`Longest Common Substring in DNA Sequence

Dipanita Saha[1], Tania Islam[2], Md. Mehedi Hasan[3]

Abstract—DNA matching is an important key to understanding genome, evolution, relationships among species, organisms, and other concepts in genomics. Much research has been doing on this problem. DNA sequence can be compared by using different known methods. These methods include dynamic programming, star alignments, tree alignments, and others which are usually based on dynamic programming. This thesis presents an algorithm that work on the DNA sequences. we design a string-matching algorithm, based on Longest Common Substring. This algorithm computes a deterministic sample of sufficiently long string in a constant time. This problem used to be a bottleneck in the pattern preprocessing for the given pattern matching. This algorithm reduces the runtime of $O(n^2)$ in Smith-Watonwan Algorithm (Dynamic Programming) to best case O(n) and worstncase r*O(n)(r<n) different from others. Our drawback is that, runtime increases when the repetition number of string increases.

Index Terms— Best case, Dynamic Algorithm, Longest common subsequence, Substring, Subsequence, Worst case, DNA.

———————————— ◆ ————————————

1 Introduction

Bioinformatics started over a century ago when Gregor Mendel, an Austrian monk cross-fertilized different colors of the same species. Bioinformatics, the discipline which studies the computational problems arising from molecular biology, poses many interesting problems to the string searching community. The LCS problem is to find the longest subsequence which is common to all sequences in a set of sequences. It is a classic computer science problem, the basis of file comparison programs, and has applications in bioinformatics. The LCS problem has an optimal substructure: the problem can be broken down into smaller, simple "sub problems", which can be broken down into simpler sub problems, and finally the solution becomes trivial. This problem also has overlapping sub problems: the solution to a higher sub problem depends on the solutions to several of the lower sub problems. Problems with these two properties—optimal substructure and overlapping sub problems—can be approached by a problem-solving technique called dynamic programming, in which the solution is built up starting with the simplest sub problems.

1.1 Problem Statement

Biological data such as DNA, proteins, genes, RNAs, *etc* are often represented as collections of sequences. Thus, many bioinformatics approaches rely on computational methods for sequence analysis.

With regard to algorithms for pattern discovery, some of the well-known ones include the PROSITE algorithm [11] and the TEIRESIAS algorithm [10]. Both are algorithms that combine PD (Pattern driven) and SD (Sequence driven) approaches. The relationship of PROSITE with the SWISS-PROT protein database allows the evaluation of the sensitivity and specificity of the PROSITE motifs and their periodic reviewing. In return, PROSITE is used to help annotate SWISS-PROT entries. However, not all patterns can be detected by the PROSITE algorithm, and the sensitivity and specificity of PROSITE patterns can be further improved.

In the TEIRESIAS algorithm [10] all elementary patterns are found in the scanning phase, and then these elementary patterns are glued with other elementary patterns at both ends. The TEIRESIAS algorithm can guarantee all the patterns that appear in at least a minimum number of sequences. The drawback of this algorithm is it does not handle flexible gaps, and only allow sole residue to occupy a single position. Recently, Ng and Shinohara [3] had proposed the minimal multiple generalization (MMG) method to find patterns in very scarce sequence samples. It requires specific initial patterns to be used.

One of the basic problems in sequence analysis is related to the extraction of the largest set of fragments that are common for a set of two or more sequences and is also known as the *Multiple* Longest Common Subsequences problem [1]. Methods that solve the MLCS problem have been successfully applied to the various areas of bioinformatics and computational genomics. However, the high complexity of macromolecular sequence data necessitates the search for new, more efficient, algorithms for solving the MLCS problem. The general problem of MLCS of an arbitrary number of sequences has been shown to be NP-hard even for a binary alphabet. Here, we introduce a general algorithm that solves LCS problem. A substring of a string is a prefix of a suffix of the string, and equivalently a suffix of a prefix. If $\bar{T}$ is a substring of T, it is also a subsequence, which is a more general concept. Given a

[1]*Institute of information Technology, Noakhali Science & Technology University, E-mail: sha.dipa.iit.nstu@gmail.com*
[2] *Department of Computer Science & Engineering, University of Barisal, E-mail: tania.bd.09@gmail.com*
[3]*Ministry of Home Affairs, mehedicse60@gmail.com*

pattern P, we can find its occurrences in a string T with a string searching algorithm. Finding the longest string which is equal to a substring of two or more strings is known as the longest common substring problem.

1.2 Objective of This Thesis

Bioinformatics was created for huge databases, such as Gene bank, EMBL and DNA Database of Japan to store and compare the DNA sequence data erupting from the human genome and other genome sequencing projects. It enables researchers to analyze the terabytes of data being produced by the Human Genome Project. Gene sequence databases and related analysis tools all help scientists to determine whether and how a molecule is directly involved in a disease process. That in turn, helps them find new and better drug targets. By the LCS and MLCS we can find the diseases pattern easily in a human body, and we can determine the percentage of diseases in his body.

2 Literature Review

2.1 Substring vs. Subsequence Study

In computer science, string is often used as a synonym for sequence, but it is important to note that substring and subsequence are not synonyms. Substrings are consecutive parts of a string, while subsequences need not be. This means that a substring of a string is always a subsequence of the string, but a subsequence of a string is not always a substring of the string. Certain known nucleotide and amino acid sequences have properties known to biologists. E.g. ATG is a string which must be present at the beginning of every DNA sequence. A target DNA sequence used to identify the location of the DNA sequence that will be used. Finding a DNA sequence contains a specific target is the main task. For this task we use the string-matching algorithm. This algorithm finds the longest common substring in DNA sequence.

2.2 PROSITE algorithm

The patterns used in PROSITE [11] have the format Y-x(1,3)-[AC], which match any sequences containing a substring starting with Y, followed by one to three arbitrary characters, followed by either A or C. However, not all patterns can be detected by the PROSITE algorithm.

2.3 TEIRESIAS algorithm

In the TEIRESIAS algorithm [10] all short patterns are found in the scanning phase, then these patterns are glued with other patterns at both ends (using depth first search) into maximal patterns. This algorithm can guarantee all the patterns that appear in at least a minimum number of sequences. The patterns used in TEIRESIAS have the format Y..A, which match any sequences containing a substring starting with Y, followed by two arbitrary characters, followed by A. The drawback of this algorithm is it does not handle flexible gaps, and only allow a single character from the alphabet set to occupy single position.

2.4 Minimal multiple generalization (MMG)

This method to find patterns in very scarce sequence samples. The patterns used in MMG [3] have the format Y*A, which match any sequences containing a substring starting. with Y, followed by any number of arbitrary characters (but usually of a limited length due to biological constraints), followed by A. This algorithm derives patterns close to known patterns, but it requires specific initial patterns to be used.

2.5 PALS Algorithm

In PALS-LCS algorithm [13]

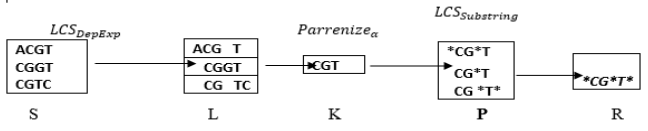


Fig. 1. Matching Process in PALS-LCS

At first take they a set of sequence. S={ACGT, CGGT,CGTC}.Then generate the Deposition and Extension algorithm for LCS(S).Here the sequence are written by matching their value. Same value is written in the same line. Then they find the longest common subsequence LCS by heuristic algorithm. K=LCS(S)= CGT. They patternize the LCS. Here they put the symbol **'*'** **,** for each sequence, where they find no match comparing with the k=LCS. At last they take the LCS substring R and append a symbol '*', at both end of the P.
The total time complexity of PALS-LCS is O($kn|\Sigma|$). The space complexity of the algorithm is O($kn|\Sigma|$).

In PALS-SCS algorithm [13]

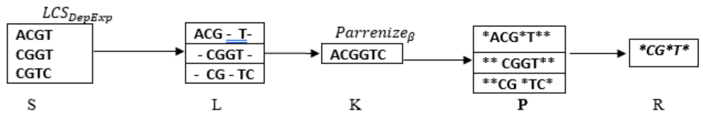


Fig. 2. Matching Process in PALS-LCS

At first, they take a set of sequence S = {ACGT, CGGT, CGTC}, they first find SCS(S) based on deposition and extension algorithm. They first generate a small set of SCS templates. Here they put the symbol '-',where they find no match sequence comparing with each of the sequences. Here k= all of the value of the of the sequences. They patternize the sequences and put the symbol '*' where they find no match comparing with each sequence. They also append '*' in the fast and last position. At, they find the common LCS from them which is R.
This analysis show that a pattern generated by MMG, which has one wildcard '*' between two alphabets, can cover about 100 sequences. And a pattern generated by PALS-LCS and

PALS-SCS, which has about five wildcards '*' between two alphabets, can also cover approximately 100 sequences. The total time complexity of PALS-SCS is $O(kn|\Sigma|+k2n)$. The space complexity of the algorithm is $O(kn|\Sigma|)$.

2.6 KMS algorithm

For Dynamic programming

s	=	AGTCGGA	(m=7)
t	=	AGCGGCTA	(n=8)

The symbol " - " indicates a missing character, or indel. Indel stands for insertion or deletion.

```
s'  =  A  G  T  C  G  G  -  A
       |  |  *  *  |  *     |
t'  =  A  G  C  G  G  C  T  A
```

In the below table we calculate the length of the alignment is *8;* the number of substitutions is 3 and the number of indels is 1, making the edit distance 4; the number of matches is 4; and the LOCKS are of lengths 2, 1, and 1

TABLE 1
Alignment Properties for DP

Length	Edit Distance	Number of Matches	Lengths of LOCKS
8	4	4	2,1,1

The KMS Algorithm [1] identifies best matches of the longest substrings of the matches of many strings

3 Methodology

3.1 Proposed Algorithm

Let, we have given two sets of string S and P. Where we denote S as source string and P is the targeted string, which would be match with S

Example S= AGCGGTACCGGGTATTTAAA
And P= AGGCTAA

3.2 Matching Process

Given a string $S=\{s_1, s_2, s_3, s_4 \dots\dots\dots.. s_n\}$ over $\sum$, where $\sum=\{A,G,C,T\}$,here S_i is an individual character in string *S* , ***where*** $1\leq i\leq n$ and pattern $P=\{p_1, p_2, p_3, p_4 \dots\dots\dots.. p_m\}$ also over $\sum$, here P_j is an individual character in the pattern, where $1\leq j\leq m$. Find the longest common substring according to the this pattern ***P*** in the given strings ***S***.

Suppose S= AGCGGTACCGGGTATTTAAA and
P= AGGCTAA

Preprocessing:

We have to store the individual positions of A, G, C, T of the given string ***S*** respectively. If $S_i=A$, then store ***ith*** position of ***S*** into an array ***A[ ][ ]*** where $1\leq i\leq n$. If $S_i=G$, then store ***ith*** position of ***S*** into an array ***G[ ][ ]*** where $1\leq i\leq n$. If $S_i=T$, then store ***ith*** position of ***S*** into an array ***T[ ][ ]*** where $1\leq i\leq n$. . If $S_i=C$, then store ***ith*** position of ***S*** into an array ***C[ ][ ]*** where $1\leq i\leq n$. Here

A[0][0]	A[0][1]
A[1][0]	A[1][1]
A[2][0]	A[2][1]
A[3][0]	A[3][1]
A[4][0]	A[4][1]

According to the occurrence of A within ***S*** string we store the position of A and the repetition number in the following way. Here we consider the string ***S***.

S = AGCGGTACCGGGTATTTAAA

Repetition	Position
1	1
1	7
1	14
	18
	19
3	20

Here position 1 contain 1 A, position 7 contain 1 A and the position 18-20 contain 3 A.

For non-repetition of character:

If $S_i \neq S_{i+1}$ where $1\leq i\leq n$ then storage *A[i][0]* =1. It will continue finding for all string ***S***.

For repetition of character:

If $S_i = S_{i+j}$,where ***j*** =1,2,3,4 …..m and $i\leq m\leq n$, then the respective array holds the consecutive position from ***i*** to ***i+j*** of source string *S. A[i][0]*=no of repetition. i.e, the value of C=(***i+j***)-***i+1*** is stored in the cell *A[i][0].*

S=AGCGGTACCGGGTATTTAAA.

Precisely we can express this string ***S*** as like.

A= [1,1],	[1,7],	[1,14],	[3,18-20]
G= [1,2],	[2,4-5],	[3,10-12]	
C= [1,3],	[2,8-9],		
T= [1,6],	[1,13],	[3,15-17]	

3.3 Findings

First of all we have $P=\{p_1, p_2, p_3, p_4 \dots\dots\dots.. p_m\}$ where $\boldsymbol{P_i}$ may be any one of {A,G,C,T} based on ***P*** .We have to search one of the four array A[ ][ ],G[ ][ ] ,T[ ][ ],C[ ][ ].

Here we have to only search one of the arrays. Here we maintain this array only for seeking starting position of searching in the main string S and skipping the repetition in source ***S.***

Suppose A[k][0] =1 and A[k][1]=5,i.e, S contains A in the 5th position. If A[*k*+*l*][0]=5 and A[*k*+*l*][1]=19,that means there are 5 consecutive A in A[*k*+*l*-1][1]=18, A[*k*+*l*-2][1]=17………. A[*k*+*l*-4][1]=15,that means 15,16,17,18,19 consecutive positions in source string S containing A. i.e., A[i][1] gives the ith position of ***S*** for ***A.*** G[i][1] gives the ith position of ***S*** for ***G.*** C[i][1] gives the ith position of ***S*** for ***C.*** T[i][1] gives the ith position of *S* for ***T.***

If A[k][1] gives us the ***ith*** position in *S*, then we know that $P_{i} = S_{i}$. here j=3,i=18. for example $P_3 = S_{18}$ that means the pattern ***P*** contain A in 3rd position and ***S*** contain A in 18th position. but we do not know $P_{i+1} = S_{i+1}$*=A or* $P_{i+1} \neq S_{i+1}$.

Now our matching procedure can be categories by the following 3 cases.

Case 1: $P_{i+l} = S_{i+l}$ *and j≤m, where j={1,2,3………m-j} and i={1,2,3………n-i} and l={0,1,2…….m-l}.*

That means the character P_i, P_{i+1}, P_{i+2}…………..P_{i+l} are matched with S_{i+1}, S_{i+2}………… S_{i+l} and we track out this position from S_i ***to*** S_{i+l}.

Case 2: $P_{i+l} \neq S_{i+l}$ *,and j≤m where j={1,2,3………m-j} and i={1,2,3………n-i} and l={0,1,2…….m-l}.*

Then stop this searching and go to the next position of A[ ][ ]or G[ ][ ] or T[ ][ ] or C[ ][ ] based on P_i. **i.e,** if ***jth*** position of ***P*** hold A, then go to the next position of A[i][1] or ***jth*** position of P hold G then go to the next position of **G**[i][1] or ***jth*** position of P hold C then go to the next position of **C**[i][1] or ***jth*** position of P hold T, go to the next position of [i][1].

Case 3: When we start from P_i,If $P_i = P_{i+l}$where P_i, contain any one of A,G,C,T ***where l*****={1,2,3……..*j-k*},**that means repetition of the same character in pattern P from the ***jth*** position to ***j+l*** position, then we have to follow the following two condition.

Condition 1: Before discuss this condition we need the following ,

When S_i to S_{i+j} (where $1 \le i \le n$, j=1,2…..*m*. Where $i \le m \le n-i$) containing the similar character (either A or G or C or T) i.e, total number of similar character is N= *(i+j)-i*+1.This similar character's positions are stored in consecutive N cells of A[i][1]or G[i][1] or T[i][1] or C[i][1] respectively.

That is A[4][1]=18, A[5][1]=19,A[6][1]=20

Again, when P_i to P_{i+j} (where 1≤ i ≤ k, j=1,2…..m. Where i ≤ m ≤ k-i) containing the similar character (either A or G or C or T) **i.e,** total number of consecutive similar character within ***P*** is **N′**= *(i+j)-i*+1. Based on the **N** and **N′** values, there are three case That is AA, GG in ***P.***

"If N > N′ then we can skip N- N′ in the source string S. After skipping matching start from i+(N- N′) th position of ***S***.And track out this matched substring of *S* from A[ ][ ] or G[ ][ ] or T[ ][ ] or C[ ][ ] and the length of this substring is N- ***N′***."

Then 3-2=1 repetition will skip for Character A and store AA.

Condition 2:

When S_i to S_{i+j} (where $1 \le i \le n$, j=1,2…..*m*. Where $i \le m \le n-i$) containing the similar character (either A or G or C or T) i.e, total number of similar character is N= *(i+j)-i*+1.This similar character's positions are stored in consecutive N cells of A[i][1]or G[i][1] or T[i][1] or C[i][1] respectively.

That is A[4][1]=18, A[5][1]=19,A[6][1]=20

Again, when P_i to P_{i+j} (where 1≤ i ≤ k, j=1,2…..m. Where i ≤ m ≤ k-i) containing the similar character (either A or G or C or T) **i.e,** total number of consecutive similar character within ***P*** is **N′**= *(i+j)-i*+1. Based on the **N** and **N′** values, there are three case That is AA, GG in ***P.***

"If N < N′ or N= N′ then there is no skipping condition and matching start from S_i. And track out this matched substring of *S* from A [ ] [ ]or G[ ][ ] or T[ ] or C[ ] [ ]and the length of this substring is **N".**

3.4 Process

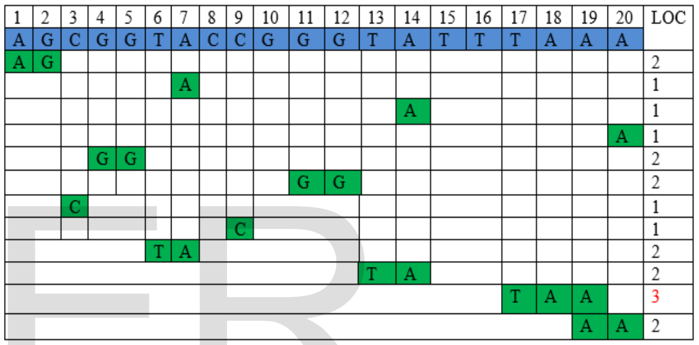
Source String S= AGCGGTACCGGGTATTTAAA

Pattern = AGGCTAA

Preprocessing:

A=	[1,1],	[1,7],	[1,14],	[3,18-20]
G=	[1,2],	[2,4-5],	[3,10-12]	
C=	[1,3],	[2,8-9],		
T=	[1,6],	[1,13],	[3,15-17]	

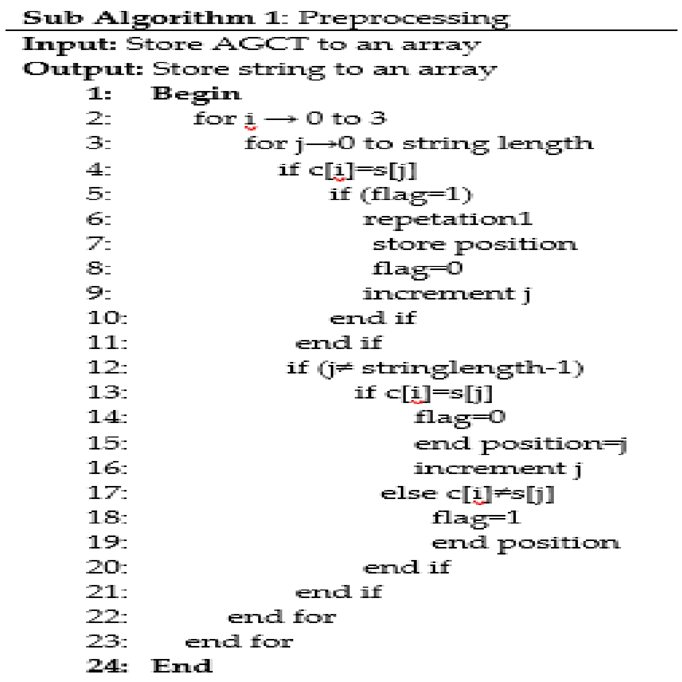
Findings:



1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	LOC
A	G	C	G	G	T	A	C	C	G	G	G	T	A	T	T	T	A	A	A	
A	G																			2
						A														1
													A							1
																			A	1
			G	G																2
										G	G									2
		C																		1
								C												1
					T	A														2
												T	A							2
																T	A	A		3
																		A	A	2

Fig. 3. Findings of our proposed algorithms

By this process we can find out longest common substring from the source string by the pattern.



```
Sub Algorithm 1: Preprocessing
Input: Store AGCT to an array
Output: Store string to an array
1:   Begin
2:      for i ⟶ 0 to 3
3:          for j⟶0 to string length
4:             if c[i]=s[j]
5:                 if (flag=1)
6:                     repetation1
7:                     store position
8:                     flag=0
9:                     increment j
10:                end if
11:            end if
12:            if (j≠ stringlength-1)
13:                 if c[i]=s[j]
14:                     flag=0
15:                     end position=j
16:                     increment j
17:                 else c[i]≠s[j]
18:                     flag=1
19:                     end position
20:                 end if
21:            end if
22:        end for
23:     end for
24:  End
```

IJSER

Sub Algorithm 2: Findings

```
1:  Begin
2:      for k→0 to t
3:              Check repetition (target.charAt(k), target, k)
4:              target A && G && T && C
5:                  p→store
6:                  po→position
7:              if (rept >=2)
8:                  for i→ 0 to store size
9:                      if (po>=rept)
10:                         skip→po-rept
11:                         p=(P-1)+skip
12:                         for j→k to t
13:                             if(k=p)
14:                                 match found
15:                                 p++
16:                 else
17:                             for i→0 to store size
18:                             po→get position
19:                                 if(K=p)
20:                                     match found
21:                                     p++
22:                                 else
23:                                     Break
24:                     end if
25:             end if
26:     end for
27:  repetion (char charAt, String target, j)
28:         count =0
29:     for k→j+1to targetlength-1
30:         if (charAt==target.charAt(k))
31:                 count++
32:         else
33:                 Break
34:         end if
35:     end for
36: End
```

3.5 Results

We take different value of input sequence & target sequence. we get two curves for two condition.

A. When input sequence variable & target value fixed (16)

TABLE 2
CPU time for different input sequence

No of experiment	X=input sequence	Y= CPU time(ms)
1	50	14
2	100	19
3	150	23
4	200	28
5	300	40
6	350	50
7	400	55
8	450	65

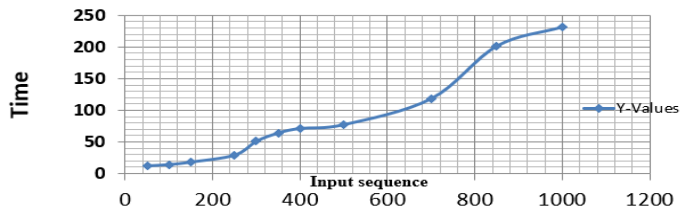


Fig. 4. Graph for input sequence vs.cpu time (java machine)

The experiment on the DNA sequences, we observe the increment of the input sequence that affect the time of execution. we perform the seven tests for various sizes of the input sequence. The curve rises up when input increases with respect to time.

B. When target variable &input sequence fixed (100)

TABLE 3
CPU time for different target sequence

No of Experiment	X=target sequence	Y=CPU time(ms)
1	10	20
2	20	40
3	30	44
4	40	47
5	50	58
6	60	62
7	70	91

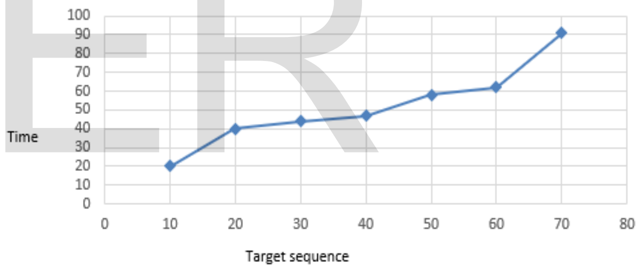


Fig. 5. Graph for target sequence vs.cpu time (java machine)

If we fixed our input size and verify our pattern, then its computation time increase with the increase of the time.

3.5 Runtime Analysis

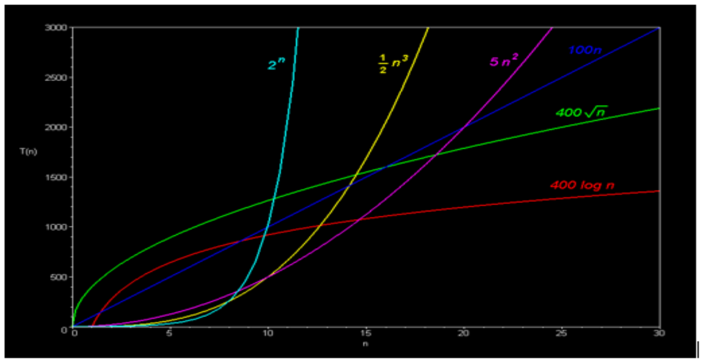


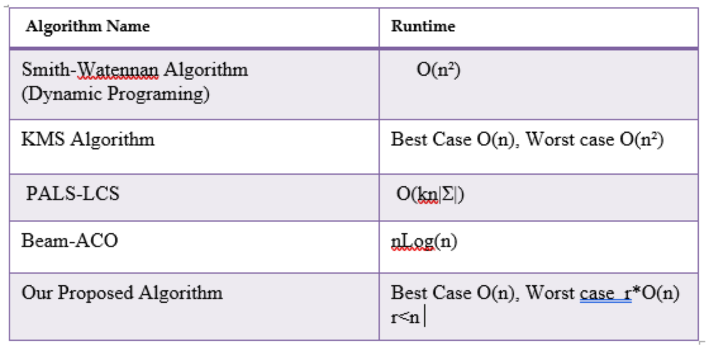
Fig. 6. Compare running times of various algorithms

This is traditional runtime for different function. We get basic idea about our runtime from this curve. We can compare our

runtime with O(n),and decide that our runtime is approximate with O(n).
Best case= O(n), when there is no repetation in source sequence.
Worst case= r* O(n), when there is repetation in source sequence. Where r is a repetation no,r<n. Runtime increases when R increases.

3.6 Comparing Runtime with existing Algorithms

TABLE 4
Comparing runtime with existing Algorithms



Algorithm Name	Runtime
Smith-Watennan Algorithm (Dynamic Programing)	$O(n^2)$
KMS Algorithm	Best Case O(n), Worst case $O(n^2)$
PALS-LCS	O(kn|Σ|)
Beam-ACO	nLog(n)
Our Proposed Algorithm	Best Case O(n), Worst case r*O(n) r<n

3.6 Limitation of our algorithm

When more repetition occurs (worst case) run time increases from O(n) to $O(n^2)$.

4.Conclusion

In this paper, we have addressed the problem of finding patterns in biological sequences. This problem is very important in bioinformatics, since the patterns in biological sequences usually indicate structure or functional relationship among sequences. The contributions of this paper include the observation that the patterns, LCS of a set of sequences are highly related and algorithms to derive patterns that is based on finding the longest common substring of the given sequences.

References

[1] M. Kaplan and J. Kaplan," Multiple DNA Sequence Approximate Matching", IEEE Computer Society, pp.79-86, 2004.
[2] L.Chen, S.Lu, and J.Ram," Compressed Pattern Matching in DNA Sequences", IEEE Computational Systems Bioinformatics Conference, pp. 1-7,2004.
[3] Y. K. Ng and T. Shinohara, "Finding Consensus Patterns in Very Scarce Biosequence Samples from Their Minimal Multiple Generalizations,"PAKDD, pp. 540-545, 2006.
[4] W.Liu, L.Chen," A Parallel Algorithm For Solving LCS Of Multiple Biosequences", IEEE,2006, pp.1-6,2005
[5] B.Ni, M.Wong, and K.Leung," N-SAMSAM : A simple and faster algorithm for solving Approximate Matching in DNA Sequences", IEEE, pp.1-7,2008.
[6] D.Korkin, Q.Wang and Y.Shang," An Efficient Parallel Algorithm for the Multiple Longest Common Subsequence Problem",IEEE,pp.1-10,2008.
[7] Q.Wang, D.Korkin, and Y.Shang," A Fast Multiple Longest Common Subsequence Algorithm" IEEE Computer Society,pp.1-17, 2011.
[8] C.Blum," Beam-ACO for the Longest Common Subsequence Problem", IEEE,pp. 1-8,2010.
[9] Y. Takekefuji, T.Td , and K Lee,"A Parallel String Search Algorithm ",IEEE Transactions an Systems Man. and Cybemetics 22. no.2, pp. 332-336, March April 1992.
[10] I. Rigoutsos and A. Floratos, "Combinatorial pattern discovery in biological sequences The TEIRESIAS algorithm," Bioinformatics, vol. 14,pp. 55-67,1998.
[11] C. J. A. Sigrist, L. Cerutti, N. Hulo, A. Gattiker, L.Falquet, M. Pagni, A.Bairoch, and P. Bucher," PROSITE: A documented database using patterns and profiles as motif descriptors," Briefings in Bioinformatics, vol. 3,pp. 265-274, 2002.
[12] N. Hulo, A. Bairoch, V. Bulliard, L. Cerutti, E. D.Castro, P. S.Langendijk -Genevaux,M. Pagni, and C. J.A. Sigrist, "The PROSITE database," Nuclei. Acids Res, vol. 34,pp.227- 230, 2006
[13] K.Ning, H.Kee Ng and H,Leong,"Finding Patterns in Biological Sequences by Longest Common Subsequences and Shortest Common Supersequences", Sixth IEEE Symposium on Bioinformatics and Bioengineering, pp.3-7,2006.
[14] Costas S. Iliopoulos and M. Sohel Rahman.'' New Efficient Algorithms for LCS an Constrained LCS Problem", Department of Computer Science, King's College London, pp.1-10,2002.
[15] Mahmoud Moh'd Mhashi," An Intelligent and Efficient Matching Algorithm to Finding a DNA Pattern", Computers and Information Technology Tabuk University, pp. 5-13,2003
[16] Raju Bhukya, DVLN Somayajulu," Exact Multiple Pattern Matching Algorithm using DNASequence and Pattern Pair", International Journal of Computer Applications Volume 17– No.8, March 2011