Improving an Approximate String Matching Algorithm

Ahmad Osama Maani[1], and Dr. Essam Al-Daoud[2]

[1] Department of Graduate Studies, Zarqa University , Zarqa 13132, Jordan

[2] Department of Computer Science Zarqa University , Zarqa 13132, Jordan

Abstract

In this paper, an improved algorithm that solves the Approximate String Matching (ASM) problem based on the Longest Common Subsequence (LCS) is presented. The LCS is defined as the longest common subsequence of characters that appears left-to-right between two given strings or substrings, a text called T, and a pattern called p. The improved algorithm solves the problem in less time and uses less space than the conventional LCS algorithm. It does this via a new technique based on the use of a one-dimensional array to order the matching characters between T and p, whereas the original LCS algorithm uses a two-dimensional matrix to order the matching characters, which requires more time and space.

Keywords: Algorithm, Approximate String Matching, Longest Common Subsequence.

1. Introduction

String matching is a conventional problem in computer algorithms, and is fundamental in many applications that require the processing of string or sequence data. It often involves finding the occurrences of a pattern string in a given text, and is utilized for spell checking in text editors, DNA strings comparison, identity and password validation and data checking at system login, and content interpretation in document and programming language parsers. Further, it is the base of many applications used in fields such as bioinformatics, linguistics, visual retrieval and classification, and pattern matching in digitalized images.

Over the years, researchers have introduced a variety of string matching algorithms that have been applied to various areas of study and research [1]. The literature also contains a large number of research papers that provide theoretical and empirical outcomes to the problem, with advanced space and time efficiencies.

1.1 Problem Definition

The String Matching Problem (SMP), which has given rise to string search algorithms, can be defined as the process of finding the approximate location of one or several strings within a larger string or text. Let T be an alphabet (finite set); then, formally, both the pattern and large texts searched are vectors of elements of *T*. It is important to take into consideration that *T* may be a human alphabet like {A-Z}; other applications may use binary alphabets, as in {1, 0}, or DNA alphabets, as in {A, C, G, T}, in bioinformatics applications.

The text T that the algorithms deal with is considered an array *T* [1 ... m] of length m, and the pattern *p* is an array [1 ... n] of length n, with n <= m. The character arrays *T* and *p* are often called strings of characters. It might be concluded that pattern *P* occurs with shift s in text *T*, if 0 <= s <= m-n and *T* [t+1 …. t + m] = *p* [1 ... M]. If *p* occurs with shift s in T, then t is called a valid shift; otherwise, it is called an invalid shift. The string matching algorithm tries to solve the problem of finding all valid shifts *p* occurring in a given text *T* [2].

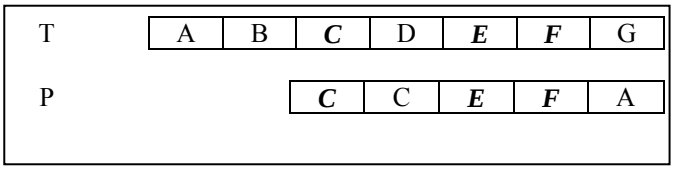

Fig.1 Approximate string matching.

2. LCS

The LCS problem is as follows. The code of the target text is given two strings—string or pattern p of length n, and string T of length m. The goal here is to produce their LCS, the longest subsequence of characters that appears left-to-right (but not necessarily in a contiguous block in both strings [3].

ENQUIRING

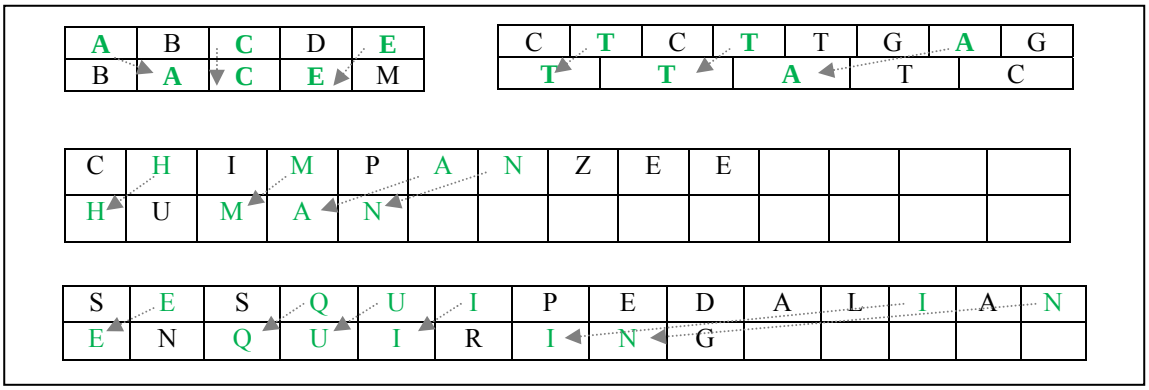

Fig.2 LCS examples.

2.1 Example 1

T = BACBADCC

p = ABAZDC

In Example 1, the LCS has length four, and is the string ABAD. Another way to look at the problem is to consider finding a 1:1 matching between some of the letters in p and some of the letters in T such that none of the edges in the matching cross each other. This type of problem frequently arises in genomics, where, given two DNA fragments, the LCS gives information about what they have in common and the best way to line them up. Let us now solve the LCS problem using dynamic programming.

As sub-problems, the code looks at the LCS of a prefix of p and a prefix of T, running over all pairs of prefixes. For simplicity, in the beginning, the code focuses on finding the length of the LCS and then the algorithm is modified to produce the actual sequence itself. Consequently, the question arises, say LCS[i, j] is the length of the LCS of p[1 … i], with T[1 ... j].
We consider how LCS[i, j] can be solved in terms of the LCS's of the smaller problems.

Case 1: If p[i] ≠ T[j], then, the preferred subsequence has to neglect one of p[i] or T[j]. This means that

$$LCS[i, j] = \max(LCS[i - 1, j], LCS[i, j - 1]).$$

Fig.3 LCS algorithm, Case 1.

Case 2: If p[i] = T[j], then, the LCS of p[1 … i] and T[1 ... j] may match them as well. For example, if a common subsequence is observed to match p[i] to an earlier location in T, for example, it can always be matched with T[j] instead. Consequently, in this case,

$$LCS[i, j] = 1 + LCS[i - 1, j - 1].$$

Fig.4 LCS algorithm, Case 2

Thus, two loops can iterate (over values of *i* and *j*), filling in the LCS using these rules. Following is a pictorial depiction of the above example, with *p* along the leftmost column and *T* along the top row.

3. Analysis of LCS

Starting with the LCS algorithm, the code needs to be traced in order to understand the LCS technique. This step initializes the search for repetitions and void steps in the code, and can either be edited or completely skipped in order to reduce the number of operations. During this phase, other techniques that eliminate the redundant operations while providing the existing results can be utilized.

```
procedure LCS(p, t) {m = |p|, n = |t|}
 begin
        for i ← 0 to m do L[i, 0] ← 0
        for j ← 0 to n do L[0, j] ← 0
        for i ← 1 to m do
                for j ← 1 to n do
                        if pi = tj then
                                L[i, j] ← L[i − 1, j − 1]+1
                        else
                                if L[i, j − 1] > L[i − 1, j] then
                                        L[i, j] ← L[i, j − 1]
                                else
                                        L[i, j] ← L[i − 1, j]
                                end if
                        end if
                end loop
        end loop
        return L[m][n]
end
```

Fig. 5 LCS pseudo code.

Reviewing the matrix form of Example 1, it is clear that each character in p must be compared to each character in T, resulting in a cost size of m × n. In order to analyze whether p(i) = T(i), an operation that facilitates the filling of all the cells in the matrix needs to be performed. Even when a match case is found, the comparison continues at the same row. For example, In Table 1, p(2) is compared to T(1) to find that both characters are (A), and the value of L[2][1] is changed to L[1][0] +1. This process keeps running on the rest of the cells in the same row. Each time this rule is applied, the match case is found. Consequently, the improved algorithm is built from that point. At the same time the code will stop comparing cells at the match case cell, and it will move to the next row; therefore, this process saves time and eliminates the redundant operations in each row.

3.1. Example 2

T = XACBZDCO

p = ABZSDC

(See Table 1)

In Table 1, if a match is found at cell L[2][1], the previous diagonal cell, L[1][0], is selected, one added to the value in it, and the new value stored in L[2][1]. Conversely, if no match is found, the largest value among L[i-1][j-1] and L [i][j-1] is selected and stored in the compared cell, and so on until i = m. Thus, the ELSE case (p ≠ t) is used to carry the value to the next cells and keep the weight running in the matrix, whereas the IF case (p = t) is used to increase the weight in the matrix as long as a match is found (See Table 2).

Table 2. LCS tracing j = 6.

	€	X	A	C	B	Z	D	C	O
€	0	0	0	0	0	0	0	0	0
A	0	0	1	1	1	1	1	1	1
B	0	0	1	1	2	2	2	2	2
Z	0	0	1	2	2	3	3	3	3
S	0	0	1	2	2	3	3	3	3
D	0	0	1	2	3	3	4	4	4
C	0	0	1	2	3	3	4	5	0

	€	X	A	C	B	Z	D	C	O
€	0	0	0	0	0	0	0	0	0
A	0	0	1	1	1	1	1	1	1
B	0	0	1	1	2	2	2	2	2
Z	0	0	1	2	2	3	3	3	3
S	0	0	1	2	2	3	3	3	3
D	0	0	1	2	3	3	4	4	4
C	0	0	1	2	3	3	4	5	5

4. Improved LCS (ImpLCS) Methodology

By analyzing the LCS code and tracing the outputs inside the matrix, two rules on which the matrix structure is based can be extracted. The first rule states that when there is a match, the algorithm should extract the value in cell L[i-1][j-1], add one to it, and store it in the current cell, L[i][j] in this case. In other words, the code extracts the largest value among cells L[1][1] to L[i-1][j-1] and stores it in L[i][j]. The second rule states that the value of the first match in the row should be carried to the other cells in the same row but to the right of the current cell, or any larger value if a new match is found.

The proposed improved algorithm is based on the above rules and changes in the structure of the original algorithm that enables it to output the same results, but using fewer operations. The LCS uses the ELSE case (p ≠ t) to carry the weight in the matrix. Further, inside the ELSE case there is another IF statement that decides which value to store in the current cell. It is worth mentioning here that the ELSE case is applied 3/4 of the times in the case of a DNA application (four characters) and 27/28 of the times for a regular text application containing all the letters of the alphabet (28 characters).

Table 1. LCS tracing j = 1.

	€	X	A	C	B	Z	D	C	O
€	0	0	0	0	0	0	0	0	0
A	0	0	1	0	0	0	0	0	0
B	0	0	0	0	0	0	0	0	0
Z	0	0	0	0	0	0	0	0	0
S	0	0	0	0	0	0	0	0	0
D	0	0	0	0	0	0	0	0	0
C	0	0	0	0	0	0	0	0	0

	€	X	A	C	B	Z	D	C	O
€	0	0	0	0	0	0	0	0	0
A	0	0	1	1	1	1	1	1	1
B	0	0	0	0	0	0	0	0	0
Z	0	0	0	0	0	0	0	0	0
S	0	0	0	0	0	0	0	0	0
D	0	0	0	0	0	0	0	0	0
C	0	0	0	0	0	0	0	0	0

This means that an additional cost, ¾ (m × n), is added to the total cost of the algorithm. At the end of the LCS algorithm, extra code to extract the output from the m × n matrix is present. The cost of the extra code is also m × n. The proposed improved algorithm avoids these extra costs by using a one-dimensional array to store the output, which results in a cost of only n when extracting the output.

4.1. Analysis of ImpLCS

As a first design of the code, the match case needs to be considered and, instead of carrying the value to the next cell in the matrix through the ELSE case (p ≠ t), an array should be used to carry the values in each time a match is found. That step is to be included in the IF case (p = t), and this array called "Last_row[ ]." The first thing that needs to be done is to place the value of (Last_row[j]+1) in the matrix at position L[j][i] where the match was found.

```
if pi = tj then
        L[j][i] ← Last_row[j]+1
end if
```

Fig. 6 ImpLCS pseudo code, Phase 1.

The new value is then carried to the rest of the cells in the array and further carries the weight of the matrix in a one-dimensional array rather than a two-dimensional array. Instead of looking for the largest value between the previous row and the previous column, as in the LCS, in ImpLCS, the value required is found immediately in the previous cell in the Last_row array. The code fills those cells that contain a value that is less than the newly carried one; otherwise, the code will skip and break the loop in order to reduce the number of steps. This may appear to be a different technique than the one in LCS, but the running time in ImpLCS is less.

Improved results are displayed later in tables and charts that show that the running time of ImpLCS is much less because the aim here is to not have to visit all cells and also not have to visit the same cell more than once. These steps are shown outlined in the following code (Fig. 7):

```
procedure LCS(p, t) {m = |p|, n = |t|}
begin
    for j← 1 to n do
        for i ← 1 to m do
            if pj = ti then
                L[i][j] ← Last_row[i]+1
                for k← i+1 to m do
                    if Last_row[i]+1 > Last_row[k]
                    Last_row[k] ← Last_row[k]+1;
                    If k=m i← m;
                else
                        i← k-1; k← m;
                    end loop
            end if
        end loop
    end loop
end
```

Fig.7 ImpLCS, Phase 2.

The above code performs fewer operations on the same p and T. In example 3, it shows in the tables that the black cells are the visited cells, which means also zero operation. The cells in blue are also visited cells but they present the number of operations in the matrix or in the array Last_row. The white cells are the non-visited cells.

4.1.1 Example 3

LCS = surey
LLCS =5

(See Table 3)

Table 3 Example 3—LCS j = 1; ImpLCS j = 1.

	LCS								
P \ T	0 €	1 S	2 U	3 R	4 G	5 E	6 R	7 Y	No. of Op
0-€	0	0	0	0	0	0	0	0	
1-S	0	1	1	1	1	1	1	1	7
2-U	0	0	0	0	0	0	0	0	
3-R	0	0	0	0	0	0	0	0	
4-V	0	0	0	0	0	0	0	0	
5-E	0	0	0	0	0	0	0	0	
6-Y	0	0	0	0	0	0	0	0	

	ImpLCS							
P \ T	0 €	1 S	2 U	3 R	4 V	5 E	6 Y	No. of Op
0-€	0	0	0	0	0	0	0	
1-S	0	1	0	0	0	0	0	7
2-U	0	0	0	0	0	0	0	
3-R	0	0	0	0	0	0	0	
4-G	0	0	0	0	0	0	0	
5-E	0	0	0	0	0	0	0	
6-R	0	0	0	0	0	0	0	
7-Y	0	0	0	0	0	0	0	
	0	1	1	1	1	1		

Table 4 Example 3—LCS j = 2; ImpLCS j = 2.

	LCS									ImpLCS								
P \ T	0 €	1 S	2 U	3 R	4 G	5 E	6 R	7 Y	No. of Op	P \ T	0 €	1 S	2 U	3 R	4 V	5 E	6 Y	No. of Op
0-€	0	0	0	0	0	0	0	0		0-€	0	0	0	0	0	0	0	
1-S	0	1	1	1	1	1	1	1	7	1-S	0	1	0	0	0	0	0	7
2-U	0	1	2	2	2	2	2	2	7	2-U	0	0	2	0	0	0	0	5
3-R	0	0	0	0	0	0	0	0		3-R	0	0	0	0	0	0	0	
4-V	0	0	0	0	0	0	0	0		4-G	0	0	0	0	0	0	0	
5-E	0	0	0	0	0	0	0	0		5-E	0	0	0	0	0	0	0	
6-Y	0	0	0	0	0	0	0	0		6-R	0	0	0	0	0	0	0	
										7-Y	0	0	0	0	0	0	0	
												0	1	2	2	2	2	

Table 5 Example 3—LCS j = 3; ImpLCS j = 3.

	LCS									ImpLCS								
P \ T	0 €	1 S	2 U	3 R	4 G	5 E	6 R	7 Y	No. of Op	P \ T	0 €	1 S	2 U	3 R	4 V	5 E	6 Y	No. of Op
0-€	0	0	0	0	0	0	0	0		0-€	0	0	0	0	0	0	0	
1-S	0	1	1	1	1	1	1	1	7	1-S	0	1	0	0	0	0	0	7
2-U	0	1	2	2	2	2	2	2	7	2-U	0	0	2	0	0	0	0	5
3-R	0	1	2	3	3	3	3	3	7	3-R	0	0	0	3	0	0	0	4
4-V	0	0	0	0	0	0	0	0		4-G	0	0	0	0	0	0	0	
5-E	0	0	0	0	0	0	0	0		5-E	0	0	0	0	0	0	0	
6-Y	0	0	0	0	0	0	0	0		6-R	0	0	0	0	0	0	0	
										7-Y	0	0	0	0	0	0	0	
												0	1	2	3	3	3	

Table 6 Example 3—LCS j = 4–5; ImpLCS j = 4–5.

	LCS									ImpLCS								
T \ P	0 €	1 S	2 U	3 R	4 G	5 E	6 R	7 Y	No. of Op	p \ T	0 €	1 S	2 U	3 R	4 V	5 E	6 Y	No. of Op
0-€	0	0	0	0	0	0	0	0		0-€	0	0	0	0	0	0	0	
1-S	0	1	1	1	1	1	1	1	7	1-S	0	1	0	0	0	0	0	7
2-U	0	1	2	2	2	2	2	2	7	2-U	0	0	2	0	0	0	0	5
3-R	0	1	2	3	3	3	3	3	7	3-R	0	0	0	3	0	0	0	4
4-V	0	1	2	3	3	3	3	3	7	4-G	0	0	0	0	0	0	0	0
5-E	0	1	2	3	3	4	4	4	7	5-E	0	0	0	0	0	4	0	2
6-Y	0	1	2	3	3	4	4	4	7	6-R	0	0	2	0	0	0	0	1
										7-Y	0	0	0	0	0	0	0	
												0	1	2	3	3	4	

Table 7 Example 3—LCS j = 6; ImpLCS j = 6.

	LCS									ImpLCS								
P \ T	0 €	1 S	2 U	3 R	4 G	5 E	6 R	7 Y	No. of Op	P \ T	0 €	1 S	2 U	3 R	4 V	5 E	6 Y	No. of Op
0-€	0	0	0	0	0	0	0	0		0-€	0	0	0	0	0	0	0	
1-S	0	1	1	1	1	1	1	1	7	1-S	0	1	0	0	0	0	0	7
2-U	0	1	2	2	2	2	2	2	7	2-U	0	0	2	0	0	0	0	5
3-R	0	1	2	3	3	3	3	3	7	3-R	0	0	0	3	0	0	0	4
4-V	0	1	2	3	3	3	3	3	7	4-G	0	0	0	0	0	0	0	0
5-E	0	1	2	3	3	4	4	4	7	5-E	0	0	0	0	0	4	0	2
6-Y	0	1	2	3	3	4	4	4	7	6-R	0	0	2	0	0	0	0	1
										7-Y	0	0	0	0	0	0	0	
												0	1	2	3	3	4	

Table 8 Example 3—LCS; ImpLCS j = 7.

	LCS									ImpLCS								
P \ T	0 €	1 S	2 U	3 R	4 G	5 E	6 R	7 Y	No. of Op	P \ T	0 €	1 S	2 U	3 R	4 V	5 E	6 Y	No. of Op
0-€	0	0	0	0	0	0	0	0		0-€	0	0	0	0	0	0	0	
1-S	0	1	1	1	1	1	1	1	7	1-S	0	1	0	0	0	0	0	7
2-U	0	1	2	2	2	2	2	2	7	2-U	0	0	2	0	0	0	0	5
3-R	0	1	2	3	3	3	3	3	7	3-R	0	0	0	3	0	0	0	4
4-V	0	1	2	3	3	3	3	3	7	4-G	0	0	0	0	0	0	0	0
5-E	0	1	2	3	3	4	4	4	7	5-E	0	0	0	0	0	4	0	2
6-Y	0	1	2	3	3	4	4	5	7	6-R	0	0	2	0	0	0	0	1
										7-Y	0	0	0	0	0	0	5	1
												0	1	2	3	3	4	20

The code works very well for simple texts and patterns; however, to verify its accuracy for ImpLCS more complex examples are applied below. It is possible to find more than one match in each row or cell. The following example shows how ImpLCS successfully improves the correct output in the same case.

Outputs

LCS = surey

LLCS =5

5. ImpLCS Results

The results show that the outputs are compatible in both ImpLCS and LCS, taking into consideration that no cell has been ignored or visited more than once in ImpLCS. After the improved outputs, the improvement in running time needs to be checked for some examples over the original LCS and ImpLCS. The length of T will first be varied in many of the examples, whereas the length of p

will be changed later in order to have the algorithms running under different conditions. Note that the tested examples use the 28 letters of the alphabet to generate random texts and patterns.

Table 9 Comparison of LCS and ImpLCS with p constant

Pattern len	Text len	LCS (ms)	ImpLCS (ms)
6	100000	0.015	0
6	300000	0.031	0.016
6	600000	0.063	0.015
6	900000	0.092	0.032
6	1200000	0.109	0.036
6	1600000	0.156	0.047
6	2000000	0.250	0.049
6	2400000	0.300	0.052
6	2700000	0.330	0.055
6	3000000	0.360	0.058

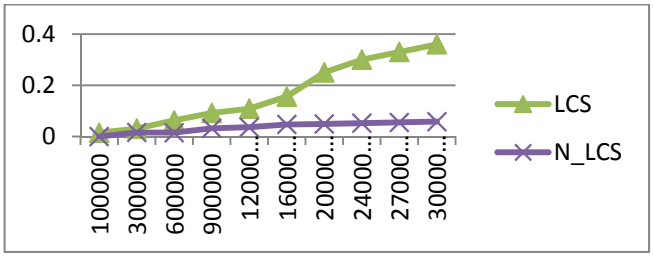

Fig.8 Graphical comparison of LCS and ImpLCS with p constant.

The results displayed in Table 9 and Fig. 8 show that ImpLCS has a lower processing time than LCS when the pattern length is six and lengths of T up to 3,000,000. The results for constant T and varying p are shown in Fig. 9.

Table 10 Comparison of LCS and ImpLCS with T constant.

Pattern len	Text len	LCS (ms)	ImpLCS (ms)
6	10000	0	0
50	10000	0.016	0
100	10000	0.019	0
200	10000	0.031	0.016
300	10000	0.046	0.017
500	10000	0.078	0.032
800	10000	0.125	0.047
1000	10000	0.156	0.051
1500	10000	0.219	0.094
2000	10000	0.359	0.141

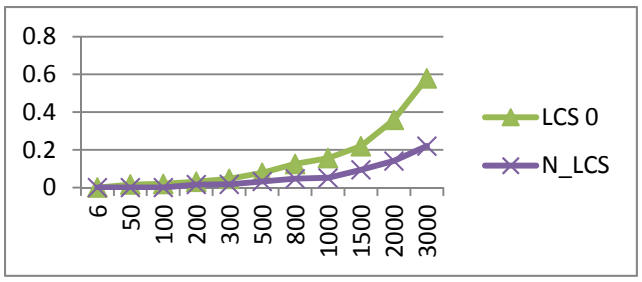

Fig. 9 Comparison of LCS and ImpLCS with T constant.

It is clear from Tables 9 and 10 that the results for ImpLCS are correct and effective. Further, the processing time is reasonable and less than that of LCS.

5.1 ImpLCS improvements

The running time in the average case for LCS can be estimated by considering only the total number of comparisons. At the start, there are two nested loops that run m × n times. These loops are followed by a comparison operation (p = t) that runs m × n times. In the case of match, which has a probability of 1/4 (in a DNA string where only four characters are used) only one operation takes place, but no comparison.

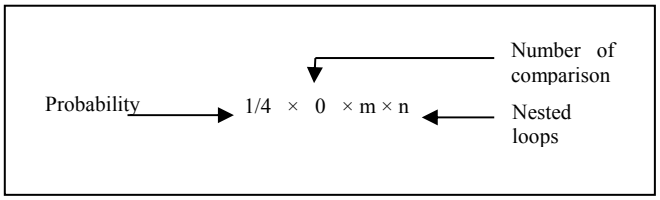

Fig.10 Cost of the IF case in the LCS alg. = 0.

The cost of the match case in the LCS algorithm is zero, which therefore does not add to the total cost. However, in the ELSE case (p ≠ t) with the probability 3/4, a comparison statement is present.

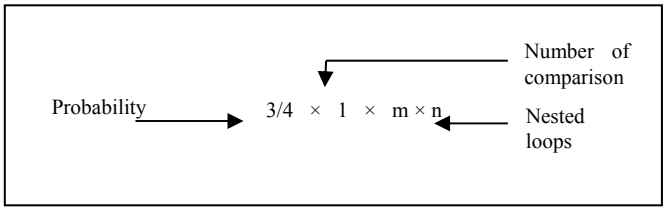

Fig.11 Cost of the ELSE case in LCS alg. = 3/4 m × n.

The total cost of LCS is the cost of (m × n) + cost of the IF case + cost of the ELSE case. This equates to a cost for

LCS of A(m × n) = (m × n) + (¼ × 0 × m × n) + (¾ × 1 × m × n) = 1.75 m × n.

To calculate the cost in the average case for ImpLCS, we start by considering the two loops of size m and n, as in the LCS m × n. However, because the second m loop is controlled by the third loop, on average, half of m will be performed. Thus, the running time for the first two loops is (m/2) × n. In the match case (p = t) with probability 1/4, there is another comparison statement inside the third loop. This statement controls and continues the running of the second loop. Moreover, it makes the probability of the inner comparison taking place m/2; on reaching the inner comparison, two probabilities are considered.

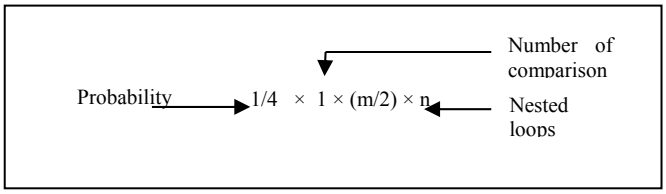

Fig.12. Cost of the IF case in the ImpLCS alg. = 1/8 × m × n.

Thus, the average cost, A(m × n) = m/2 × n + 1/8 × m × n = 0.62 m × n. This is less than the average case for LCS (i.e., less than 1.75 m × n).
In addition, after the LCS algorithm, another algorithm used to print out the results. This code has cost m × n, whereas in ImpLCS, the extra code only runs with a cost of n.

LCS cost without printing outputs	ImpLCS cost without printing outputs
1.75 m × n	0.62 m × n
LCS cost with printing outputs	ImpLCS cost with printing outputs
2.75 m × n	0.62 m × n + n

Fig.13 Cost of LCS and ImpLCS.

The big-O notation for both algorithms is O(m × n). However, a comparison of the algorithms at the second level (the average case) shows that ImpLCS performs better when A (m × n) is considered. Further, the results of tabulation of the running time for both algorithms confirm that ImpLCS is better.

6. Conclusion

The paper has presented a new, improved approximate string-matching algorithm, which is reasonably accepted on average for low and intermediate deference ratios (up to 1/2). Improving the results from original algorithms, the ImpLCS algorithm gave improved results that save time and space. Also proved theoretically and experimentally. It also gave the needed results plus alternative results. The algorithm may give better results based on the structure of the used text and patterns implemented.

The complexity of the improved algorithm was the same as compared to the original algorithm which is O(m.n). However, the cost in the average case has reduced in a noticeable way. Based on that, it is not a big change if only the complexity is considered. On the other hand, as far as enhancing the algorithm is concerned, it is a positive step to achieve more improvements. The concept of solving this problem was by using a matrix of size (m.n). The possibility might exist in the future to solve this problem considering any other alternative by trying a liner method that would cost less time and space comparing to the previous cost.

In a future research, this algorithm could be combining to another improved algorithm in order to achieve a special function in one of the computation theories fields or other fields such as bioinformatics and DNA formulas or even image processing and text searching methods. The scope to use such enhanced algorithm is wide since many life applications use the string matching and comparing.

References

[1] Jokinen, P., Tarhio, J., & Ukkonen, E. A comparison of approximate string matching algorithms. *Software: Practice and* Experience, Vol. 26, No. 12, 1996, pp. 1439-1458.
[2] Navarro, G. A guided tour to approximate string matching. ACM computing surveys (CSUR),Vol. 33, No.1, 2001, 31-88.
[3] Manolis Christodoulakis, Costas S. Iliopoulos, Yoan José Pinzón Ardila .Simple Algorithm for Sorting the Fibonacci String Rotations,Theory and Practice of Computer Science, 2006, pp. 218-225. Springer Berlin Heidelberg.

Ahmad Maani graduated from Zarqa University with a MSc. in Computer Science. He also worked for Zarqa University as a programmer for two years while doing his research. His research interests include Computer Science, Mathmatics, and Algorhims.

Essam Al-Daoud is an Associate Professor in the Department of Computer Science at Zarqa University. He earned his PhD in Computer Science from Putra University, Malaysia in 2002. His research areas are Computer Science, Advance Machine Learning, and Cryptography. Al Daoud is also the author of over 26 peer- reviewed publications.