Proceedings of the International Conference on Intelligent Sustainable Systems (ICISS 2017)
IEEE Xplore Compliant - Part Number:CFP17M19-ART, ISBN:978-1-5386-1959-9

An Analysis on Computation of Longest Common Subsequence Algorithm

Gaurav Kawade
Department of Computer Application
Shri Ramdeobaba College of Engineering and Management
Nagpur, India
kawadegv@rknec.edu

Santosh Sahu
Department of Computer Application
Shri Ramdeobaba College of Engineering and Management
Nagpur, India
sahusk1@rknec.edu

Sachin Upadhye
Department of Computer Application
Shri Ramdeobaba College of Engineering and Management
Nagpur, India
upadhyesd@rknec.edu

Nilesh Korde
Department of Computer Application
Shri Ramdeobaba College of Engineering and Management
Nagpur, India
kordens@rknec.edu

Manish Motghare
Department of Computer Application
Shri Ramdeobaba College of Engineering and Management
Nagpur, India
motgharemm@rknec.edu

***Abstract*—**There are many algorithms for computing the longest common subsequence, which are especially used in comparing files, text, comparison of DNA and protein sequences. In this paper we had done comparison among various algorithms which works on two or more strings. In our second approach we had done comparison of algorithms which is able to work on thousands of strings. As per the latest condition there are very few algorithm which works on multiple strings. The spotlight is on development of algorithm which is space efficient and reduced time complexity. In conclusion of the work we put the new proposals for the development of new algorithms for more strings.

Keywords*—Longest common subsequence; Dynamic Programming, Memory Management, Time complexity; Space complexity.*

I. INTRODUCTION

The longest common subsequence problem is defined as finding a longest subsequence common to all input strings, so the subsequence is contained in all strings. The ability to determine the longest common subsequence for two or more strings have numerous applications, including spelling correction systems, file comparison utilities, and the study of genetic evolution and genetic engineering. Scientist would like to know which species have common proteins and how similar they are. And if sequences are similar then determine when the evolution of the common predecessor created two new species. The computation of the LCS and other derived problems belongs to NP-problems [1]. Existing algorithms have in the average better performance, but still the worst case complexity remains proportional to the product of input strings length. The LCS problem is often defined by recursive formula Equation 1. Implementing the formula would led to the exponential time complexity. The dynamic programming can solve this kind of problems in the time proportional to the product of inputs length and use a special table called the dynamic programming table of the size (n × m), where m and n are lengths of two strings.

$$LCS(X_i, Y_i) = \begin{cases} \emptyset & if\ i = 0\ or\ j = 0 \\ LCS(X_{i-1}, Y_{j-1}) + 1 & if\ X_i = Y_i \\ \max\left(LCS(X_i, Y_{j-1}), LCS(X_{i-1}, Y_j)\right) & if\ X_i \neq Y_i \end{cases}$$

A. Principle of Dynamic Programming

The dynamic programming was firstly used by Wagner-Fischer [3] in 1974. Unfortunately with the quadratic time and space complexity it is not usable for long strings and more input strings. Several improvements have been published since the traditional approach. New algorithms based on the dynamic programming table are divided into three categories according to the method they compute cells of a dynamic programming table. Row-by-row algorithms scan rows and may advance from contour to contour on the same line. Advancing contour to contour and skipping rows is how the second approach Contour-to-Contour works. The last dynamic programming approach called Diagonal-wise advances the table in a greedy manner. Rather than processing rows or columns it use a specific diagonal and process only cells located on the diagonal. Cells on the same diagonal have for example the same delete distance.loud computing is distributed processing, parallel processing and the development of grid computing, or is it the computer science concept of commercial realized. The analysis of algorithms and exploring the performance is important when developing new algorithms. We will make experiments and measure how algorithms perform for different types of inputs. We also measure performance as functions of the number of dominant matches and match pairs, which is the key for new algorithms, where contours are processed. Memory requirements and time complexity measurement are also not missing, we will focus on memory space requirements, because is often forgotten and it is limiting factor for large instances. The work is organized as follows: section II Summarizes the previous research in the field of the longest common subsequence computation. In section III, implementation of comparison of finite automata

and the dynamic programming. In section IV Summarizes results of experiments on implemented algorithms.

II. RELATED WORK

Following sections describe common paradigms used by algorithms to compute the longest common subsequence. First three paradigms are based on the dynamic programming. All of them were developed for two strings, and differ in the way the dynamic programming table is computed and which cells are not omitted. The memory requirement of presented algorithms is usually proportional to the sum of length of input strings.

A. Row by row filling paradigm

The procedure is derived from traditional way of computing a dynamic programming table. The table is filled row after row and the time complexity is always $O(m * n)$. The straightforward implementation is known as the Wagner-Fischer [3] algorithm. Look at Figure 1, the result of the Wagner-Fisher algorithm. The total number of comparison made is 9 * 6=$O(m * n)$. Obviously the number of comparison can be reduced, because for example in the first row last three values cannot change anymore. Following algorithm Hirschberg [10], Hunt-Szymanski [4], Kuo-Cross[8] Algorithm, Mukhopadhyay [5] Algorithm, Wagner-Fischer[3] Algorithm uses the same paradigm.

	c	b	a	c	b	a	a	b	a
a	0	0	1	1	1	1	1	1	1
b	0	1	1	1	2	2	2	2	2
c	1	1	1	2	2	2	2	2	2
d	1	1	1	2	2	2	2	2	2
b	1	2	2	2	3	3	3	3	3
b	1	2	2	2	3	3	3	4	4

Fig. 1. The dynamic programming table for ”cbacbaaba” and ”abcdbb”

B. Contour to Contour

A disadvantage of the row-wise approach is that we have to scan the matches corresponding to the entire Y string for each row. Rick’s method avoids this (partly) by trying to compose whole contours as soon as possible. In the genuine contour-to-contour approach, we scan through all dominant matches on contour k and for each such match, determine all possible dominant k + 1 -matches as ’seen’ by the current dominant k-match.

C. Table Filling Diagonalwise

In this section we study methods which fill the dynamic programming table greedily, trying to reach the goal entry R[m,n]as fast as possible. Most of these algorithms are based on the calculation of the edit distance between the input strings. To accomplish this, the diagonals are numbered so that entry (i , j) is on diagonal j - i. Thus, elements (O,O), (1, l) , ..., (m,m) are on diagonal zero, diagonals 1,2,3, ..., n lie above it and diagonals - 1, -2, ..., -m below it. Diagonal n - m ends in the lower right corner (m,n). The diagonals are processed in a systematic manner and each of them is extended in two steps. First, we check the diagonals immediately above and below (if they exist), which one reaches lower down in the table. The initial position (i , j) for the current diagonal is determined either by moving one row down from the diagonal above or one column right from the diagonal below. Once the initial position (and the edit distance value) has been determined, we move further down along the current diagonal as long as the symbols of X and Y match.

Hirschberg introduced a divide and conquer technique for the dynamic programming and computes the longest common subsequence in the linear space and the quadratic time. The middle of the LCS is computed and stored on the stack. The procedure is performed for the left and the right half of the table until the remaining length of strings is zero. Hunt-Szymanski computes a special arrays called the matchlist, the threshold array and store information where the contour lines are crossing the horizontal axis. Kuo-Cross made an improvement to Hunt Szymanski by using different technique of checking if a contour crosses the horizontal axis. More is discussed in the description of algorithms. The Apostolico-Guerra algorithm uses similar arrays also for rows and as others create an area of interest, where cells are computed. The area dynamically changes as new matches are discovered. Algorithms using contour-to-contour paradigm beat others when the large alphabet is used and the number of dominant matches is close to the number of match pairs. The Rick algorithm[9] performs well even for small alphabets.

Sr. no	Source	Time Complexity	Space	Paradigm	Description
1	Wagner and Fischer[3]	$O(mn)$	(mn)	*Row-by-Row*	The string-to-string correction problem is to determine the distance between two strings as measured by the minimum cost sequence of "edit operations" needed to change the one string into the other. The edit operations investigated allow changing one symbol of a string into another single symbol, deleting one symbol from a string, or inserting a single symbol into a string. An algorithm is presented which solves this problem in time proportional to the product of the lengths of the two strings. Possible applications are to the problems of automatic spelling correction and determining the longest subsequence of characters common to two strings.easured by the minimum cost sequence of "edit operations" needed to change the one string into the other.
2	Hunt and Szymanski[4]	$O(\|M\| \ logn)$	$O(\|M\|)$	*Row-by-Row*	An algorithm for this problem is presented which has a running time of O((r + n) log n), where r is the total number of ordered pairs of positions at which the two sequences match. Thus in the worst case the algorithm has a running time of O(n 2 log n). However, for those applications where most positions of one sequence match relatively few positions in the other sequence, a running time of O(n log n) can be expected.
3	Mukhopadhyay[5]	$O(\|M\| \ Mn)$	$O(n+r)$	*Row-by-Row*	In this paper a fast algorithm for the longest common subsequence problem is present which runs in O((p+n)logn) time where p is the total number of pairs of matched positions between the strings. Thus, the average performance of this algorithm is much better than those of the quadratic algorithms proposed earlier and takes only a linear amount of space.
4	Hsu and Du[6]	$O(rm \ log(r/n) + rm)$		*Row-by-Row*	In Hirschberg's algorithm, the lower bound for the determination of a ***K-key*** at a row, say row ***i,*** is set by the value of P_{k-1}^{i-1}**,.** Essentially, the algorithm searches ***sequentially*** through the j-values that are located between the lower bound and the upper bound (high[θ], if ***A[i] = θ).*** We propose an algorithm which preserves the same basic scheme (that avoids duplicated searches) but may benefit additionally from the power of binary searches.
5	Apostolico and Guerra[7]	$O(mlogn + dlog(2m\,n/d))$	$O(\sigma m + n)$	*Row-by-Row*	
6	Kuo and Cross[8]	$O(\|M\| + n(r + logn))$	$O(n+r)$	*Row-by-Row*	
7	Rick[9]	$O(\sigma n + \min\{r(n - r), rm\})$		*Row-by-Row*	

8	Rick[9]	$O(\sigma n + \min\{\sigma d, rm\})$		*Row-by-Row*	
9	Hirschberg [10]	$O(rn + nlogn)$		*Contour*	
10	Hsu and Du[6]	$O(rmlog(n/m) + r$		*Contour*	In Hirschberg's algorithm, the lower bound for the determination of a ***K-key*** at a row, say row ***i,*** is set by the value of P_{k-1}^{i-1}**.** Essentially, the algorithm searches ***sequentially*** through the j-values that are located between the lower bound and the upper bound (high[θ], if ***A[i] = θ).*** We propose an algorithm which preserves the same basic scheme (that avoids duplicated searches) but may benefit additionally from the power of binary searches.
11	Apostolico and Guerra[7]	$O(rm + \sigma n + nlog\sigma)$		*Contour*	
12	Chin and Poon[11]	$O(\sigma n + \min\{\sigma d, rm\})$		*Contour*	
13	Nakatsu et al. [12]	$O(n(m-r))$		Diagonal	Efficient algorithms for computing the longest common subsequence (LCS for short) are discussed. O(p n) algorithm and O(p(m -p) log n) algorithm seem to be best among previously known algorithms, where p is the length of an LCS and m and n are the lengths of given two strings (m < n). There are many applications where the expected length of an LCS is close to m. In this paper, O(n(m-p))algorithm is presented. When p is close to m (in other words, two given strings are similar), the algorithm presented here runs much faster than previously known algorithms.
14	Miller and Myers[13]	$O(m(m-r))$		Diagonal	
15	Wu et al. [14]	$O(n(m-r))$		Diagonal	This algorithm work by using a path-compression technique that has been used as a heuristic for shortest paths problems. This technique was also used by Hadlock to give an O(NP) sequence comparison algorithm, however, Hadlock used a version of Dijkstra's algorithm and thus the expected running time of his algorithm is also O(NP), whereas the expected running time of our algorithm is O(N + PD). Our fusion of a notion of compressed distances and Myers's greedy approach give an O(NP) algorithm that is very simple and thus very efficient in practice. The algorithm's dependence on P implies that it is particularly efficient when A is similar to a subsequence of the longer sequence B. In fact, the algorithm is O(N) when A is a subsequence of B. By using Hirschberg's divide-and-conquer technique, the algorithm can be modified to deliver a shortest edit script using only linear space.

16	Jiaoyun Yang et. al SA-MLCS[15	O(cmNlogN)	O(cnm)		SA-MLCS is an anytime algorithm with a small space increase rate. SA-MLCS uses an iterative beam widening search strategy to reduce space usage during the iterative search process of finding better solutions.
17	Jiaoyun Yang et. al SAL-MLCS[16]	O(c|Σ|nm)	O(cnm)		SLAMLCS, a space-bounded algorithm, is developed based on SA-MLCS. SLA-MLCS uses a replacing strategy when SA-MLCS reaches a given space bound and can effectively handle much larger size problem instances.
18	Qingguo Wang et. al Pro-MLCS[17]	O(c|Σ|nm)	O(c|Σ|nm)		In Pro-MLCS, a heuristic best-first search is applied to search the tree constructed in the last section. Each layer maintains one priority queue that contains the nodes at the same layer.[2]
19	Qingguo Wang et. al DPro-MLCS[17]	O(log2m + logn)	O(Mn/logm))		The main idea of DPro-MLCS is to compute different layers in parallel, which is a major advantage over existing parallelalgorithms that can only parallelize computation in the same layer. [2]
20	Qingguo Wang et. al DSDPro-MLCS[17]	O(log2m loglogm)	O(mn/log2m l		DSDPro-MLCS, the parallelization between hardware nodes is based on the distributed memory, while the parallelization within each hardware node is based on shared memory. [2]

Fig. 2.List of existing LCS algorithms for two strings

III. IMPLEMENTATION AND COMPARISON

A. *Performance and memory consumption measurement*

To be able to measure the real memory consumption, we have implemented a special version of C functions free and malloc by overriding the default implementation. Our implementation helped us to find memory leaks during the development and optimize the allocated memory space.

B. *Implementation of extended Wagner-Fischer for more strings*

We did not find any references in articles about using the Wagner-Fischer [6] algorithm for more strings and no reference implementation which our implementation can be compared to Wagner-Fischer algorithm can be modified to work with multiple inputs we covered in previous sections. Now we look at how we implemented it. No static structures can be used when dealing with the multiple numbers of inputs. Each loop and a comparison must be done in the further loop which takes in account the number of inputs. For example a test whether symbols at given position are equal must be done also in the loop. The complication caused that no part of the original algorithm could be used. The computation is done by iterating over each dimension. For mentioned 3D-dimensional Cube, first the plane (i, j, 0) is solved and then planes (i, j, 1), (i, j, 2), (i, j, 3). For more dimensional space the advancing is similar and after completion of 3. Dimension (for example (i, j, 4, 0)), the 4. Dimension is calculated (for example (i, j, 1, 1)). Each vector must be mapped using a mapping function from the n-dimensional space to the linear memory block of the size equal to the product of lengths of input strings. This mapping is done simply by get coordinates function (Algorithm 16).

The memory mapping for the Extended Wagner-Fischer algorithm

Input: A set of sequences, where |sequences| = count and vector denotes an order set of indices.
Output: The index in the linear array
1: index 0
2: for i =1 to count do
3: index = index * |sequences[i]|;
4: index = index + $vector_i$

5: end for
6: return index

IV. CONCLUSION & FUTURE SCOPE

Computation of the longest common subsequence is a key for a lot of fields of research like genetic engineering, file comparison, string matching problems and others. We have recalled the problem of the LCS and other related problems derived from the LCS plus gave a list of algorithms capable of computing these problems mainly the longest common subsequence. First of all we presented existing algorithms for two strings and described some of them more deeply. Except the finite automata approach are all based on the dynamic programming. To improve the running time algorithms use different structures and pre-computations. Unfortunately the speed up is always for specific types of inputs. Finite automata seem to be a very straightforward approach, but are beaten by fast dynamic programming algorithms for two strings, because of large overhead factor.

References

[1] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. Basic local alignment search tool. J Mol Biol, 215(3):403–410, October 1990.

[2] H.-Y. Ann, C.-B. Yang, C.-T. Tseng, and C.-Y. Hor. A fast and simple algorithm for computing the longest common subsequence of run-length encoded strings. Inf. Process. Lett., 108(6):360–364, 2008.

[3] Wagner, R.A. & Fischer, M.J.: The String-to-String Correction Problem, Journal of the ACM, Vol. 21, No. I , Jan- uary 1975, pp. 168-173

[4] Hunt, J.W. & Szymanski, T.G.: A Fast Algorithm for Computing Longest Common Subsequences, Comm. of the ACM, Vol. 20, NO. 5, 1977, pp. 350-353

[5] Mukhopadhyay, A.: A Fast Algorithm for the Longest- Common-Subsequence Problem, Information Sciences, Vol. 20, 1980, pp. 69-82

[6] Hsu, W.J. & Du, M.W.: New Algorithms for the LCS Problem, J. Comp. and Svstem Sc., Vol. 29, 1984, pp. 133-152

[7] Apostolico, A. & Guerra, C.: The Longest Common Subsequence Problem Revisited, Algorithmica, No. 2, 1987, pp. 315-336

[8] Kuo, S. & Cross, G.R.: An Improved Algorithm to Find the Length of the Longest Common Subsequence of ' b o Strings, ACM SlGlR Forum, Vol. 23, No. 3 4 , 1989, pp.89-99

[9] Rick, C.: A New Flexible Algorithm for the Longest Common Subsequence Problem, in Galil, Z. & Ukkonen, E.(eds): Proc. of Combinatorial Pattern Matching, 6th Annual Symposium, Espoo, Finland, July 1995, pp. 340-351. Appeared also as Lecture Notes in Computer Science, Vol.937

[10] Hirschberg, D.S.: Algorithms for the Longest Common Subsequence Problem, Journal of the ACM, Vol. 24, No. 4, 1977, pp. 664-675

[11] Chin, F.Y.L. & Poon, C.K.: A Fast Algorithm for Computing Longest Common Subsequences of Small Alphabets Size, J. of In$ Proc., Vol. 13, No. 4, 1990, pp. 463469

[12] Nakatsu, N., Kambayashi, Y. & Yajima, S.: A Longest Common Subsequence Algorithm Suitable for Similar Texts, Acta Informatica. Vol. 18, 1982, pp. 171-179

[13] Miller, W. & Myers, E.W.: A File Comparison Program, Softw. Pract. Exp., Vol. 15, No. 11, November 1985, pp. 1025-1040

[14] Wu, S., Manber, U., Myers, G. & Miller, W.: An O(NP) Sequence Comparison Algorithm, In$ Proc. Lett., Vol. 35, September 1990, pp. 317-323

[15] Jiaoyun Yang, Yun Xu, Yi Shang, and Guoliang Chen,"A Space-Bounded Anytime Algorithm for the Multiple Longest Common Subsequence Problem"IEEE Transactions on Knowledge and Data Engineering, NO. 1-13, 2014.

[16] Jiaoyun Yang, Yun Xu, Guangzhong Sun, and Yi Shang, "A New Progressive Algorithm for a Multiple Longest Common Subsequences Problem and Its Efficient Parallelization",IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS, VOL. 24, NO. 5, MAY 2013.

[17] Qingguo Wang, Dmitry Korkin, and Yi Shang,"A Fast Multiple Longest Common Subsequence (MLCS) Algorithm",IEEE TRANSACTIONS ON KNOWLEDGE AND DATA ENGINEERING, VOL. 23, NO. 3, MARCH 2011.