Fast Lempel-ZIV (LZ'78) Algorithm Using Codebook Hashing

Megha Atwal, Lovnish Bansal

***Abstract*— Every communication system must be able to replicate the transmitted data at the receiver approximately the same if not exact. Smaller the amount of data lesser is the bandwidth consumption and also the probability of erroneous reception also decreases. Source coding schemes such as Lempel-Ziv coding help compress the data before transmission and thus help in bandwidth saving. But the compressed data must be decompressed before use. Also the compression time required leads to a delay in data transmission which may prove fatal in real time systems.**

To reduce the compression or encoding time of Lempel-Ziv coding, hashing technique can be used which is the core of this work. Hashing allows fast and easy access to the stored data, thereby reducing the search time from exponential to linear or in some cases constant or nearly stable.

I. INTRODUCTION

Claude Shannon ignited the area of source coding with his ground breaking papers [1] in the late 1940s and early 1950s. Shannon essentially forged the theoretical background of compression using information theory for lossless compression and rate-distortion theory for lossy compression. In information theory, data compression or source coding involves encoding information using fewer bits than the original. Compression can be either lossy or lossless. Lossless compression reduces bits by removing statistical redundancy. Lossy compression reduces bits by removing unnecessary information. The process of reducing the size of a file is referred to as data compression, though formally it is called source coding.

Compression is useful as it helps reduce resources usage, such as storage space or transmission capacity. But we must decompress the compressed data before use. This overhead

Processing imposes extra computational costs. For instance, a compression scheme for video may require expensive hardware for the video to be decompressed fast enough to be viewed as it is being decompressed, [2] and the option to decompress the video in full before watching it may be inconvenient or require additional storage. The design of data compression schemes [3] involves trade-offs among various factors, including the degree of compression, the amount of distortion introduced and the computational resources required to compress and un-compress the data [4].

Manuscript received March 20, 2015.

Megha Atwal, Student, Department of Computer Science & Engineering, Bhai Maha Singh College of Engineering, Shri Muktsar Sahib, Punjab, India,

Lovnish Bansal, Assistant Professor, Department of Computer Science & Engineering, Bhai Maha Singh College of Engineering, Shri Muktsar Sahib, Punjab, India,

II. DICTIONARY

A. *Dictionary Coding*

Dictionary coding techniques rely upon the [5] observation that there are correlations between parts of data (recurring patterns). The basic idea is to replace those repetitions by (shorter) references to a "dictionary" containing the original.

B. *Static Dictionary*

The simplest forms of dictionary coding use a static dictionary. Such a dictionary may contain frequently occurring phrases of arbitrary length, di-grams (two-letter combinations) or n-grams. This kind of dictionary can easily be built upon an existing coding such as ASCII by using previously unused codewords or extending the length of the codewords to accommodate the dictionary entries [6].
A static dictionary achieves little compression for most data sources. The dictionary can be completely unsuitable for compressing particular data, thus resulting in an increased message size (caused by the longer codewords needed for the dictionary)[7].

C. *Semi Adaptive Dictionary*

The aforementioned problems can be avoided by using a semi-adaptive encoder. This class of encoders creates a dictionary custom-tailored for the message to be compressed. Unfortunately, this makes it necessary to transmit/store the dictionary together with the data. Also, this method usually requires two passes over the data, one to build the dictionary and another one to compress the data. [8] A question arising with the use of this technique is how to create an optimal dictionary for a given message. Fortunately, there exist heuristic algorithms for finding near-optimal dictionaries.

D. *Adaptive Dictionary*

The ***Lempel Ziv*** algorithms belong to this third category of dictionary coders. The dictionary is being built in a single pass, while at the same time also encoding the data. As we will see, it is not necessary to explicitly transmit/store the dictionary because the decoder can build up the dictionary in the same way as the encoder while decompressing the data[9].

III. LEMPEL ZIV ALGORITHM

The Lempel Ziv Algorithm is an algorithm[10] for lossless data compression which employs adaptive dictionary coding. It is not a single algorithm, but a whole family of algorithms, stemming from the two algorithms proposed by Jacob Ziv and Abraham Lempel in their landmark papers in 1977 and 1978 as shown below. In this work we will only consider the original Lempel-Ziv algorithm proposed in 1978, also known as LZ'78 algorithm.

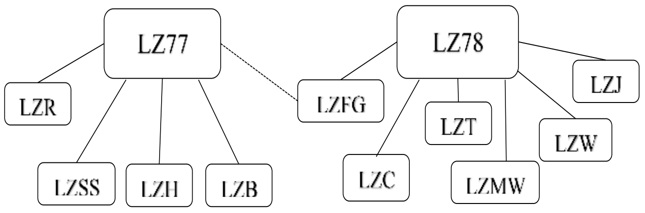


Figure 1: The Lempel Ziv Algorithm Family [6].

A. *Principal*

The LZ'78 is a dictionary-based compression algorithm that maintains an explicit dictionary. The codewords output by the algorithm consist of two elements: an index referring to the longest matching dictionary entry and the first non-matching symbol.In addition to outputting the codeword for storage/transmission, the algorithm also adds the index and symbol pair to the dictionary. When a symbol that not yet in the dictionary is encountered, the codeword has the index value 0 and it is added to the dictionary as well. With this method, the algorithm gradually builds up a dictionary [10]. This simplified pseudo-code version of the algorithm does not prevent the dictionary from growing forever. There are various solutions to limit dictionary size, the easiest being to stop adding entries and continue like a static dictionary coder or to throw the dictionary away and start from scratch after a certain number of entries has been reached.

IV. OBJECTIVE

Main objective of this work is to minimize the encoding time using the *hashing* technique of a finite long data sequence. Hashing is widely used technique used to accelerate table lookup or data comparison tasks such as finding items in a database, detecting duplicated or similar records in a large file, finding similar stretches in DNA sequences, and so on. The main focus of this work is to use implement hashing in LZ'78 algorithm and thereby making the LZ'78 phrase dictionary easily and swiftly searchable, without sacrificing the coding efficiency and the compression ratio attained by the LZ'78 algorithm. The following sections explain the implementation details. The software used to illustrate the results and findings is – Visual Studio2012.

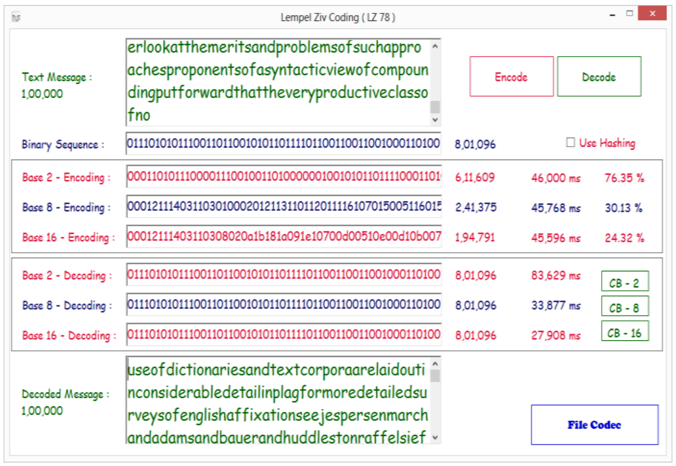
V. LZ'78 ENCODING USING HASHING

LZ'78 dictionary grows without bounds and yields long phrases but every phrase is ***unique***, i.e., no two phrases would match each other. We exploited this condition of LZ'78 and used it to build a *Hashtable*. At every point when a new unique phrase is generated while encoding using LZ'78, we need to save that phrase in the dictionary along with its corresponding index (generated continuously). But in case of *hashtable*, we used *reverse mapping*. At every point when a new unique phrase is generated while encoding using LZ'78, we would add a new entry in the *hashtable* with the generated unique phrase as *key* and the corresponding index as *value*. In this way for every new unique phrase generated instead of new dictionary entry we made a new *hash* entry. In this way by using *hashing* to create the LZ'78 dictionary, which always consists of unique phrases, we can quickly search through the previous hash entries and increasing the speed of the whole encoding process on the whole. The following section shows the LZ'78 encoding algorithm using hashtable and is self-explanatory[10].

VI. RESULTS

A. *Test Case #1*

Test Case #1 uses a string of 1,00,000 characters as input to LZ'78 encoder. The following sub-sections illustrate the result of test case #1 using LZ'78 with and without hashing. The GUI used is generated using Visual Studio 2012.



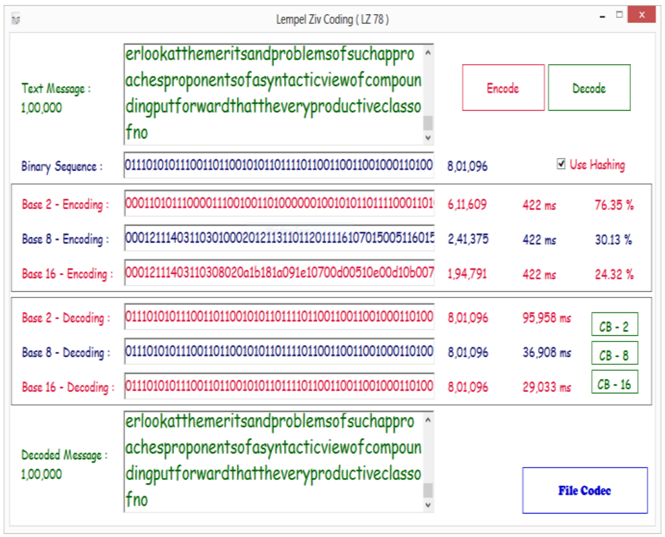
Text Message : 1,00,000
erlookatthemeritsandproblemsofsuchappro
achesproponentsofasyntacticviewofcompoun
dingputforwardthattheveryproductiveclasso
fno

Binary Sequence : 0111010101110011011001010110111101100110011001000110100 8,01,096 ☐ Use Hashing

Base 2 - Encoding :	000110101110000111001001101000000100101011011110001101	6,11,609	46,000 ms	76.35 %
Base 8 - Encoding :	000121114031103010002012113110112011116107015005116015	2,41,375	45,768 ms	30.13 %
Base 16 - Encoding :	00012111403110308020a1b181a091e10700d00510e00d10b007	1,94,791	45,596 ms	24.32 %
Base 2 - Decoding :	0111010101110011011001010110111101100110011001000110100	8,01,096	83,629 ms	CB - 2
Base 8 - Decoding :	0111010101110011011001010110111101100110011001000110100	8,01,096	33,877 ms	CB - 8
Base 16 - Decoding :	0111010101110011011001010110111101100110011001000110100	8,01,096	27,908 ms	CB - 16

Decoded Message : 1,00,000
useofdictionariesandtextcorporaarelaidouti
nconsiderabledetailinplagformoredetailedsu
rveysofenglishaffixationseejespersenmarch
andadamsandbauerandhuddlestonraffelsief

Fig. 2: Test Case #1 – LZ'78 Coding (Without Hashing)



Text Message : 1,00,000
erlookatthemeritsandproblemsofsuchappro
achesproponentsofasyntacticviewofcompoun
dingputforwardthattheveryproductiveclasso
fno

Binary Sequence : 0111010101110011011001010110111101100110011001000110100 8,01,096 ☑ Use Hashing

Base 2 - Encoding :	000110101110000111001001101000000100101011011110001101	6,11,609	422 ms	76.35 %
Base 8 - Encoding :	000121114031103010002012113110112011116107015005116015	2,41,375	422 ms	30.13 %
Base 16 - Encoding :	00012111403110308020a1b181a091e10700d00510e00d10b007	1,94,791	422 ms	24.32 %
Base 2 - Decoding :	0111010101110011011001010110111101100110011001000110100	8,01,096	95,958 ms	CB - 2
Base 8 - Decoding :	0111010101110011011001010110111101100110011001000110100	8,01,096	36,908 ms	CB - 8
Base 16 - Decoding :	0111010101110011011001010110111101100110011001000110100	8,01,096	29,033 ms	CB - 16

Decoded Message : 1,00,000
erlookatthemeritsandproblemsofsuchappro
achesproponentsofasyntacticviewofcompoun
dingputforwardthattheveryproductiveclasso
fno

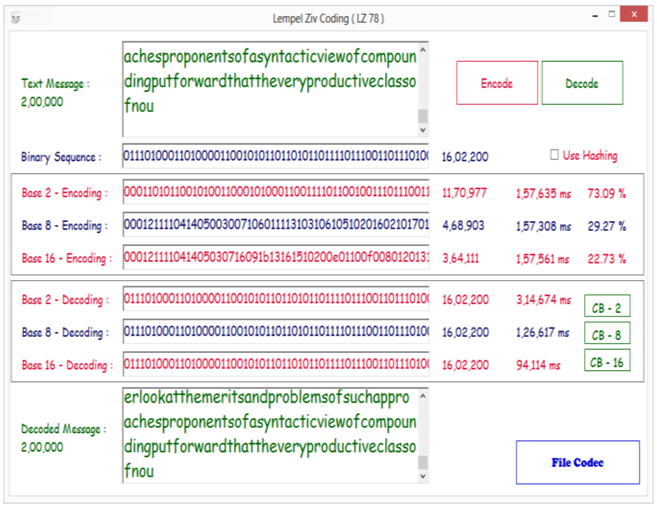
Fig. 3: Test Case #1 – LZ'78 Coding (With Hashing)

The user needs to enter a text message to encode. The Encode button will encode the entered text using LZ'78 without Hashing (as the 'Use Hashing' checkbox is not checked.) using base-2, 8 and 16 encoding. At first entered text is converted to binary sequence and then encoded using the binary, octal and hexadecimal encoding schemes with the help of LZ'78 and Hashing. The encoding results are displayed in the GUI corresponding to each encoding type as shown in the above figure. The decode button will decode the corresponding encoded data and display the results as shown above.

The Encode button will encode the entered text using LZ’78 with Hashing (as the ‘Use Hashing’ checkbox is checked). Rest all the process of encoding and decoding is same as explained in the previous section with LZ’78 without hashing.

B. *Test Case #2*

Unlike the previous test case the Test Case #2 uses a string of 2,00,000 characters as input to LZ’78 encoder and performs encoding with and without hashing. The following two figures show the test results.



Lempel Ziv Coding (LZ 78)

Text Message : 2,00,000 — achesproponentsofasyntacticviewofcompoundingputforwardthattheveryproductiveclassofnou

Binary Sequence : 0111010001101000011001010110110101101111011100110111010(— 16,02,200 — ☐ Use Hashing

Base 2 - Encoding : 0001101011001010011000101000110011110110010011101110011 — 11,70,977 — 1,57,635 ms — 73.09 %

Base 8 - Encoding : 00012111104140500300710601111310310610510201602101701 — 4,68,903 — 1,57,308 ms — 29.27 %

Base 16 - Encoding : 000121111041405030716091b13161510200e01100f008012013: — 3,64,111 — 1,57,561 ms — 22.73 %

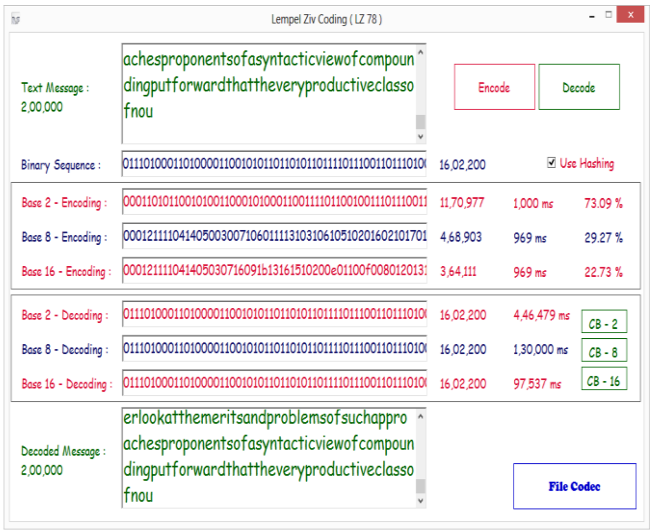
Base 2 - Decoding : 0111010001101000011001010110110101101111011100110111010(— 16,02,200 — 3,14,674 ms — CB - 2

Base 8 - Decoding : 0111010001101000011001010110110101101111011100110111010(— 16,02,200 — 1,26,617 ms — CB - 8

Base 16 - Decoding : 0111010001101000011001010110110101101111011100110111010(— 16,02,200 — 94,114 ms — CB - 16

Decoded Message : 2,00,000 — erlookatthemeritsandproblemsofsuchapproachesproponentsofasyntacticviewofcompoundingputforwardthattheveryproductiveclassofnou

Fig. 4: Test Case #2 – LZ’78 Coding (Without Hashing)



Lempel Ziv Coding (LZ 78)

Text Message : 2,00,000 — achesproponentsofasyntacticviewofcompoundingputforwardthattheveryproductiveclassofnou

Binary Sequence : 0111010001101000011001010110110101101111011100110111010(— 16,02,200 — ☑ Use Hashing

Base 2 - Encoding : 0001101011001010011000101000110011110110010011101110011 — 11,70,977 — 1,000 ms — 73.09 %

Base 8 - Encoding : 00012111104140500300710601111310310610510201602101701 — 4,68,903 — 969 ms — 29.27 %

Base 16 - Encoding : 000121111041405030716091b13161510200e01100f008012013: — 3,64,111 — 969 ms — 22.73 %

Base 2 - Decoding : 0111010001101000011001010110110101101111011100110111010(— 16,02,200 — 4,46,479 ms — CB - 2

Base 8 - Decoding : 0111010001101000011001010110110101101111011100110111010(— 16,02,200 — 1,30,000 ms — CB - 8

Base 16 - Decoding : 0111010001101000011001010110110101101111011100110111010(— 16,02,200 — 97,537 ms — CB - 16

Decoded Message : 2,00,000 — erlookatthemeritsandproblemsofsuchapproachesproponentsofasyntacticviewofcompoundingputforwardthattheveryproductiveclassofnou

Fig. 5: Test Case #2 – LZ’78 Coding (With Hashing)

VII. COMARISON

We compare the above test cases using two parameters, *viz.*, encoding time and compression ratio. Encoding time should be as small as possible and compression ratio should ideally be less than 100%. Less ratio implies that the encoded sequence length is less than the original message length (in binary) and that the data is compressed.

$$\text{Compression Ratio} = \frac{\text{Length of Encoded Binary Message}}{\text{Length of Original Binary Message}} \times 100\ \%$$

The following tables summarize the results obtained from the previous test cases for base-2 encoding only.

Table 1.1: Compression Ratio comparison for different message length.

Input Binary Message Length	Encoded Message Length	Code Book Entries	Compression Ratio
801,096	611,609	39,833	76.35 %
1,602,200	1,170,977	72,337	73.09 %
2,402,448	1,706,459	102,086	71.03 %

Table 1.2: Encoding time comparison with and without Hashing.

Input Message Length (Binary Message)	Time taken to Encode	
	With Hashing	Without Hashing
801,096	0.422 seconds	46.000 seconds
1,602,200	1.000 seconds	157.635 seconds
2,402,448	1.562 seconds	319.818 seconds

We can see from the results obtained in table 1, larger the input message length more is the compression ratio. Also the number of code book entries increase with input message length. With such a huge amount of entries it is virtually impossible for any system to perform a real-time search.
Form the results obtained in table 2, we can see that with increase in the number of codebook entries the encoding time increases from 46 seconds to nearly 320 seconds without hashing. But as explained earlier about fastness of hashing, we can see that the encoding time remains almost unchanged with increase in number of codebook entries. It increases but slowly as compared to the one without hashing. Hence, our results.

VIII. CONCLUSION

In this paper we presented a source coding scheme that we call *Hashed Lempel-Ziv coding*, as an extension for the LZ’78 coding scheme, without sacrificing the coding efficiency and the compression ratio attained by the original LZ’78 algorithm.
In addition to outputting the codeword for storage/transmission, the algorithm also adds the index and symbol pair to the dictionary. When a symbol that not yet in the dictionary is encountered, the codeword has the index value 0 and it is added to the dictionary as well. With this method, the algorithm gradually builds up a dictionary.
But LZ’78 has several weaknesses. First of all, the dictionary grows without bounds. Various methods have been introduced to prevent this, the easiest being to become either static once the dictionary is full or to throw away the dictionary and start creating a new one from scratch. There are also more sophisticated techniques to prevent the

dictionary from growing unreasonably large. The dictionary building process of LZ'78 yields long phrases only fairly late in the dictionary building process and only includes few substrings of the processed data into the dictionary.

Larger the dictionary, better is the coding efficiency. But the main drawback of large dictionary is the amount of time required to look for uniqueness of every new phrase in the dictionary. Each time we have to search the whole dictionary to check whether the phrase encountered is already present in the dictionary or not. Thus, searching the dictionary takes a very-very-very long time, because each search needs '*r*' comparisons ('*r*' is the current number of phrases in the codebook). Also, we can't use Binary search too, as the phrases in the dictionary are unsorted.

To solve this problem, we used Hashing technique. Hashing can be used for faster search in a dictionary. As seen in the results obtained, hashing only makes the encoding process of LZ'78 coding faster, with no change in the coding efficiency or compression ratio. Irrespective of the size of the codebook the time taken to encode using Hashed LZ'78 was more or less the same. But without hashing there was a huge difference in time taken to encode and hence our results were justified.

REFERENCES

[1] A.-R. Elabdalla and M. Irshid, "An efficient bit-wise source encoding technique based on source mapping," in *Devices, Circuits and Systems, 2000. Proceedings of the 2000 Third IEEE International Caracas Conference on*, 2000.

[2] D. Kirovski and Z. Landau, "Generalized Lempel--Ziv compression for audio," *Audio, Speech, and Language Processing, IEEE Transactions on,* vol. 15, no. 2, pp. 509-518, 2007.

[3] M. B. B. Loranca and L. A. O. Santos, "Multiple correspondences and log-linear adjustment to test single correspondence relationships and categorization of qualitative data in e-commerce," in *Electronics, Communications and Computers, 2004. CONIELECOMP 2004. 14th International Conference on*, 2004.

[4] C. E. Shannon, "A mathematical theory of communication," *ACM SIGMOBILE Mobile Computing and Communications Review,* vol. 5, no. 1, pp. 3-55, 2001.

[5] J. Ziv and A. Lempel, "Compression of individual sequences via variable-rate coding," *Information Theory, IEEE Transactions on,* vol. 24, no. 5, pp. 530-536, 1978.

[6] K. Sayood, Introduction to data compression, Newnes, 2012.

[7] T. Weissman, N. Merhav and A. Somekh-Baruch, "Twofold universal prediction schemes for achieving the finite-state predictability of a noisy individual binary sequence," *Information Theory, IEEE Transactions on,* vol. 47, no. 5, pp. 1849-1866, 2001.

[8] T. A. Welch, "A technique for high-performance data compression," *Computer,* vol. 17, no. 6, pp. 8-19, 1984.

[9] M. E. Hellman, "An extension of the Shannon theory approach to cryptography," *Information Theory, IEEE Transactions on,* vol. 23, no. 3, pp. 289-294, 1977.

[10] R. N. Williams, "An extremely fast Ziv-Lempel data compression algorithm," in *Data Compression Conference, 1991. DCC'91.*, 1991.