \\
&= T1(c) + 2\{\, 2T(n/4) + T1(n/2) \,\} \\
&= 2T1(c) + 4T(n/4) \\
&= 2T1(c) + 4\{\, 2T(n/8) + T1(n/4) \,\} \\
&= 3T1(c) + 8T(n/8) \\
&\text{------------------------} \\
&\text{------------------------} \\
&= kT1(c) + 2\hat{}k\ T(n/2\hat{}k) \text{ -------- (i)}
\end{aligned}
$$

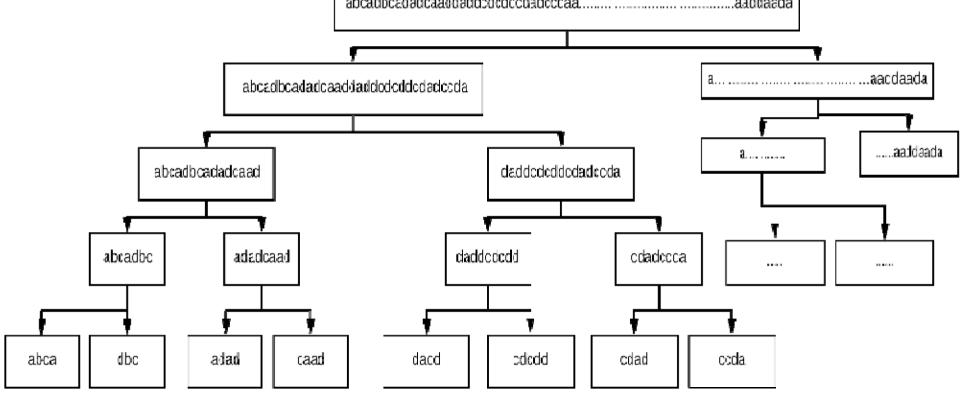
abcadbcadadcaaddaddcdcdccdadcccaa........ ....aaddaada
abcadbcadadcaaddaddcdcddcdadccda
a... ....aacdaada
abcadbcadadcaad
daddcdcddcdadccda
a.........
......aaddaada
abcadbc
adadcaad
daddcdcdd
cdadccca
.....
......
abca
dbc
adad
caad
dacd
cdcdd
cdad
ccda

Fig. 4 Divide-and-Conquer method of the largest string

Let, n/2^k = 1

⇨ n = 2^k

⇨ log(n) = k [log base 2]

Substituting k in (i),

T(n) = log(n)T1(c) + n*log(n) -------- (ii)

So, the general time complexity is log(n)T1(c) + n*log(n).The best, worst and the average case is depends on the T1(n) term.

B. Best Case Analysis

From section VI A, the general time complexity is log(n)T1(c) + n*log(n). Let the new sting D and the second string is B. If the new string is the proper subset of the second string (D ⊂ B), then the T1(c) will the constant time. Because all the elements of D are within in B, and the size of the D is less than 4, then the second algorithm will run on constant time. So, T1(c) will be 1.

From equation (ii),T(n) = log(n)*1 + n*log(n)

So, the best case of this algorithm is n*log(n) where n is the size of min (|A|, |B|).

C. Average Case Analysis

The Average case of this algorithm is similar to the best case of the algorithm. Let the new sting D and the second string is B. In the average case, D is the subset of B, means that D has some elements which are in the B. So every recursive method the size of B is reducing, for that reason the algorithm has to go through the size of the B string. The second algorithm will run on linear time. So, T1(c) will be m.

From equation (ii),T(n) = log(n)*m + n*log(n)

So, the best case of this algorithm is k*log(n) where n is the size of min(|A|, |B|) and k is the size of max(|A|, |B|)

D. Worse Case Analysis

The general time complexity is log(n)T1(n) + n*log(n). Let the new sting D and the second string is B. If D' is the proper subset of B means that there has no element in D which is in B. So, it will iterate through all the elements in D so the algorithm will run on O(k*m) where the size of the D is k and the B is m. So, the worst case time complexity will be T(n) = log(n)* k*m + n*log(n) where k is log(n)/2.

Then, T(n) = m*log(n)^2 + n*log(n). So, the worst case of this algorithm is m*log(n)^2, where n is the size of min(|A|, |B|) and m, is the size of max(|B|,|A|).

E. Space Complexity

The space complexity of the first algorithm is O(n/2+1)~ O(n/2) and the space complexity of the second algorithm is O(n) because instead of a 2D array, there is used a data structure queue. The space complexity of queue for the best and worst case is O(n). So the space complexity of the full algorithm is O(n) means that the program is running on a linear space algorithm.

A comparison between our proposed method and the proposed framework of other authors is given in Table 1. From Table 1, it is clearly seen that our algorithm's space complexity is better than any other of these mentioned algorithms and the time complexity becomes a logarithmic function which is optimal than some of the other algorithms.

TABLE I Comparison of time and space complexity

Algorithm	**Best time complexity**	**Average time complexity**	**Worst time complexity**	**Space complexity**
LCS	Ω(m*n)	Θ(m*n)	O(m*n)	O(m*n)
Ref. [1]	Ω(n*n)	Θ(n*n)	O(n*n)	O(m+n)
Ref. [2]	Ω(n(m-p))	Θ(n(m-p))	O(n*m)	O(m+n)
Ref. [3]	Ω(m*n)	Θ(m*n)	O(m*n)	O(n)
Ref. [11]	Ω(ns+min(ds,lm))	Θ(ns+min(ds,lm))	O(ns+min(ds,lm))	O(n*m)
Our Algorithm	Ω(n*log(n)), n=max(|A|, |B|)	Θ(k*log(n)), k=max(|A|, |B|)	m*log(n)^2 + n*log(n), m=max(|A|, |B|), n=max(|A|, |B|)	O(n)

TABLE II Comparison of time with General LCS algorithm

Length of two string (n and m)	**LCS(sec)**	**Our Algorithm (sec)**
n<100 && m<100	0.932	0.812
n<10000 && m<10000	3.003	2.855

In this paper, the space complexity of LCS is highlighted. The space complexity of this algorithm is linear. The best space complexity of queue is O(n) as well as the space complexity is O(n). Divide-conquer approach has been used here. For this approach, this will take constant time to solve a problem. Later the data structure has been changed from a two-dimensional array to queue. For this, space will be reduced. In Table II it has shown time comparison between general LCS and our algorithm.

Generally, for LCS, the worst time complexity becomes exponential and it becomes very complex to find a subsequence from a very large string. But in the proposed case of this paper, the worst time complexity is O(m*((log n)^2)) for which it becomes very optimal to get the final result from a very large string. But not for only the worst case, In the time complexity for best case in the proposed LCS is O(n*log(n)). It is very optimum and time-consuming to get the final result.

V. Conclusion

The LCS algorithm is studied and researched by a lot of researchers. Here, we have developed an algorithm by using a divide and conquer approach and a queue. By using this approach we have made the space complexity linear O(n). The time complexity becomes a function of the logarithm. For the best case, it becomes O(n*log(n)) where n is the length of the smallest string between the two string. For average time complexity, it becomes O(k*log(n)) where k is the length of the largest string between the two string. For the worst case, the time complexity is O(m*log^2(n)) where m is the length of the bigger string. Normally the time complexity for the general LCS algorithm over the world is

O (n^2). We are able to show that our algorithm is better which gives results faster and use linear space than the used algorithm.

References

[1] J. Liu and S. Wu, "Research on longest common subsequence fast algorithm," in *2011 International Conference on Consumer Electronics, Communications, and Networks, CECNet 2011 - Proceedings*, 2011.

[2] N. Nakatsu, Y. Kambayashi, and S. Yajima, "A longest common subsequence algorithm suitable for similar text strings," *Acta Inform.*, 1982.

[3] D. S. Hirschberg, "Algorithms for the Longest Common Subsequence Problem," *J. ACM*, 1977.

[4] K. Kwarciak and P. Formanowicz, "A greedy algorithm for the DNA sequencing by hybridization with positive and negative errors and information about repetitions," *Bull. POLISH Acad. Sci. Tech. Sci.*, vol. 59, no. 1, 2011.

[5] J. F. Myoupo and D. Semé, "Time-Efficient Parallel Algorithms for the Longest Common Subsequence and Related Problems," *J. Parallel Distrib. Comput.*, 1999.

[6] A. Apostolico, M. Attalah, L. Larmore, and S. Mcfaddin, Efficient parallel algorithms for string editing and related problems, SIAM J. Comput. 19 (1990), 968?988.

[7] X. Tang, R. Tian, and D. F. Wong, "Fast evaluation of sequencepair in block placement by longest common subsequence computation," *IEEE Trans. Comput. Des.Integr. Circuits Syst.*, 2001.

[8] L. L. Cheng, D. W. Cheung, and S. M. Yiu, "Approximate string matching in DNA sequences," in *Proceedings - 8th International Conference on Database Systems for Advanced Applications, DASFAA 2003*, 2003.

[9] R. L. Goldfeder, D. P. Wall, M. J. Khoury, J. P. A. Ioannidis, and E. A. Ashley, "Human Genome Sequencing at the Population Scale: A Primer on High-Throughput DNA Sequencing and Analysis," *Am. J. Epidemiol.*, 2017.

[10] J. Eid *et al.*, "Real-time DNA sequencing from single polymerase molecules," *Science (80-.).*, 2009.

[11] F. Chin, and C.K. Poon, Chung, "Fast algorithm for computing longest common subsequences of small alphabet size," *Journal of Information Processing. 13. 463-469, 1991.*