I.J. Information Technology and Computer Science, **2013, 10, 39-61**
Published Online September 2013 in MECS (http://www.mecs-press.org/)
DOI: 10.5815/ijitcs.2013.10.05

A Compression & Encryption Algorithm on DNA Sequences Using Dynamic Look up Table and Modified Huffman Techniques

Syed Mahamud Hossein
Regional Office, Kolaghat; Directorate of Vocational Education & Training, West Bengal, India
E-mail: mahamud123@gmail.com

S.Roy
HIT, Haldia, India

Abstract— Storing, transmitting and security of DNA sequences are well known research challenge. The problem has got magnified with increasing discovery and availability of DNA sequences. We have represent DNA sequence compression algorithm based on Dynamic Look Up Table (DLUT) and modified Huffman technique. DLUT consists of 4^3(64) bases that are 64 sub-stings, each sub-string is of 3 bases long. Each sub-string are individually coded by single ASCII code from 33(!) to 96(`) and vice versa. Encode depends on encryption key choose by user from four base pair {a,t.g and c }and decode also require decryption key provide by the encoded user. Decoding must require authenticate input for encode the data. The sub-strings are combined into a Dynamic Look up Table based pre-coding routine. This algorithm is tested on reverse; complement & reverse complement the DNA sequences and also test on artificial DNA sequences of equivalent length. Speed of encryption and security levels are two important measurements for evaluating any encryption system. Due to proliferate of ubiquitous computing system, where digital contents are accessible through resource constraint biological database security concern is very important issue. A lot of research has been made to find an encryption system which can be run effectively in those biological databases. Information security is the most challenging question to protect the data from unauthorized user. The proposed method may protect the data from hackers. It can provide the three tier security, in tier one is ASCII code, in tier two is nucleotide (a,t,g and c) choice by user and tier three is change of label or change of node position in Huffman Tree. Compression of the genome sequences will help to increase the efficiency of their use. The greatest advantage of this algorithm is fast execution, small memory occupation and easy implementation. Since the program to implement the technique have been written originally in the C language, (Windows XP platform, and TC compiler) it is possible to run in other microcomputers with small changes (depending on platform and Compiler used). The execution is quite fast, all the operations are carried out in fraction of seconds, depending on the required task and on the sequence length. The technique can approach an effective compression ratio of 1.98 bits/base and even lower. When a user searches for any sequence for an organism, an encrypted compressed sequence file can be sent from the data source to the user. The encrypted compressed file then can be decrypted & decompressed at the client end resulting in reduced transmission time over the Internet. An encrypt compression algorithm that provides a moderately high compression with encryption rate with minimal decryption with decompression time.

Index Terms— Compression, Security,

Abbreviation— DLUT-Dynamic Look up Table

I. Introduction

With more and more complete genomes of prokaryotes and eukaryotes becoming available and the completion of Human Genome Project on the horizon, fundamental questions regarding the characteristics of these sequences arise. Life represents order. It is not chaotic or random [1]. Thus, we expect the DNA sequences that encode life to be nonrandom. In other words, they should be very compressible. There is also strong biological evidence that supports this claim, it is well known that DNA sequences only consist of four nucleotide bases {*a*, t,*g*,c},(note that T is replaced with U in the case of the RNA), and one byte are enough to store each base. All this evidence gives more concrete support that the DNA sequences should be reasonably compressible. It is well recognized that the compression of DNA sequences is a very difficult task [2-6]. However, if one applies standard compression tools such as the Unix "compress" and "compact" or the MS-DOS archive programs "pkzip" and "arj", they all expand the file. These tools are designed for text

compression [2], while the regularities in DNA sequences are much subtler. It means that DNA sequences do not have the same properties for the traditional compression algorithms to be counted on. This requires a better model for computing the DNA content such that better data compression results can be achieved. In fact, it is our purpose to reveal such subtleties, such as dynamic Look Up Table of 3 letter 64 sub-string, match with source DNA sequences by using a more appropriate compression algorithm. In this article, we will present a DNA compression algorithm, DLUT, based on exact matching between Look Up Table and source file and that gives the best compression results on standard benchmark DNA sequences. We will present the design rationale of dynamic LUT based on exact matching, discuss details of the algorithm, provide experimental results and compare the results with the one most effective compression algorithm for DNA sequence (gzip-9).

We can find the compression rate and compression rate over reverse, complement and reverse complement of DNA sequences result of same cellular DNA sequences. Also we can find the compression rate, compression ratio of artificially sequence generated by randomly of equivalent length of cellular DNA sequence. Compare all result to each other. For that purpose we can generate two different algorithms.

We devised a new DNA sequence compression algorithm based on dynamic Look Up Table pre-coding routine which maps the 3 letter 64 sub-sting into 64 ASCII characters start from 33(!) to 96 (`) and vice versa. Since the essence of compression is a mapping between source file and destination file, the compression algorithm dedicates to find the relationship. We migrate this idea to our research on DNA sequence compression. We are trying to build a finite DLUT which implements the mapping relationship of our coding process. Some experiments indicate that the compression ratio is 3.1 bits/base.

Huffman CODING: Statistical codes represent data blocks of fixed length with variable-length code[7] words. Huffman coding is one type of statistical code[8-9]. This coding is also one type of entropy coding. Entropy encoding is a lossless data compression scheme that is independent of the media's specific characteristics. Entropy coding[10-11] assigns codes to symbols so as to match code lengths with the probabilities of the symbols. Typically, these entropy encoders are used to compress data by replacing symbols represented by equal-length codes with symbols represented by codes where the length of each codeword is proportional to the negative logarithm (is $-\log_b P$, where b is the number of symbols used to make output codes and P is the probability of the input symbol) of the probability. Therefore, the most common symbols use the shortest codes.

The efficiency of a Huffman[12-13] code depends on the frequency of occurrence of all distinct fixed length blocks in a set of data. The most frequently occurring blocks are encoded with short code words, whereas the less frequently occurring ones are encoded with large code words. In this way, the average codeword length is minimized. It is obvious however that, if all distinct blocks in a data set appear with the same (or nearly the same) frequency, then no compression can be achieved. Among all statistical codes, Huffman offer the best compression since they provably provide the shortest average codeword length. Another advantageous property of a Huffman code is that it is prefix free; i.e., no codeword is the prefix of another one. This makes the decoding process simple and easy to implement.

Let T be the fully specified test set. Let us also assume that if we partition the test vectors of T into blocks of length l, we get k distinct blocks $b1, b2, \ldots, bk$ with frequencies (probabilities) of occurrence $p1, p2, \ldots, pk$, respectively. The entropy of the test set is defined as

$$H(T) = -\sum_{i=1}^{k} P_i(\log_2 P_i) \quad (1)$$

and corresponds to the minimum average number of bits required for each codeword. The average codeword length of a Huffman code is closer to the aforementioned theoretical entropy bound compared to any other statistical code. In practice, test sets have many don't care (x) bits. In a good encoding strategy, don't cares must be assigned such that the entropy value $H(T)$ is minimized. In other words, the assignment of the test set's x values should skew the occurrence frequencies of the distinct blocks as much as possible. We note that the inherent correlation of the test cubes of T (test vectors with x values) favors the targeted occurrence frequency skewing and, consequently, the use of statistical coding. To generate a Huffman code, we create a binary tree. A leaf node is generated for each distinct block bi, and a weight equal to the occurrence probability of block bi is associated with the corresponding node. The pair of nodes with the smallest weights is selected first, and a parent node is generated with weight equal to the sum of the weights of these two nodes. The previous step is repeated iteratively, selecting each time the node pair with the smallest sum of weights, until only a single node is left unselected, i.e., the root (we note that each node can be chosen only once). Starting from the root, we visit all the nodes once, and we assign to each left-child edge the logic 0 values and to each right-child edge the logic 1 value. The codeword of block bi is the sequence of the logic values of the edges belonging to the path from the root to the leaf node corresponding to bi. If $c1, c2, \ldots, ck$ are the codeword lengths of blocks $b1, b2, \ldots, bk$, respectively, then the average codeword length is

Copyright © 2013 MECS

$$C(T) = \sum_{i=1}^{k} P_i C_i \qquad (2)$$

Data Compression and ENCRYPTION: There is a similarity between the process of data compression and process of encryption. The goal for both the processes is to reduce the redundancy in the source message. According to Shannon [14], for perfect lossless compression algorithm, the average bit rate is equal to the source entropy. The redundancy of a source message is the difference between average bit rate and the source entropy of the message. The purpose of reducing the redundancy in case of compression algorithm is to conserver the storage space or communication bandwidth. The goal of reduction of redundancy in case encryption process is to thwart the different cryptanalysis attack based on statistical property of the source message. If we combine both the process of compression and encryption as shown in the fig below, then we can utilize another property of the compression algorithm that the decompression information are concentrated in a few portion of the bit stream, to selectively encrypt those portion of the bit stream which has got more impact on the reconstruction of the message during decompression process, keeping the remaining uncompressed bit stream in the clear.

For a perfect compression scheme, the plain text of the unencrypted portion of the message is statistically independent of the encrypted plain text message. So by knowing the unencrypted plain text, cryptanalyst cannot infer anything for the encrypted plain text.

Due to the combination of the process of compression and the process of encryption, two benefits are realized:

1. Conservation of storage space and communication bandwidth

2. Encryption cost is reduced.

3. The attacks on the basis of statistical property of the source bit stream are thwarted.

Client side decompression: We use compression & selection encryption techniques for the general purpose of sequence data delivery to the client. Existing DNA search engines do not utilize DNA sequence compression algorithms & encryption for high security for client side decryption & decompression, i.e. where encrypted compressed DNA sequence is decrypted & decompressed at the client end for the benefit of faster transmission & information security. Because most of the existing DNA sequence compression algorithms aim for higher compression ratios or pattern revealing, rather than client side & decryption decompression, their decompression times are longer than necessary information security. This makes these compression techniques unsuitable for the "on the fly" decompression. We use encrypted compression technique designed for client side decrypted followed by decompression in order to achieve faster sequence secure data transmission to the client.

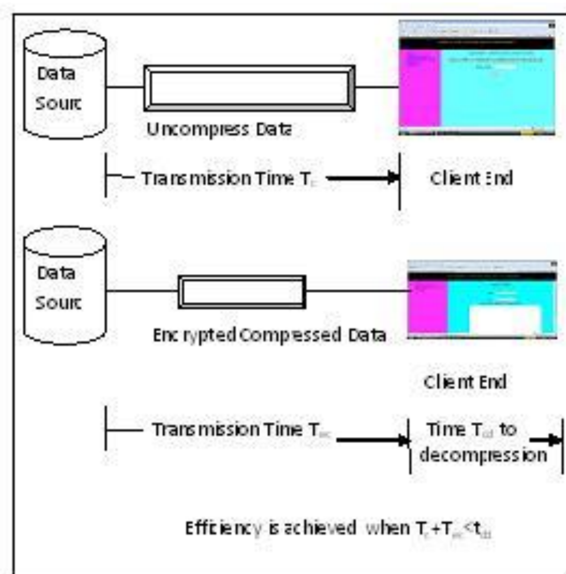

Fig. 1: Sender & receiver site encryption & Decryption process

If encrypted compressed sequence data is sent from the data source to be decrypted decompressed at the client end and the decryption to decompression time along with the encrypted compressed file transmission time is less than the transmission time for uncompressed data transfer from the source to the client, then efficiency is achieved. Fig. 1 illustrates the situation. Note that the sequence data should be kept pre-compressed within the data source.

A Sequence compression algorithm with reduced decompression time and moderately high compression rate is the preferred choice for efficient sequence data delivery with faster data transmission. As our target is to minimize decompression time and high information security, we use similar compression techniques to those used in [15], based on a "Two phase" approach, meaning, that the file is compressed followed by encryption or decrypt followed by decompressed while reading it. Unlike "four phase" algorithms there is no need to re-read the input file. Our compression technique is essentially a symbol substitution compression scheme that encodes the sequence by replacing four consecutive nucleotide sequences with ASCI characters. Our technique is to find the best solution for a client side decompression.

Work already carried out **:** So many biological compression algorithms is available in market as in paper[100] where showing that Huffman's code also fails badly on DNA sequences both in the static and adaptive model, because there are only four kind symbols in DNA sequences and the probabilities of

Copyright © 2013 MECS

occurrence of the symbols are not very different. Here this two phase technique solved this problem because after the 1st phase compression we get the 252 ASCII characters' with nucleotide base pair a, t,g & c.

II. Flowchart:

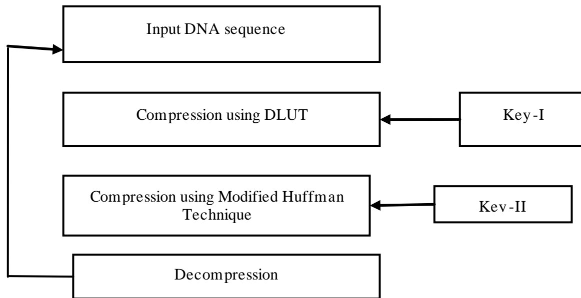

Fig. 2: Process Diagram

III. Methods

3.1 File Format:

We will begin discussing file type is text file (file extension is dot txt) contain a series of successive four base pair (a,t,g and c) and end with blank space ahead the end of file. Text file is the basic element to which we consider in compression and decompression. The output file also text file, contains the information of both unmatched four base pair and a coded value of ASCII character.

The coded valued are located in the encoded section. The coded information is written into destination file byte by byte. The file size depends on number of base pair present in the input file and output file measured by byte, i.e. File size (in byte)= number of base pair in a file(in byte).As per example total no. of base pair in a file is n, so the file size is n byte. ASCII character also required one byte for storing. On the basis of ASCII code availability, we can take input as a lower case letter of a,t,g and c, if upper case input are taken, algorithm convert into upper case letter to lower case letter.

3.2 The Four Bases Pair a,t,g and c

The four bases pair a,t,g and c possible orientation 4^3=64. All sub-string has 3 bases long. The look-up table describes a mapping relationship between DNA segment and its corresponding characters. We assume that every three characters in source RNA sequence without N^2 will be mapped into a character chosen from the character set which consists of 64 ASCII characters.

Consider a string S= aaaaaagaacatgatcttccc ………n where n is the length of string.

So, n=Length of the string = Total no. of base pair in S = File size in byte (n>0)

Due to DLUT facility we can take input from four base pair {a,t,g & c}in 24(Factorial of 4=24) orientation and all times starting and ending ASCII code range from 33(!) to 96(`) and vice versa. That means DLUT variation are 24X2 i,e 48 no different table. We can create 48 different LUT in this way. But all times sub-string in a table is 64 (4^3=64), 64 ASCII code are sufficient to define the one particular LUT in each encoding and decoding time.

^{2}N Refers to those not available base in DNA sequence

3.3 Encoding steps:

First input file encoded by Look Up Table-I, encoded data store on another output file.

Look up Table structure is

Table 1: Lookup Table

LUT INPUT		Structure LUT		Range of ASCII code A[j]	
Sr. No.	**Input character**	**Sr. No.**	**Sub-string for LUT S[i]**	**Start from(Increasing)**	**Start from(Decreasing)**
1	atgc	1	aaa	33/96	96/33
		3	aag		.
		4	aac		
		7	atg		
		8	atc		
		24	ttc		
		33	gaa		
		64	ccc	96/33	33/96

We match the input string with pre coded LUT. For that purpose we have break the string S into (n-2) sub string, each sub-string has length 3 bases long. n-2 because each sub-string length is 3.

S[i]= 1 2 3 4 5 6 7 8 9 10 11 12 13 14 ……n [1≤i≤n]

As for example: S = a a a a a a g a a c a t g ……..n

n = 1 2 3 4 5 6 7 8 9 10 11 12 13 14……n

Where S[i] array store each sub-string of S and i=1 to n-1.

S[1]=aaa,S[2]=aaa,S[3]=aaa,……………………….. S[n-2]ccc

Table 2: Sub Group press

Input string(S)	LUT starting charecter	Sub-string A[i]	Match condition S[i]=A[j]	ASCII code A[j]	Output
aaaaaagaacatgatcttccc	atgc	aat	S[1]=A[1] aat=	-	a
		att	S[2]=A[2]		

First match S[1] with A[1] to A[64] , if match occur place ASCII character in 1st position otherwise left one character in left hand side in S, place 1st character in output file 1st position. This left single character are place output file in 1st position. Progress one character toward in right direction, take next sub-string and match S[2] with A[1] to A[64], if match occur place appropriate ASCII character in 2nd place. This ASCII character is put into the output file in 2nd position. This process will continue until and unless n-2 position reaches.

The Encoding procedure mentioned this rule and produce compression output file.

Match found then S[i]=A[j] ,place ASCII character in the output file ith position. Each matching cases the value of i is incremented by i=no. of unmatched character+ (no. of sub-string match X 3+1)

Otherwise S[i]≠A[j] place base pair in output file ith position. If unmatched occur, the value of i is incremented by one and j is increased by one.

At the end, we can get the compress output file O

So, O= !!Ac'(8cc………….n_1 where n_1 is the length of output file. Output file size is n_1 byte

3.4 Decoding

Decoding time, first create LUT(fix base pair which are use in encoding time for single case , its depends on user authenticate value and decryption key and authenticate ASCII code starting position otherwise proper decoding are not possible.

On this basis of input character set and ASCII code starting value, the actual LUT are crated. On this particular value, the encoded output string is decoded and produces the output original file.

Look Up

Table 3: ASCII staring ending point

INPUT	**LUT**		**ASCII**
Authenticate Input	**Sr. No.**	**Sub-string for LUT S[i]**	**Start from(Increasing/decreasing) A[j]**
atgc	1	aaa	33
	.	.	
	3	aag	
	4	aac	
	7	atg	
	8	atc	
	24	ttc	
	33	gaa	
	64	ccc	

O=!!Ac'(8cc ……………………n1 where n1 is the length of output string (n>n1).

At the time of decoding each character match with ASCII code that is A[j] with O[i] one by one. If match occur in between A[1] to A[64] with O[1], place ASCII equivalent sub-string in 1st places in output file. The value of i is incremented by one. If unmatched found in between A[1] to A[64] with O[2], place base pair in 2nd position in output file. The value of i is incremented by one. This process will continue until i=n1 position will appear.

Copyright © 2013 MECS

Table 4: ASCII code decompression

Input string(O)	LUT starting character	Match condition A[j]=S[i]	ASCII code A[j]	Output
!!Ac'(8cc	atgc	O[1]=A[1] aat=	-	a
		O[2]=A[2]		

The Decoding process mentioned this rule and produce original output string.

Match found if O[i]=A[j] place ASCII character equivalent sub-string in ith position. If match found, the value of i is incremented by one.

Otherwise S[i]≠A[j] place base pair in ^{ith} position in output file. If unmatched occur , the value of i is incremented by one.

For easy implementation, characters a,u,g,c will no longer appear in pre-coded file and A,T,G,C will appear in pre-coded file. For instance, if a segment "aaaaaagaacatgatcttccc ……..n " has been read, in the destination file, we represent them as "aDNc9txa…….n_1".Obviously, the destination file is case-sensitive

But at the end of file two base segments are remaining (ideal case where total number of base pair are not divisible by 3), We cannot find any arrangement in ASCII table. In these circumstances, we just write the original segment into destination file.

We know that each character require 1 byte (8 bit) for storing. Using lookup table compaction ration per word is 3:1 when match is found.

In case of above example string length = 21 that means 21 byte require for storing this string. After encoding on the basis of lookup table of 3 sub string length sizes , reduce string length 9, require 9 byte for storing this string.

3.5 Huffman Encoding:

Consider the test set shown in column 1 of Table I, which consists of four test vectors of length 16 (64 bits overall). If we partition the test vectors into blocks of length 4, we get 16 occurrences of five distinct blocks. Columns 2 and 3 present those five blocks and their frequencies of occurrence, respectively. Column 4 presents the codeword of Table-5

Table 5.1: Encoded data set

Encoded data set			
10	0	110	0
110	10	0	1110
0	10	1111	0
10	0	10	0

Table 5: Huffman encoding example

Test set	Block	Frequency	Code word
0000 1010 1111 1010 1111 0000 1010 0001 1010 0000 0010 1010 0000 1010 0000 1010	1010	7/16	0
	0000	5/16	10
	1111	2/16	110
	0001	1/16	1110
	0010	1/16	1111

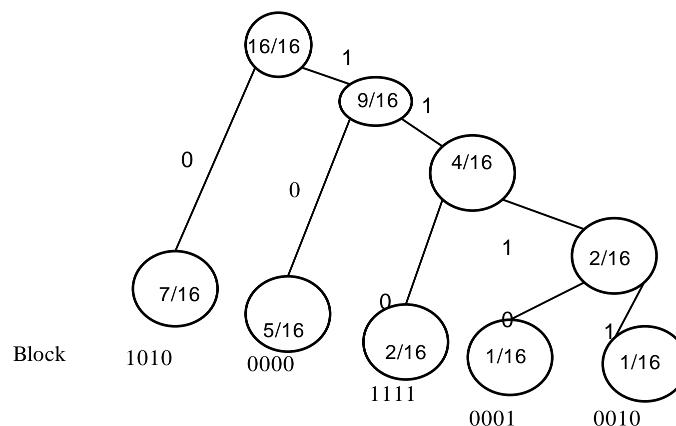

Fig. 3: Left & right node

Each block the corresponding Huffman tree and the encoded data set are shown in Fig.2.3. The size of the encoded data set is 31 bits, and the average codeword length is

C(T)=1*(7/16)+2*(5/16)+3*(2/16)+4*(1/16)+4*(1/16) = 1.9375

Note that the entropy is equal to

$H(T) = - [(7/16) * \log_2(7/16) + (5/16) * \log_2(5/16) + (2/16) * \log_2(2/16) + (1/16) * \log_2(1/16) + (1/16) * \log_2(1/16)] = 1.9218$

The size of a Huffman decoder depends on the number of distinct blocks that are encoded. Increased encoded-block volumes lead to big decoders due to the big size of the corresponding Huffman tree. For that reason, a selective Huffman approach was adopted in our project, according to which only the most frequently occurring blocks are encoded, whereas the rest are not.

3.6 Proposed Methods

3.6.1 Methodology of Experiments PERFORMED:

We have conducted our experiments in normal text files of different sizes and on the basis of the statistical property generated by Huffman tree for each text files. Since our objective is to find out the selective portion i.e. R part (discussed previously) from the text message we made swapping of the branches in the Huffman tree on at a particular level on the basic of a key and decode the encoded symbols using the modified Huffman tree which are specified in scheme I and II. In scheme-I we apply swapping method on two nodes at specified level on Huffman tree, and in scheme-II we perform swapping method between two specified nodes at different level on Huffman tree. When we generate the Huffman tree using the statistical property of symbols, first we consider each character of input text message as a symbol and later each word as a symbol.

Illustrating Huffman Code With Example: We take a simple text containing alphabets A, B, C, D, E, F and their frequencies 8, 3, 2, 4, 2, 1 respectively. The Huffman tree in this case would be as in figure below (fig. 4.1).

Table 6: Coding scheme

Char	Codeword
A	0
C	100
B	101
F	1100
E	1101
D	111

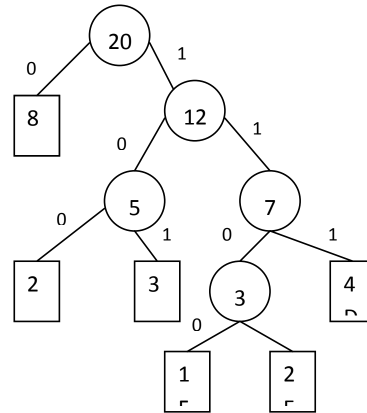

Fig. 4: Binary coding scheme

Example: We take a simple test their frequencies 8,3,2,4,2,1 would be as in figure above

Suppose we take a string “AABBC”, for example, would be encoded as “00101101100”. And when we decompress it according to 0 and 1 , we traverse the tree from root to left or right until we get child node. E.g. the bit stream “00101101100”. The first two“0” will be decode traversing from root 20→8(A) [left traversing since 0] and next “1”, then traversing from root to left 20→12, next “0”, then traversing from 12 to left 12→5, next “1”, traversing from 5 to right 5→3(B) and get leaf node B. Similarly we get original text “AABBC”. Thus the original text is decompressed from compressed text.

3.6.2 Swapping Nodes at Specified Level (Scheme-I)

Our objective is to find out the selective portion i.e. R part (discussed previously) from the text message. We made swapping of the branches in the Huffman tree on at a particular level on the basis of a key and decode the encoded symbols using the modified Huffman tree. Now for selecting the R part, my first experiment is swapping two nodes at specified levels. Here I exchange by left most nodes with right most nodes at specified level. So only those nodes, which are changed their positions after swapping, are affects and also corresponding codes are also altered. Remaining other nodes is kept unchanged. Hence selective bits are altered.

Let illustrate this with an example, Fig.4.2 is the original tree and Huffman codes of W=00, X=01, Y=10, Z=11.

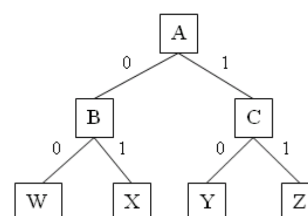

Fig. 5: Code word

Suppose we apply swapping at level 1 then we find out A is single node i.e. root node at level 0 i.e. root node at level 0. Then we interchange the position of left and right child node with their sub tree as shown in fig.4.3. So corresponding code of W, X, Y, Z are totally changed. W=10, X=11, Y=00, Z=01. So if the original text is "WWXYXZ" then it will be encrypted "101011001101". If we decode it without change the level the text will be "YYZWZX". D_{SID} in this case is 6.

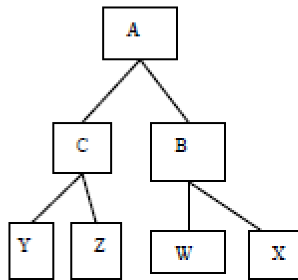

Fig. 6: Tree

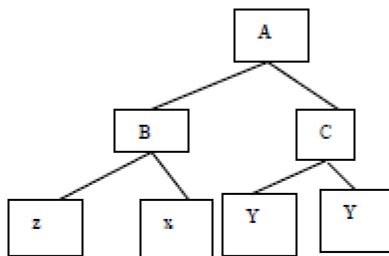

Fig. 7: Tree

Table-7: Binary code

Character	Before Encrypt	After Encrypt	
		Swapping at Level 1	Swapping at Level 2
W	00	10	11
X	01	11	01
Y	10	00	10
Z	11	01	00

Now we apply swapping at level 2(here B, C) then we interchange the position of left child of B and right child of C and with their sub tree as shown in fig.4.4. In this case corresponding codes are selectively changed. i.e. since in fig.4.4 we see on W and Z are interchange their positions so code of Z and W are only changed, other should be unchanged according to table 4.1. So if the original text is "WWXYXZ" then it will be encrypted "111101100100". If we decode it without change the level the text will be "ZZXYXW". D_{SID} in this case is 3.

The results diverged from our expectations in some simple cases due to the complexities in the alignments of characters when calculating D_{SID}. In our experiment, when we measure Lavenstein distance and effectness on actual text we face some minor problems.

Suppose we compress a text file-applying node swapping method at particular level. But if we measure effectness on actual text by decoding the encoded text without any swapping method apply, then in some cases all codes may not be retrieved, for e.g. suppose frequency of A=2, B=1 and C=1 then tree will be generate like

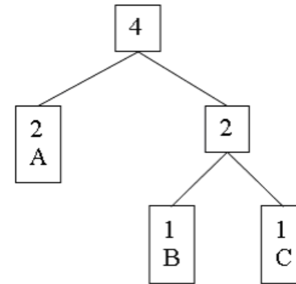

Fig. 8: Tree structure

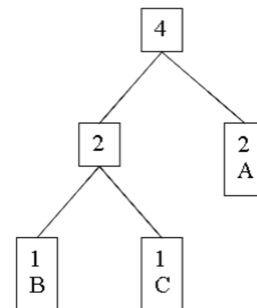

Fig. 9: Tree structure

And their corresponding codeword A=0, B=10, C=11.

And suppose the string 'AABBC' would be encoded as 00101011.

Now if we apply swapping method at level 0 then the tree will generate like fig. 4.6

And their corresponding code A=1, B=00, C=01.

And same string then encoded as 11000001.

Now if we measure Lavenstein distance & % of effectness on actual text then we must decode without apply swapping method at level 0. Then that encoded string is decoded using original tree (fig.4.5).

Table 8: Code Table

1	1	0	0	0	0	0	1
C	C	A	A	A	A	A	-

The last 1 is left over because there is no such code of only 1. So for measurement purpose the size of original text is altered in these cases.

3.6.3 Swapping between Two Specified Nodes at Different Level (Scheme-II)

In my second experiment we get another approach for selecting the r part (discussed previously). In this approach swapping can be perform at any specified two nodes. By this approach we can interchange any two nodes with its sub tree of the Huffman tree at any level. Hence this scheme has flexibility to modify Huffman tree and also use more than one key so it obviously increase security concern. In this scheme we need to specify two level values of two nodes and two binary values. Number of binary digit must be same with level value with respect to nodes. If we consider above specified two values as a key then security concern is improved than before experiment. E.g. Fig.4.7 is the original tree and Huffman codes of W=00, X=01, Y=10, Z=11.

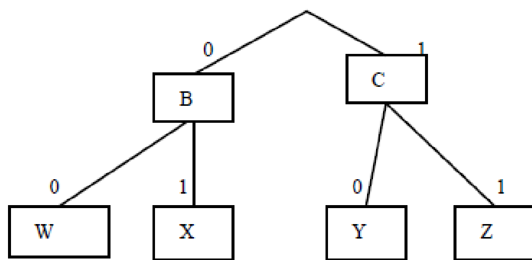

Fig. 10: Binary Tree

Suppose I want to swap between two nodes B and Z then we need to specify first, level number, in this case 1 and binary digit 0 for B node and level number for Z, in this case 2 and binary digit 11. Then new tree will generate like

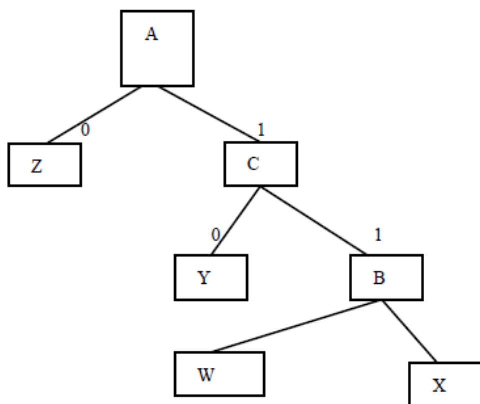

Fig. 11: Binary Tree

After interchanging the position of left and right child node with their sub tree as shown in fig.4.8 so corresponding codes of W, X, Y, Z are totally changed according to table 4.2. So if the original text is “WWXYXZ” then it will be encrypted “110110111101110”. If we decode it without change the level the text will be decrypt like “ZXYZZXZ”. D_{SID} in this case is 6.

Table 9: Character corresponding code word

Character	Before Encrypt	After Encrypt	
		Swapping between B and Z	Swapping between C and X
W	00	110	00
X	01	111	1
Y	10	10	010
Z	11	0	011

3.7 Random String Generation Method:

We have generate a string of four symbols (a,t,g and c) of any arbitrary length, it is user requirement.

This method simply uses random function in C++ language.

3.8 Reverse, Complement and Reverse Complement Method

Suppose the original string is

S = aaaaaagaacatgatcttccc ………n

Reverse string is R=n......……..cccttctagtacaagaaaaa

Complement string is =tttttcttgtctagaggg…………n

Reverse Complement is RC=n….gggaagatgttgttttt

Where n is the length of the string.

3.9 We Have Develop Four Algorithms

First: encoding (compression) algorithm, Second: decoding(decompression) algorithm, Third: Random string generator algorithm and Fourth is Reverse, complement and Reverse complement algorithms.

IV. Following Algorithm Are:

4.1 Algorithm for Random String Generator

Procedure Generate

```
do
        Integer i,j;
        character A[]="atcg";
        Input("%ld",&j);
        FILE * fp;
        fp=Open_file("input.txt","w");
        for(i=0;i<j;i++)
        do
        Write_charcter(A[generate_random_num(4)],
fp);
        end
        fclose(fp);
end
```

4.2 Reverse, Complement and Reverse Complement the String Algorithm:

Procedure comp(Character x,Integer p)

```
do
        switch(x)
        do
                case 'A':
                                if(p=0)
                                        return 'T';
                                else
                                        return 'U';
                case 'T':
                        return 'A';
                case 'U':
                        return 'A';
                case 'C':
                        return 'G';
                case 'G':
                        return 'C';
        end
        return x;
end
Procedure REVERSE_FILE
do
        clrscr();
        FILE* fin,*fout[3];
        Character * in,*out;
        Character c1,c2,c3,c4;
        Character p;
    Integer count:=1,point:=1;
        Display("\n Enter Source File:");
        Input("%s",in);
        Display("\n The generated files are
        rev.txt,r_comp.txt,comp.txt");
        Display("\n\n File type ? 0 -DNA / 1 -RNA :");
        Input("%d",&p);
        system("mkdir out");
        fin:=Open_file(in,"r");
        if(fin=NULL)
        do
                Display("\n Could not open Source
File.");
                wait_for_keypress();
                exit(1);
        end
        fout[0]:=Open_file(".//out//rev.txt","w");
        if(fout=NULL)
        do
                Display("\n Could not open reverse
File.");
                wait_for_keypress();
                exit(1);
        end
        fout[1]:=Open_file(".//out//r_comp.txt","w");
        if(fout=NULL)
        do
      Display("\n Could not open reverse
complement File.");
                wait_for_keypress();
                exit(1);
        end
        fout[2]:=Open_file(".//out//comp.txt","w");
        if(fout=NULL)
        do
        Display("\n Could not open complement
File.");
                wait_for_keypress();
                exit(1);
        end
        c1:=fgetc(fin);
        while(c1<>EOF)
        do
         Put_Char_in_File(comp(c1,p),fout[2]);
         c1:=fgetc(fin);
         count++;
        end
        point:=count-1;
        dodo
                point--;
                fseek(fin,point,SEEK_SET);
                c1:=fgetc(fin);
                (c1>90)?c1-:=32:c1-:=0;
                Put_Char_in_File(c1,fout[0]);

        Put_Char_in_File(comp(c1,p),fout[1]);
        endwhile(point);
        close_file(fin);
        close_file(fout[0]);
        close_file(fout[1]);
        close_file(fout[2]);
end
```

4.3 Algorithm for DLUT

4.3.1 DLUT Encoding algorithm

Integer : eger : parse(Character : *tx);

```
Procedure check (Character : ch)
        do
        if (ch < 97 AND ch > 64)
                return ch+32;
        return ch;
        end;
Character : cod[4][4][4];
Character : LUT[4],vbx[20];
Real : ibytes,obytes;
Procedure ret_pos(Character : x)
        do
        Integer : eger : i;
        for(i:=0;i<4;i++)
        if(LUT[i]=x)
                break;
        return i;
        end;
Procedure isin(Character ch)
do
        if (ch>= 33 && ch<=96)
                return 1;
        else
```

```
                    return 0;
    end;
    Procedure parse(Character : *tx)
    do
            Integer : eger : p,q,r;
            p:=ret_pos(tx[0]);
            q:=ret_pos(tx[1]);
            r:=ret_pos(tx[2]);
            return cod[p][q][r];
    end;
    Procedure Compress_Input()
    do
    struct time st,sp;
    Real : cr1,cr2,crate;
    FILE *fp,*ip,*op;
    Character : BUT[3];
    Integer : i,p,q,r,order,rorder;
    Character : ch;
    Integer : match,t1,t2;
    Display " Input 3 LUT initializing Character : acters:
(a,t,c,g)";
    Display " 1:";
    LUT[0]:=get_character();
    Display " 2:";
    LUT[1]:=get_character();
    Display " 3:";
    LUT[2]:=get_character();
    if(LUT[0]<>'a' AND LUT[1]<>'a' AND
LUT[2]<>'a')
            LUT[3]:='a';
    if(LUT[0]<>'t' AND LUT[1]<>'t' AND
LUT[2]<>'t')
            LUT[3]:='t';
    if(LUT[0]<>'g' AND LUT[1]<>'g' AND
LUT[2]<>'g')
            LUT[3]:='g';
    if(LUT[0]<>'c' AND LUT[1]<>'c' AND
LUT[2]<>'c')
            LUT[3]:='c';
    Display " 4:"<<LUT[3];
    do
    Display " LUT ordering (33/96)";
    Get_User_Input order;
    while(order<>33 AND order <>96);
    if(order=33)
            rorder:=96;
    else
            rorder:=33;
    for(i:=0;i<64;i++)
    do
            p:=i/16;
            q:=(i-p*16)/4;
            r:=i-p*16-q*4;
            cod[p][q][r%4]:=(order=33)?order+i:order-
i;
    end;
    Display "LUT generated:";
    fp:=fopen("LUT.txt","w");
    for(i:=0;i<64;i++)
    do
            p:=i/16;
            q:=(i-p*16)/4;
            r:=i-p*16-q*4;
            Write_to_FIle :
(fp," %d  %c%c%c  : %c",i+1,LUT[p],LUT[q],LUT[r]
,cod[p][q][r]);
    end;
    Close_File (fp);
    Display " Enter input file:";
    Get_User_Input vbx;
    ip:=fopen(vbx,"r");
    if(ip=NULL)
    do
            Display " Unable to open input file.";

            exit(1);
    end;
    Display " Enter name for output file:";
    Get_User_Input vbx;
    op:=fopen(vbx,"w");
    if(op=NULL)
    do
            Display " Unable to open output file.";

            exit(1);
    end;
    ibytes:=obytes:=match:=0;
    gettime(&st);
    gettime(&sp);
    gettime(&st);
    Display (" Start time
is : %d:%d:%d.%d",st.ti_hour,st.ti_min,st.ti_sec,st.ti_h
und);
    do
    ch:=fgetc(ip);
    ch:=check(ch);
    if(ch=EOF)
            break;
    if(ret_pos(ch)>2)
            do
            Put_Char_to_File(ch,op);

            end;
    else
            do
            BUT[0]:=ch;
            if(ch=EOF)
                    break;
            BUT[1]:=fgetc(ip);
            BUT[1]:=check(BUT[1]);
            if(BUT[1]=EOF)
                    do
                    Put_Char_to_File(ch,op);
                            break;
                    end;
            BUT[2]:=fgetc(ip);
            BUT[2]:=check(BUT[2]);
            if(BUT[2]=EOF)
                    do
                    Put_Char_to_File(ch,op);
```

```
        Put_Char_to_File(BUT[1],op);
                break;
            end;

        Put_Char_to_File(parse(BUT),op);
        match++;
            end;
    while(ch<>EOF);
    gettime(&sp);
    Display (" Completed at : %d:%d:%d.%d",sp.ti_hour,sp.ti_min,sp.ti_sec,sp.ti_hund);
    t1:=st.ti_hund+st.ti_sec*100+st.ti_min*6000+st.ti_hour*360000;
    t2:=sp.ti_hund+sp.ti_sec*100+sp.ti_min*6000+sp.ti_hour*360000;
    ibytes:=ftell(ip);
    obytes:=ftell(op);
    Close_File (op);
    Close_File (ip);
    cr1:=((ibytes-obytes)/ibytes*100);
    crate:=(obytes*8)/ibytes;
    cr2:=((1-crate/2)*100);
    Display " Input Size: "<<ibytes<<"bytes";
    Display " output Size: "<<obytes<<"bytes";
    Display " Total words matched from LUT :"<<match;
    Display " Compresstion ratio: "<<cr1;
    Display " Biological compression ratio : "<<cr2;
    Display " Compresstion rate :"<<crate;
    Display " Total Time taken : "<<(t2-t1)<<" hundreths seconds";
    end;
```

4.3.2 DLUT Decoding algorithm

Character: LUT[4],vbx[20];

Procedure Decompress ()

```
    do
    struct time st,sp;
    FILE *fp,*ip,*op;
    Integer :  i,p,q,r,order,rorder;
    Integer : t1,t2;
    Character  : ch;
    Display " Input 3 LUT initializing Characters: (a,t,c,g)";
    Display " 1:";
    LUT[0]=get_character();
    Display " 2:";
    LUT[1]=get_character();
    Display " 3:";
    LUT[2]=get_character();
    if(LUT[0]!='a' && LUT[1]!='a' && LUT[2]!='a')
            LUT[3]='a';
    if(LUT[0]!='t' && LUT[1]!='t' && LUT[2]!='t')
            LUT[3]='t';
    if(LUT[0]!='g' && LUT[1]!='g' && LUT[2]!='g')
            LUT[3]='g';
    if(LUT[0]!='c' && LUT[1]!='c' && LUT[2]!='c')
            LUT[3]='c';
    Display " 4:"<<LUT[3];
    do
    Display " LUT ordering (33/96)";
    Get_User_Input  order;
    while(order!=33 && order!=96);
    if(order==33)
            rorder=96;
    else
            rorder=33;
    Display " Enter compressed file:";
    Get_User_Input  vbx;
    ip=fopen(vbx,"r");
    if(ip==NULL)
    do
            Display " Unable to open file.";
            exit(1);
    end;
    Display " Enter name for output file:";
    Get_User_Input vbx;
    op=fopen(vbx,"w");
    if(op==NULL)
    do
            Display " Unable to open output file.";
            exit(1);
    end;
    gettime(&st);
    gettime(&sp);
    gettime(&st);
    Display (" Start time is : %d:%d:%d.%d",st.ti_hour,st.ti_min,st.ti_sec,st.ti_hund);
    do
    ch=fgetc(ip);
    if(ch==EOF)
            break;
    if(!isin(ch))
            do
            Put_Char_to_File(ch,op);
            end;
    else
            do
              rorder=Integer : (ch);
              rorder=rorder-order;
              if(rorder<0)
                    rorder*=-1;
            p=rorder/16;
            q=(rorder-p*16)/4;
            r=rorder-p*16-q*4;
            Put_Char_to_File(LUT[p],op);
            Put_Char_to_File(LUT[q],op);
            Put_Char_to_File(LUT[r],op);
            end;
    end;
    while(ch!=EOF);
    gettime(&sp);
    Display (" Completed at : %d:%d:%d.%d",sp.ti_hour,sp.ti_min,sp.ti_sec,sp.ti_hund);
```

```
t1=st.ti_hund+st.ti_sec*100+st.ti_min*6000+st.ti_hour*360000;
t2=sp.ti_hund+sp.ti_sec*100+sp.ti_min*6000+sp.ti_hour*360000;
Display " Total Time taken : "<<(t2-t1)<<" hundreths seconds";
Close_File(op);
Close_File(ip);
end;
```

4.4 Huffman Algorithm:

This algorithm recursively find a weighted binary tree with n given weights $w_1, w_2, \ldots w_n$. (Here weights mean frequency of n characters in text).

1. Arrange the weights in increasing weights.

2. Construct two leaf vertices with minimum weights, say w_i and w_j in the given weight sequence and parent vertex of weight $w_i + w_j$.

3. Rearrange remaining weights (excluding w_i and w_j but including parent vertex of weight $w_i + w_j$) in increasing order.

4. Repeat step 2 until no weight remains.

5. To find out Huffman code for each given weights (i.e. frequency of characters) traversing tree from root assign 0 when traverse left of each node & 1 when traverse right of each node.

4.5 Proposed Algorithm for Scheme-I:

This algorithm recursively find a weighted binary tree with n given weights $w_1, w_2, \ldots w_n$. (Here weights mean frequency of n characters in text). LEVEL is the input where the tree is altered.

1. Arrange the weights in increasing weights.

2. Construct two leaf vertices with minimum weights, say w_i and w_j in the given weight sequence and parent vertex of weight $w_i + w_j$.

3. Rearrange remaining weights (excluding w_i and w_j but including parent vertex of weight $w_i + w_j$) in increasing order.

4. Repeat step 2 until no weight remains.

5. Find out left most nodes and right most nodes at specified LEVEL and interchange their position with respect to their parent node.

6. To find out code for each given weights (i.e. frequency of characters) traversing tree from root assign 0 when traverse left of each node & 1 when traverse right of each node.

4.6 Proposed Algorithm for scheme-II:

This algorithm recursively find a weighted binary tree with n given weights $w_1, w_2, \ldots w_n$. (Here weights mean frequency of n characters in text). LEVEL is the input where the tree is altered.

1. Arrange the weights in increasing weights.

2. Construct two leaf vertices with minimum weights, say w_i and w_j in the given weight sequence and parent vertex of weight $w_i + w_j$.

3. Rearrange remaining weights (excluding w_i and w_j but including parent vertex of weight $w_i + w_j$) in increasing order.

4. Repeat step 2 until no weight remains.

5. Find out two nodes at specified LEVEL by binary digits and interchange their position with respect to their parent node.

6. To find out code for each given weights (i.e. frequency of characters) traversing tree from root assign 0 when traverse left of each node & 1 when traverse right of each node.

4.7 Algorithm for file mapping

- *Step1 : frame_size=LENGTH(String_1);*
- *Step2 : Repeat step 3 to 5 while String_1 is NULL.*
- *Step3 : Index=MISMATCH-INDEX(String_1,String_2).*
- *Step4 : IF Index>Length(String_1)-1 then goto step 6.*
- *Step5 : IF Index=Length(String_1)-1*

  then String_1=NULL.

  ELSE

  String_1=SUBSTRING(String_1,(Index+1)).

  String_2=SUBSTRING(String_2,(Index+1)).
- *Step6 : Error_no=Error_no + 1.*
- *Step7 : Percentage = ((Frame_size-Error_no)/Frame_size)*100.*
- *Step8 : Return Percentage.*

V. Algorithm Evaluation

5.1 Accuracy

As to the DNA sequence storage, accuracy must be taken firstly in that even a single base mutation, insertion, deletion or SNP would result in huge change of phenotype as we see in the sicklemia. It is not tolerable that any mistake exists either in compression or in decompression. Although not yet proved

Copyright © 2013 MECS

mathematically, it could be infer from DLUT that our algorithm is accuracy, since every base arrangement uniquely corresponds to an ASCII character.

5.2 Efficiency

You can see that the pre-coding algorithm can compress original file from 3 characters into 1 character for any DNA segment and destination file uses less ASCII character to represent successive DNA bases than source file.

5.3 Time Elapsed

Today many compression algorithms are highly desirable, but they require considerable time to execute. As our algorithm is based on a DLUT rather than sequence statistics, it can save the time of obtaining statistic information of sequence, and more, after the pre-coding routine, the character number is 1/3 of source one. You can see the elapsed time of our algorithm is in fraction of second.

5.4 Space Occupation

Our algorithm reads characters from source file and writes them immediately into destination file. It costs very small memory space to store only a few characters. The space occupation is in constant level. In our experiments, the OS has no swap partition. All performance can be done in main memory which is only 512 MB on our PC.

VI. Experimental Results

We tested DLUT on standard benchmark data used in [17]. For testing purpose we use two types of data sets. These standard sequences come from a variety of sources and include the complete genomes of two mitochondria: MPOMTCG, PANMTPACGA (also called MIPACGA), two chloroplasts: CHNTXX and CHMPXX (also called MPOCPCG); five sequences from humans: HUMGHCSA, HUMHBB, HUMHDABCD, HUMDYSTROP, HUMHPRTB; and finally the complete genome from two viruses: VACCG and HEHCMVCG (also called HS5HCMVCG).

These tests are performed on a computer whose CPU is Intel P-IV 3.0 GHz core 2 duo(1024FSB), Intel 946 original mother board, IGB DDR2 Hynix, 160GB SATA HDD Segate.

The definition of the compression ratio[1] is the same as in [18-19]; i.e., $1-(|O|/2|I|)$, where $|I|$ is number of bases in the input DNA sequence and $|O|$ is the length (number of bits) of the output sequence, other compression ratio[2] which is defined as $1-(|O|/|I|)$, where $|I|$ is the length(number of byte) number of bases in the input DNA sequence and $|O|$ is the length (number of byte) of the output sequence. The compression rate, which is defined as $(|O|/|I|)$, where $|I|$ is number of bases in the input DNA sequence and $|O|$ is the length (number of bits) of the output sequence. The improvement[9] over gzip-9, which is defined as (Ratio_of_gzip-9 – Ratio_of_LUT-3)/Ratio_of_gzip-9 *100 or Improvement =((Ratio_of_HUFFLUT-Ratio_of_DLUT)/ Ratio_of_HUFFLUT).The compression ratio and compression rate are presented in Table-10 & Table-11. Our result compared with gzip-9[20] in the same table.

Compression ratio and Compression rate and speed for the DNA sequences shown in Table I to Table-IV. From top to bottom, each row displays the result for an algorithm showing average compression and decompression speed in seconds per input byte (average computed over five runs for each sequence).“encode” means compression while “decode” mean decompression. Each operation is evaluated in two units, CPU clock and second.

Using Scheme-I

Table 14: Encryption using following levels

File size (byte)	Weighted Frequency for input file	level	Weighted frequency output file
7550	162	2	20
		3	20
		4	19
11358	183	2	26
		3	25
		4	25
26655	434	2	59
		3	59
		4	58
59890	778	2	124
		3	124
		4	123

We measure weighted frequency for different input and output text shown in table-14. We can conclude that weighted frequency of an input text and encrypted text are maintaining approx 8:1 ratio. That mean in case of input text and output text, number of characters having same frequency maintains the ratio nearly 8:1. So the chance of frequency analysis attack is reduced.

Copyright © 2013 MECS

Table 10: Compression Ratio & Rate are shown in the Table for first data set

Sequence	Base pair	File Size byte	Cellular DNA										Artificial DNA						
			Normal Sequences		Reverse Sequences		Complement sequences		Reverse Complement Sequence		Average Speed(in second)		Reduce file size byte	Compression ratio	compression rate(bits /base)	compare with gzip-9	Improvement	Average Speed(in second)	
			Compression ratio	compression rate (bits /base)	Compression ratio	compression rate (bits /base)	Compression ratio	compression rate (bits /base)	Compression ratio	compression rate (bits /base)	Encode	Decode						Encode	Decode
MTPACGA	100314	100314	-45.59881	2.91197	-45.59881	2.91197	-48.31827	2.96636	-48.32625	2.96652	<100	<100	40146	-60.08133	3.20162	4.62605	0.37096	<100	<100
MPOMTCG	186608	186608	-56.06619	3.12132	-56.06619	3.12132	-56.71353	3.13427	-56.71353	3.13427	.05	.03	74472	-59.63303	3.19266			<100	<100
CHNTXX	155844	155844	-53.71781	3.07435	-53.71781	3.07435	-52.75275	3.05505	-52.75275	3.05505	<100	<100	62254	-59.78542	3.19570			<100	<100
CHMPXX	121024	121024	-45.28688	2.90573	-45.28688	2.90573	-45.49842	2.90996	-45.49842	2.90996	.05	.02	48424	-60.04759	3.20095			<100	<100
HUMGHCSA	66495	66495	-60.34288	3.20685	-60.35491	3.20709	-60.29475	3.20589	-60.29475	3.20589	<100	<100	26733	-60.81209	3.21624			<100	<100
HUMHBB	73308	73308	-53.76221	3.07524	-53.77311	3.07546	-54.51514	3.09030	-54.50428	3.09008	<100	<100	29328	-60.02619	3.20052			<100	<100
HUMHDAB CD	58864	58864	-60.43761	3.20875	-60.43761	3.20875	-62.70725	3.25414	-62.69366	3.25387	<100	<100	23596	-60.34248	3.20685			<100	<100
HUMDYST ROP	38770	38770	-52.59220	3.05184	-52.61284	3.05225	-51.70493	3.03409	-51.68429	3.03368	<100	<100	15494	-59.85555	3.19711			<100	<100
HUMHPRT B	56737	56737	-54.93946	3.09878	-54.93946	3.09878	-55.23556	3.10471	-55.23556	3.10471	<100	<100	22653	-59.70530	3.19410			<100	<100
VACCG	191737	191737	-48.28019	2.96560	-48.28019	2.96560	-48.29688	2.96593	-48.29688	2.96593	<100	<100	76789	-60.19651	3.20393			<100	<100
HEHCM VCG	229354	229354	- 59.71816	3.19436	- 59.71467	3.19429	- 60.23091	3.20461	- 60.23091	3.20461	<100	<100	91794	- 60.09138	3.20182			<100	<100
Average				3.07407		3.07414		3.08411		3.08405					3.08405				

Table 11: Compression Ratio & Rate are shown in the Table for second data set

Sequence	Base pair	File Size byte	Cellular DNA										Artificial DNA						
			Normal Sequences		Reverse Sequences		Complement sequences		Reverse Complement Sequence		Average Speed(in second)		Reduce file size byte	Compression ratio	compression rate(bits /base)	compare with gzip-9	Improvement	Average Speed(in second)	
			Compression ratio	compression rate(bits /base)	Compression ratio	compression rate(bits /base)	Compression ratio	compression rate(bits /base)	Compression ratio	compression rate(bits /base)	Encode	Decode						Encode	Decode
atatsgs	9647	9647	-50.30579	3.00611	-50.30579	3.00611	-51.88140	3.03762	-51.88140	3.03762	<100	<100	3819	-58.34973	3.166995			<100	<100
atef1a23	6022	6022	-51.31186	3.02623	-51.17901	3.02358	-54.63301	3.09266	-54.63301	3.09266	<100	<100	2370	-57.42278	3.148456			<100	<100
atrdnaf	10014	10014	-56.18134	3.12362	-56.18134	3.12362	-64.56960	3.29139	-64.56960	3.29139	<100	<100	4006	-60.01597	3.20032			<100	<100
atrdnai	5287	5287	-52.75202	3.05505	-52.90334	3.05806	-63.19273	3.26385	-63.04142	3.26082	<100	<100	2119	-60.31775	3.206355			<100	<100
celk07e12	58949	58949	-52.26042	3.04520	-52.24685	3.04493	-48.92195	2.97843	-48.90837	2.97816	<100	<100	23589	-60.06377	3.201276	4.62605	0.30958	<100	<100
hsg6pdgen	52173	52173	-63.44852	3.26897	-63.46385	3.26927	-61.60849	3.23217	-61.60849	3.23217	<100	<100	20879	-60.07514	3.201503			<100	<100
mmzp3g	10833	10833	-59.62337	3.19246	-59.69722	3.19394	-63.09424	3.26188	-63.09424	3.26188	<100	<100	4377	-60.61727	3.23236			<100	<100
xlxfg512	19338	19338	-56.16920	3.12338	-56.21056	3.12421	-56.99658	3.13993	-56.95521	3.13910	<100	<100	7720	-59.68558	3.193712			<100	<100
Average				3.10512		3.10546		3.16224		2.76933									

Using modified Huffman algorithm Scheme-I and Scheme-II, the result are shown in below Table-12 &13

Table 12: Comparison Compression Ratio & Rate are shown in the Table for first data set

Sequence	Base pair	File Size byte	Using LUT algo.							Using HUFF algo.							Using LUTHUFF algo.							Improvement over LUT(%)	Improvement over HUFF(%)
			Reduce file size byte	Compression ratio	Compression rate(bits /base)	compare with gzip-9	Improvement	Speed(in second) Encode	Speed(in second) Decode	Reduce file size byte	Compression ratio	Compression rate(bits /base)	Compare with gzip-9	Improvement	Speed(in second) Encode	Speed(in second) Decode	Reduce file size byte	Compression ratio	Compression rate(bits /base)	compare with gzip-9	Improvement	Speed(in second) Encode	Speed(in second) Decode		
MTPACGA	100314	100314	36514	-45.59881	2.91197			<100	<100	24373	0.028131	1.943736			0.109	0.109	23960	0.044599	1.910800			0.109	0.219		
MPOMTCG	186608	186608	72808	-56.06619	3.12132			.05	.03	46653	-2.144E-05	2.000042			0.274	0.384	46780	-0.002743	2.005487			.0164	0.384		
CHNTXX	155844	155844	59890	-53.71781	3.07435			<100	<100	38962	-2.567E-05	2.000051			0.219	0.329	39011	-0.001283	2.002566			0.109	0.329		
CHMPXX	121024	121024	43958	-45.28688	2.90573			.05	.02	29208	0.03463776	1.930724			0.164	0.274	28432	0.060285	1.879428			0.109	0.219		
HUMGHCSA	66495	66495	26655	-60.34288	3.20685			<100	<100	16625	-7.519E-05	2.000150			0.054	0.164	16661	-0.002240	2.004481			0.054	0.109		
HUMHBB	73308	73308	28180	-53.76221	3.07524	4.62605	0.33548	<100	<100	18328	-5.456E-05	2.000109	4.62605	0.570109	0.109	0.164	18226	0.005510	1.988978	4.62605	0.571888	0.054	0.109	-0.552195	-0.415407
HUMHDABCD	58864	58864	23610	-60.43761	3.20875			<100	<100	14717	-6.795E-05	2.000135			0.054	0.109	14926	-0.014270	2.028540			0.054	0.109		
HUMDYS TROP	38770	38770	14790	-52.59220	3.05184			<100	<100	9694	-0.0001548	2.000309			0.054	0.109	9618	0.007686	1.984627			0.109	0.109		
HUMHPRTB	56737	56737	21977	-54.93946	3.09878			<100	<100	14186	-0.0001234	2.000246			0.054	0.109	14243	-0.004141	2.008283			0.054	0.054		
VACCG	191737	191737	71077	-48.28019	2.96560			<100	<100	47936	-3.651E-05	2.000073			0.219	0.384	46875	0.022097	1.955804			0.164	.0439		
HEHCMVCG	229354	229354	91580	-59.71816	3.19436			<100	<100	57340	-2.616E-05	2.000052			0.329	0.494	57801	-0.008066	2.016132			0.219	0.494		
Average					3.07407							1.988693							1.980466						

Table 13: Table shown the comparative result for second data set

Sequence	Base pair	File Size byte	Using LUT algo.							Using HUFF algo.							Using LUTHUFF algo.							Improvement over LUT	Improvement over HUFF
			Reduce file size byte	Compression ratio	Compression rate(bits /base)	compare with gzip-9	Improvement	Speed (in second)		Reduce file size byte	Compression ratio	Compression rate(bits /base)	Compare with gzip-9	Improvement	Speed (in second)		Reduce file size byte	Compression ratio	Compression rate(bits /base)	compare with gzip-9	Improvement	Speed (in second)			
								Encode	Decode						Encode	Decode						Encode	Decode		
atatsgs	9647	9647	3625	-50.30579	3.00611	4.62605	0.32877	<100	<100	2413	-0.000518	2.001037	4.62605	0.567435	0.054	<100	2371	0.016896	1.966207	4.62605	0.567685	0.054	0.054	-0.552632	-5.800269
atef1a23	6022	6022	2278	-51.31186	3.02623			<100	<100	1507	-0.000996	2.001993			<100	<100	1483	0.014945	1.97011			<100	<100		
atrdnaf	10014	10014	3910	-56.18134	3.12362			<100	<100	2505	-0.000599	2.001198			<100	0.054	2545	-0.016577	2.033154			<100	<100		
atrdnai	5287	5287	2019	-52.75202	3.05505			<100	<100	1323	-0.000946	2.001891			<100	0.054	1329	-0.005485	2.01097			<100	<100		
celk07e12	58949	58949	22439	-52.26042	3.04520			<100	<100	14739	-0.000119	2.000237			0.054	0.109	14628	0.007413	1.985174			0.054	0.109		
hsg6pdgen	52173	52173	21319	-63.44852	3.26897			<100	<100	13045	-0.000134	2.000268			0.109	0.109	13144	-0.007724	2.015449			0.054	0.109		
mmzp3g	10833	10833	4323	-59.62337	3.19246			<100	<100	2710	-0.000646	2.001292			<100	0.054	2713	-0.001754	2.003508			0.054	<100		
xlxfg512	19338	19338	7550	-56.16920	3.12338			<100	<100	4836	-0.00031	2.000621			0.054	0.054	4870	-0.007343	2.014686			<100	0.054		
Average					3.10512							2.001067							1.999907						

Table 15: Percentage of encryption

Input file size (byte)	Compress (%)	Swapping at Level	Lavenstein Dist.	% effect on actual text	% of encryption
71077	58	1	64471	94.58	100
		2	64278	93.893	60.3
		3	64178	93.685	51.85
		4	63985	93.049	44.32
43958	60	1	45365	94.22	100
		2	45123	93.717	58.17
		3	44930	93.52	52.93
		4	44838	92.78	47.67
36514	61	1	28499	93.897	100
		2	28432	93.7	62.44
		3	28350	93.329	51.33
		4	28087	92.251	44.48
14790	62	1	16266	93.85	100
		2	16242	93.62	61.14
		3	16238	92.8	55.67
		4	16204	91.9	48.95
7550	67	1	8582	93.65	100
		2	8508	93.525	58.33
		3	8482	92.588	50.02
		4	8462	91.48	44.16

From the above table-4 we can conclude with the ratio of encryption of about 45% to 60%, we could achieve the ratio of damage to the file of nearly 94% in some cases.

We also observe that if we change at the top level (1) then Lavenstein Distance is at highest point & % of damage is highest. When we change at lower level Lavenstein Distance is decrease & % of damage also decrease. According to above results we draw two graphs for five different real text cases; one is effectiveness on the output text by different input file size at different level (Graph-I) and other is effectiveness on the output text file by increasing encryption. From fig. a, we observe if the file size is decreased then %of modification from the actual text is decreased and from Graph-II; if %of encryption is increased then %of modification from the actual text is also increased.

Effectiveness on the output text by different input size at different level.

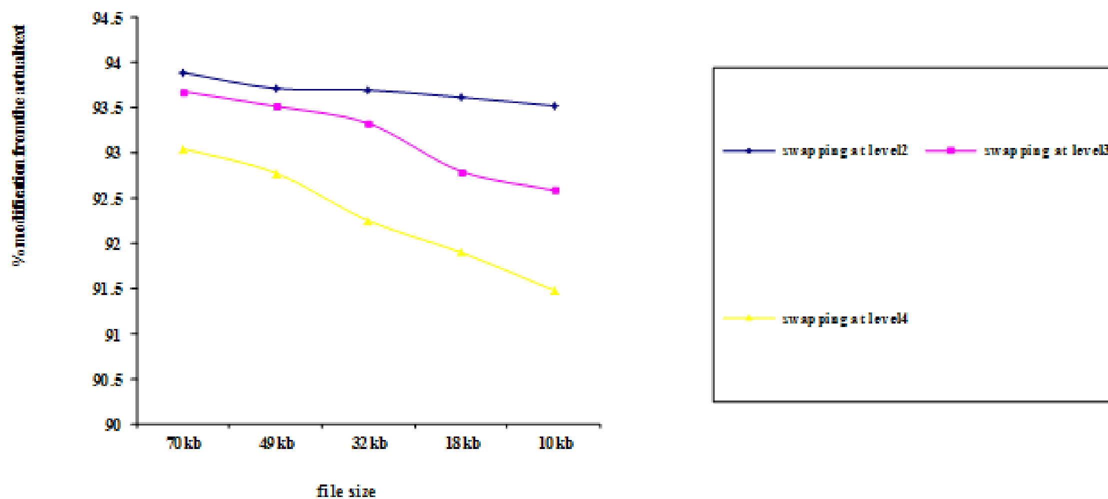

Fig. 12: (% modification for the actual text vs. file size)

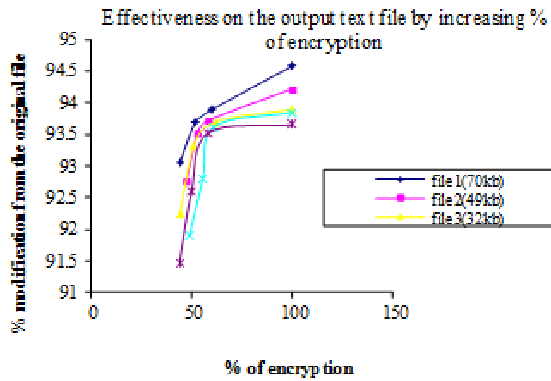

Fig. 13: (% modification from the original file vs. % of encryption)

Using Scheme-II

From the table 16 we observe that if we apply swapping at lower level then percentage of encryption is also reduced but original text will affect minimum 85%. This also provides higher security if we consider two binary values as a key and encrypt them.

According to above results we draw graph for three different real text cases. From Graph-III; if %of encryption is increased then %of modification from the actual text is also increased.

Table 16: Effect using level charge

Input file size (byte)	Compress (%)	Specification of two nodes				Lavenstein Dist.	% of encryption	% effect on actual text
		1st node		2nd node				
		Level	Binary value	Level	Binary value			
71077	58	3	110	4	0100	64182	51	92.13
		6	111111	4	0010	60985	47.19	89.049
		4	0101	5	11111	63116	49.3	90.6
		5	00001	6	111111	61089	46.5	87.69
43958	60	5	10100	4	0001	41545	49.8	90.15
		2	11	3	000	44838	53.13	93.125
		3	001	4	1110	44797	51.06	91.98
		2	00	2	10	45790	54.68	94.02
36514	61	4	1000	3	101	27997	51	93.65
		5	00101	3	101	28087	47.98	91.051
		2	01	2	10	29502	53.39	94.65
		4	0000	4	1111	28511	49.35	91.47

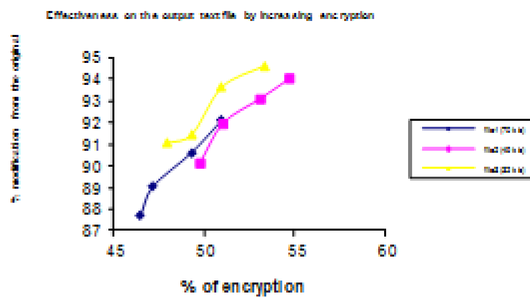

Fig. 14: (% modification from the original file vs. % of encryption)

Table 17:

Sequence	Base pair/File size	GZIP	BZIP2
atatsgs	9647	2.1702	2.15
atef1a23	6022	2.0379	2.15
atrdnaf	10014	2.2784	2.15
atrdnai	5287	1.8846	1.96
celk07e12	58949		
hsg6pdgen	52173	2.2444	2.07
mmzp3g	10833	2.3225	2.13
xlxfg512	19338	1.8310	1.80
Average			

Table 18:

Sequence	Base pair /File size	GZIP	BZIP2
MTPACGA	100314	2.2919	2.12
MPOMTCG	186609	2.3288	2.17
CHNTXX	155844	2.3345	2.18
CHMPXX	121024	2.2818	2.12
HUMGHCSA	66495	2.0648	1.31
HUMHBB	73308	2.2450	
HUMHDABCD	58864	2.2389	2.07
HUMDYSTROP	38770	2.3618	2.18
HUMHPRTB	56737	2.2662	2.09
VACCG	191737	2.2518	2.09
HEHCMVCG	229354	2.3275	2.17

The results from Table 12 & 13 show our algorithms to be the best solution for client side decryption - decompression with the shortest and linearly increasing decompression time. However, our algorithm doesn't compress sequences as much as others for many of the cases in the compression ratio table 17 & 18 but it provide high information security. This is because our algorithm uses less than 2 bits to represent one nucleotide

Copyright © 2013 MECS

In order to compare the overall performance, we conducted further studies involving sending actual sequence files of varying sizes (without compression) to measure the calculated time (T_c) needed for the transmission from the source to the destination. Then we compressed those files using both compression & encryption algorithms. The total time T, defined as the sum of the encryption compressed file transmission time (T_{ec}) plus the client side decompression time (T_{dd}), is measured by both these methods.

VII. Result Discussion

We can feel the time of program running since they are in second level. Some ones even cost minutes or hours of time to run. But our algorithm runs almost 10^3 times faster. Our algorithm performances is better than it in both compression ratio and elapsed time. Our algorithm is very useful in database storing. We can keep sequences as records in database instead of maintaining them as files. By just using the pre-coding routine, users can obtain original sequences in a time that can't be felt. Additionally, our algorithm can be easily implemented while some of them will take you more time to program.

From these experiments, we conclude that pre coding matching patter are same in all type of sources and pre coding Look up Table plays a key role in finding similarities or regularities in DNA sequences. Straight line graph declared that compression rate are same in all type of sources. Output file contain ASCII character with unmatched a,u,g and c so, it can provide information security which is very important for data protection over transmission point of view. Also, at the time of encoding require authenticate input base pair (from a,t,g and c) and ASCII character starting position for derivate LUT table, produce original sequence which is encode by this techniques. These techniques provide the high security to protect nucleotide sequence in a particular source. Not necessary to derive all combinational result because this method created 24X2 =48 different types of LUT but total sub-string are same in all cases but their position is different only.

Here we can get better security than static LUT. In static LUT encode/decode does not depends on LUT sub-string input value from a,t,g and c, but dynamic LUT must depends on LUT sub-string input value from a,t,g and c. In that situation authentication is very important.

Encoding time "Sub-sequence size-1" base segment are remaining, (if at the end of file segment are not match exactly with pre-coded table)We cannot find any arrangement in table-I or table-II. In these circumstances, we just write the original segment into destination file. To increase the probability of compaction we match the sequence in other orientation such as reverse , complement and reverse complement the input file. But experimental result showing no meaningful changes are found using other orientation taking as input.

The ratio of decompression time to original transmission time of the uncompressed sequence file (T_{dd} / T_c), reduces with increasing file size. This means our client side decryption decompression technique with our algorithm is a better choice for larger sequence files. Our client side decryption decompression technique can be implemented by a genome search agent and decryption decompression time can be estimated by two empirical equations according to our experiments.

The graph of compression time versus sequence file length is somewhat non-linear, because of the complexity that arises when the sequence length is not divisible by 4, which means that not all the bit pairs in the last byte of the compressed file may represent valid nucleotides. One solution to this problem is to add an extra byte at the end of compressed file which is the count (1-4) of the number of valid nucleotides in the previous byte.

Our algorithm combines moderate encryption compression with reduced decryption decompression time to achieve the best performance for client side sequence delivery compared with existing techniques. Its linearity in decompression time and close linearity in compression time make it an effective compression tool for commercial usage. Given, for a particular connection speed, the efficiency achieved using our algorithm; this compression technique is recommended for transmission of queried sequence files.

VIII. Conclusion

In this article, we discussed a new DNA compression algorithm whose key idea is dynamic LUT. This compression algorithm gives a good model for compressing DNA sequences that reveals the true characteristics of DNA sequences. The compression results of dynamic LUT for DNA sequences also indicate that our method is more effective than many others. Dynamic LUT is able to detect more regularity in DNA sequences, such as mutation and crossover, and achieve the best compression results by using this observation. Dynamic LUT fails to achieve higher compression ratio than others standard method, but dynamic LUT has provide very high information security and high authentication user.

In this work we have performed computational experiments to selectively encrypt the compressed text of different sizes generated through static Huffman encoding technique and compare the effectiveness in terms of dissimilarity from the original file if one has to decrypt without the key and the resistance of the cipher text from the attacks based on statistical property of the plain text. We have used two different schemes; in scheme-I swapping of nodes is done at

Copyright © 2013 MECS

specified level based on key and in scheme-II swapping is done between two specified nodes at different levels. We have found from our experiments, the effectiveness of the encryption system increases as the level at which swapping is done, increases. We have achieved in the both the scheme with 45% to 60% encryption of original text, near about 90% to 93% of damage in original file.

We have also measures the redundancy on the basis of weighted frequency. We have observed weighted frequency of an input text and encrypted text are nearly 8:1 ratio. That mean in case of input text and output text, number of characters having same frequency maintain the ratio approx 8:1. So the probability of frequency analysis attack is low.

In our experiments of selective encryption of text, on the basis of statistical property of words therein, we have found that to get 90% effectness on actual text then %of encryption should be more than 60, since number of distinguishable words are huge and their frequency is much less than number of frequency of characters. This approach has a good scope as a selective encryption scheme because of the fact that in a text of any language the articles, verbs, and prepositions have a higher frequency compared to the other words relevant to the core content of the text.

The problem small key space has to be sorted out to effectively apply this encryption system in real world.

IX. Future Work

We are trying to do more, such as combining our dynamic LUT pre-coding routine with other compression algorithms, to revise our algorithm in order to improve its performance. We are trying to build a finite LUT which implements the mapping relationship of our coding process.

Here in this work, we have taken into consideration the statistical property of a character or a word while doing compression. Instead, one can consider the statistical property of any number characters or bits, the number of bits may be provided by the user depending on the application or may be chosen automatically on the basis of entropy. In that case this encryption technique may be extended to any type of media. The effectiveness of selective encryption may be studied for the other statistical compression algorithms available.

Acknowledgements

Above all, author is grateful to all our colleagues for their valuable suggestion, moral support, interest and constructive criticism of this study.

References

[1] M. Li and P. Vitányi, An Introduction to Kolmogorov Complexity and Its Applications, 2nd ed. New York: Springer-Verlag, 1997.

[2] R. Curnow and T. Kirkwood, “Statistical analysis of deoxyribonucleic acid sequence data-a review,” J. Royal Statistical Soc., vol. 152, pp. 199-220, 1989.

[3] S. Grumbach and F. Tahi, “A new challenge for compression algorithms: Genetic sequences,” J. Inform. Process. Manage., vol. 30, no. 6, pp. 875-866, 1994.

[4] É. Rivals, O. Delgrange, J.P. Delahaye, M.Dauchet, M.O. Delorme et al., “Detection of significant patterns by compression algorithms: the case of Approximate Tandem Repeats inDNAsequences,” CABIOS, vol. 13, no. 2, pp. 131-136,1997.

[5] K. Lanctot, M. Li, and E.H. Yang, “Estimating DNA sequence entropy,”in Proc. SODA 2000, to be published.

[6] D. Loewenstern and P. Yianilos, “Significantly lower entropy estimates for natural DNA sequences,” J. Comput. Biol., to be published (Preliminary version appeared in a DIMACS workshop, 1996.)

[7] Two algorithms for constructing efficient huffman-code based reversible variable length Codes Chia-Wei Lin; Ja-Ling Wu; Yuh-Jue Chuang

[8] Bentley J. L., Sleator D.D., Tarjan R.E., and Wei V., "A locally adaptive data compression scheme", Communications of the *ACM*, **29**(4), 320-330, 1986.

[9] J. G. Cleary and I. H. Witten. Data compression using adaptive coding and partial string matching. IEEE Trans. Comm., COM-32(4):396–402, April 1984.

[10] C. E. Shannon, “A mathematical theory of communication,” The Bell System Technical Journal, vol. 27, 1948.

[11] D. A. Huffman, “A method for the construction of minimum-redundancy codes,“Proc. IRE, vol. 40, pp. 1098-1101,1952.

[12] On the competitive optimality of Huffman codes by Thomas. M. Cover.

[13] Guaranteed Synchronization of Huffman Codes with Known Position of Decoder Marek Tomasz Biskup, Wojciech Plandowski,

[14] C. E. Shannon, “Communication theory of secrecy systems,” Bell Systems Technical Journal, v. 28, October 1949, pp. 656-715.

Copyright © 2013 MECS

[15] Chen, L., Lu, S. and Ram J. 2004. "Compressed Pattern Matching in DNA Sequences". Proceedings of the 2004 IEEE Computational Systems Bioinformatics Conference (CSB 2004)

[16] Toshiko Matsumoto, Kunihiko Sadakane and Hiroshi Imai. " Biological Sequence Compression Algorithms", Genome Informatics 11 : 43-52 (2000)

[17] S. Grumbach and F. Tahi, "A new challenge for compression algorithms: Genetic sequences," J. Inform. Process. Manage., vol. 30, no. 6, pp. 875-866, 1994.

[18] Xin Chen, San Kwong and Mine Li, "A Compression Algorithm for DNA Sequences Using Approximate Matching for Better Compression Ratio to Reveal the True Characteristics of DNA", IEEE Engineering in Medicine and Biology,pp 61-66,July/August 2001.

[19] Adam Drozdek " Elements of Data Compression", Vikas Publishing House (2002)

[20] T. Matsumoto,K.Sadakame and H. Imani, "Biological sequence compression algorithm", Genome Informatics 11:43-52 (2000).

[21] ASCII code. [Online]. Available: http://www.asciitable.com

[22] National Center for Biotechnology Information, http://www.ncbi.nlm.nih.gov

Author's Profiles

Syed Mahamud Hossein is perusing Ph.D for Computer Science in Vidyasagar University. He had received his post graduate degree in Computer Applications from Swami Ramanand Teerth Marathawada University, Nanded and Master of Engineering in Information Technology from West Bengal University of Technology, Kolkata. He has worked as the Senior Lecturer in Haldia Institute of Technology, Haldia, Lecturer on contract basis in Panskura Banamali College, Panskura and Lecturer in Iswar Chandra Vidyasagar Polytechnic, Govt. of West Bengal, Jgargram. Presently he is working as a District Officer, Regional Office, Kolaghat, Directorate of Vocational Educational & Training, West Bengal since 2010. His research interests includes Bioinformatics, Compression Techniques & Cryptography, Design and Analysis of Algorithms & Development of Software Tools. He is a member of professional societies like Computer Society of India & Indian Science Congress Association.

How to cite this paper: Syed Mahamud Hossein, S.Roy,"A Compression & Encryption Algorithm on DNA Sequences Using Dynamic Look up Table and Modified Huffman Techniques", International Journal of Information Technology and Computer Science(IJITCS), vol.5, no.10, pp.39-61, 2013. DOI: 10.5815/ijitcs.2013.10.05