SN Computer Science (2022) 3:314
https://doi.org/10.1007/s42979-022-01210-0

ORIGINAL RESEARCH

An Efficient Compression Scheme for Natural Language Text by Hashing

Md. Ashiq Mahmood[1] · **K. M. Azharul Hasan**[1]

Received: 22 June 2020 / Accepted: 15 May 2022 / Published online: 4 June 2022

Abstract

Data compression means the route towards adjusting, encoding or changing the bit structure of information so that it requires less space. The fundamental standard behind compression is to build up a strategy or convention for utilizing less bits to express the actual data. Character encoding is fairly identified with compression of data that illustrate a character by a kind of encoding system. We proposes an efficient and simple compression algorithm for large natural text named *n-Sequence* based *m* Bit Compression (*nSmBC*) which can beat WinZip and WinRAR in terms of compression ratio. WinZip and WinRAR are two well-known compression techniques used for text compression in the industry. The scheme provides an efficient encoding algorithm that converts an 8 bit character by 5 bits utilizing a look up table. The look up table is produced by utilizing Zipf distribution that represents a discrete dispersion of ordinarily utilized characters in various languages. 8 bit characters are converted to 5 bits by partitioning the characters into 7 sets. After converting the characters into 5 bit, an *n-sequence* scheme is developed to logically calculate the location number of a particular combination of characters. The reverse algorithm to recover the actual input is further demonstrated. The *nSmBC* is finally compared with the well-known WinZip, WinRAR, Huffman and LZW techniques. Promising performance is demonstrated both by theoretical and experimental analysis.

Keywords Data compression · Encoding · Decompression · n-sequence dictionary · Look up table · Zipf distribution

Introduction

Data compression is a procedure of changing a data stream of one form into another that has fewer size than the original [1]. The stream can be a document, a bit stream, or individual bits sent to a channel. The primary targets of data compression are to lessen the size of data and increase the exchange rate. Data compression covers a huge area of applications including data communication, data storage and database technology. Text compression is a field of data compression, which utilizes the lossless compression method to change over information to another type of document that is able to decrease the room required to store the data. In this way, the most clear favorable position of data compression is that of lessening the capacity prerequisite to store the information. Lessening the capacity prerequisite is equal to expanding the limit of the capacity medium. Since compacted text are encoded utilizing fewer bits, moving of packed data starting with one spot then onto the next requires less time and subsequently brings about a higher successful exchange rate. Since the pressure decreases the stacking of I/O channels, it gets practical to process more I/O demands every second and thus accomplish higher and viable channel use. One of the most important application of data compression is the reducing the cost of data communication in distributed networks [2]. There are some systems that have been proposed for text compression in the literature. The majority of which depends on a similar standard of expelling or lessening redundancies from the intended text record. The repetition can show up at character, syllable, or word levels. This standard proposed a component for text compression by relegating short codes to regular parts, that is, characters, syllables, words, or sentences, and long codes to uncommon parts. Lately, a few strategies have been created for text compression. These strategies can be further classified

✉ Md. Ashiq Mahmood
ashiqmahmoodbipu@gmail.com

K. M. Azharul Hasan
azhasan@gmail.com

[1] Khulna University of Engineering and Technology, Khulna 9203, Bangladesh

into four types of techniques namely, substitution, measurable, lexicon, and context-based technique. The substitution text compression strategies replace a specific longer reiteration of characters with a shorter one. A strategy that is a representative of this method is run-length encoding [3].

The factual strategies more often than not figure the likelihood of characters to create the briefest normal code length, for example, Huffman coding [4], and arithmetic coding [5]. The lexicon techniques include substitution of a substring of text by a file or a pointer code. They identify with a situation in the dictionary of the substring. Agents of these strategies are LZW [6], LZ77 [7], and LZ78 [8]. The last sort is context-based methods, which include the utilization of insignificant earlier suspicions about the measurements of the text. Ordinarily, they utilize the context of the text being encoded and the historical back drop of the text to give progressively proficient compression. Representatives of this sort are Prediction by Partial Matching (PPM) [9] and Burrow Wheeler change (BWT) [10]. Compression methods can likewise be partitioned into two essential classifications to be specific lossless and lossy. In lossless compression, the decompressed information is a precise imitation of the first information. Despite what might be expected, in lossy compression, the decompressed information might be not quite the same as the first information. Commonly, there is some contortion between the first and recreated information in lossy compression [11].

In this paper, we propose a dictionary based lossless compression technique that converts the 8 bit character to 5 bits using a look up table. The look up table is produced with the general printable characters which are used in English and Bangla text. The lookup table is constructed using the Zipf distribution [12, 13]. After converting the character into 5 bits we propose an idea of *n-sequence* to create a dictionary. The dictionary is implemented by a hash function to logically calculate the value of dictionary entry. Therefore, the dictionary is a logical implementation and does not take any physical storage. Compression and decompression algorithms have been proposed using the converted the 5 bit stream and the *n-sequence* characters. We call our proposed scheme as *n-sequence* based *m* Bit Compression (nSmBC). We compared our scheme with Huffman [4] and LZW [6] and found better performance to them. We also compared the proposed scheme with WinRAR and WinZip that also shows promising efficiency. The nSmBC technique can be able to compress any text by more than 92% of the actual text. The proposed scheme can not only be applied to text compression but also to text mining [14], text encoding [15] and database compression [16]. The rest of the paper is organized as follows. The next section presents some related works, the following section explains the proposed text compression scheme, the next section describes the analytical evaluation of the scheme, the next section depicts the experimental result and finally last section outlines some conclusion.

Related Works

Most of the text compression systems [17–21] are based on a dictionary of word, or character levels. An n-gram based dictionary approach is presented in [17] for compressing Vietnamese text. They used window size of bigram to five grams dictionary to encode a string. The compression ratio is good but the scheme will fail to work if the input string is not a traditional word of the language. In [18], the authors proposed a technique to convert the characters in the original file to a binary code. In this method, the most common characters have the shortest binary codes whereas the least common characters have the longest binary codes. The binary codes are originated based on the estimated probability of the character within the file and using 8-bit character word length. In [19], the authors proposed a technique that combined word with LZW algorithm. The method divides the input text to word and non-word and after that uses them as initial alphabet of LZW. But dividing word and non-word is a costly operation. [20] proposed a method to compress shorter text on the basis on two stages. At the first stage, it alters the source input including letters, numbers, spaces, and punctuation marks used in English language. And in the 2nd stage, it introduces a transformation that reduces the length of the text by a fixed fraction of the length of its input. In [21], the authors introduced a word-based compression on the basis on the LZ77 algorithm and proposed and implemented different ways of sliding windows and various possibilities of output encoding. Some other techniques of text compression are also appeared based on syllables such as The Burrows–Wheeler transform. These techniques include few languages which have morphology in the organization of words or morphemes (e.g., German, Arabic, Turkish, and Czech) such as in [22–25]. In [22] the authors introduced a lossless text compression algorithm that uses syllable-based morphology of multisyllabic languages. The proposed technique is implemented by partitioning words into its syllables and then producing their little bit organizations for compression. A genetic algorithm based technique is introduced in [23]. This technique was utilized by determining the characteristics

of syllables. Then it stores the characteristics into a dictionary, that happens in the compression process and it is not needed placing the characteristics into compressed data. This phenomena has led to the compression of the space has been used. Lansky and his colleagues [24, 25] proposed a technique for syllable-based text compression techniques. They emphasizes on specification of syllables, methods for decomposition of words into syllables, and using syllable-based compression in combination of the principles of LZW and Huffman coding.

A 6 bit representation of printable characters is presented in [26]. It converts the 8 bit characters to 6 bits by partitioning the characters into 5 sets and utilizing them in a look up table. This method has a compression ratio that converges to around 50% and it is suitable for small text files. Another system by utilizing the algorithm of [26] for database operation is proposed in [27].

Most of the techniques mentioned above uses a dictionary and applies Run length encoding or Huffman encoding or LZW technique. In this paper, we introduced a new idea namely *n-sequence* based *m* bit compression Scheme. The basic difference is that we have implemented the dictionary in logical fashion and it does take any physical space. The scheme shows superior performance than existing systems

Compression Scheme for Natural Language Text

Definition 1 (*n-sequence*) *n-sequence* is a sequence of characters taken from a character set with n characters which are arranged together in a sequence. It is possible that all combinations take *n* characters from a given character set. For example, if a character set contains the characters {A, B} then 1-sequence is <A, B>, 2-sequence is <AA, BB, AB, BA>, and 3-sequence is <AAA, BBB, AAB, ABA,…>. Each of the members in the *n-sequence* is identified by an index which is a number. If there are *k* members in an *n-sequence* set, the index numbers are from 0 to $(k-1)$. For example, the index of "BB" in the 2-sequence is 1.

Definition 2 (Set tag) *Set tag* is defined as a group of characters tagged to a particular set number. For example, the capital letters, small letters, digits and symbols are tagged to particular set number as shown in Table 1. We call it *set tag* representation.

Look Up Table Construction

We construct the look up table by dividing the characters into 7 set tags namely Set-1, Set-2… Set-7. Each of the set tag contains 25 characters. The characters are placed in a lookup table as shown in Table 1. The entry in Table 1 is organized as follows:

1. Characters of the Bangla alphabet are placed in Set-1, Set2 and Set-3. Set 3 also contains the Bangla digits as well. The positions 21–24 in Set-3 are blank and can be used in future.
2. Characters of the English alphabet are placed in Set-4, Set-5, Set-6 and Set-7. The English digits and special characters are placed in Set-6 and Set-7 respectively. Position 19–24 of Set-7 is empty and can be filled with any missing characters.
3. The rest of the 7 combinations are filled with the 7 *set tags* as shown in Table 1.

Therefore, the Table 1 contains 32 characters (0 to 31). These 32 (2^5 =32) combinations can be represented by 5 bits. Within the 32 combinations 25 combinations are utilized for converting the original 8 bit character to 5 bit and the rest of the 7 combinations are utilized for representation of the *set tags*. Therefore, we can use $(2^5-7)\times 7$ =175 characters in Table 1. If we can take 6 bits then there can be $(2^6-7)\times 7$ =399 characters can be handled. We call it *m* bit representation scheme. In the following we represent $m=5$ bits to explain our proposed method. We represent any character using the encoding scheme (see Table 1). We call it *set tag* representation. For example, if we have a character stream "ABCDabcd956" then the *set tag representation* is "Set4ABCDSet5abcdSet6956". Since "A" is located in Set4 we start with Set4 followed by "A", "a" is represented in Set5 we put Set5 before "a" and so on. When a set change occurs, we insert a Set number to distinguish it with others. In these stage we can say this representation as a variants of Run Length Encoding.

The placement of characters in the look up table is optimized using Zipf distribution which is a discrete dispersion of ordinarily utilized characters in various dialects [12]. Zipf's law is an empirical law formulated using mathematical statistics. It states that given a large sample of words used, the frequency of any word is inversely proportional to its rank in the frequency table. So word number n has a frequency proportional to 1/n. Thus the most frequent word will occur about twice as often as the second most frequent word, three times as often as the third most frequent word [13]. The objective of using Zipf's distribution is to place the characters in Table 1 such that the minimum number of set change occurs to handle the input string.

n-Sequence Dictionary Construction

After converting the 8 bit characters into 5 bits, we have a bit stream of 5 bits of each character. For any input text T, we create a bit stream (5 bits for each character) and from this bit stream we divide it by 4 to take 4 bits each. We put trailing the last set number to make it mod 4 equal to zero if the length of the bit stream is not mod 4 equal to zero. From this 4 bits, we have $2^4 = 16$ different combinations of bits. Since each of the characters is represented by 8 bits, we add

Table 1 Lookup table for *set tag*

Decimal value	Binary value	Set-1	Set-2	Set-3	Set-4	Set-5	Set-6	Set-7
0	00000	অ	ণ		E	e	1	Q
1	00001	আ	ত		T	t	2	X
2	00010	ই	থ		A	a	3	Z
3	00011	এ	দ	ঃ	O	o	4	J
4	00100	া	ধ	ৎ	R	r	5	,
5	00101	ি	ঘ	ঢ়	I	i	6	?
6	00110	ী	প	ঐ	N	n	7	‘
7	00111		ফ	ঊ	S	s	8	!
8	01000	ে	য়	ও	H	h	9	"
9	01001	ো	ভ	ঈ	D	d	0	#
10	01010	ৌ	ছ	।	L	l	+	\
11	01011	ক	ট	০	C	c	-	~
12	01100	খ	ঞ	১	U	u	*	^
13	01101	গ	ঠ	২	P	p	/	|
14	01110	ন	ঙ	৩	M	m	=	$
15	01111	শ	ষ	৪	W	w	(	:
16	10000	স	চ	৫	F	f	)	;
17	10001	ম	য	৬	G	g	{	_
18	10010	জ	ড়	৭	Y	y	}	`
19	10011	ব	ঝ	৮	B	b	<	
20	10100	র	উ	৯	V	v	>	
21	10101	ট	ও		K	k	[	
22	10110	ল	ঋ		z	x	]	
23	10111	ড			j	.	%	
24	11000	হ	ং		q	space	&	
25	11001	Set-1						
26	11010	Set-2						
27	11011	Set-3						
28	11100	Set-4						
29	11101	Set-5						
30	11110	Set-6						
31	11111	Set-7						

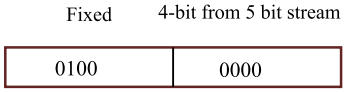


Fig. 1 Adding fixed bit pattern

a fixed bit pattern in front of each of the 4 bits. Figure 1 shows an example. The fixed bit pattern is 0100.

After adding fixed 4 bit pattern (0100) we get ASCII value range 64–79. Table 2 shows the characters along with its decimal and ASCII values. Hence any of the characters shown in Table 1 becomes a character shown in Table 2.

Example 1

Original Text: “*Test Text*”

Set Representation:

Set4 T Set5 est space Set4 T Set5 ext

Decimal Representation:

28 1 29 0 7 1 24 28 1 29 0 22 1

5 bit representation:

11100 00001 11101 00000 00111 00001 11000 11100 00001 11101 00000 10110 00001

After Dividing by 4:

1110 0000 0111 1010 0000 0011 1000 0111 0001 1100 0000 1111 0100 0001 0110 0000 1111

Adding 0100 to every combination:

01001110 01000000 01000111 01001010 01000000 01000011 01001000 01000111 01000001 01001100 01000000 01001111 01000100 01000001 01000110 01000000 01001111

Corresponding

ASCII Character: N@GJ@CHGAL@ODAF@O

Dictionary Construction

Using the characters of Table 2, we generate a dictionary of *n-sequence* (See Definition 1) of different values of n. Figure 2 shows the *n-sequence* for $n = 1$, 2 and 3.

Key Generation

From the *n-sequence* dictionary we generate a key of the form $< n, (v_1, v_2, v_3, v_4, \ldots) >$ where n is the number of the *n-sequence* and v is the index value of the corresponding *n-sequence* dictionary (see Fig. 2). We store the $(v_1, v_2, v_3, v_4, \ldots)$ the secondary storage and store the n in main memory.

Logical Dictionary

The dictionary we generate using *n-sequence* generation is not stored in the physical memory. We implement the dictionary using a hash function *h(s)*. The function *h()* takes a string *s* which is member of the *n-sequence* dictionary as input and returns the corresponding index of the dictionary. Hence the dictionary becomes a logical one and does not take any physical memory.

Table 2 Converted list of characters

Serial no.	Characters	Decimal value	Binary value
1	@	64	01000000
2	A	65	01000001
3	B	66	01000010
4	C	67	01000011
5	D	68	01000100
6	E	69	01000101
7	F	70	00100110
8	G	71	01000111
9	H	72	01001000
10	I	73	01001001
11	J	74	01001010
12	K	75	01001011
13	L	76	01001100
14	M	77	01001101
15	N	78	01001110
16	O	79	01001111

Hash Function Development

Forward Hash Function Firstly, we assign all the 16 characters (see Table 2) a value as follows $V_0 = V_7 = @$, $V_1 = V_8 = A$, $V_2 = V_9 = B$, $V_3 = V_{10} = C$, $V_4 = V_{11} = D$, $V_5 = V_{12} = E$, $V_6 = V_{13} = F$, $V_{14} = N$, $V_{15} = O$.

where $V_1 = 1, V_2 = 2, \ldots, V_i = i\ (1 \le i \le 15)$.

The index value for *n-sequence* dictionary for different values of n is calculated as follows.

For $n=1$,

$h(s)=V_i + 1$

If $s=$"A" then $h(\varepsilon A\varepsilon) = V_1+1=1+1=2$ [where $i=1$].

For $n=2$,

$h(s) = (V_i * 16) + V_j + 1$

If $s =$ "AM" then $h(\text{AM}) = (V_1 * 16) + V_j + 1 = (1*16)+13+1=30$ [where $i=1$ and $j=13$].

For $n=3$,

$h(s) = (((V_i * 16) + V_j)*16) + V_k + 1$.

If $s=$"BAG" then $h(\varepsilon BAG\varepsilon) = (((V_i * 16) + V_j) *16) + V_k + 1 = (((2*16)+1) * 16)+7+1=536$ [where $i=2$, $j=1$ and $k=7$].

Finally we generalize *h(s)* as

$h(s)=(((V_i * 16)+V_j) * 16+V_k) * 16+V_l)+\ldots+1$

where $h(s)$ a string which is a member of *n-sequence* dictionary, V_i assigned number of the 1st character, V_j assigned number of the 2nd character, V_k assigned number of the 3rd character, V_l assigned number of the 4th character, and so on.

After getting the indexes using the above hash function we represent each index with 1 byte using the Java *OutputStreamWriter()* function in Java platform which is used to convert the written characters to the bytes written to the underlying *OutputStream*. Here we convert the written index to ASCII which defines 1 byte.

Reverse hash function For $n=1$,

$h=V_i + 1$

$Y = V_1\ [h\text{-}1=Y]$.

$V_1 = Y \bmod 16$.

For $n=2$,

$h = (V_i * 16) + V_j + 1$

$Y = (V_1 * 16) + V_2 + 1\ [h\text{-}1=Y]$.

$V_2 = Y \bmod 16$.

$V_1 = Y/16$.

For $n=3$,

$h = (((V_i * 16) + V_j)*16) + V_k + 1$.

$Y = ((V_1 * 16) + V_2)*16) + V_3 + 1\ [h\text{-}1=Y]$

$V_3 = Y \bmod 16$

$V_1=[Y/16]/16$

$V_2=[Y/16] \bmod 16$

Hence the general equations becomes

$V_n = Y \bmod 16$

$V_1=[[[Y/16]/16]/16 \ldots]/16$

$V_{i[2<i<n-1>} = [[[Y/16]/16]/16 \ldots] \bmod 16$

We use the idea of *n-Sequence* for *m* bit representation hence call the scheme *nSmBC* (*n-Sequence* based *m* bit Compression).

Compression and Decompression Algorithm

In this section, the compression and decompression algorithms for *nSmBC* is briefly described in different steps. After the compression and decompression technique, an example is provided to show the working procedure of the algorithms.

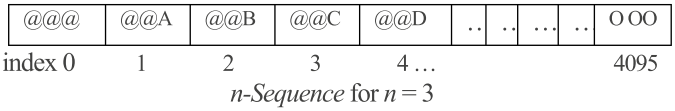
Forward Mapping

Input**:** A string S to be compressed,

Output**:** An encoded compressed string S_c.

Step 1: Represent S to S′ as *set tag representation* adding Set tag.

Step 2: Using the look up table, convert the string S' by 5 bit stream. Let, in this stage the bit stream contains k bits.



n-Sequence for $n=3$

Fig. 2 *n-Sequence* for $n=1$, 2 and 3.

Step 3: d = k%4; if (d ≠ 0) add trailing 0 bits to the last set number to make d = 0.
Step 4: Store every 4 bit combinations of k.
Step 5: Add 0100 in front of to every 4 bit combination of k to make the binary combination only limited to the characters Table 2.
Step 6: Divide k by 8 to find the corresponding ASCII characters.
Step 7: Create the logical *n-sequence* dictionary using forward hash function *h()* and store < *n, index* >

Example 2
Original Text (Input): "Test Text "

Step 1: Set Representation: Set4 T Set5 est space Set4 T Set5 ext
Step 2: Decimal Representation: 28 1 29 0 7 1 24 28 1 29 0 22 1
Step 3: 5 bit representation: 11100 00001 11101 00000
Step 4: 00111 00001 11000 11100 00001 11101 00000 10110 00001
Step 5: After Dividing by 4: 1110 0000 0111 1010 0000 0011 1000 0111 0001 1100 0000 1111 0100 0001 0110 0000 1111
Step 6: Using Adding 0010 to every combination: 01001110 01000000 01000111 01001010 01000000 01000011 01001000 01000111 01000001 01001100 01000000 01001111 01000100 01000001 01000110 01000000 01001111
Step 7: ASCII Representation:N@GJ@CHGAL@ODAF@O
Step 8: Generate *n-Sequence* to get the < *n, index* > (n = 4 used here): < 4, (57467, 904, 7184, 16737, 63422) >

Backward Mapping

Input: Compressed String, S_c.
Output: Uncompressed original string, S

Step 1: Representing the string Sc by its corresponding < n, index > pair using reverse hash function.
Step 2: From the location of the pair < *n, index* > find the exact *n-sequence* character combination and store it in S_c'.
Step 3: From S_c', find its corresponding binary combination from the ACII Table (Table 2) and store the resultant binary bits in k.
Step 4: Remove 0100 from every 8 bit binary combinations.
Step 5: From the remaining bits stream, take 5 bits and representing it by the character set of the look up table (Table 1).
Step 6: Remove the set number to get the original string S.

Example 3:
Compressed String: < 4, (57467, 904, 7184, 16737, 63422) >

Step 1: Corresponding string in *n-sequence* dictionary: N@GJ@CHGAL@ODAF@O
Step 2: From 8 bit Representation: 01001110 01000000 01000111 01001010 01000000 01000011 01001000 01000111 01000001 01001100 01000000 01001111 01000100 01000001 01000110 01000000 01001111
Step 3: Removing 0100 from every 8 bit combination: 1110 0000 0111 1010 0000 0011 1000 0111 0001 1100 0000 1111 0100 0001 0110 0000 1111
Step 4: Fro5 bit representation: 11100 00001 11101 00000 00111 00001 11000 11100 00001 11101 00000 10110 00001
Step 5: Decimal Number corresponding to 5 bits: 28 1 29 0 7 1 24 28 1 29 0 22 1
Step 6: Corresponding Set Representation: Set4 T Set5 est space Set4 T Set5 ext
Step 7: Original Text: Test Text

The proposed nSmBC compression algorithm is designed for all the natural characters that are found in a standard keyboard. These characters include English characters, special characters and Bangla natural characters. Hence we believe we have handled the special characters as well as natural English and Bangla characters. Moreover the algorithm can easily be extended for other characters as the last column of lookup table (Table 1) contains blank cell that can be used for other characters. The look up table can also be extended to accommodate other characters (if necessary) for compressing other characters using the nSmBC compression algorithm.

Analytical Evaluation

In this section, the analytical evaluation of the proposed scheme is done. Table 3 shows the parameters for analytical evaluation. Some parameters are provided as input while others are derived from the input parameters. All lengths and sizes are in bits.

Therefore,

$$\eta = \frac{q*\alpha}{N*8} = \frac{v*q*\alpha}{n*N*8} = \frac{S_2*\alpha}{\beta*n*N*8} = \frac{N*m+\varepsilon*\alpha}{\beta*n*N*8},$$

Table 3 Parameters for analytical evaluation

Parameter	Description
N	Total number of characters in the input string
S_1	Size of the input string, $S_1 = N \times 8$ bit
m	Number of bits used to compress the input character using lookup table (Table 1)
ε	Number of bits required to store *Sets* for *set tag* representation
β	Number of bits used to create the converted characters of Table 2
S_2	Size of input string using m bit representation, $S_2 = N \times m + \varepsilon$ bit
v	Number of characters generated from S_2 by taking β bits, $v = \frac{S_2}{\beta}$
q	Number of indices to store, $q = \frac{v}{n}$
α	Size of one index
S_3	Size of q, $S_3 = q \times \alpha$(Compressed file size)
η	Compressed ratio, $\eta = \frac{S_3}{S_1}$
σ	Savings of space $\sigma = (1- \eta) \times 100\%$

[We assume the size of the *Set tags* are negligible $\varepsilon \approx 0$].

$= \frac{N*m*\alpha}{\beta*n*N*8} = \frac{m*\alpha}{\beta*n*8} = \frac{m*8}{\beta*n*8}$, [$\alpha = 1$ byte $= 8$ bit].

$\eta = \frac{m}{\beta*n}$

Using the above equation, we evaluated the trend of η with varying values of n (6 to 15). Figure 3 shows the analytical result.

The performance of the proposed *nSmBC* scheme depends on the Value of n i.e. if the length of *n-sequence* is large the performance will be better. The performance also depends on the value of m and β. If m is large then performance will degrade but if m is very small then the number of characters that can be accommodated in the lookup table will be small. Therefore, moderate value of m is necessary. If the value of β increases, then performance will also be improved but when β increase then the number of characters also increase in Table 2. Hence number characters to generate the *n-sequence* will be large.

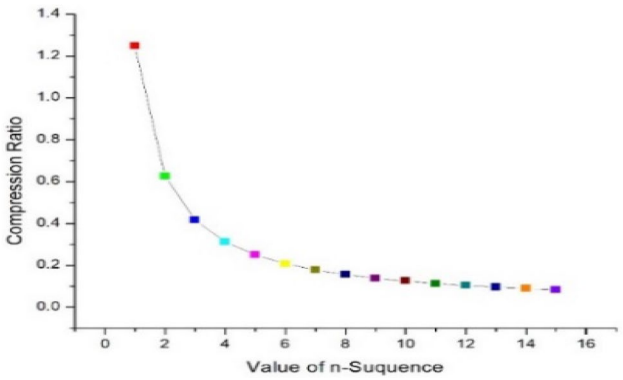


Fig. 3 Analytical evaluation for compression ratio for different n-Sequence value

Table 4 Parameters for experimental evaluation

n	*S1* (MB)	m	β	α
6–15	0.5,1.05,2.0,3.07,5.03	5	4	1

Experimental Results

We have implemented a prototype system with our proposed algorithm in Java NetBeans IDE 8.2 with the parameter values shown in Table 4. In this Section we present the experimental results. In our experiment, we used the data set collected from Microsoft Research (MSR) Abstractive Text Compression Dataset. The dataset is available at [28]. The MSR dataset contains substantial amount of text data collected from diverse source and genre including business letters, newswire, journals, and Non-fiction academic publications, such as PLoS Medicine, open access journal from the Open American National Corpus. The MSR text maintains a strong correlation with the human judgments of meaning. Therefore, we believe, the data set becomes a good natural text. Since our main concern was to compress natural text, we used MSR natural dataset in our experimental evaluation. The details of the dataset can be found in [29]. Table 5 shows the description of the dataset.

Figure 4a shows the experimental results for compression ratio with varying values of n for *nSmBC*. It demonstrates that when the value on n increases the value of η decreases. For $n = 15$, η reduces to 0.08 which means, σ is 92% as shown in Fig. 4b. When n increases, η reduces because, η

depends mainly on n, m and β. For increasing the value of n, more characters can be increased to include to a single index. Hence the η will reduce and the σ will increase as shown in Fig. 4. This is what we shown in our analytical evaluation in Sect. 5 (see Fig. 3). Hence we validate our analytical model.

We compare our proposed technique with well-known Huffman technique [4] and LZW [6]. The comparison for compressed file size with LZW and Huffman is shown in Fig. 5a. The result for *nSmBC* is shown for $n=6$, 8, 14 and 15. The *nSmBC* outperforms Huffman technique for $n=6$, 8, 14 and 15. LZW shows good results but the *nSmBC* scheme outperforms LZW for $n=14$ and 15. For all the cases Huffman shows worst result.

We also compare our technique with two industrial systems WinZip and WinRAR. The WinZip and WinRAR compress all character set along with images. The scope of this paper is to compress natural text. The image compression is another research issue and, therefore, image compression was out of the scope of this paper. The MSR dataset used in the experimental evaluation is a pure natural text. And the same data set was used to evaluate the compression schemes namely proposed nSmBC, WinZip and WinRAR.

Figure 5b shows the comparison with WinZip and WinRAR for compressed file size. The *nSmBC* performs well than WinZip and WinRAR for $n=14$ and 15. The WinZip performs well for small S_1, when S_1 increases the *nSmBC* performs well even for $n=8$.

The comparison for η with LZW and Huffman is shown in Fig. 6a. The result for *nSmBC* is shown for $n=6$, 8, 14 and 15. The Huffman shows poor result among the schemes. The reason behind Huffman technique to be poor is that the data is derived by Huffman from the frequency of occurrence of the possible values in the source symbol [4]. So if the size of data is quite large then a large number of individual symbols will be created. As a result, it shows poor η comparing to others. η ranges from 0.35 to 0.4 leading to $\sigma=60$–65%. The result demonstrates that *nSmBC* provides the best

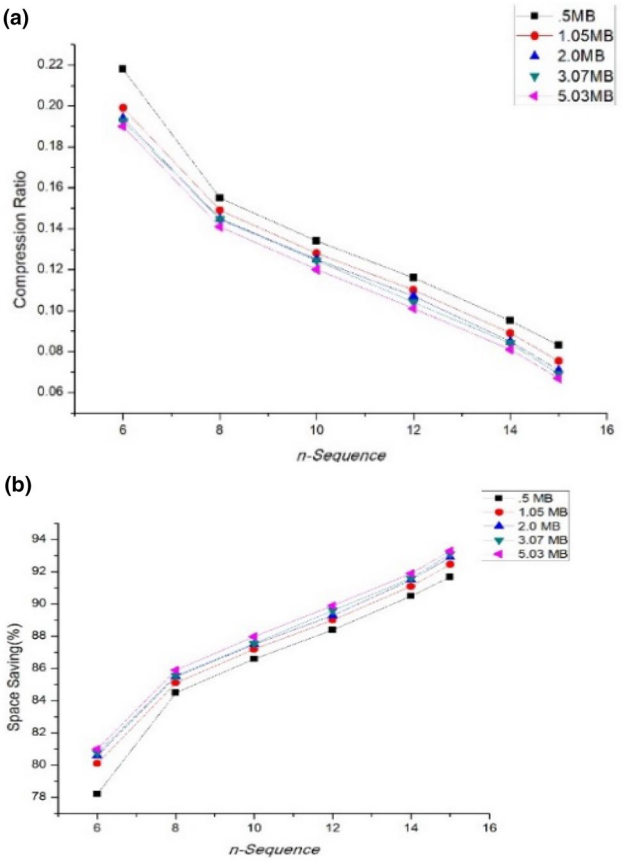


Fig. 4 **a** Compression ratio for different *n-Sequence* value. **b** Savings of space for different *n-Sequence* values

compression ratio. It reaches to $\eta=0.08$ ($\sigma=92\%$) for $n=15$. The other values n also provides good performance.

Figure 6b shows the comparison for $\eta$with WinZip and WinRAR. The WinRAR shows $\eta=0.10$ ($\sigma=90\%$) at initial level. But at the increasing S_1, η degrades to 0.15 ($\sigma=85\%$). In case of WinZip, it also shows same type of values for η as WinRAR at the initial stage but not as good as WinRAR. At the increasing S_1, the compression ratio fluctuates in between 0.16 to 2.0 ($\sigma=80$–85%). The *nSmBC* shows better performance and it outperforms WinZip and WinRAR for $n=14$ and 15. Finally, we conclude that the *nSmBC* outperforms other techniques.

Figure 7 provides the compression time of different algorithms. We use Java *currentTimeMillis()* function to calculate the time of the *nSmBC*, LZW and Huffman method. From Fig. 7, it shows that *nSmBC* provides better result than Huffman. Initially LZW provides bad

Table 5 Dataset description

Sl. No.	Description
Dataset 1	Size: 0.5 MB, no. of characters: 514,055
Dataset 2	Size: 1.05 MB, no. of characters: 1,102,371
Dataset 3	Size: 2.0 MB, no. of characters: 2,103,453
Dataset 4	Size: 3.07 MB, no. of characters: 3,228,053
Dataset 5	Size: 5.03 MB, no. of characters: 5,283,107

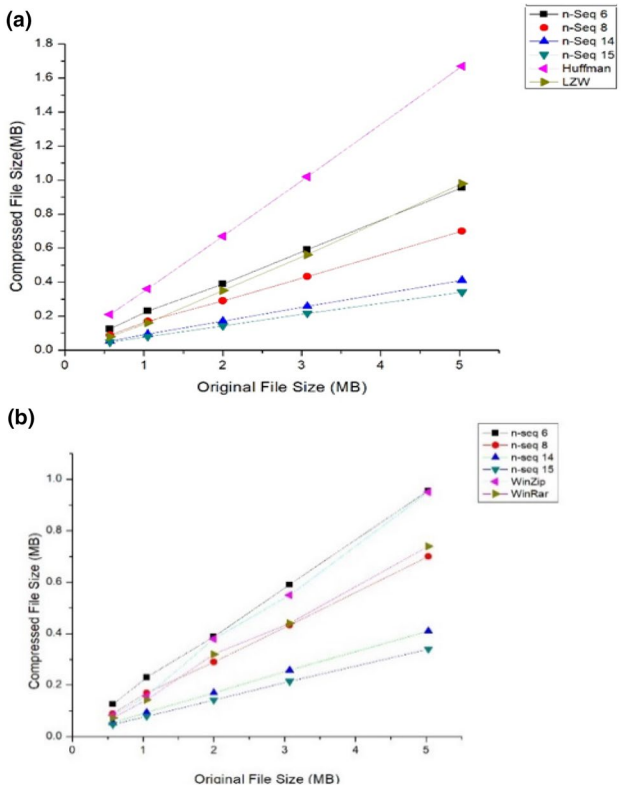


Fig. 5 **a** Comparison of compressed file size for LZW and Huffman. **b** Comparison of compressed file size for WinZip and WinRAR

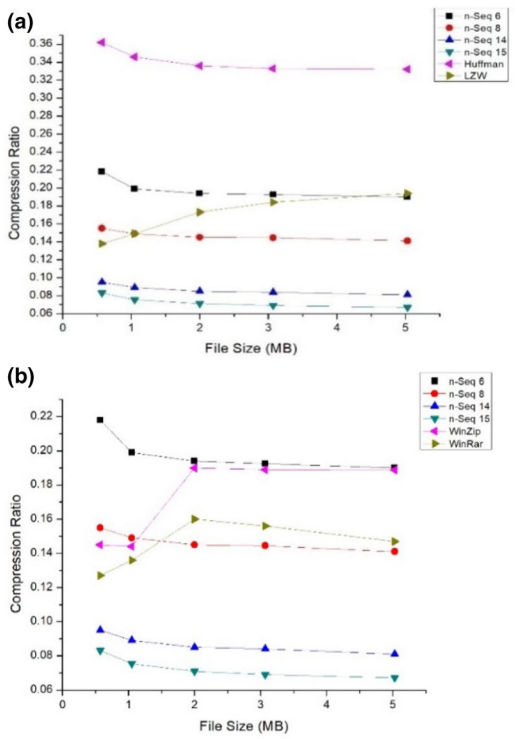


Fig. 6 **a** Comparison of compression ratio for LZW and Huffman technique. **b** Comparison of compression ratio for WinZip and WinRAR

result but when file size is increasing, it will show the similar range of time as *nSmBC*. But WinZip and WinRAR provides best result that means these two algorithms need a very short run time. We think this is the only shortcoming of our algorithm in respect to WinZip and WinRAR. But this time issue can be resolved by utilizing this algorithm in a high configuration computers.

Conclusion

In this paper, we present a novel method for text compression. The paper proposes the idea on *n-sequence* and construction of logical dictionary. The large dictionary is implemented of a hash function. The proposed *nSmBC* takes 5 bits for each character using a lookup table. Analytical and experimental results are presented to show the superiority of the scheme. The scheme IS able to compress up to 92% for web-based diverse data set. The scheme shows superior performance to existing schemes namely LZW, Huffman and also for WinZip and WinRAR. The technique can easily be utilized to compress large amount of natural language text. Both the forward and backward mapping algorithms are presented. This technique can also be utilized to parallel processing environment as well as load balancing technique to achieve promising encoding

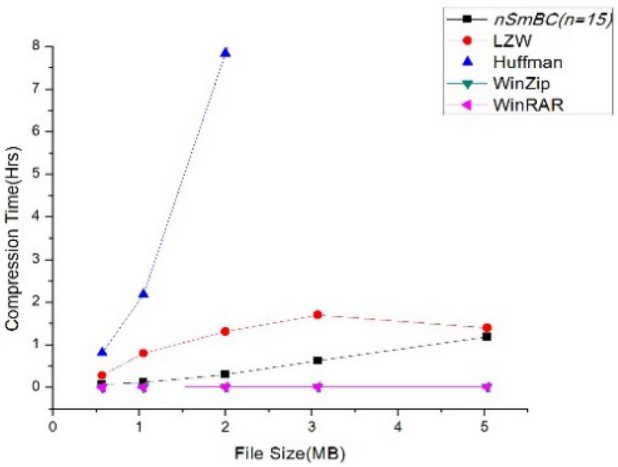


Fig. 7 Comparison of compression time for LZW, Huffman, WinZip and WinRAR

time. We believe, the *nSmBC* is an efficient algorithm for compression that has the potential to compete with the existing text compression techniques.

Funding None.

Declarations

Conflict of interest The authors declare that they have no conflict of interest.

Ethical approval All procedures performed in studies involving human participants were in accordance with the ethical standards of the institutional and/or national research committee and with the 1964 Helsinki Declaration and its later amendments or comparable ethical standards.

Informed consent Informed consent was obtained from all individual participants included in the study.

References

1. NguyenVH, Nguyen HT, Duong HN, Snasel V. Trigram-based Vietnamese text compression. In: Recent developments in intelligent information and database systems, studies in computational intelligence, vol 642. Springer; 2016. p. 297–307.
2. Bassiouni MA. Data compression in scientific and statistical databases. IEEE Trans Softw Eng. 1985;11(10):1047–57.
3. Žalik B, Lukač N. An chain code lossless compression using move-to-front transform and adaptive run-length encoding. Signal Process Image Commun. 2014;29(1):96–106.
4. Wu J, Wang Y, Ding L, Liao X. Improving performance of network covert timing channel through Huffman coding. Math Comput Model. 2012;25(1–2):69–79.
5. Witten IH, Neal RM, Cleary JG. Arithmetic coding for data compression. Commun ACM. 1987;30(6):520–40.
6. Welch TA. Technique for high-performance data compression. IEEE Comput. 1984;17(6):8–19.
7. Travis GagieJ, Gawrychowski P, Kärkkäinen J, Nekrich Y, Puglisi SJ (2014) LZ77-based self-indexing with faster pattern matching. In: Pardo A, Viola A, editors. LATIN 2014, LNCS 8392. Berlin: Springer; 2014. p. 731–742.
8. Bannai H, Inenaga S, Takeda M. Efficient LZ78 factorization of grammar compressed text. In: Caldron-Benavides L et al, editors. SPIRE 2012, LNCS 7608. Berlin: Springer; 2012. p. 86—98.
9. Cleary J, Witten I. Data compression using adaptive coding and partial string matching. IEEE Trans Commun. 1984;32(4):396–402.
10. BurrowsM, Wheeler D. A block-sorting lossless data compression algorithm. Digital SRC Research Report. 1994.
11. Azharul HasanKM. Compression schemes of high dimensional data for MOLAP. In: Furtado P, editor. Evolving application domains of data warehousing and mining: trends and solutions, University of Coimbra, Portugal. Chapter IV. 2010. p. 64–81.
12. Wentian L. Random texts exhibit WinZipfs-law-like word. IEEE Trans Inf Theory 1992;38(6).
13. Fagan S, Gençay R. An introduction to textual econometrics. In: Handbook of empirical economics and finance. 2010. p. 133–153.
14. Aggarwal CC, Zhai CX. A survey of text clustering algorithms. In: Recent developments in database management & information retrieval, chapter 4 of mining text data. Springer; 2012. p. 1–123.
15. Taeho J. Text encoding. In: Recent studies in big data, vol 45, sec 3.1 of text mining. Springer; 2018. p. 41–58.
16. Satir E, Isik H. A compression-based text steganography method. J Syst Softw. 2012;85(10):2385–94.
17. Nguyen VH, Nguyen HT, Duong HN, Snasel V. n-gram-based text compression. Comput Intell Neurosci. 2016;2016:1–11.
18. Al-Bahadili H, Hussain SM. An adaptive character word length algorithm for data compression. Comput Math Appl. 2008;55(6):1250–6.
19. Dvorsk J, Pokorn J, Sna´sel J. Word-based compression methods and indexing for text retrieval systems. In: Proceedings of the 3rd East European conference on advances in databases and information systems (ADBIS '99), Maribor, Slovenia. 1999. p. 75–84.
20. Kalajdzic K, Ali SH, Patel A. Rapid lossless compression of short text messages. Comput Stand Interfaces. 2015;37:53–9.
21. Platos J, Dvorskþ J. Word-based text compression. 2008. http://arxiv.org/abs/0804.3680.
22. Akman I, Bayindir H, Ozleme S, Akin Z, Misra S. A lossless text compression technique using syllable based morphology. Int Arab J Inf Technol. 2011;8(1):66–74.
23. Kuthan T, Lansky J. Genetic algorithms in syllable-based text compression. In: Proceedings of the Dateso annual international workshop on databases, texts, specifications and objects, Desna, Czech Republic, 2007. p. 21–34.
24. Lansky, Zemlicka M. Text compression: syllables. In: Proceedings of the Dateso annual international workshop on databases, texts, specifications and objects, Desna, Czech Republic, April 2005. p. 32–45.
25. LanskyJ, Zemlicka M. Compression of small text files using syllables. In: Proceedings of the data compression conference, Snowbird. 2006.
26. Mahmood A, Latif T, Azharul Hasan KM. An efficient 6 bit encoding scheme for printable characters by table look up. In: International conference on electrical, computer and communication engineering (ECCE). 2017. p. 468–472.
27. MahmoodMA, Latif T, Azharul Hasan KM, Islam R. A feasible 6 bit text database compression scheme with character encoding (6BC). In: 2018 21st international conference of computer and information technology (ICCIT). 2018. p. 1–6.
28. https://www.microsoft.com/enus/download/details.aspx?id=54262. 2020.
29. Toutanova C, Brockett C, Tran KM, Amershi S. A dataset and evaluation metrics for abstractive compression of sentences and short paragraph. In: Empirical methods in natural language processing, EMNLP. 2016. p. 340–350.