iJRASET
International Journal For Research in Applied Science and Engineering Technology

INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY

Volume: 3 **Issue:** VII **Month of publication:** July 2015

DOI:

www.ijraset.com

Call: 08813907089 | E-mail ID: ijraset@gmail.com

International Journal for Research in Applied Science & Engineering Technology (IJRASET)

An Efficient Algorithm for Approximate String Matching

Neha[1], Rachna Dhaka[2]
[1]M.Tech. Scholar
Department of Computer Science & Engineering
Deenbandhu Chhotu Ram University of Science & Technology (DCRUST), Sonepat.

Abstract— The 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). In our classical approach which is studied academically, we form a dynamic programming matrix to find our solution. In mathematics, computer science, and economics, dynamic programming is a method for solving complex problems by breaking them down into simpler sub problems. It is applicable to problems exhibiting the properties of overlapping sub problems which are only slightly smaller and optimal substructure. When applicable, the method takes far less time than naive methods.
Keywords— Longest Common Subsequence, Dynamic Programming, Space and Time Complexity.

I. INTRODUCTION

Approximate string matching is used is finding similar matches in search engines like yahoo or *google* when the user may want to see similar items, or may be 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:

A. Search engines
B. Database searching:
C. Spell Checking
D. Signal processing
E. File comparison
F. 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

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 complexity of Levenshstien distance is O(mn)

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.

The complexity for hamming distance is O(n)

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.

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.

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

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.

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.

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 represent ant 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.
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 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.
5) *Myers-Miller Algorithm***:** It is a diagonal-wise 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.

III.OUR ALGORITHM FOR LCS

In our classical approach which is studied academically, we form a dynamic programming matrix to find our solution. In mathematics, computer science, and economics, dynamic programming is a method for solving complex problems by breaking them down into simpler subproblems. It is applicable to problems exhibiting the properties of overlapping subproblems which are only slightly smaller and optimal substructure. When applicable, the method takes far less time than naive methods. The key idea behind dynamic programming is quite simple. In general, to solve a given problem, we need to solve different parts of the problem (subproblems), then combine the solutions of the subproblems to reach an overall solution. Often, many of these subproblems are really the same. The dynamic programming approach seeks to solve each subproblem only once, thus reducing the number of computations: once the solution to a given subproblem has been computed, it is stored or "memo-ized": the next time the same solution is needed, it is simply looked up. This approach is especially useful when the number of repeating subproblems grows exponentially as a function of the size of the input.

So in the LCS problem we are taking the dynamic programming approach. We construct a dynamic programming matrix which has m rows and n columns, corresponding to the two sequences of length m and n, respectively. So the number of cells in our matrix are m*n. Since we are processing each cell one by one, and the time for processing each cell is O(1), the total time required in the classical approach is O(mn).

In our algorithm we would attempt to reduce the time taken to solve the problem by processing only those cells (i, j) which are classified as matches; i.e. for which we have $X_i = Y_j$, where X and Y are the two respective sequences. The processing time for each match would be greater than O(1), but still this approach would be faster because the number of matches are generally very less as compared to the total number of cells.

A. Pre-processing: The inverted arrays

We talk of finding the matches first. But how do we point to the location of a particular match and how do we know which match to go to next? The first question would be how to find all the matches? Do we scan all the elements in the array to find the matches? But that would take a time of O(m*n), which would defeat our purpose. The solution to this dilemma is a tricky one: We can invert the second array, that is to make the value at an index into the index, and the index would become the value which will be fetched from the index which was previously the value. And since the values in a sequence may be repeated, now there will be multiple values associated with a single index.

Let us take the input string Y: DHBBAAA

The array can be represented as:

Index	Value
1	D
2	H
3	B
4	B
5	A
6	A
7	A

The inverted array y_inv would be as follows:

INDEX	VALUE
A	5, 6, 7
B	3, 4
D	1
H	2

Now, when we traverse the first array X: AAAA

Index	Value
1	A
2	A
3	A
4	A

We can directly assess the location of the character 'A' in the Y array because the index is stored the inverted array. For each index in X we know the value and we can get the index corresponding to the same character in Y through the inverted Y array. The matches are thus found as:

Index(X)	Value(X)	Index(Y_inv)	Value(Y_inv)	Matches
1	A	A	5, 6, 7	(1,5),(1,6),(1,7)
2	A	A	5, 6, 7	(2,5),(2,6).(2,7)
3	A	A	5, 6, 7	(3,5),(3,6),(3,7)
4	A	A	5, 6, 7	(4,5).(4,6),(4,7)

Hence the link is made from value(X) to index(Y_inv). The set of matches is given by:

{ (1,5) , (1,6) , (1,7) ,
(2,5) , (2,6) , (2,7) ,
(3,5) , (3,6) , (3,7) ,
(4,5) , (4,6) , (4,7) }

So we get total 3 * 4 = 12 matches corresponding to the matching character 'A'

B. Calculating the Rank of a match

Let $A = a_1a_2 \ldots a_n$ and $B = b_1b_2 \ldots b_n$ be the two input sequences of n numbers. We need some definitions. For $1 \leq i \leq n$, denote A_i the prefix of A with the first i numbers, i.e., $a_1a_2 \ldots a_i$. Similarly, we have $B_i = b_1b_2 \ldots b_i$. We say that (i, j) is a match of A

and B if $a_i = b_j$, and a_i (or b_j) is called the match value. Note that the rank R(i, j) is an intermediate result for the longest common subsequence and

$$R(i, j) = LCS[(X_1X_2\ldots X_i)(Y_1Y_2\ldots Y_j)]$$

1) *The dominating matches:* We say that a match (i', j') dominates another match (i, j) or that (i', j') is a dominating match of (i, j) if ai' < ai and i' < i and j' < j. It is a necessary condition for both ai' and ai (or bj' and bi) to appear in the LCIS. For every match (i, j), define the rank of the match, denoted by R(i, j), to be the length of the LCIS of Ai and Bj such that ai (and bj) is the last element of the LCIS. If the length of the LCIS of Ai and Bj corresponding to match (i, j) is k which is greater than 1, then there must be an LCIS of A[i] and B[j] corresponding to a match (i , j) with length k − 1 and i' < i, j' < j and ai' = bj' < ai. Therefore, R(i, j) can be defined by the following recurrence equation.

$$R(i,j) = \begin{cases} 0 & \text{if } (i,j) \text{ is not a match,} \\ 1 & \text{if } (i,j) \text{ is a match and thereis no match } (i',j') \text{ that} \\ & \text{dominates } (i,j), \\ 1+\max\{R(i',j') \mid (i',j') \text{ is a match that dominates } (i,j)\} & \\ & \text{otherwise.} \end{cases}$$

In other words, we are finding the rank of a given match from the matches in the candidate matrix of the match.

2) *The critical match:* We understand that we have to find the rank of a given match from among the dominating matches of that match. We can compare with each of the dominating matches, which would give us a running time of O(r^2). Note that r can be as large as m*n, so this is giving us a time of $O(m^2n^2)$ which is way too high. So we need to devise ways to find the largest dominating match efficiently. This is where the critical match comes in. Typically the match having the least value of i and j is the critical match, along with the condition that the rank of the critical match should be maximum.

	A(1)	B(2)	C(3)	D(4)	A(5)	B(6)	C(7)
A(1)	1				1		
B(2)		***2***				2	
C(3)			**3**				3
D(4)				4			
E(5)							

Consider the above example in which we have found Ranks of each of the matches. Note that the dominating matches for (4,4) are (1,1), (2,2) and (3,3). In this case (3,3) becomes the critical match since it is having maximum rank. Also note that the match(3,7) has two dominating matches with highest rank: (2,2) and (2,6). But the critical match would be (2,2) since it is having lower value of j.

3) *The partitions:* We partition the set of matching points $M=\{(i_1,j_1),(i_2,j_2)\ldots.(i_r,j_r)\}$ as per the corresponding ranks. At any time, let M be the set of matches whose ranks have been computed so far. Since the length of the LCS is l, there are at most l (disjoint) partitions of M, namely $M_1, M_2, \ldots, M_l$ where M_k contains the matches with rank k. Note that some of the partitions may be empty. Since the ranks of the matches are computed in ascending order of their values, when the rank of a match is to be computed, the ranks of all dominating matches of this match (if exist) should have been computed and those matches should appear in M. The l partitions of M possess a kind of monotone property that helps locating the largest rank dominating match of a given match efficiently. The property is that if no dominating match of a given match exists in a partition, say Mk, then no partition corresponding to rank larger than k contains dominating match of the given match.

C. Length of LCS of the two sequences

Length of LCS of the two sequences can be given by l, where l is the number of partitions we finally have for M. we need to store only the two critical points in each partition of M, corresponding to minimum i and j, respectively.

D. Constructing the LCS

We can construct the LCS backwards by finding the character corresponding to one of the critical matches in M_l. Then we check find the corresponding dominating match in M_{l-1} and note the corresponding character. We backtrack this way upto the match in M_1. Now the LCS may be displayed.

E. The algorithm

1) Start
2) Input char X[ ]
3) Input char Y[ ]
4) M = length(X)
5) N = length(Y)
6) For (i = 1 to n) do steps 7 to 9
7) j = Y[i]
8) Y_inv[j] -> value = i
9) Y_inv[j]= Y_inv[j] -> next
10) r = 0
11) for (i = 1 to m) do steps 12 to 18
12) yval = X[i]
13) Y_inv[yval] = head[yval]
14) If (Y[Y_inv[yval] -> value = yval) do steps 15 to 18
15) While(Y_inv[yval]) do steps 16 to 18
16) Rpl[r] = (I, Y_inv[yval] -> value,0)
// rpl[ ] stores the position of matches
// and corresponding length of common subsequence
17) r++
18) Y_inv[yval = Y_inv[yval] -> next
19) Maxlength = 0
20) For (i = 1 to r)
21) Flag[i] = 0
22) For (a = 1 to r) do steps 23 to 51
23) (i,j) = rpl[a] // take the position i,j of the match
24) k = min(maxlength,i,j)
25) if (k = 0) then do steps 26 to 32 else do steps 33 to 39
26) if(flag[k] = 0) then do steps 27 to 32
27) CP[0, 0] = (i, j)
// CP[ ] stores the critical points. Array size is l * 2
28) CP[0, 1] = (i, j)
29) Flag[0] = 1
30) Rpl[a] = (X,X,1) //change only the length of a^{th} match
31) Maxlength++
32) Goto 22
33) If(flag[k] = 0) then do steps 34 to 39
34) If a critical match at CP[k-1] dominates (i, j) then do steps 35 to 39
35) CP[k,0] = (i, j)
36) CP[k,1] = (i, j)
37) Rpl[a] = (X, X, k + 1)
38) Maxlength++
39) Goto 22
40) If (k != 0) then do steps 41 to 46
41) If a critical match at CP[k-1] dominates (i, j) then do steps 42 and 43 else do steps 44 to 46
42) If (k = maxlength) then do step 43
43) Maxlength++

44) If critical match at CP[0] dominates (i, j) do step 45 else do step 46
45) K = binary_search(i, j, 0, k-1)
46) K = 0
47) Rpl[a] = (X,X,k+1)
48) If(CP[k,0].X > i)
49) CP[k, 0] = (i, j)
50) If (CP[k, 1].Y > j)
51) CP[k,1] = (i, j)
52) Ch = 0
53) For (k = maxlength down to 1) do steps 54 to 58
54) i = CP[k, ch].X
55) lcs[k] = X[i]
56) if(CP[k-1.0] dominates CP[k, ch] then do step 57 else do step 58
57) ch = 0
58) ch = 1
59) print lcs
60) Stop

F. Complexity for the algorithm

1) Preprocessing

a. Inverting the second array Y takes time of O(m)

b. For each element X_i in X we can directly find the matches in Y. Time for this is given by O(n + r)

So the total time is given by O(m + n + r)

2) Time for finding R(i, j) for a given match (i, j) is done in time O(log l)

3) Processing time for all the matches = r * O(log l)
= O(r log l)

4) Total time = O(m + n + r log l)

a) Best case: The best case comes when r = 0, so the time = O(m + n) = O(n) for m < n.
b) Average case: The average case may be found by using probability.
The probability of any cell having a particular character = 1/σ, where σ is the size of the alphabet. Since there are total m*n cells in our dynamic programming matrix, so the average value of r becomes m * n / σ. Note that the actual value is more complex than this, but this value is sufficient for our purpose.
Also, the maximum length of string will be = m, assuming that m < n. we know that probability of a match of one character is 1 / σ, so the average length of the LCS will be m / σ.

So the complexity in average case = O(m + n + r lg l)
= O(m + n + mn/ σ *lg(m / σ))

c) Worst case: We have time = O(m + n + r log l). When r approaches m * n, time of the algorithm may exceed O(m * n) . Maximum value of l is given by minimum(m, n). Maximum time approaches O(mn log m) assuming that m < n.

Time = O(mn log m)

Note that when r = m * n, then in that case time for finding the rank of any match would be O(1). So the time would always be lesser than the above mentioned value.

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 **$n - lcs \leq ed \leq n + m - 2\ lcs$**, and if n = m then we can say that **$n - lcs \leq ed \leq 2\ (n - lcs)$.**

Our algorithm gives significantly better time in the average case and best case, though it is slightly degrading in the worst case, as compared to Levenshstien distance, LCS classical approach, Hunt and Szymanski algorithm and Masek and Paterson algorithm

One type of edit distance is the **Hamming distance , which gives us a linear time but it does not give us optimal results** for $\sigma > 2$. Even for $\sigma = 2$ 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.

So on the basis of our analysis and our findings we can say that our algorithm is best suited for the problem of approximate string matching.

V. FUTURE SCOPE

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

A. 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.

B. 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.

C. Gene LCS for more strings

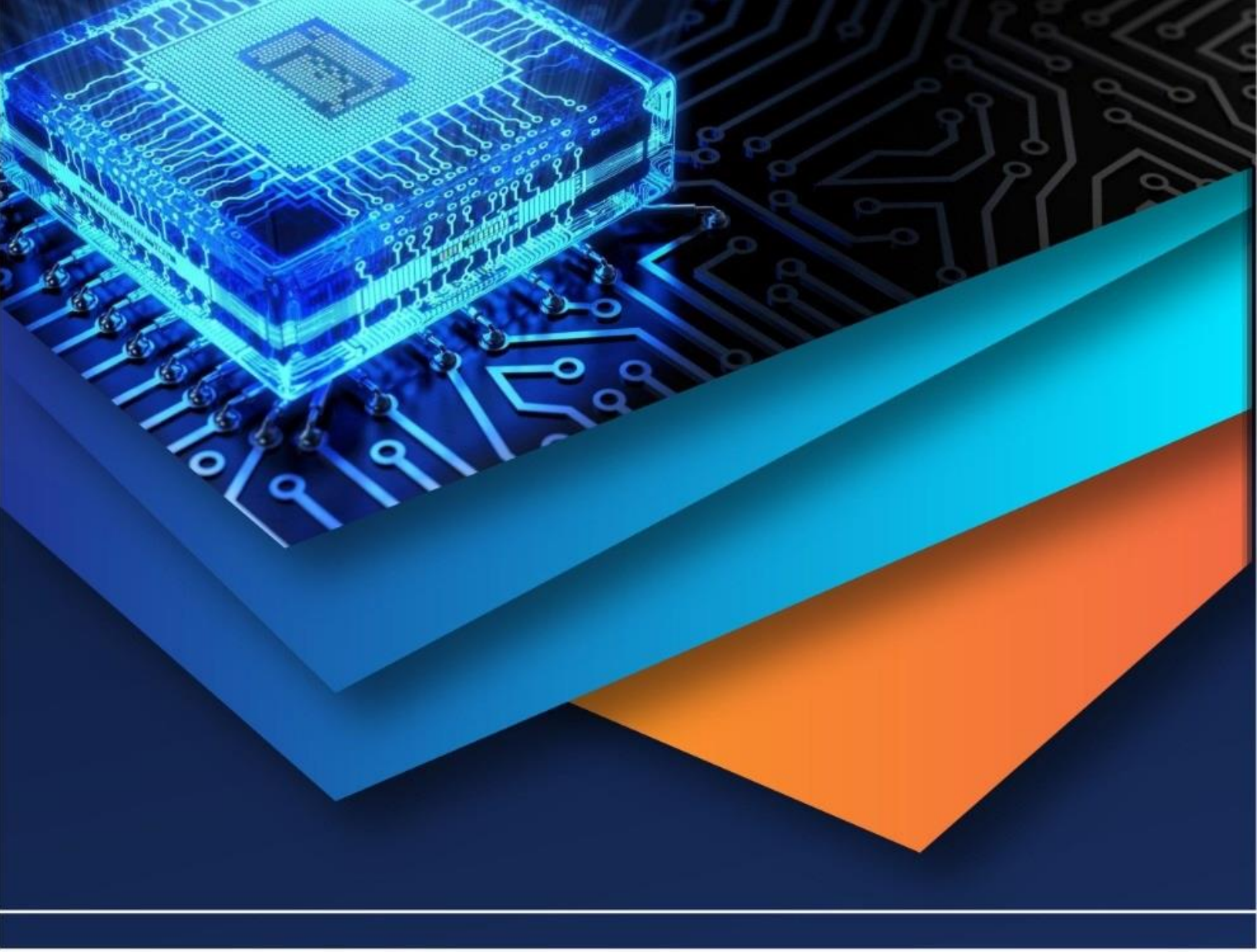
String alignment algorithms have not been covered in the work, but the longest common subsequence problem has a huge potential in this area. FASTA and BLAST algorithms and its modifications are used for genetic, but are only approximate not exact. Developing an algorithm for comparing millions of DNA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

D. 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.

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.
[21] WONG, C. K., AND CHANVRA, A. K. "Bounds for the string editing problem," J. ACM 23, 1 (Jan. 1976), 13-16.



10.22214/IJRASET

45.98

IMPACT FACTOR: 7.129

IMPACT FACTOR: 7.429

INTERNATIONAL JOURNAL FOR RESEARCH

IN APPLIED SCIENCE & ENGINEERING TECHNOLOGY