Journal of Prime Research in Mathematics Vol. **4***(2008), 165-170*

A GREEDY APPROACH FOR COMPUTING LONGEST COMMON SUBSEQUENCES

AFROZA BEGUM*

Abstract. This paper presents an algorithm for computing Longest Common Subsequences for two sequences. Given two strings X and Y of length m and n, we present a greedy algorithm, which requires $O(n \log s)$ preprocessing time, where s is distinct symbols appearing in string Y and $O(m)$ time to determines Longest Common Subsequences.

Key words : Algorithms, alphabet, longest common subsequences, greedy algorithm.

1. Introduction

Let $X = x_1, x_2, x_3, ..., x_m$ and $Y = y_1, y_2, y_3, ..., y_n$ be two strings on an alphabet $\sum$ of constant size σ. A subsequence $\bigcup$ of a string is defined as any string which can be obtained by deleting zero or more elements from it, i.e. $\bigcup$ is a subsequence of X when $\bigcup = x_{i1}x_{i2}...x_{ik}$ and $i_q < i_q + 1$ for all q and $1 \leq q < k$. Given two strings X and Y, a longest common subsequence (LCS) of both strings is defined as any string which is a subsequence of both X and Y and has maximum possible length [10].

The problem of finding the longest common subsequence (LCS) of two given sequences is well studied and has a lot of applications in various fields, DNA or protein alignments, file comparison, speech recognition, gas chromatography, etc. Over the last two decades, many efficient algorithms have been designed to solve LCS problem.

This paper describes a simple greedy approach to find LCS. Given two strings X and Y of length m and n, an algorithm is presented which determines Longest Common Subsequences in $O(m)$ time with $O(n \log s)$ preprocessing time, where s is distinct symbols appearing in string Y.

*Department of Computer Science and Engineering, International Islamic University Chittagong, Chittagong, Bangladesh. E-mail: afrozactg@yahoo.com.

The paper is organized as follows. In the next section, a brief literature review has been presented. Section 3 gives an algorithm for computing the LCS of two strings, followed by the complexity analysis of the proposed algorithm. Finally section 4 presents the conclusion of the paper.

2. Literature Review

The classic dynamic programming solution to LCS problem was invented by Wagner and Fischer [11]. This dynamic programming algorithm defines the dynamic programming matrix $L_{0...m,0..n}$ as follows:

$$L_{ij} = \begin{cases} 1 & \text{if } i = 0 \text{ or } j = 0, \\ L_{i-1,j-1} + 1 & \text{if } x_i = y_i, \\ \max(L_{i,j-1}, L_{i-1,j}) & \text{if } x_i \neq y_i, \end{cases}$$

Using dynamic programming, the values in this matrix can be computed in $O(mn)$ time and space [1]. But Hirschberg [4] shows that in fact, only linear space is needed to find the length, since the computation of each row only needs the preceding row. An LCS can be retrieved by backtracking through the matrix, which would imply that the computation of the whole matrix L, requiring $O(mn)$ space. The corresponding string editing problem by Masek and Peterson [13] that uses "Four Russians trick" [12] is the fastest general solution for LCS in $O(nm/\log n)$ time. There are several algorithms that exist with complexities depending on other parameters. For example, Hunt and Szymanski gave a faster algorithm that runs in $O((r+n)\log n)$ time, where r is the total number of matching pairs of X and Y [6]. Also, Myers in [5] and Nakatsu et al. in [9] presented an $O(nD)$ algorithm, where the parameter D is the simple Levenshtein distance between the two given strings. Crochemore et al. presented a practical bit vector algorithm in $O(nm/w)$ time and $O(m/w)$ additional/working space, where w is the number of bits in a machine word [7].

There are also a number of problems related to LCS. The constrained LCS problem finds the LCS that contains a specific subsequence. Tsai [14] introduced the problem and presented an algorithm based on dynamic programming running on $O(m^2n^2r)$ time and $O(mnr)$ space. Another generalization of LCS problem is the All-substrings Longest Common Subsequence (ALCS) problem. Given two strings A and B of lengths m and n, respectively, the ALCS problem obtains the lengths of all the longest common subsequences for string A and all substrings of B. The sequential algorithm designed by Alves et al. [2] for this problem takes $O(mn)$ time and $O(n)$ space. Later a time-and space-efficient parallel algorithm is proposed in [3]. Illiopoulos and Rahman [8] introduce the notion of gap-constraints in the LCS problems and present efficient algorithms to solve several variants of LCS problem.

3. The Algorithm

3.1. **Preprocessing.** Given two strings $X = x_1, x_2, x_3, ..., x_m$ and $Y = y_1, y_2, y_3, ..., y_n$, first the lists of coincident points or matches for each distinct symbol in Y are computed. Lists of matches are the lists of ordered pairs of integers (i, j) such that $x_i = y_j$. It is sufficient to record only the set of j values (the positions in Y) corresponding to each distinct symbol, since from this set, the set of i values (the positions in X) can easily be obtained. For example, let two strings $X = ABCBDABE$ and $Y = BDCABA$. The lists of matches for string Y are shown in Table 1.

Table 1. Lists of matches for string Y

Symbol,s	Mathches in Y	Count[s]
A	4,6	2
B	1,5	2
C	3	1
D	2	1

To compute lists of matches for a string of length n, it requires $O(n \log s)$ time using $O(n)$ space, where s is the total number of distinct symbols appearing in the string. For a known symbol set, calculating lists of matches can be accomplished more efficiently, usually with $O(n)$ time.

3.2. **A Greedy Approach.** Only the matched symbol can constitute a LCS. So, at the time of scanning the string X, when we want to decide whether the symbol being examined is to select next, we only look at the lists of matches for that symbol. We can only consider the symbols for which lists of matches are constructed. All other symbols can be disregarded. The preprocessing phase gives opportunity to correctly find next available positions of each matched symbol.

The idea behind the algorithm is that, at every choice we pick the symbol that comes earlier. This is greedy because it leaves as much opportunity as possible for the remaining symbols to be selected.

The proposed greedy algorithm uses the following steps to compute LCS:

1. Preprocessing phase: Constructs lists of matches for all distinct symbols in Y.
2. Scan X from left to right. For those symbols of X that have match lists do the following:
   a. Let P_i and P_{i+1} be the two positions of i^{th} and $i + 1^{st}$ symbol respectively that are obtained from lists of matches.

b. Compare them. If P_i is larger than P_{i+1} then we disregard P_i and P_{i+1} is added to the set L. L is the set of positions of selected symbol that will constitute LCS, which is initially empty.
c. If P_i is smaller than P_{i+1} then P_i is added to the set L.

This algorithm records the position of last selected symbol in Y. If any of P_i and P_{i+1} denotes a position that belongs to the left of last selected position, it is assigned ∞ to ignore that position. Also, match lists traces the next matched positions of each symbol.

GreedyLCS

/$*$ Takes two sequences $X = x_1, x_2, x_3, ..., x_m$ and $Y = y_1, y_2, y_3, ..., y_n$ as inputs. Lists of matches of all distinct symbols in Y are provided by preprocessing phase. A one dimensional array count[s] maintains the total number of coincident points for symbol s. R records the position of last selected symbol. L is the set of positions of selected symbol that will constitute LCS, which is initially empty. $*$/

```
L = ∅
R = 0
i = 1
P_i= Position in Y of i^th symbol
while i < m
   do P_{i+1}= Position in Y of i + 1^st symbol
      count[P_{i+1}]=count[P_{i+1}] − 1
      if P_{i+1} < R
         then R = ∞
      if P_i > P_{i+1}
         then L = L ⌣ P_{i+1}
              R = P_{i+1}
              i = i + 1
              P_i= Position in Y of i^th symbol
              count[P_i]=count[P_i] − 1
         else
              L = L ⌣ P_i
              R = P_i
              P_i = P_{i+1}
return L
```

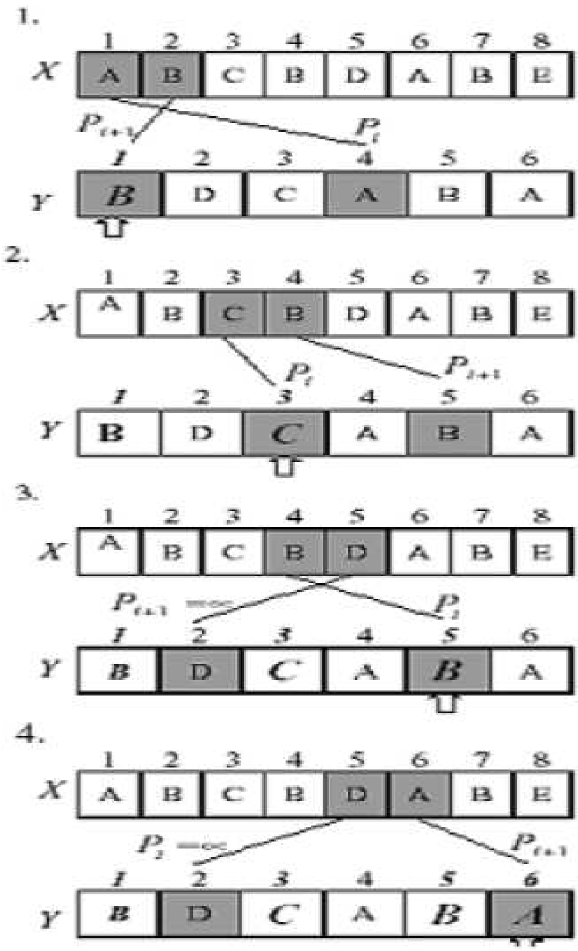

FIGURE 1. The execution of GreedyLCS on two given strings X and Y. Lightly shaded elements denote the symbols being examined. In each iteration, hollow arrows indicate the last selected position. The resulting set of selected positions is {1, 3, 5, 6}.

3.3. **Complexity Analysis.** This algorithm determines LCS in $O(m)$ time assuming that match lists are provided by a preprocessing phase that requires $O(n \log s)$ time, where s denotes the total number of distinct symbols in string Y and m and n are the length of two given strings.

4. Conclusion

The LCS problem was first studied by molecular biologists while studying similar amino acids. Subsequently, many applications in computer science found the use of LCS as a certain similarity measure of the objects represented by the strings. This paper proposes a greedy algorithm for the computation of the Longest Common Subsequences of two strings X and Y that achieves complexity of $O(m)$ time with $O(n \log s)$ preprocessing time, where m and n are the lengths of two original strings and s denotes the total number of distinct symbols in string Y.

References

[1] Corman, T.H., Leiserson, C.E., Riverst, R.L., Stein, C., *Introduction to Algorithms.* 2nd edn. MIT Press, Cambridge, MA, 2001.

[2] C. E. R. Alves, E. N. Caceres, and S. W. Song, "An all-substrings common subsequence algorithm", 2nd Brazilian symposium on graphs, algorithms and combinatorics. Electronics Notes in Discrete Mathematics, 19:133-139, 2005.

[3] C. E. R. Alves, E. N. Caceres, and S. W. Song, "A BSP/CGM algorithm for the all-substrings longest common subsequence problem", Proceedings of the 17th IEEE/ACM IPDPS, pages 1-8, 2003.

[4] D.S. Hirschberg, "A linear space algorithm for computing maximal common subsequences", Comm. Assoc. Comput. Mach., 18:6, 341-343, 1975.

[5] Eugene W. Myers, "An $O(ND)$ difference algorithm and its variations", *Algorithmica*, 1(2):251-266, 1986.

[6] J.W. Hunt, T.G. Szymanski, "A fast algorithm for computing longest common subsequences", Comm. ACM 20 (1977) 350-353.

[7] M. Crochemore, C.S. Iliopoulos, Y.J. Pinzon, J.F. Reid, "A fast and practical bit vector algorithm for the longest common subsequence problem", Inform. Process. Lett. 80 (2001) 279-285.

[8] M. Sohel Rahman and Costas Iliopoulos, "Algorithms for Computing Variants of the Longest Common Subsequence Problem", In Proceedings of the Algorithms and Computation (ISAAC 2006), India, December 2006, LNCS 4288, pp. 399-408.

[9] Narao Nakatsu, Yahiko Kambayashi, and Shuzo Yajima, "A longest common subsequence algorithm suitable for similar text strings", *Acta Inf.*, 18:171-179, 1982.

[10] Peter Krusche and Alexander Tiskin, "Efficient Longest Common Subsequence Computation using Bulk-Synchronous Parallelism", Workshop on Parallel and Distributed Computing (PDC-2006), 165-174.

[11] Robert A. Wagner and Michael J. Fischer. "The string-to-string correction problem", *J. ACM*, 21(1):168-173, 1974.

[12] V.L. Arlazarov, E.A. Dinic, M.A. Kronrod, I.A. Faradzev, "On economic construction of the transitive closure of a directed graph", *Dokl. Akad. Nauk SSSR*, 194, 487-488, 1970.

[13] W.J. Masek and M.S. Paterson, "A faster algorithm computing string edit distances", J. Comput. System. Sci., 20, 18-31, (1980).

[14] Y.-T. Tsai. "The constrained common subsequence problem. Information Processing Letters", 88:173-176, 2003.