Pertanika J. Sci. & Technol. 28 (4): 1427 - 1444 (2020)

PERTANIKA JOURNALS

SCIENCE & TECHNOLOGY

Journal homepage: http://www.pertanika.upm.edu.my/

A Compression System for Unicode Files Using an Enhanced Lzw Method

Rincy Thayyalakkal Anto[1]* and Rajesh Ramachandran[2]

[1]Department of Computer Science, Prajyoti Niketan College, Pudukad, Thrissur, Kerala, 680301 India
[2] Department of Computer Science, CHRIST (Deemed to be University), Hosur Road, Banglore, 560029 India

ABSTRACT

Data compression plays a vital and pivotal role in the process of computing as it helps in space reduction occupied by a file as well as to reduce the time taken to access the file. This work relates to a method for compressing and decompressing a UTF-8 encoded stream of data pertaining to Lempel-Ziv-welch (LZW) method. It is worth to use an exclusive-purpose LZW compression scheme as many applications are utilizing Unicode text. The system of the present work comprises a compression module, configured to compress the Unicode data by creating the dictionary entries in Unicode format. This is accomplished with adaptive characteristic data compression tables built upon the data to be compressed reflecting the characteristics of the most recent input data. The decompression module is configured to decompress the compressed file with the help of unique Unicode character table obtained from the compression module and the encoded output. We can have remarkable gain in compression, wherein the knowledge that we gather from the source is used to explore the decompression process.

Keywords: Compression algorithms, dictionary-based data compression, LZW, unicode encoding, UTF-8

ARTICLE INFO

Article history:
Received: 17 April 2020
Accepted: 27 July 2020
Published: 21 October 2020

DOI: https://doi.org/10.47836/pjst.28.4.16

E-mail addresses:
rincyanto@gmail.com (Rincy Thayyalakkal Anto)
ryanrajesh@hotmail.com (Rajesh Ramachandran)
*Corresponding author

INTRODUCTION

Since from the distant past, every human society has undoubtedly tried to compress data which is very evident as attempts are made to make devices for briefing as well as for abridgment of relayed messages. The limited transmission bandwidth and necessity of quick and prompt transmission

ISSN: 0128-7680
e-ISSN: 2231-8526

in the digital world makes data compression an imperative and indispensable method. The methodology of minimizing the potential size of a data block with the exception of diminution the quality of information that it is carrying, can be termed as Data Compression. Compression is helpful because it helps in reducing the data storage such as hard disk space and increases its feasibility to be transmitted in the allowed transmission bandwidth. We can tenably state that devoid of fruitful compressions many of the sumptuous and contemporary contrivances, formulations as well as progressions, will be a mirage for all the populations.

Shannon (1948), actualized a formal intellectual discipline for compression, even though there existed compression of data in a very informal manner. Numerous methods are available for data compression. According to the flexibility of methods, compression methods are broadly classified into different types as follows: By pruning and sacrificing some information, lossy compression makes it feasible to attain a better compression rate. Certain files which will become defective and futile if even a bit gets altered, must be compressed merely by lossless compression method. The dissimilarity among lossless and lossy codecs can be evidently featured by taking a cascade of compressions. Perceptive compression methods take advantage of the data being compressed by perceiving our psycho-acoustic and psycho-visual perception. In Symmetric compression, essentially the same algorithm is made to work in "opposite" directions both in the cases of compressor and decompressor. A non-adaptive compression technique which is very inflexible and does not recast its parameters, process, or table in reciprocation to the distinct data selected for compression.

Compression ratio, compression factor and compression gain. are the criteria's commonly used to express the efficiency and performance of a compression method. For byte-based compressors, the common test data used for compression algorithms and implementations are Calgary Corpus and Canterbury Corpus.

Welch (1984) proposed Lempel-Ziv-Welch (LZW) algorithm that belonged to the category of dictionary compressors. If the size of the file is immense, the dictionary based LZW compressor works as an entropy encoder. Always it is desirable to select an adaptive method which is dictionary-based. While static dictionary is a good choice for a special-purpose compressor, an adaptive dictionary-based method is always favoured for a general-purpose compressor. The ideas of Ziv and Lempel (1977; 1978), led to the development of many common LZ compression methods. In LZW, which itself is a leading derivative of LZ78, the token comprises only a pointer to the dictionary. Before inputting any data, the LZW compression commences by initializing the dictionary with all the symbols of the alphabet in the first 256 entries. Hereafter, the succeeding character can be visualized in the dictionary, as it is initialized. Many personages could come up with divergent application and derivatives of this method, which is an evidence of intimidation created by publication of LZW algorithm in 1984.

The most important derivatives of LZW are LZMW (LZ Miller and Wegman), LZAP (LZ All Prefixes), LZY (LZ Yabba) and LZP (LZ Predictor). As the string in the dictionary expand only one character lengthier at each instance, LZW, which is an adaptive method, lags to adjust to its inputs. The LZMW method survives the problems as the existing string is summated to the whole adjacent phrase to the dictionary, rather than adding character one by one. In LZAP method, rather than directly concatenating the last two phrases and positioning the result in the dictionary, it keeps all of the prefixes of the concatenation in the dictionary itself. LZP is a LZ77 variant and is based on the principle of context prediction.

Now-a-days, it is feasible to display and print characters of any shape and size. As a result, 128 characters ASCII (American Standard Code for Information Interchange) code now stands superannuated for modern computing. The needs of present hardware and software are very well met with Unicode encodings. The world's most supreme Unicode character encoding UTF-8, presently hosts the greater part of online contents penned in Non-English dialects. The prevailing compression algorithms routinely act on discrete bytes. Even though, this mode of approach finishes poorly in case of UTF-8, as characters generally span multiple bytes, it operates well for the single-byte ASCII encodings. Repertoire of Unicode 12.1, which spans 150 novel and monumental scripts, have its justified ground with a solid base consisting 137,994 characters in total, as well as multiple symbol sets and emoji. This firm base owes for the dominance of Unicode at unifying character sets, have become the reason of its extensive and tremendous use in the global platforms as well as for the computer software localization.

Even though, various compression techniques are in service for English as well as for other languages, the grammatical structure and syntactic structure differs from one dialect to another. This fact necessitates the need of a peculiar compression technique for natural languages. A small corpus of Unicode files has been compressed on several widely available text compressors of the various types by Fenwick and Brierley (1998), confirming that Unicode files have different compression characteristics from those known for 8-bit data. The Malayalam text compression by variable length encoding was explained by Divakaran et al. (2013), after an informational analysis of Malayalam Language. Barua et al. (2017) projected an enhanced LZW compression technique for Bangla dialect considering the unique features of that language. Gleave et al. (2017) represented modified techniques with escaping on LZW and PPM (Prediction with Partial Matching). An abridged bit representation in the dictionary is an indicative for each Unicode character. A novel approach that suggests modifications to cater the characteristic of Gujarati text to achieve better compression was discussed by Maniya et al. (2012). Vijayalakshmi and Sasirekha (2018) used a static dictionary compression technique in which Unicode characters were replaced with ASCII characters for compression and the original file was retained in the decompression process. Marjan et al. (2014) had proposed a novel approach in which Bengali text was represented efficiently with a better compression ratio.

Yamagiwa et al. (2019) proposed a work known as LCA-DLT (Lowest Common Ancestor- Dynamic invalidation and Lazy compression Technique) and it focused on a technique to reduce the number of searches in the dictionary using a bank separation technique. The aptitude of wavelet transform to be multilingual lossy text compression was discussed and a new strategy that is based on the application of wavelet transform to text compression of files was proposed by Al-Dubaee et al. (2010). A novel technique of compressing a more symbolic dialect like Bengali through a less symbolic language like English was proposed by Hossain et al. (2014) in which Huffman principle was used to its maximum.

The established algorithms such as bzip2, usually dominates over newly developed Unicode compressors from scratch, as per research findings. Here, we are trying to introduce a technique to modify byte-based compressors to percolate directly on Unicode characters and to execute a variant of LZW to accomplish better compression.

In this article, we apply an interactive approach to data compression and discuss its prominent benefits as well as significant advantages. To execute this, we can presume some degree of interaction between the compressor and the decompressor. It can allow a more competent usage of the information gained during compression of the original file, whilst the data compression is used for data transmission purposes.

The systematic organization of the paper is presented as follows: In section II, LZW Compression technique is explained for both ASCII text and Unicode Text. In section III, we propose our new method for Unicode text as a variation of existing LZW Compression method. We are discussing the results of our innovative method in section IV. The future research directions are concluded in Section V.

LZW COMPRESSION

LZW is characterized by the fact that its conditions are fulfilled only by one pass over the given data. A code table is created wherein each entry is made up of peculiar strings of characters. As detailed by the algorithm in Figure 1, the principle of LZW derived by Welch (1984) is that the encoder puts in symbols one after another, to get them accumulated in a string α. The dictionary is searched for string α, after receiving and concatenating each discrete symbol to α. The process continues as long as α is discovered in the dictionary. At a certain point, adding the immediate successive symbol x causes a failure of search characterized by the presence of string α and absence of string αx in the dictionary. At this juncture, the encoder

a. Output the dictionary pointer that points to string α
b. Saves string αx, in the next available dictionary entry, and
c. Initializes string α to symbol x.

Algorithm 1 The Existing LZW Encoder

```
1: for i←0 to 255 do
2:    append i as 1-symbol string to the dictionary;
3: end for
4: α← empty string
5: x ← first input character
6: while not end of stream do
7:     if α+x is in the dictionary then
8:         α← α+x
9:     else
10:        output code for α
11:    end if
12:    append α+x to the dictionary
13:    α ← x
14:    x ← next input character
15: end while
16: output code for α
```

Figure 1. Algorithm for the existing LZW Encoder

The decoder starts to make its operations with the first 256 entries of its dictionary initialized to the whole symbols of the alphabet. Later, it reads the input stream which includes pointers to the dictionary and utilizes each pointer to retrieve uncompressed symbols from its dictionary and writes them on its output stream. The algorithm for the existing LZW decoder is provided as Figure 2. In this process, it constructs its dictionary in lockstep with the encoder.

Algorithm 2 The Existing LZW Decoder

```
1:  for i← 0 to 255 do
2:     append i as 1-symbol string to the dictionary
3:  end for
4:  read first input code
5:  output translation of code, α
6:  w ← α
7:  while not end of input do
8:     x ←read next input code
9:     β← translation of x
10:    output β
11:    append w+β[0] to dictionary
12:    w←β
13:  end while
```

Figure 2. Algorithm for the existing LZW Decoder

When we encode the string “agh eats aghagha” with LZW, the dictionary entries obtained are shown in Figure 3.

0	NULL	256	ag	264	␣a
1	SOH	257	gh	265	agh
.....		258	h␣	266	ha
32	SP	259	␣e	267	agha
.....		260	ea		
97	a	261	at		
.....		262	ts		
255	255	263	s␣		

Figure 3. LZW dictionary entries for the string “agh eats aghagha”.

But, if we compress a string including Unicode characters, then it will also do byte-wise compression. Figure 4 shows the dictionary entries for the Malayalam (A Dravidian language spoken over the Indian territory of Kerala) string 'കേന്ദ്രതാഴില് മന്ത്രാലയം'.

0	NULL	260	´?	272	à´°	284	²à	296	°à
1	SOH	261	?à	273	°	285	àµ?	297	à´¾
.....		262	àµ	274	à	286	à´	298	¾à
32	SP	263	µ?	275	à´¤	287	´®	299	à´²
.....		264	?à	276	¤à	288	®à	300	²à´
97	a	265	à´¨	277	àµ?	289	à´¨à	301	´¯
.....		266	¨à	278	?à	290	àµ?à´	302	¯à
255	255	267	àµ?	279	à´´	291	´¤	303	à´?
256	ï»	268	?à	280	´à	292	¤àµ		
257	»¿	269	à´	281	à´¿	293	µ?		
258	¿à	270	¦à	282	¿à´	294	?à´		
259	à´	271	àµ?à	283	´²	295	´°		

Figure 4. LZW dictionary entries for the string 'കേന്ദ്രതാഴില് മന്ത്രാലയം'.

A code table (dictionary), with 4096 as a usual choice for the number of table entries, is utilized in LZW compression. At the commencement of encoding, the code table comprises of only the initial 256 entries of single byte characters, with the rest of the table being kept blank. Codes 256 through 4095 are used to indicate the sequential bytes for achieving the compression. While the encoding advances, LZW determines frequented sequences in the data and adds them to the code table. For the above text, it builds the dictionary with 303 entries, even though the text contains only 25 Unicode characters. This prompts to making an attempt to recommend a technique to alter byte-based compressors to permeate

precisely on Unicode characters and to actualize a versatile variant of LZW to attain better compression.

METHODOLOGY

An optimised method and system for compressing and decompressing a UTF-8 encoded stream of data using an enhanced LZW algorithm was made available. In this method of compression, the initial part is to read data from the input stream present in the form of UTF-8 characters. The only known work which reads data in the form of UTF-8 characters belongs to Gleave et al. (2017). In their work, they had investigated the effectiveness of different token distributions while being used as a base distribution for LZW. They had proposed the algorithms of LZW and PPM with three base models. The initial two models characterised by uniform base distribution over the byte alphabet as well that of token alphabet, and another one with Polya tree-based model over the alphabet. They could find that adaptive character model based on Polya trees was well suited for learning Unicode character distributions. In this method, UTF-8 text is decoded into a sequence of Unicode code points. Beginning with an empty initial dictionary, the method is implemented by escaping to a base model over tokens, immediately when a new symbol is seen. This implementation places no upper limit on the dictionary size N and hence, encodes each index in $\log_2 N$ bits using arithmetic coding. As the mapping used by them had the appealing property of allocating semantically related tokens to nearby integers, they claimed that their approach yielded a 12.2% average improvement in compression effectiveness for LZW over a corpus of UTF-8 files. We propose a work without converting UTF-8 text to Unicode code points and with no initial identification of token distributions.

In our work, we propose a different version of the traditional LZW encoding algorithm in which no characters are preloaded in the dictionary. The newly derived algorithm for the encoder is shown in Figure 5. As the dictionary is empty, the occurrence of each new character creates a new entry in the dictionary. As the process of encoding goes on, it further decides whether to output a code associated with the data, in case if the data is already in the dictionary or to add the character to the dictionary as a new character, in case if it is not found in the dictionary. The system identifies duplicated sequences in the data as well as the new characters which are unavailable in the dictionary and adds them to the dictionary. All the new characters along with its position in the dictionary are stored in a separate table and it is passed as part of the compressed data to the decompressor. Even though this may be considered as an overhead, the price we accept to pay is negligible compared to the benefits we achieve. In the existing LZW method, 4096 is used as the dictionary size and 12 bits are used to output the code. When the dictionary has filled up, existing LZW becomes static. Further it compresses with unchanging dictionary and may lead to degradation in compression performance. So in our proposed system, when the dictionary

had been filled up to a specified maximum, we cleared the dictionary after checking the boundary conditions and then reloaded the dictionary with all new characters found so far along with a certain number of most frequently used character sequences from the previous dictionary. Rather than using 12 bits for every outputted code, we used a variable-width coding system in which each code was represented exactly as their binary representation.

LZW methods convert a sequence of strings (<string (1)> . . . <string (n)>) to a sequence of codes (<code (1)> . . . <code (n)>). The proposed strategy and methodology lessen the number of entries in the dictionary and hence caters the need of better compression in a file. It can boost the activities of many LZW-based applications.

Algorithm 3 The Proposed LZW Encoder

```
1:  α ← empty string
2:  x ← first input character
3:  while not end of stream do
4:     if x is not in the dictionary then
5:        append x to dictionary
6:        add x to a string table 'dict_add' with its dictionary index position
7:     end if
8:     s← α+x
9:     if s is in the dictionary then
10:       α ← s
11:    else
12:       output code for α
13:       append s to the dictionary
14:       α←x
15:    end if
16:    x← next input character
17: end while
18: output code for α
19: pass the string table 'dict_add' to the decoder
```

Figure 5. Algorithm for the proposed LZW Encoder

The same string in Malayalam 'കുന്നത്തൊഴില് മന്ത്രാലയം'. was tested with the proposed compression and the number of dictionary entries was reduced to 37. Malayalam which is one among 22 scheduled Indian languages is a one with Dravidian origin spoken in the state of Kerala. With the largest number of alphabets accounting to 56, Malayalam tops the other Indian languages in this aspect. The dictionary entries created are shown in Figure 6.

Decompressor built the dictionary in the same pattern as that of compressor with the help of the table that it received as part of the compressed file. The decompressor reads the input stream containing the pointers to the dictionary and used each pointer to recover original symbols from its dictionary, if found present in the dictionary and wrote them on its output stream. On the other hand, if it was not present in the dictionary, the system extracted the next character from the table that it received from the compressor to create a

0	ക	10	ര	20	ല	30	ത്
1	◌േ	11	◌്ര	21	◌ില	31	◌ാ
2	കേ	12	ത	22	ല്	32	◌്രാ
3	ന	13	രത	23		33	◌ാല
4	◌േന	14	◌ൊ	24	◌്	34	യ
5	◌്	15	തൊ	25		35	ലയ
6	ന്	16	ഴ	26	മ	36	◌ം
7	ദ	17	◌ൊഴ	27	മ	37	യം
8	◌്ദ	18	◌ി	28	മന		
9	ദ്	19	ഴി	29	ന്ത		

Figure 6. The proposed LZW dictionary entries for the string 'കേന്ദ്രതൊഴില് മന്ത്രാലയം'.

new dictionary entry, if the position extracted from the table matched with the dictionary index. The number of dictionary entries and the search time are considerably reduced due to the proposed system and it contributes directly to faster compression and decompression. Thus, we have made an attempt to recommend a technique to alter byte-based compressors to permeate precisely on Unicode characters and to actualize a versatile variant of LZW to attain better compression.

In the proposed LZW decoding method, based on algorithm detailed in Figure 7, the decoder reads the first character α from the string table 'dict_add' as the first decoding step and append it to dictionary. For every following steps of decoding after the initial one, the decoder inputs the next pointer, fetches the immediate string β from the dictionary if it is present, writes it on the output, followed by isolation of its initial symbol, and saves string w+ initial character of β in the next obtainable entry in the dictionary. If not present in the dictionary, it extracts the next character from the table 'dict-add' and adds it to the dictionary if the position matches with the dictionary index. String β is the extracted character in this case, and it is noted down on the output. The decoder then proceeds w to β and is all set for further step.

Some of the existing LZW compression or decompression methods use 8 to 16 bits to represent the dictionary entries. Accordingly, the number of entries in the LZW dictionary stands limited to 2^number of bits. The application program will now be in a position either to fortify the dictionary once it is full, or to cast off a few entries in the dictionary once it becomes filled. Henceforth, we can undoubtedly say that the capacity of LZW dictionary is obtained from the size of code (i.e., compression code) used in a particular implementation of the LZW variant (algorithm). The system and method of the present disclosure provided

Algorithm 4 The Proposed LZW Decoder

```
1: read first character α from the string table 'dict_add'
2: append it to the dictionary
3: remove the first input code
4: output α
5: w ← α
6: while not end of input do
7:     x ← read next input code
8:     if dictionary contains translation of x then
9:         β ← translation of x
10:    else if the position of next character in 'dict_add' = current dictionary
                index then
11:        extract the next character y from the string table
12:        append y to the dictionary
13:        β ← y
14:    end if
15: output β
16: append w+β[0] to dictionary
17: w ← β
18: end while
```

Figure 7. Algorithm for the proposed LZW Decoder

for enhanced LZW compression or decompression utilizes the method of clearing the dictionary once it is full and reloading the dictionary with all the new characters found so far. Implementation of this algorithm with dictionary update makes use of a certain number of most frequently used entries from the previous dictionary.

The most frequently used entries in the dictionary were found by using an auxiliary dictionary. Whenever the code corresponding to a character sequence in the dictionary was added to the output of an encoder, the count corresponding to the character sequence was incremented in the auxiliary dictionary. Once the dictionary reaches its specified maximum, the dictionary was cleared, and all the new characters found so far were added to the dictionary. Thereafter, the auxiliary dictionary was sorted and a specified number of most recently used entries are added to the dictionary. This helps the compression process very much as there is no need to start all from scratch.

We have tested this algorithm both with 32 KB and 64 KB dictionaries. We were using auxiliary dictionary of the same size, with dictionary update variant of this algorithm. The results obtained and its comparison with two popular methods Unix compress and gzip are detailed below.

RESULTS AND DISCUSSION

The modus operandi of the present invention in one actualization executes an enhanced LZW procedure to create dictionary entries in a dynamic manner and acquires codes from those entries at the stage of compression. It retrieves the code and reinstates the original stream.

After testing the proposed system with Malayalam String, we had also done it with many Unicode files in languages like Tamil, Hindi, Bengali and Arabic. Some of the sample files, number of dictionary entries (with less than 4096 entries) and execution time are given in Table 1. From Table 1, we could see that number of dictionary entries was considerably reduced in the proposed LZW compression scheme and the time spent for compression and decompression was much lesser as the search time was reduced.

Table 1

Comparison between The Existing LZW method and The Proposed LZW method

File Name	No. of dictionary entries		Time spent (in milliseconds)			
			ELZW[1]		PLZW[2]	
	ELZW[1]	PLZW[2]	Comp[3]	Decomp[4]	Comp.[1]	Decomp.[2]
malayalam.txt	3411	2433	112	21	23	7
tamil.txt	1334	823	71	17	17	4
hindi.txt	1896	1277	85	21	21	5
Bengali.txt	4096	3873	103	28	25	8
Arabic.txt	3267	2581	117	20	40	10

[1] Existing LZW [2]Proposed LZW [3]Compression [4]Decompression

According to Uthayakumar et al. (2018), the performance of data compression algorithms can be evaluated in multiple dimensions. The competence of compression can be estimated using parameters like algorithm complexity, computational memory, speed, amount of compression and quality of reconstructed data. Even though the performance of data compression can be evaluated using multiple parameters, most frequently it is done by Compression Ratio. It is defined as the ratio of total number of bits required to store uncompressed data and total number of bits required to store compressed data and is given in Equation 1.

$$CR = \frac{\text{No: of bits in uncompressed data}}{\text{No: of bits in compressed data}} \quad [1]$$

Another measure called Space savings is also used, which defines the reduction in file size relative to the uncompressed size and is given in Equation 2.

$$\text{Space Savings} = 1 - \frac{\text{No : of bits in compressed data}}{\text{No : of bits in uncompressed data}} \quad [2]$$

Table 2

Compression Ratio – The Existing LZW method and The Proposed LZW method

File Name	Original File Size (KB)	Compressed File Size		Additional overhead	Total File Size	Compression Ratio		Space Savings (%)	
		ELZW[1] (KB)	PLZW[2] (KB)			ELZW[1]	PLZW[2]	ELZW[1]	PLZW[2]
malayalam.txt	14.7	4.62	3.46	0.37	3.83	3.18	3.83	68.57	73.94
tamil.txt	3.89	1.58	1.13	0.26	1.39	2.46	2.8	59.38	64.26
hindi.txt	6.16	2.39	1.77	0.36	2.13	2.58	2.9	61.2	65.58
Bengali .txt	23.1	7.28	5.55	0.46	6.01	3.17	3.85	68.48	74.02
arabic.txt	10.2	4.41	3.68	0.28	3.96	2.31	2.57	56.76	61.17

[1] Existing LZW [2]Proposed LZW

Table 2 shows the improved compression performance for the same files in terms of Compression Ratio and Space Savings. It describes the total compressed file size after adding the additional overhead caused by the proposed algorithm. By additional overhead, we mean the space required to store the various new characters detected and its position during the compression process. As the dictionary is initialized as empty with this algorithm, it is passed as part of the compressed file. Compression performance is substantially improved as per the data in Table 2.

From the above tables, it is clear that the Unicode character-based compressor based on an enhanced LZW scheme outweighs the traditional byte based LZW compressors in performance. The number of dictionary entries is considerably reduced. Both the sender and the receiver are able to construct a code table either synchronously or in lockstep. Though the code table may occupy more space with the Unicode characters than that with ASCII characters, but it is likely to be compensated considerably as the number of entries is less in the code table.

After the analysis of the proposed algorithm with some small files, we made a thorough analysis of it with some large files from Canterbury Corpus and from the public domain. This analysis was done with 32 KB and 64 KB dictionaries. The first one is the proposed LZW method wherein the dictionary size used is 32 KB. When the dictionary size reaches its specified maximum, it clears the dictionary and only the new characters found so far is used to initialize the dictionary. In the second method, we were updating the dictionary as in the previous method and in addition, a certain number of most recently used dictionary entries from the previous dictionary were also appended. Auxiliary dictionaries of the same size were used with this algorithm. Unix Compress which is a popular method that use LZW algorithm was also tested with 32KB and 64KB dictionaries. Another LZ variant compression tool gzip was also used for the comparison.

The details of test data files are given in Table 3 and comparison between the compression ratios of different LZ implementations are detailed in Table 4.

Table 3
Test Corpus

File	Size in KB	Description
ben-kobita.txt	23.1	Bengali Novel
hin-baital.txt	420	Hindi
mal-travelblog.txt	209	Malayalam
Indulekha.txt	290	Malayalam Novel
Mal-Bible.txt	1453	Malayalam Bible
ben-blog.txt	1104	Bengali blog

Table 3 *(Continued)*

File	Size in KB	Description
Bhagavat Geetha.txt	1068	Malayalam
Kundalatha.txt	174	Malayalam Novel
Umakeralam.txt	347	Malayalam Poetry
alice29.txt	676	Alice in wonderland, English
Asyoulik.txt	148	As you like it, Early modern English
lcet10.txt	122	Technical Writing, English
plrabn12.txt	416	Paradise Lost, English Novel
ara-tabula.txt	470	Arabic
Beowulf.txt	2277	Old English Poetry
crime_and_punishment.txt	156	Russian Novel
genchi-all.txt	1889	Japanese Novel
Kokoro.txt	1421	Japanese Novel
ziemia_obiecana.txt	473	Polish Novel
cedict_small.txt	1202	Mixed language, Chinese & English

Table 4

Compression Ratio with proposed LZW, Unix Compress and gzip. Highest Compression Ratio is shaded in grey color

File name (.txt File)	**Original File Size (KB)**	**Compression Ratio**						
		Proposed LZW				**Unix Compress**		**Gzip**
		32 KB	**64 KB**	**32KB Dictionary update**	**64KB Dictionary update**	**32KB**	**64KB**	
ben-blog	23.1	4.17	4.17	4.17	4.17	3.30	3.30	4.15
ben-kobita	420	5.24	5.42	5.42	5.42	4.31	4.29	4.42
hin-baital	209	6.13	6.13	6.13	6.13	4.55	4.55	5.89
mal-travelblog	290	5.40	6.12	6.12	6.12	4.65	4.63	5.27
Indulekha	1453	7.30	7.30	7.38	7.41	4.96	5.36	6.08
Mal-Bible	1104	6.92	7.34	7.46	7.51	5.21	5.66	6.24
Khuran	1068	7.80	7.80	7.85	7.91	5.42	5.84	6.47

Table 4 *(Continued)*

File name (.txt File)	Original File Size (KB)	Compression Ratio						
		Proposed LZW				Unix Compress		Gzip
		32 KB	64 KB	32KB Dictionary update	64KB Dictionary update	32KB	64KB	
Bhagavat Geetha	174	5.34	5.34	5.34	5.34	4.24	4.24	4.73
Kundalatha	347	6.46	6.46	6.46	6.46	4.82	4.82	5.48
Umakeralam	676	5.24	5.24	5.28	5.28	4.12	4.17	4.23
alice29	148	2.39	2.43	2.44	2.44	2.45	2.44	2.79
asyoulik	122	2.27	2.27	2.27	2.27	2.27	2.27	2.55
lcet10	416	2.41	2.51	2.57	2.57	2.43	2.62	2.95
plrabn12	470	2.26	2.35	2.39	2.40	2.34	2.37	2.47
ara-tabula	2277	5.17	5.41	5.42	5.53	3.92	4.13	4.69
beowulf	156	2.54	2.58	2.59	2.59	2.52	2.50	2.69
crime_and_ punishment	1889	3.80	3.99	4.03	4.09	3.25	3.37	3.65
genchi-all	1421	3.90	4.10	4.14	4.24	2.99	3.17	3.30
Kokoro	473	3.45	3.62	3.58	3.64	2.83	2.99	3.20
ziemia_ obiecana	1202	2.28	2.40	2.42	2.47	2.25	2.36	2.54
cedict_small	745	1.65	1.72	1.75	1.77	1.71	1.80	1.97

Figure 8 depicts a comparison between the compression ratios obtained with the above said implementations with 32 KB dictionaries. Figure 9 depicts the same with 64 KB dictionaries. From the results obtained, it is evident that the proposed method compressed UTF-8 files by an average of 5% while using 32 KB dictionaries and 4% in case of 64 KB dictionaries than Unix Compress. The proposed method also showed a better compression ratio than gzip for languages with 3 to 4 bytes encodings.

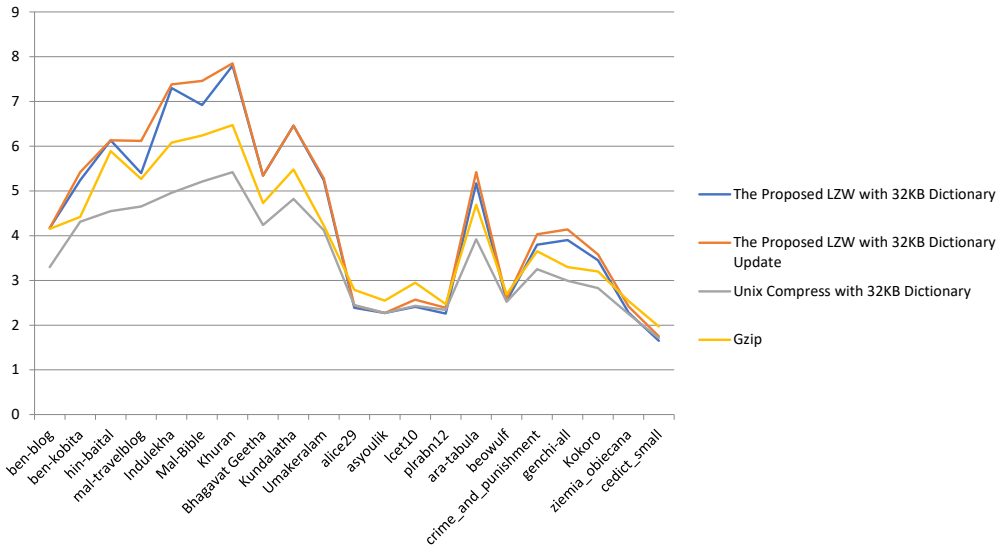

Figure 8. Compression Ratios with 32 KB dictionaries and Gzip

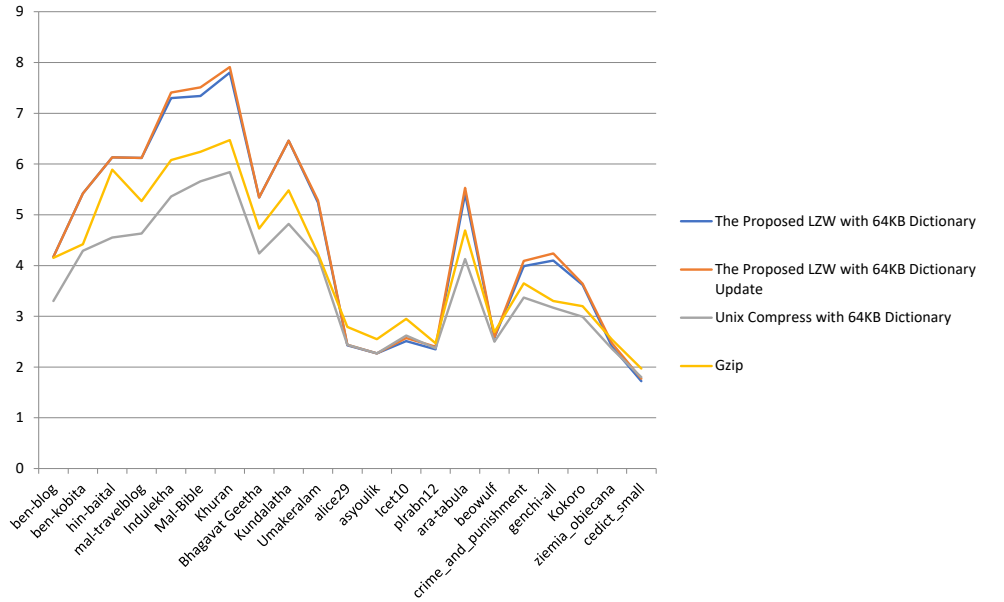

Figure 9. Comparison of Compression Ratios with 64 KB dictionaries and Gzip

CONCLUSION

This research demonstrates the use of enhanced LZW approach which permits empty dictionaries and it offers an efficient dictionary update mechanism whenever the dictionary reaches its specified maximum. This approach makes use of Unicode characters to lessen the number of dictionary entries and to achieve better compression. From the experiments, we can see that the proposed method is highly efficient in compressing text files with 3 or 4 byte encodings, even though there is performance degradation while compressing files of 1 or 2 byte encodings (Figure 8 & 9). This research will be the basis for future research to improve compression rate. It may be interpreted that the current invention is restricted to these specific patterns, but rather consider them as few explanatory examples alone. Hence, diverse alteration and improvisation shall be invoked by anyone who is talented in the art, without departing from the spirit or scope of the invention.

ACKNOWLEDGEMENT

We thank the Bharathiar University Research Centre, Coimbatore for all the facilities extended so far during the period of research.

REFERENCES

Al-Dubaee, S. A., & Ahmad, N. (2010, August 5-7). Multilingual lossy text compression using wavelet transform. In *2010 First International Conference on Integrated Intelligent Computing* (pp. 39-44). Bangalore, India.

Barua, L., Dhar, P. K., Alam, L., & Echizen, I. (2017, February 16-18). Bangla text compression based on modified Lempel-Ziv-Welch algorithm. In *2017 International Conference on Electrical, Computer and Communication Engineering (ECCE)* (pp. 855-859). Cox's Bazar, Bangladesh.

Divakaran, S., Anjali, C., Biji, C. L., & Nair, A. S. (2013). Malayalam Text Compression. *International Journal of Information Systems and Engineering, 1*(1), 7-11.

Fenwick, P., & Brierley, S. (1998, August 6). Compression of unicode files. In *Proceedings of DCC '98 Data Compression Conference* (pp. 547). Snowbird, Utah, USA.

Gleave, A., & Steinruecken, C. (2017, April 4-7). Making compression algorithms for Unicode text. In *Proceedings of the Data Compression Conference* (pp. 22-25). Snowbird, Utah, USA.

Hossain, M. M., Habib, A., & Rahman, M. S. (2014, May 23-24). Transliteration based bengali text compression using huffman principle. In *2014 International Conference on Informatics, Electronics & Vision* (ICIEV) (pp. 1-6). Dhaka, Bangladesh.

Maniya, S. V., Sheth, M. J., & Lad, K. (2012). Compression Technique based on Dictionary approach for Gujarati Text. *International Journal of Engineering Research and Development, 4*(8), 101-108.

Marjan, M. A., Uddin, M. P., Afjal, M. I., & Haque, M. D. (2014, October 21-23). Developing an efficient algorithm for representation and compression of large Bengali text. In *2014 9th International Forum on Strategic Technology (IFOST)* (pp. 22-25). Cox's Bazar, Bangladesh.

Shannon, C. E. (1948). A mathematical theory of communication. *The Bell System Technical Journal, 27*(3), 379-423.

Uthayakumar, J., Vengattaraman, T., & Dhavachelvan, P. (2018). A survey on data compression techniques: From the perspective of data quality, coding schemes, data type and applications. *Journal of King Saud University-Computer and Information Sciences, 2018*, 1-22.

Vijayalakshmi, B., & Sasirekha, N. (2018). Lossless text compression for unicode Tamil documents. *ICTACT Journal on Soft Computing, 8*(2), 1635-1640.

Welch, T. A. (1984). A technique for high-performance data compression. *Computer, 17*(6), 8-19.

Yamagiwa, S., Morita, R., & Marumo, K. (2019, March 26-29). Bank select method for reducing symbol search operations on stream-based lossless data compression. In *2019 Data Compression Conference (DCC)* (p. 611). Snowbird, Utah, USA.

Ziv, J., & Lempel, A. (1978). Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory, 24*(5), 530-536.

Ziv, J., & Lempel, A. (1977). A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory, 23*(3), 337-343.