Indu et al. International Journal of Recent Research Aspects ISSN: 2349-7688, Vol. 3, Issue 2, June 2016, pp. 70-75

A Comparison of Different Algorithms for Approximate String Matching

Indu, Prerna

Department of Computer Science, Gateway Institute of Engineering & Technology (GIET), Deenbandhu Chhotu Ram University of Science & Technology (DCRUST), Sonepat

Abstract— In computer science, approximate string matching is the technique of finding strings that match a pattern approximately (rather than exactly). Most often when we need to match a pattern exact matching is not possible, due to insufficient data, broken data, or other such reasons. So we try to find a close match instead of an exact match. And for this we need to find the distance between two strings. We have different approaches for the same such as edit distance in the form of Hamming distance, Levenshstien distance, Dameru-Levenshstein distance, Jaro-Winkler distance and Longest Common Subsequence (LCS). Different algorithms have been made for these different approaches, and we will try to analyze some of these algorithms.

Keywords— Edit Distance, Longest Common Subsequence, Pattern Matching , Space and Time Complexity.

I. Introduction

Approximate string matching is used is finding similar matches in search engines like *google* when the user may want to see similar items, or maybe sometimes the user does not enter the exact words occurring in the content on the item to be searched, and this is done through approximate string matching. It is used for the following purposes:

1) Search engines
2) Database searching:
3) Spell Checking
4) Signal processing
5) File comparison
6) Screen redisplay

II. Approaches

Two primary methods used for approximate string matching are **Edit distance and longest common subsequence.**

A. Edit Distance

The closeness of a match is measured in terms of the number of primitive operations necessary to convert the string into an exact match. This number is called the edit distance between the string and the pattern. The usual primitive operations are:

insertion:	*cot → coat*
deletion:	*coat → cot*
substitution:	*coat → cost*

There are several different ways to define an edit distance, depending on which edit operations are allowed: replace, delete, insert, transpose, and so on. There are algorithms to calculate its value under various definitions: Levenshtein distance & Hamming distance

A.1 Levenshtein distance

The Levenshstein distance is the number of insert, delete and substitution operations require to transform one string into another. Levenshstein distance is defined as:

$$\mathrm{lev}_{a,b}(i,j) = \begin{cases} \max(i,j) & ,\min(i,j)=0 \\ \min \begin{cases} \mathrm{lev}_{a,b}(i-1,j)+1 \\ \mathrm{lev}_{a,b}(i,j-1)+1 \\ \mathrm{lev}_{a,b}(i-1,j-1)+[a_i \neq b_j] \end{cases} & ,\ \text{else} \end{cases}$$

For example, the Levenshtein distance between "kitten" and "sitting" is 3, since the following three edits change one into the other, and there is no way to do it with fewer than three edits:

kitten → sitten (substitution of "s" for "k")
sitten → sittin (substitution of "i" for "e")
sittin → sitting (insertion of "g" at the end).

The matrix used for the comparison of two strings is shown below:

		m	e	i	l	e	n	s	t	e	i	n
	0	1	2	3	4	5	6	7	8	9	10	11
l	1	1	2	3	3	4	5	6	7	8	9	10
e	2	2	1	2	3	3	4	5	6	7	8	9
v	3	3	2	2	3	4	4	5	6	7	8	9
e	4	4	3	3	3	3	4	5	6	6	7	8
n	5	5	4	4	4	4	3	4	5	6	7	7
s	6	6	5	5	5	5	4	3	4	5	6	7
h	7	7	6	6	6	6	5	4	4	5	6	7
t	8	8	7	7	7	7	6	5	4	5	6	7
e	9	9	8	8	8	7	7	6	5	4	5	6
i	10	10	9	8	9	8	8	7	6	5	4	5
n	11	11	10	9	9	9	8	8	7	6	5	4

Finding the Levenshstein distance between two given strings “meilenstein” and “levenshstein” through dynamic programming. The Levenshstien distance comes out to be 4, as given in the bottom right corner of the matrix. The complexity of Levenshstien distance is O(mn)

A.2 Hamming distance

In information theory, the Hamming distance between two strings of equal length is the number of positions at which the corresponding symbols are different. In another way, it measures the minimum number of substitutions required to change one string into the other, or the minimum number of errors that could have transformed one string into the other.

Examples:

The Hamming distance between:
"toned" and "roses" is 3.
1011101 and 1001001 is 2.

2173896 and 2233796 is 3.

Special properties:

For a fixed length n, the Hamming distance is a metric on the vector space of the words of length n, as it obviously fulfils the conditions of non-negativity, identity of indiscernible and symmetry, and it can be shown easily by complete induction that it satisfies the triangle inequality as well. The Hamming distance between two words *a* and *b* can also be seen as the Hamming weight of a−b for an appropriate choice of the − operator.
For binary strings *a* and *b* the Hamming distance is equal to the number of ones (population count) in a XOR b. The metric space of length-n binary strings, with the Hamming distance, is known as the Hamming cube; it is equivalent as a metric space to the set of distances between vertices in a hypercube graph. One can also view a binary string of length n as a vector in by treating each symbol in the string as a real coordinate; with this embedding, the strings form the vertices of an n-dimensional hypercube, and the Hamming distance of the strings is equivalent to the Manhattan distance between the vertices.
The complexity for hamming distance is O(n)

A.3 Comparisons:

1. The complexity of Levenshtein distance is O(mn)
   The complexity for hamming distance is O(n)
2. The Levenshtein distance works for unequal string.
   The Hamming distance works for equal string.
3. The no. of operations used in Levenshtein distance are 3 (Insertion, Deletion and Substitution)
   The no. of operations used in Hamming distance is 1 (Substitution)

B. Longest Common Subsequence (LCS)

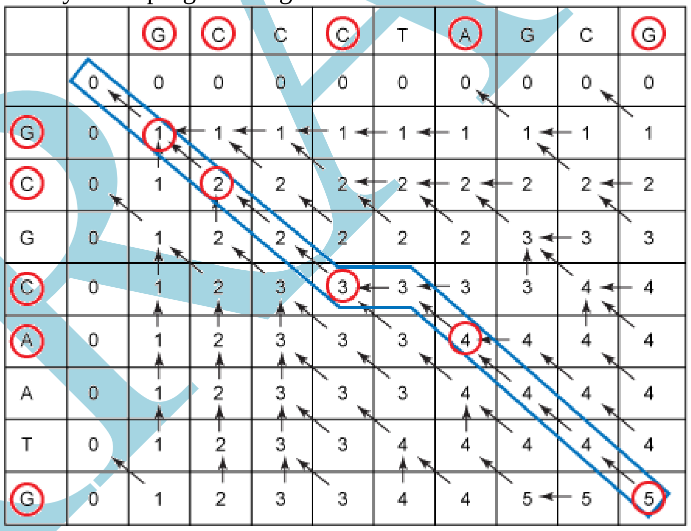
The longest common subsequence is a classical problem which is solved by using the dynamic programming approach. The LCS problem has an optimal substructure: the problem can be broken down into smaller, simple "subproblems", which can be broken down into yet simpler subproblems, and so on, until, finally, the solution becomes trivial. The LCS problem also has overlapping subproblems: the solution to a higher subproblem depends on the solutions to several of the lower subproblems. Problems with these two properties—optimal substructure and overlapping subproblems—can be approached by a problem-solving technique called dynamic programming, in which the solution is built up starting with the simplest subproblems. The procedure requires memoization—saving the solutions to one level of subproblem in a table (analogous to writing them to a memo, hence the name) so that the solutions are available to the next level of subproblems.

B.1 Wagner-Fischer Algorithm

Wagner-Fischer developed in 1974 the one of first algorithms which can compute the LCS of two strings. Originally the algorithm was intended to compute an edit distance between two strings called the problem of string to string correction (the usage of the dynamic programming to solve this kind of problems was invented by Richard Bellman in 1953). This means the number of remove, replace, and insert operations needed to change the string into another.

$$D[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ D[i-1, j-1] + 1 & \text{if } X[i] = Y[j] \\ \max(D[i-1,j], D[i,j-1]) & \text{if } X[i] \neq Y[j] \end{cases}$$

The disadvantage of the Wagner-Fischer algorithm is the equal time for all types of inputs no matter how similar inputs are. This is impractical when we have some expectation about a structure of input strings or about the similarity (input strings can be similar at 99%). More complicated algorithms have been developed since the straightforward dynamic programming and we will investigate principles in next sections, however algorithms based on the Wagner-Fischer have the worst case time complexity O(mn) even if the average time complexity is better.
The dynamic programming matrix



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

The above example shows two strings "GCCCTAGCG" and "GCGCAATG" for which the dynamic programming matrix is computed and the length of LCS comes out to be 5.

B.2 Hirschberg Algorithms

Hirschberg presented in 1977 an algorithm for computing the LCS using the dynamic programming and a divide and conquer paradigm, runs in O(mn) time and O(m + n) space Using the dynamic programming matrix the time complexity is proportional to the product of the lengths of strings. Previously Hirschberg presented a modification which uses only linear space O(m + n). However, by computing the table in the linear space we lose the ability to backtrack the longest common subsequence, because only the last row is stored in the memory. To obtain the LCS sequence and keep the linear space complexity, Hirschberg proposed a divide and conquer technique for the dynamic programming table. To be able to use the C&D technique, we need to calculate where the optimal LCS path is crossing the first half of the table (coordinates of that pair). Lets assume that we have two strings x and y, where the length of strings are equal without loss of generality. Create substrings x1 = $x_{1..n}$, y1 = $y_{1..m/2}$, x2 = $x_{n..1}$, y2 = $y_{m..m/2}$, where the length |x1| = |x2| and |y1| = |y2|. y1 is a prefix of y of the length m/2. y2 is a reversed suffix of y of the length m/2 . Then calculate the longest common subsequence for x1 and y1, and the longest common subsequence of reversed strings x2 and y2. Formally LCS($x_{1..n}$, $y_{1..m/2}$) and LCS($x_{n..1}$, $y_{m..m/2}$).

Table 1 and Table 2 shows the computed LCS for that strings and Table 3 sums values of last rows. The first highest value in the row, the value of 8 in Table 3 denotes the middle of the LCS sequence.

	a	c	b	d	e	a	c	b	e	d
a	1	1	1	1	1	1	1	1	1	1
c	1	2	2	2	2	2	2	2	2	2
e	1	2	2	2	3	3	3	3	3	3
b	1	2	3	3	3	3	3	4	4	4
d	1	2	3	4	4	4	4	4	4	5
a	1	2	3	4	4	5	5	5	5	5

Table 1: The dynamic programming table for ”acebda” and ”acbdeacbed”

	d	e	b	c	a	e	d	b	c	a
d	1	1	1	1	1	1	1	1	1	1
e	1	2	2	2	2	2	2	2	2	2
b	1	2	3	3	3	3	3	3	3	3
a	1	2	3	3	4	4	4	4	4	4
b	1	2	3	3	4	4	4	5	5	5
b	1	2	3	3	4	4	4	5	5	5

Table 2: The dynamic programming table for ”debabb” and ”debcaedbca”

		a	c	b	d	e	a	c	b	e	d
a		1	1	1	1	1	1	1	1	1	1
c		1	2	2	2	2	2	2	2	2	2
e		1	2	2	2	3	3	3	3	3	3
b		1	2	3	3	3	3	3	4	4	4
d		1	2	3	4	4	4	4	4	4	5
a		1	2	3	4	4	5	5	5	5	5
$\sum$	5	6	7	7	8	8	8	8	7	6	5
b	5	5	5	4	4	4	3	3	2	1	
b	5	5	5	4	4	4	3	3	2	1	
a	4	4	4	4	4	4	3	3	2	1	
b	3	3	3	3	3	3	3	3	2	1	
e	2	2	2	2	2	2	2	2	2	1	
d	1	1	1	1	1	1	1	1	1	1	
	a	c	b	d	e	a	c	b	e	d	

		1	2	3	4	4	5	5	5	5	5
	5	5	5	4	4	4	3	3	2	1	
=	5	6	7	7	8	8	8	8	7	6	5

Table 3: Sum of last rows of Tables 1.2 and reversed 1.3

B.3 Hunt-Szymanski Algorithm

The Hunt-Szymanski algorithm solves the LCS problem in O ((r + n) log(n)) time and in O(r + n) space, where r denotes the number of match points. When the number of match points is small, the algorithm is very efficient.

The algorithm is a representant of row-by-row paradigm presented previously. It tries to optimize the number of comparison by remembering where a contour line crosses the X axis vertically. Hunt-Szymanski developed an array called the THRESHOLD where stores indices of contours crossing the axis and speed up the computation by decreasing the number of cell comparisons.

Lets first discover how the THRESHOLD array is being computed and then how the Hunt-Szymanski algorithm works.

The THRESHOLD array (Figure 2, symbol “-” denotes undefined value) has the length of the first string. All values are initialized to |n|+1 that is undefined. After processing a row of the dynamic programming table, the array contains indices where contour lines are crossing the X axis. When a new contour is found on the current row. The rank of the pair is computed and is also the value of the cell of the table. Afterwards we only find the index in the THRESHOLD array which satisfies the condition array[i - 1] < k ≤ array[i + 1] by using the binary search (this step corresponds to moving the contour line to the left). The last step is to update the array and move to next pairs. The modified algorithm will run in O(n^3 log(n)) that is much worse than the simple Wagner-Fischer algorithm

A simple improvement proposed by Hunt-Szymanski can be made. Sort input strings (Figure 3) alphabetically in the decreasing order and remember original indices of symbols in the string. Create an array (Figure 1) of the linked list of the length of the second string. And for each symbol from the first string precompute all pairs and put indices of equal symbols from the second string in the decreasing order in the linked list (for example, the second line ”c” in Figure 1 denotes that only symbols at 6, 5, 2 equal to ”c”). This structure is called MATCHLIST and decrease the number of comparison because we are advancing only over symbols which match on the current line.

The MATCHLIST can be created in O(n log(n)) by using for example the quick sort.

c	3		
b	6	5	2
a	1		
c	3		
b	6	5	2
a	1		
a	1		
b	6	5	2
a	1		

Figure 1: The Hunt-Szymanski MATCHLIST array for strings ”cbacbaaba” & ”abcdbb”

	1	2	3	4	5	6
0	-	-	-	-	-	-
1	3	-	-	-	-	-
2	2	5	-	-	-	-
3	1	5	-	-	-	-
4	1	3	-	-	-	-
5	1	2	5	-	-	-
6	1	2	5	-	-	-
7	1	2	5	-	-	-
8	1	2	5	6	-	-
9	1	2	5	6	-	-

Figure 2: The Hunt-Szymanski THRESHOLD array for strings ”cbacbaaba” & ”abcdbb”

c	b	a	c	b	a	a	b	a
1	2	3	4	5	6	7	8	9

a	a	a	a	b	b	b	c	c
9	7	6	3	8	5	2	4	1

a	b	c	d	b	b
1	2	3	4	5	6

a	b	b	b	c	d
1	6	5	2	3	4

Figure 3: Sorted strings ”cbacbaaba” and ”abcdbb”

B.4 Kuo-Cross Algorithm

The construction of the THRESHOLD array is not optimal in terms of unnecessary updates. Kuo-Cross presented a modification of the original algorithm, where a matchlist is sorted in the increasing order and matches are processed left to right. The different direction avoids unnecessary updates (Figure 4 & 5) to the array and a value of the array is rewritten only once (for the worst case scenario for the Hunt-Szymanski algorithm, it is the number of elements in the matchlist for a symbol).

Time and space complexity The algorithm avoids unnecessary updates to the THRESH-OLD array and runs in O(σ + n(r + log(n))) compared to the original Hunt-Szymanski O((r + n) log(n)). The algorithm uses O(r + n) as much memory as the Hunt-Szymanski algorithm.

c	3		
b	2	5	6
a	1		
c	3		
b	2	5	6
a	1		
a	1		
b	2	5	6
a	1		

Figure 4: The Kuo-Cross MATCHLIST array for strings ”cbacbaaba” and ”abcdbb”

	1	2	3	4	5	6
0	-	-	-	-	-	-
1	1	-	-	-	-	-
2	1	2	-	-	-	-
3	1	0	-	-	-	-
4	0	1	-	-	-	-
5	0	1	2	-	-	-
6	0	0	0	-	-	-
7	0	0	0	-	-	-
8	0	0	0	1	-	-
9	0	0	0	0	-	-

Figure 5: The number of rewrites for the HS THRESHOLD array (strings ”cbacbaaba” and ”abcdbb”)

B.5 Myers-Miller Algorithm

It is a diagonalwise algorithm for two strings with the linear space complexity. The algorithm solves the LCS and the SES problem in O(n(m-r)), where m and n denote lengths of input strings and r the total number of ordered pairs. The only structure the algorithm uses is the linear array O(m + n) of temporary x values.

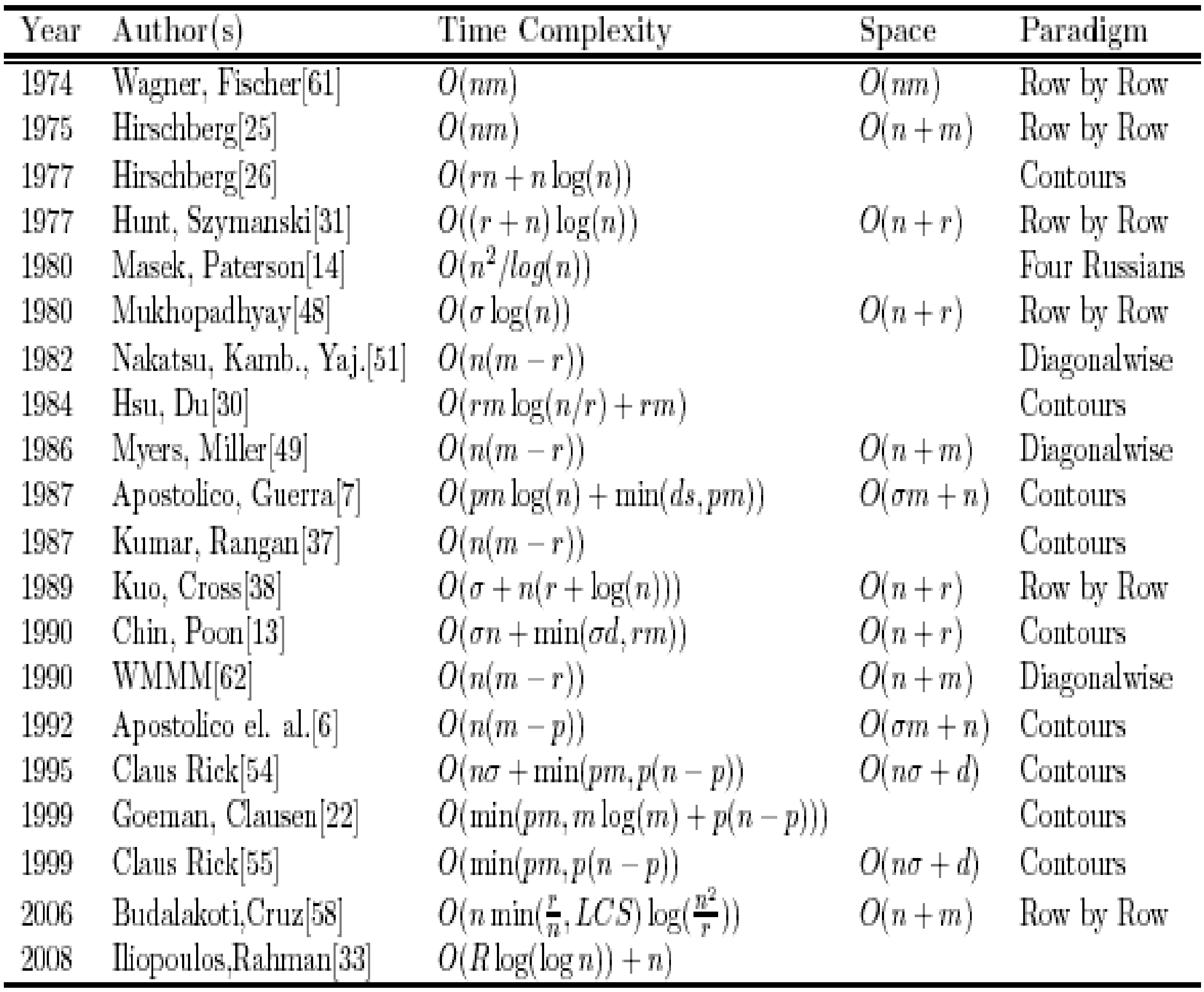
To understand how the algorithm works, lets first define an edit graph. The edit graph is a graph corresponding to the dynamic programming table. The difference is that cells are replaces by vertices and three types of edges. Diagonal edges represent cases where symbols at given positions are equal. Horizontal and vertical edges where symbols are different and a remove or an insert instruction is required. Lets introduce the cost of the edge; horizontal and vertical edges have the cost of one and diagonal edges the cost of zero. All paths (traces) from [0, 0] to [m, n] with the same cost are isomorphic. When two paths are isomorphic then there is no difference which path is chosen. Furthermore, some cells are not needed to be computed. Look at Figure 6, the cell [3,5 = e,a] do not need to be computed because from the upper cell the optimal path is connected by either a horizontal or a vertical edge.

In contrast with the row by row or the contour to contour approach, the algorithm advances based on calculating the number of delete instructions (the Levenshtein distance). The advancing is greedy because discards paths corresponding to common subsequences with a large Levenshtein distance. These paths cannot form the longest common subsequence since other paths with a smaller distance exist.

	a	c	e	b	d	a	b	b	a	b	e	d
a	0											
c		0										
b			2	1								
d			3		1							
e			2			3	4	5	6	7		
a				4		2						
c				5			4	5	6	7		
b				4			3					
e					6			5	6	7	6	
d					5			6	7	8		6

Figure 6: Cells filled by the Myers-Miller algorithm for strings ”acebdabbabed” & ”acbdeacbed”

III. COMPARISONS OF EXISTING LCS ALGORITHMS



Year	Author(s)	Time Complexity	Space	Paradigm
1974	Wagner, Fischer[61]	$O(nm)$	$O(nm)$	Row by Row
1975	Hirschberg[25]	$O(nm)$	$O(n+m)$	Row by Row
1977	Hirschberg[26]	$O(rn+n\log(n))$		Contours
1977	Hunt, Szymanski[31]	$O((r+n)\log(n))$	$O(n+r)$	Row by Row
1980	Masek, Paterson[14]	$O(n^2/log(n))$		Four Russians
1980	Mukhopadhyay[48]	$O(\sigma\log(n))$	$O(n+r)$	Row by Row
1982	Nakatsu, Kamb., Yaj.[51]	$O(n(m-r))$		Diagonalwise
1984	Hsu, Du[30]	$O(rm\log(n/r)+rm)$		Contours
1986	Myers, Miller[49]	$O(n(m-r))$	$O(n+m)$	Diagonalwise
1987	Apostolico, Guerra[7]	$O(pm\log(n)+\min(ds,pm))$	$O(\sigma m+n)$	Contours
1987	Kumar, Rangan[37]	$O(n(m-r))$		Contours
1989	Kuo, Cross[38]	$O(\sigma+n(r+\log(n)))$	$O(n+r)$	Row by Row
1990	Chin, Poon[13]	$O(\sigma n+\min(\sigma d,rm))$	$O(n+r)$	Contours
1990	WMMM[62]	$O(n(m-r))$	$O(n+m)$	Diagonalwise
1992	Apostolico el. al.[6]	$O(n(m-p))$	$O(\sigma m+n)$	Contours
1995	Claus Rick[54]	$O(n\sigma+\min(pm,p(n-p))$	$O(n\sigma+d)$	Contours
1999	Goeman, Clausen[22]	$O(\min(pm,m\log(m)+p(n-p)))$		Contours
1999	Claus Rick[55]	$O(\min(pm,p(n-p))$	$O(n\sigma+d)$	Contours
2006	Budalakoti,Cruz[58]	$O(n\min(\frac{r}{n},LCS)\log(\frac{n^2}{r}))$	$O(n+m)$	Row by Row
2008	Iliopoulos,Rahman[33]	$O(R\log(\log n))+n)$		

Table: List of existing LCS algorithms for two strings

IV. Conclusions

We may say that edit distance and LCS are dual of each other as edit distance gives us the number of operations required to transform one string into another, whereas LCS gives us the common subsequence in two strings. LCS is a special case of edit distance as by allowing only insertions and deletions at cost 1 we can compute the LCS. If we consider two strings of length m and n, with m<n, then we can form a relation between edit distance ed and lcs as **n-lcs ≤ ed ≤ n+m – 2lcs**, and if n=m then we can say that **n-lcs ≤ ed ≤ 2(n-lcs).**

One type of edit distance is the **Hamming distance , which gives us a linear time but it does not give us optimal results** for σ>2. Even for σ=2 it would give us near optimal results and in the worst case it would give way below optimal results. So we rule out Hamming distance for our problem.

Then we come to **Myers**' differential algorithm, which has complexity in terms of D, the edit distance. For our problem of approximate string matching we consider m<<n so we would get the edit distance as ed ≥ n-m, and since m<<n, we can write ed ≥ n. So our **complexity becomes O((n+m)*n) = O(n^2 + nm) = O(n^2)** which is asymptotically greater than O(mn) for classical approach to LCS or edit distance. So Myers' algorithm is not well suited to our problem. It is more suited for DNA comparisons where D is very small as compared to lengths of the strings, and the lengths of the two strings are approximately equal. Also note that the worst case of Myers' algorithm corresponds to the best case of our algorithm, and its best case corresponds to the worst case of our algorithm as it is based on edit disatnce and our algorithm is based on the number of matches, which have an approximately inverse square relationship.

And lastly we come to Masek and Paterson algorithm. It has the best time for worst case among all algorithms. However, it has some limitattions as it works for finite alphabets only, and costs of edit operations are integral multiples of a single positive real number. So it is not a practical algorithm.

V. Future Scope

In this section we present new ideas and further improvements in the field of computing longest common subsequence.

1. What to do next?

Even the field of the longest common subsequence has been intensively studied over past fifty years, there is still a lot to do. We will consider only the LCS problem, not the constrained longest common subsequence and other problems.

2. Reduce the overhead factor of the finite automata approach Finite automata are able to explore fewer states than other algorithms and the experiments have confirmed it. Because no algorithm for more strings is developed, finite automata remain the best choice. For two strings, WMMM and Kuo-Cross are said to be fastest known. Problem is in the large overhead factor of a lookup table, a queue and other parts.

3. Gene LCS for more strings String alignment algorithms have not been covered in the work, but the longest common subsequence problem has a huge potential in this area. FASTA and BLAST algorithms and its modifications are used for genetic, but are only approximate not exact. Developing an algorithm for comparing millions of DNA sequences in real time would revolutionize genetic engineering.

4. Memory management and cache optimization As the length of strings grows, the memory exceeds the capacity of a computer and swapping plus page faults slow down the system. With a little effort a strategy of computing could be changed to minimize page faults. But this should be the last chance to improve the performance of implemented algorithms. More effective would be to think about a different algorithm with the lower time complexity suitable for average input strings.

5. Algorithms for more strings We have presented some algorithms which work for two and more strings. There are algorithms developed for two strings; are fast and do not require a large amount of the memory space. Almost all of them run in the linear time proportional to the sum of lengths of strings. The question is, are paradigms extendable to more dimensions for more strings? The general problem is in mapping of multi dimensional space to linear memory block.

6. Parallel algorithms Since 1990s the field of parallel algorithms of the LCS has been extensively studied. Parallel algorithms have been completely omitted in the work. We only refer readers to articles discussing parallel modifications of algorithms.

7. Algorithms for compressed strings Even if the algorithm is efficient, it still could not be enough to meet low demands on the memory usage. Consider we have a large set of input strings of billion symbols (for example a set of DNA sequences), it will be impractical to have all inputs in the main memory. A group of authors from Tohoku University(Japan) developed algorithms for computing the longest common subsequence of two given SLP(straight line programs)-compressed strings in $O(n^4 \log(n))$ time with $O(n^3)$ space, and in $O(n^4)$ time with $O(n^2)$ space, respectively, where n denotes the size of the input SLP-compressed strings. Some algorithms which are able to compute the LCS of run-length-encoded strings have been presented recently, but no algorithm for widely used compression algorithms (for example LZW or DCA) exists and remains for further research.

References

[1] Efficient Algorithms for Finding a Longest Common Increasing Subsequence: Wun-Tat Chan, Yong Zhang, Stanley P.Y. Fung, Deshi Ye, and Hong Zhu X. Deng and D. Du (Eds.): ISAAC 2005, LNCS 3827, pp. 665–674, 2005.

[2] J. W. Hunt and T. G. Szymanski. A fast algorithm for computing longest common subsequences Commun. ACM 20(5), 350–353 (1977)

[3] Wagner, R.A., Fischer, M.J.: The string-to-string correction problem J. ACM 21(1), 168–173 (1974)

[4] Masek, W.J., Paterson, M.: A faster algorithm computing string edit distances J. Comput. Syst. Sci., 20(1):18–31, 1980.

[5] Myers, E.W.: An o(nd) difference algorithm and its variations Algorithmica 1(2), 251–266 (1986)

[6] Levenshtein, V.I.: Binary codes capable of correcting deletions, insertions, and reversals Probl. Inf. Transm. 1, 8–17 (1965)

[7] Edit distance for a run-length-encoded string and an uncompressed string J.J. Liu , G.S. Huang , Y.L. Wanga , R.C.T. Lee Information Processing Letters 105 (2007)

[8] Introduction to algorithms Thomas Cormen, Charles Lieserson, Ronald Rivest MIT press, Mc Graw Hill publications , Twenty-fifth printing 1990, Page 301-31

[9] Error detecting and error correcting codes, R.W. Hamming, The Bell System Technical Journal, Volume 29, Number 2 , 60-74, April 1950

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

[11] V. L. ARLAZAROV, E. A. DINIC, M. A. KRONROD, AND I. A. FARADZEV, On economic construction of the transitive closure of a directed graph, Dokl. Akad, Nauk SSSR 194 (1970), 487-488

[12] J. E. HOPCROFT, W. J. PAUL, AND L. G. VALIANT, On time versus space and other related problems, in Proceedings, 16th Annual Symposium on Foundations of Computer Science, Berkeley, 1975," pp. 57-6

[13] A. V. AHO, J. E. HOPCROFT, AND J. D. ULLMAN, "The Design and Analysis of Computer Algorithms," Addison-Wesley, Reading, Mass., 1974.

[14] C. S. Iliopoulos and M. S. Rahman. New efficient algorithms for the lcs and constrained lcs problems. Inf. Process. Lett., 106(1):13–18, 2008.

[15] R. W. Irving and C. Fraser. Two algorithms for the longest common subsequence of three (or more) strings. In CPM '92: Proceedings of the Third Annual Symposium on Combinatorial Pattern Matching, pages 214–229, London, UK, 1992. Springer-Verlag.

[16] T. Jansen and D. Weyland. Analysis of evolutionary algorithms for the longest common subsequence problem. In GECCO '07: Proceedings of the 9th annual conference on Genetic and evolutionary computation, pages 939–946, New York, NY, USA, 2007. ACM.

[17] T. Jiang and M. Li. On the approximation of shortest common supersequences and longest ommon subsequences. SIAM J. Comput., 24(5):1122–1139, 1995.

[18] S. K. Kumar and C. P. Rangan. A linear space algorithm for the lcs problem. Acta Inf., 24(3):353–362, 1987.

[19] S. Kuo and G. R. Cross. An improved algorithm to find the length of the longest common subsequence of two strings. SIGIR Forum, 23(3-4):89–99, 1989.

[20] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions, & reversals. Technical Report8, 1966.