IOSR Journal of Computer Engineering (IOSRJCE)
ISSN: 2278-0661, ISBN: 2278-8727 Volume 6, Issue 1 (Sep-Oct. 2012), PP 42-50
www.iosrjournals.org

An Efficient Biological Sequence Compression Technique Using LUT and Repeat in the Sequence

Subhankar Roy[1], Sunirmal Khatua[2], Sudipta Roy[3], Prof. Samir K. Bandyopadhyay[4]
[1]*Department of Computer Science & Engineering, GNIT, WBUT, Kolkata, India*
[2,3,4]*Department of Computer Science and Engineer, University of Calcutta,*
92 A.P.C. Road, Kolkata-700009, India

Abstract. *Data compression plays an important role to deal with high volumes of DNA sequences in the field of Bioinformatics. Again data compression techniques directly affect the alignment of DNA sequences. So the time needed to decompress a compressed sequence has to be given equal priorities as with compression ratio. This article contains first introduction then a brief review of different biological sequence compression after that my proposed work then our two improved Biological sequence compression algorithms after that result followed by conclusion and discussion, future scope and finally references. These algorithms gain a very good compression factor with higher saving percentage and less time for compression and decompression than the previous Biological Sequence compression algorithms.*

Keywords: *Hash map table, Tandem repeats, compression factor, compression time, saving percentage, compression, decompression process.*

I. Introduction

Compression arises because approximately 139,266,481,398 bases (139.3 billion) from 151,824,421(153 million) reported sequences are there in the GenBank database in April 15, 2012[1] and the database size getting two or three times bigger annually. Compression required so that large data sequence after compression can easily be transferred through network channel to analyst. Data compression decrease communication cost by speed up transmission due to limited bandwidth communication of modern world and storage cost. This is very useful when processing, storing or transferring a huge file, which needs lots of resources. It is a well known fact that biological sequence compression is a useful tool to recover information from biological sequences. It has also been observed that better compression gives better understanding as it is sent to different station for its observation. Hence, compression of Biological sequences is a need for analyst for its storage, alignment and analyzing from remote locations and indeed a very challenging task. The Biological sequence consist of four nucleotide bases {A, C, G, T}, so only two bits are enough to store each base. Sometimes a special character N appears in some sequence which have equal probability of being A, C, G and T but it is an exceptional case. Other character like R, Y, W, S, M, K, H, B, V and D may appear in some sequence but they are very rare cases.

General text compression algorithms do not utilize the specific characteristics of DNA sequences i.e. more redundant nature of Biological sequences. The standard techniques of text compression do not compress these sequences; rather they expand the size of file with more than 2 bits per bases. Even though these compression tools are universal compression algorithms are designed for text compression. The regularities in Biological sequences are more.

It is known that the compression means mapping between source file and destination file, so our compression algorithm leads to find those relationship.

The objective of this paper is to build a finite LUT in which the combination of three and four characters of characters A, C, G, T and N are taken as input and by mapping relationship during compression process output correspond to ASCII character and then find tandem repeats in the sequence for the first case. We compress the individual DNA sequences, not the whole genome sequence [2].

II. Brief review

Several compression algorithms specialized for Biological sequences have been developed in the last decade and some of these are; DNABIT compress tool (DBC) [3], GENBIT compress tool (GBC) [4], HUFFBIT COMPRESS [5] etc. One knows that all such algorithms take a long time but achieving high speed and best compression factor remains to be a challenging task. In this article, it has been tried to cope the above said problem. In this work it has been tried to achieve a better compression factor, saving percentage and runs significantly faster than existing compression program.

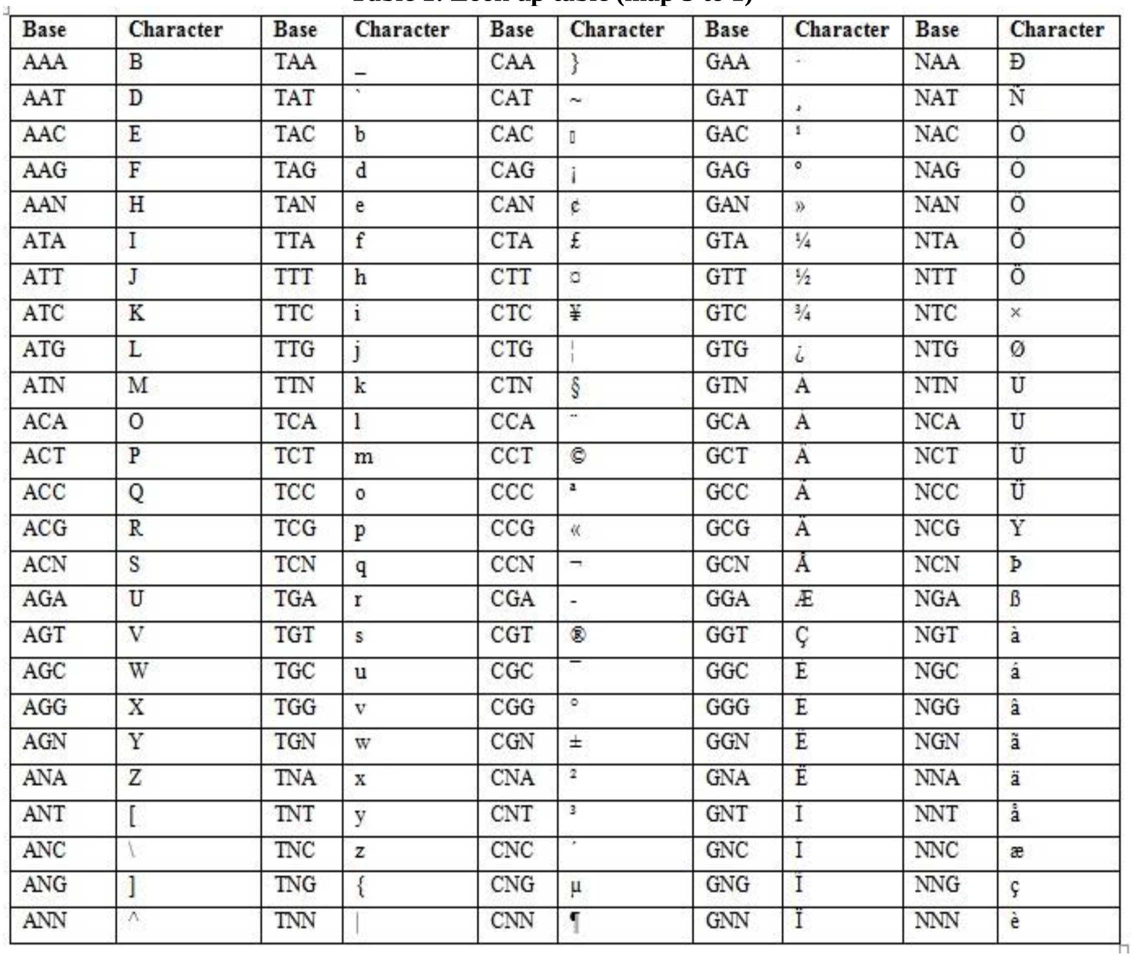
Research work has been carried out for developing efficient Biological sequence compression technique. Some of them use the property similarity of genomic sequence [6], approximate matching [7], complementary palindrome [8] and cross chromosomal similarities within sequences [9] etc. In present work the properties tandem repeat in the sequences has been utilized.

III. Proposed work

First step of coding rule is based on LUT.

For algorithm-1, we take all possible three characters combination of characters {A, T, C, G or N}, so total 5^3=125 combinations are formed. Then we mapped those 125 combinations into 125 chosen ASCII character 8 bit each. The generated LUT is given in table 1.Here it has been also observed that any sequence file of characters a, t, g, c or n and A, T, G, C or N have same meaning. As an example if a line segment "AAANNNTTTGGGCCC" appeared in the input text, in the output file, it is represented as "BèhÉª". The generated output is become case-sensitive.

Table 1: Look up table (map 3 to 1)



Base	Character	Base	Character	Base	Character	Base	Character	Base	Character
AAA	B	TAA	_	CAA	}	GAA	·	NAA	Ð
AAT	D	TAT	`	CAT	~	GAT	¸	NAT	Ñ
AAC	E	TAC	b	CAC	▯	GAC	¹	NAC	Ò
AAG	F	TAG	d	CAG	¡	GAG	º	NAG	Ó
AAN	H	TAN	e	CAN	¢	GAN	»	NAN	Ô
ATA	I	TTA	f	CTA	£	GTA	¼	NTA	Õ
ATT	J	TTT	h	CTT	¤	GTT	½	NTT	Ö
ATC	K	TTC	i	CTC	¥	GTC	¾	NTC	×
ATG	L	TTG	j	CTG	¦	GTG	¿	NTG	Ø
ATN	M	TTN	k	CTN	§	GTN	À	NTN	Ù
ACA	O	TCA	l	CCA	¨	GCA	Á	NCA	Ú
ACT	P	TCT	m	CCT	©	GCT	Â	NCT	Û
ACC	Q	TCC	o	CCC	ª	GCC	Ã	NCC	Ü
ACG	R	TCG	p	CCG	«	GCG	Ä	NCG	Ý
ACN	S	TCN	q	CCN	¬	GCN	Å	NCN	Þ
AGA	U	TGA	r	CGA	-	GGA	Æ	NGA	ß
AGT	V	TGT	s	CGT	®	GGT	Ç	NGT	à
AGC	W	TGC	u	CGC	¯	GGC	È	NGC	á
AGG	X	TGG	v	CGG	°	GGG	É	NGG	â
AGN	Y	TGN	w	CGN	±	GGN	Ê	NGN	ã
ANA	Z	TNA	x	CNA	²	GNA	Ë	NNA	ä
ANT	[	TNT	y	CNT	³	GNT	Ì	NNT	å
ANC	\	TNC	z	CNC	´	GNC	Í	NNC	æ
ANG	]	TNG	{	CNG	µ	GNG	Î	NNG	ç
ANN	^	TNN	|	CNN	¶	GNN	Ï	NNN	è

For algorithm-2, we take all possible four characters combination of characters {A, T, C and G}, so total 4^4=256 combinations are formed. Then we mapped those 256 combinations into 256 ASCII character 8 bit each. We have not taken into account character 'N', because this is an exceptional case. The generated LUT is shown in table 2.For example for line segment "CAAACTTACTGA" output is "€"œ".

Table 2: Look up table (map 4 to 1)

Base	Character	Base	Character	Base	Character	Base	Character
AAAA	(char)0	TAAA	(char)64	CAAA	(char)128	GAAA	(char)192
AAAT	(char)1	TAAT	(char)65	CAAT	(char)129	GAAT	(char)193
AAAC	(char)2	TAAC	(char)66	CAAC	(char)130	GAAC	(char)194
AAAG	(char)3	TAAG	(char)67	CAAG	(char)131	GAAG	(char)195
AATA	(char)4	TATA	(char)68	CATA	(char)132	GATA	(char)196
AATT	(char)5	TATT	(char)69	CATT	(char)133	GATT	(char)197
AATC	(char)6	TATC	(char)70	CATC	(char)134	GATC	(char)198
AATG	(char)7	TATG	(char)71	CATG	(char)135	GATG	(char)199
AACA	(char)8	TACA	(char)72	CACA	(char)136	GACA	(char)200
AACT	(char)9	TACT	(char)73	CACT	(char)137	GACT	(char)201
AACC	(char)10	TACC	(char)74	CACC	(char)138	GACC	(char)202
AACG	(char)11	TACG	(char)75	CACG	(char)139	GACG	(char)203
AAGA	(char)12	TAGA	(char)76	CAGA	(char)140	GAGA	(char)204
AAGT	(char)13	TAGT	(char)77	CAGT	(char)141	GAGT	(char)205
AAGC	(char)14	TAGC	(char)78	CAGC	(char)142	GAGC	(char)206
AAGG	(char)15	TAGG	(char)79	CAGG	(char)143	GAGG	(char)207
ATAA	(char)16	TTAA	(char)80	CTAA	(char)144	GTAA	(char)208
ATAT	(char)17	TTAT	(char)81	CTAT	(char)145	GTAT	(char)209
ATAC	(char)18	TTAC	(char)82	CTAC	(char)146	GTAC	(char)210
ATAG	(char)19	TTAG	(char)83	CTAG	(char)147	GTAG	(char)211
ATTA	(char)20	TTTA	(char)84	CTTA	(char)148	GTTA	(char)212
ATTT	(char)21	TTTT	(char)85	CTTT	(char)149	GTTT	(char)213
ATTC	(char)22	TTTC	(char)86	CTTC	(char)150	GTTC	(char)214
ATTG	(char)23	TTTG	(char)87	CTTG	(char)151	GTTG	(char)215
ATCA	(char)24	TTCA	(char)88	CTCA	(char)152	GTCA	(char)216
ATCT	(char)25	TTCT	(char)89	CTCT	(char)153	GTCT	(char)217
ATCC	(char)26	TTCC	(char)90	CTCC	(char)154	GTCC	(char)218
ATCG	(char)27	TTCG	(char)91	CTCG	(char)155	GTCG	(char)219
ATGA	(char)28	TTGA	(char)92	CTGA	(char)156	GTGA	(char)220
ATGT	(char)29	TTGT	(char)93	CTGT	(char)157	GTGT	(char)221
ATGC	(char)30	TTGC	(char)94	CTGC	(char)158	GTGC	(char)222
ATGG	(char)31	TTGG	(char)95	CTGG	(char)159	GTGG	(char)223
ACAA	(char)32	TCAA	(char)96	CCAA	(char)160	GCAA	(char)224
ACAT	(char)33	TCAT	(char)97	CCAT	(char)161	GCAT	(char)225
ACAC	(char)34	TCAC	(char)98	CCAC	(char)162	GCAC	(char)226
ACAG	(char)35	TCAG	(char)99	CCAG	(char)163	GCAG	(char)227
ACTA	(char)36	TCTA	(char)100	CCTA	(char)164	GCTA	(char)228
ACTT	(char)37	TCTT	(char)101	CCTT	(char)165	GCTT	(char)229
ACTC	(char)38	TCTC	(char)102	CCTC	(char)166	GCTC	(char)230
ACTG	(char)39	TCTG	(char)103	CCTG	(char)167	GCTG	(char)231
ACCA	(char)40	TCCA	(char)104	CCCA	(char)168	GCCA	(char)232
ACCT	(char)41	TCCT	(char)105	CCCT	(char)169	GCCT	(char)233
ACCC	(char)42	TCCC	(char)106	CCCC	(char)170	GCCC	(char)234
ACCG	(char)43	TCCG	(char)107	CCCG	(char)171	GCCG	(char)235
ACGA	(char)44	TCGA	(char)108	CCGA	(char)172	GCGA	(char)236
ACGT	(char)45	TCGT	(char)109	CCGT	(char)173	GCGT	(char)237
ACGC	(char)46	TCGC	(char)110	CCGC	(char)174	GCGC	(char)238
ACGG	(char)47	TCGG	(char)111	CCGG	(char)175	GCGG	(char)239
AGAA	(char)48	TGAA	(char)112	CGAA	(char)176	GGAA	(char)240
AGAT	(char)49	TGAT	(char)113	CGAT	(char)177	GGAT	(char)241
AGAC	(char)50	TGAC	(char)114	CGAC	(char)178	GGAC	(char)242
AGAG	(char)51	TGAG	(char)115	CGAG	(char)179	GGAG	(char)243
AGTA	(char)52	TGTA	(char)116	CGTA	(char)180	GGTA	(char)244
AGTT	(char)53	TGTT	(char)117	CGTT	(char)181	GGTT	(char)245
AGTC	(char)54	TGTC	(char)118	CGTC	(char)182	GGTC	(char)246

AGTG	(char)55	TGTG	(char)119	CGTG	(char)183	GGTG	(char)247
AGCA	(char)56	TGCA	(char)120	CGCA	(char)184	GGCA	(char)248
AGCT	(char)57	TGCT	(char)121	CGCT	(char)185	GGCT	(char)249
AGCC	(char)58	TGCC	(char)122	CGCC	(char)186	GGCC	(char)250
AGCG	(char)59	TGCG	(char)123	CGCG	(char)187	GGCG	(char)251
AGGA	(char)60	TGGA	(char)124	CGGA	(char)188	GGGA	(char)252
AGGT	(char)61	TGGT	(char)125	CGGT	(char)189	GGGT	(char)253
AGGC	(char)62	TGGC	(char)126	CGGC	(char)190	GGGC	(char)254
AGGG	(char)63	TGGG	(char)127	CGGG	(char)191	GGGG	(char)255

Second step of coding rule is based on ASCII character.

In algorithm-1 as the total number of ASCII required 125 so we can choose ASCII value out of total 256 ASCII value. That is why in this algorithm no problem occurred during the hash mapping.

But in algorithm-2 as all the ASCII value needed some problem arise during the mapping. The problem is that ASCII character of ASCII value 10 and 13 giving new line, as we compress and decompress file line by line so if we put those character in LUT during writing to file by hash mapping line break occurs. So we write the original characters block corresponding to these two values to the compress file. Another problem is five extended ASCII value (129, 141, 143, 144 and 157) not giving the correct characters so they cannot be map. So during compression if those ASCII value appear we do not map them using hash map table rather than we write the original block of characters value. That is why during decompression we have to take into account those characters block i.e. we have to search for those characters block because for them no reverse mapping is required.

Third step of coding rule is based on tandem repeats.

For algorithm-1 this feature can be used because during compression we count the number of repeats and write the corresponding character value to the compressed file. It has been observed that maximum number of repeats block never exceed 65 i.e. 195 character for any of the used sequences. So value 1 to 65 is used for handling those repeats except value 10 and 13 cause giving above. The repeat character followed by the number of repeats is written to the file. For e.g. input "AATAATAATAATAATAAC" have 4 tandem repeats. So these sequence represented by total 3 characters.

For algorithm-2 this feature cannot used because all ASCII value already used during mapping.

Fourth step of coding rule is based on Segment which consists of 1 or 2 characters.

We compress the input sequence line by line both for algorithm-1 and algorithm-2. For first algorithm we divide each line of the input sequence into segment of length 3 so if we modulate the line length by 3 i.e. Line length % 3 = 2 or 1 or 0. So the segments of 2 or 1 character are written as it is to the output file in the same line i.e. no mapping.

For second algorithm we divide each line of the input sequence into segment of length 4 so if we modulate the line length by 4 i.e. Line length % 4 = 3 or 2 or 1 or 0. So the segments of 3 or 2 or 1 character are written as it is to the output file no mapping but after a line break because they cannot be identify as 256 ASCII value already being used during mapping.

Algorithm for block size 3 with tandem repeats is given below

3.1 Algorithm-1

3.1.1 Compression algorithm

Input : Text file contain characters A,C,G and T or a,c,g and t or A,C,G,T and N or a,c,g,t and n respectively.

Output: Compressed file in terms of ASCII characters.

```
Function Compression (Original file)
Step1: An ArrayList is made of all possible input characters.
Step2: A HashMap table map string to character is made dynamically at the run time of compression.
Step3: While string <> null do //string reads the content of the file line by line
    While string.length > 3 do
        Read a combination of three characters block and add them in a list.
    End while
    Add the last block to the same list.
    For String string1: list do
        If string1.length = 3 then
            If string1= temp then //where temp store previous three character, initially null
                count = count+1 //count the number of repeats
            Else {
                If count > 0 then
```

```
            Write the ASCII value of count.
        Else
            Write this character to output file by mapping
        End if
        }
        temp = string1
    Else if string1.length < 3 then
        If count > 0 then
            Write the ASCII value of count.
        End if
        Write string1 value to the output file.
    End if
    End for
    If count > 0 then
        Write the ASCII value of count.
    End if
    Read the next line.
    temp = null
End while
Step4: Close the output file.
```

3.1.2 Decompression algorithm

Input : Compressed file.
Output : Original file.

```
Function Decompression (Compressed file)
Step1: An ArrayList is made of all possible input characters.
Step2: A HashMap table is made dynamically at the run time of compression.
Step3: Read the content of text file
While string<>null do
    For 1 to string.length in step 1 do
        Read a character from the string
        If character = 'A' or 'a' or 'C' or 'c' or 'G' or 'g' or 'T' or 't' or 'N' or 'n' then
            Write character to the output file.
        Else {
            If count>=1 and <=65 then
                For 1 to count in step 1 do
                    Write string of 3 characters to the file.
                End for
            Else
                Write string of 3 characters to the file.
            End if
        }
        End if
    End for
    Read the next line
End while
Step4: Close the output file.
```

Algorithm for block size 4 without tandem repeats is given below

3.2 Algorithm-2

3.2.1 Compression algorithm

Input : Text file contains character A,C,G and T or a,c,g and t respectively.
Output : Compressed file in terms of all ASCII characters.

```
Function Compression (Text file)
Step1: An ArrayList is made of all possible input characters.
Step2: A HashMap table map string to character is made dynamically at the run time of compression.
Step3: Read the content of file.
While string <> null do
    temp = string //temp store each line temporarily
    While string.length() > 4 do
```

```
        Read a combination of four characters block.
        Add them in a list.
    End while
    Add the last block to the same list.
    For String string1: list do
        If string1.length() = 4 then
            Read the mapped character of block.
            If  i=10 or 13 or 129 or 141 or 143 or 144 or 157 then // i store the integer value of the character.
                Write the original block.
            Else
                Write the character.
            End if
        Else {
            Write newline character followed by block.
        }
        End if
    End for
    Read the next line.
End while
Step4: Close the output file.
```

3.2.2 Decompression algorithm

```
Input : Compressed file.
Output : Original file.
Function Decompression (Compressed file)
Step1: An ArrayList is made of all possible input characters.
Step2: A HashMap table map string to character is made dynamically at the run time of compression.
Step3: Read the content of file.
While string <> null do
    If (count mod 2 <> 0) then //Initially count=1
        A HashMap table, hMap, map integer to string value.
        A List intList of integer initially set to null.
        Add block corresponding to 10, 13, 129, 141, 143, 144 and 157 to an ArrayList strList.
        For String str : strList do
            intList := findIndexes(string, str)
            For int index : intList  do
                hMap.put (index, str)
            End for
        End for
        A Set keys of integer store the key value of hMap.
        An ArrayList L of integer addAll the keys.
        A Collection sort the value in L.
        For int midIndex : L do
            For startIndex to midIndex in step 1 do //startIndex = 0
                Read c character from string.
                Write the corresponding string value by mapping to file.
            End for
            Write string value to file of midIndex by mapping.
            StartIndex := midIndex+4;
        End for
        For startIndex to string.length() in step 1 do //Read remaining character
            Read c character from string.
            Write the corresponding string value by mapping to file.
        End for
        count := count+1;
    Else {
        Write the same characters to file.
        count := count+1;
        Write newline character.
    }
```

```
Read the next line.
End while
Step4: Close the output file.
```

IV. Results

We have used three Homo sapiens mRNA: IL4-transcript variant 1, IL4-transcript variant 2 and MOPT, five funguses Rhizopus oryzae glucoamylase A precursor, gene, partial cds: strain 7, strain 15, strain 30, strain 45 and strain 52 that lives worldwide in dead organic matter and four Zea mays mRNA: ZM_BFb0129K09, ZM_BFb0203I02, ZM_BFc0025I23 and ZM_BFc0038A03 known corn or mielie/mealie, is a grain.

Table 3: Comparison of Compress Sequence Length (bytes) for different techniques

Sequence Name	Original Seq. Length	Deflate + IMPLODE +LWZ	LZSS + PPM	Block3 + Tandem repeats	Block4
IL4, transcript variant 1	660	402	358	234	225
IL4, transcript variant 2	610	379	333	214	208
MOPT	712	377	355	254	241
strain 7	591	341	314	209	192
strain 15	589	344	316	210	199
strain 30	445	294	264	156	150
strain 45	594	341	311	210	201
strain 52	542	334	301	192	180
ZM_BFb0129K09	682	385	353	244	231
ZM_BFb0203I02	787	425	391	282	266
ZM_BFc0025I23	811	417	390	290	262
ZM_BFc0038A03	765	412	381	266	262

Table 4: Comparison of Compression Factor for different techniques

Sequence Name	Deflate + IMPLODE +LWZ	LZSS+ PPM	Block3 + Tandem repeats	Block4
IL4, transcript variant 1	1.642	1.843	2.821	2.933
IL4, transcript variant 2	1.609	1.832	2.851	2.933
MOPT	1.889	2.005	2.803	2.954
strain 7	1.733	1.882	2.827	3.078
strain 15	1.712	1.864	2.804	2.960
strain 30	1.514	1.686	2.853	2.967
strain 45	1.742	1.909	2.829	2.955
strain 52	1.623	1.801	2.823	3.011
ZM_BFb0129K09	1.771	1.932	2.795	2.952
ZM_BFb0203I02	1.852	2.013	2.791	2.959
ZM_BFc0025I23	1.945	2.079	2.797	3.095
ZM_BFc0038A03	1.857	2.008	2.876	2.921

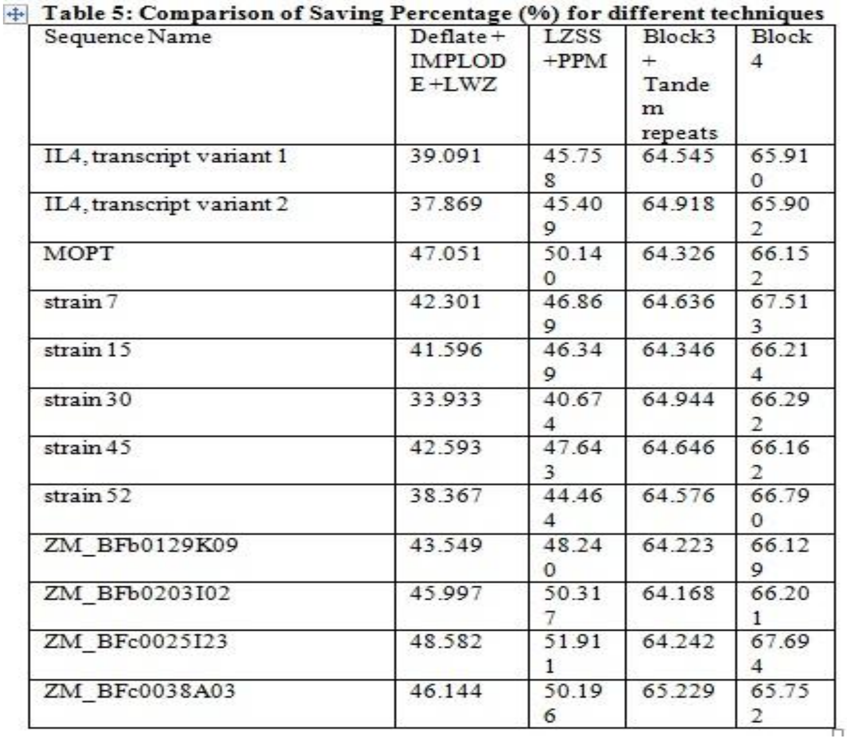


Table 5: Comparison of Saving Percentage (%) for different techniques

Sequence Name	Deflate+ IMPLODE+LWZ	LZSS +PPM	Block3 + Tandem repeats	Block 4
IL4, transcript variant 1	39.091	45.758	64.545	65.910
IL4, transcript variant 2	37.869	45.409	64.918	65.902
MOPT	47.051	50.140	64.326	66.152
strain 7	42.301	46.869	64.636	67.513
strain 15	41.596	46.349	64.346	66.214
strain 30	33.933	40.674	64.944	66.292
strain 45	42.593	47.643	64.646	66.162
strain 52	38.367	44.464	64.576	66.790
ZM_BFb0129K09	43.549	48.240	64.223	66.129
ZM_BFb0203I02	45.997	50.317	64.168	66.201
ZM_BFc0025I23	48.582	51.911	64.242	67.694
ZM_BFc0038A03	46.144	50.196	65.229	65.752

The graphical representation of table 3 is shown in fig-1

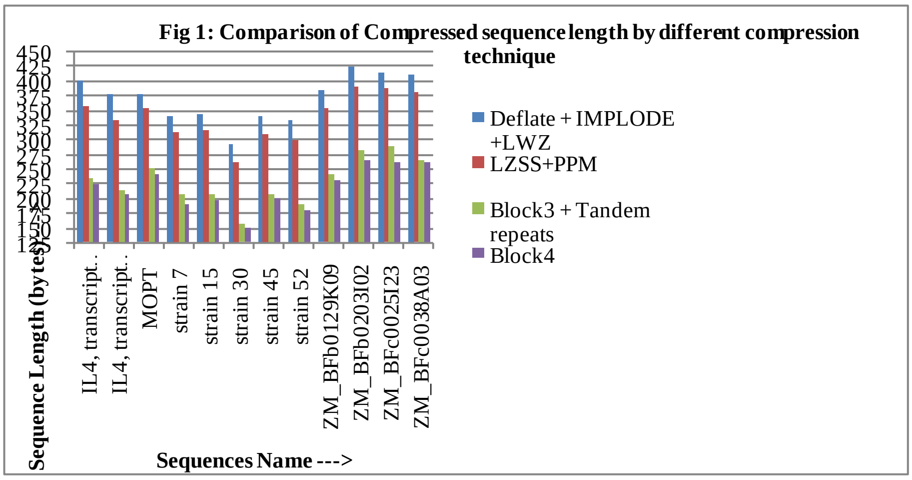


Fig 1: Comparison of Compressed sequence length by different compression technique

The graphical representation of table 4 is shown in fig-2

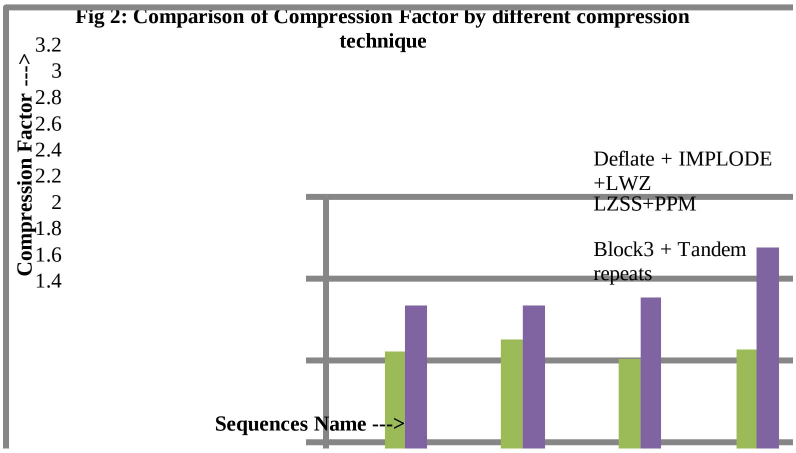


Fig 2: Comparison of Compression Factor by different compression technique

The graphical representation of table 5 is shown in fig-3

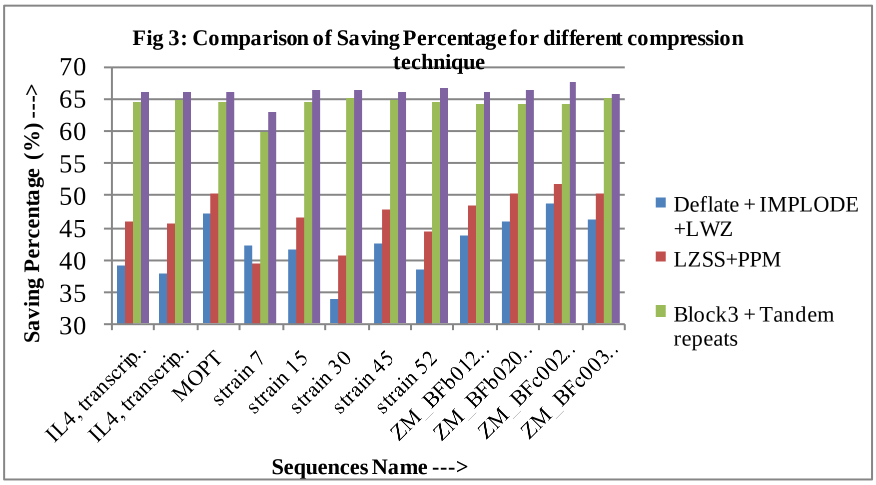


Fig 3: Comparison of Saving Percentage for different compression technique

V. Conclusion and Discussion

In this article, we have presented two new compression algorithms that are based on LUT of block size three with tandem repeats and LUT of block size four.

There are various most common compression algorithms like LZSS, WinZIP use deflate, deflate64, IMPLODE, dynamic LWZ , WinRAR use LZSS and PPM etc but they are not giving good result for the compression of biological sequences as the result shown above that our propose compression algorithms gives better result.

The propose algorithms have high compression factor and giving good saving percentage to other exiting Biological Sequence Compression. This technique also requires less memory and less coding effort compare to the other algorithms.

The proposed techniques compress Biological sequences taking block size of three, four and using tandem repeat in the sequence line. So it will be easier to make sequence alignment and sequence analysis between compressed sequences.

References

[1] Genbank size from: ftp://ftp.ncbi.nih.gov/genbank/gbrel.txt

[2] Hyoung Do Kim and Ju-Han Kim , “DNA Data Compression Based on the Whole Genome Sequence”, Journal of Convergence Information Technology Vol. 4, No. 3, September 2009.

[3] Pothuraju Rajarajeswari, Allam Apparao, “DNABIT Compress – Genome compression algorithm” Biomedical Informatics, volume 5, Issue 8, pp. 350-360, 2011.

[4] P.Raja Rajeswari and Dr. Allam AppaRao, “GENBIT COMPRESS TOOL (GBC): A java-based tool to compress DNA sequences and compute compression ratio (bits/base) of genomes”, IJCSIT, Vol. 2, No. 3, pp. 181-191, June 2010.

[5] P.Raja Rajeswari, Dr. Allam AppaRao and Dr. R. Kiran Kumar, “HUFFBIT COMPRESS – Algorithm to compress DNA sequences using extended binary trees”, JTAIT, pp. 101-106, 2010.

[6] Heba Afify, Muhammad Islam and Manal Abdel Wahed, “DNA lossless differential compression algorithm based on similarity of genomic sequence database”, IJCSIT, Vol. 3, No 4, August 2011.

[7] R.K.Bharti, “A Biological sequence compression Based on Approximate repeat Using Variable length LUT”, International Journal of Advances in Science and Technology, Vol. 3, No.3,PP:71-75, 2011.

[8] R.K. Bharti, Prof. R.K. Singh, “A Biological Sequence Compression based on Look up Table (LUT) using Complementary Palindrome of Fixed Size”, ICJA (0975–8887), Volume 35– No.11, December 2011.

[9] R.K.Bharti, “Biological sequence Compression Based on Cross chromosomal properties using variable length LUT”, CSC Journal, Vol. 4 Issue 6, PP: 217-223. , 2011.

[10] Sequences are taken from : http://www.ncbi.nlm.nih.gov/.

[11] Suman Chakraborty, Sudipta Roy, Prof. Samir K. Bandyopadhyay, “Image Steganography Using DNA Sequence and Sudoku Solution Matrix”, International Journal of Advanced Research in Computer Science and Software Engineering(IJARCSSE), Volume 2, Issue 2, February 2012.