Università degli Studi di Padova - Department of Information Engineering

MULTIMEDIA CODING COURSE

Final Project

IMPLEMENTATION OF LZ77 AND LZSS ALGORITHMS

Yükselcan Sevil

February 23, 2021

Abstract

In digital communications, compression algorithms are used to transmit the data by reducing similar sequences from the original data and decompress the data into its original form These techniques provide more efficient transmission time and more capacity in storage and they can be used for the transmission of the data such as text, audio, image.

The purpose of this report is to:

1. give a short introduction to compression techniques.

2. give a brief explanation of the dictionary-based LZ77 and LZSS algorithms.

3. show an evaluation of the performances of dictionary-based compression techniques.

4. discuss the performance results

1 Introduction

Compression algorithms are generally referring to two main types. These are

1. ***lossless coding***: Lossless compression techniques provide to compress and decompress data without loss of information [1]. To do that, it removes the redundant information from the input source and recovers the data by using unique data information . As their name indicates, can not recover the input source if the loss occurs during the compression. These techniques are perfectly reversible, can be applied only to digital sources, and need variable length coding.
2. ***lossy coding***: Lossy compression is a method of data compression that removes redundant information from the source input [1]. This technique has some loss of information and can not be invertible perfectly. In lossy coding, the main goal is to achieve the highest possible reconstruction fidelity subject to constraints of compressed data size.

In this report dictionary-based techniques will be discussed, one of the techniques involved in lossless coding. Dictionary-based techniques are generally divided into

1. *static dictionary*: in a static dictionary, a fixed dictionary is used for multiple compression process [1]. It sometimes allowing the addition of new entry but deletion not allowed.
2. *adaptive dictionary*: in the adaptive dictionary, each entry in the dictionary is pre-read entries of the input source [1]. Original ideas of the adaptive dictionary-based techniques are published by Jacob Ziv and Abraham Lempel in 1977 (LZ77) and 1978 (LZ78).

In this report, the first published LZ77 algorithm and its variant LZSS will be implemented and the compression results will be tested according to the size of the sliding window.

2 Technical approach

2.1 Objectives

The objective of this section is that of providing a brief description huffman codin, LZ77, LZSS algorithms.

2.2 Huffman Coding

Huffman coding was published by David Huffman. In this procedure, the codes are prefix code and optimum code according to given set of probabilities [1]. The Huffman algorithm is based on the idea of representing more common symbols in a data set with less length code and uncommon symbols with larger length codes. The algorithm is as follows:

1. Find the frequency of the all symbols.
2. Sort the symbol probabilities in non increased.
3. Group together two least probable symbols into a new symbol with assign probability equal to the sum of the probabilities.
4. Repeat the procedures 2 and 3 until get a reduced source with two symbols.

2.3 LZ77 Algorithm

This compression algorithm focus on keeping encoded sequences with its distance where the same sequence happened before. The encoder checks the input sequence which is bounded with sliding window size. The sliding window is divided into two parts. These are search window which holds the previous encoded sequences and lookahead buffer which holds the next sequences to be encoded [1]-[2].

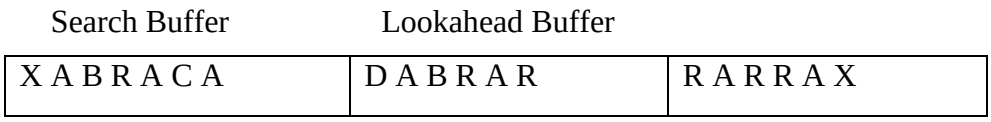
To encode a sequences, the sequence in the looakahead buffer is compared with sequences in the search window, if the longest sequence matches a sequence in the search window side [1]-[4], the mapped sequence and the next unmapped character are placed into search buffer and the previous mapped sequence with the same code length as the next matched sequence and the next character is removed at the beginning of the search window.

The dictionary is consist of a set of tuples and each tuple [1] is shown as (o, l, c). Each tuple shows the distance between the matched sequence in the search window and matched sequence in the lookahead buffer which is called offset and shown as ‘o’. The codeword

length of the matched sequence is shown as ‘l’. Also the tuple holds the next character after the matched sequences and the next character is shown as ‘c’.

Let's look at what this algorithm means with an example.

Encoding:



1. Next symbol is D and we are looking for D in out search buffer. Offset and the length of the codeword is zero. Because there is no any matched sequences.

⟶ (0, 0, D)

2. The next sequence is ABRA and the offset of this sequences is 7 and the length of the codeword is 4. The next incoming character after this sequence is R.

⟶ (7, 4, R)

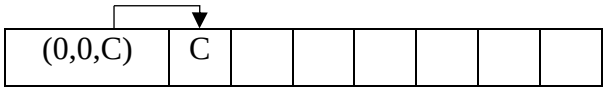
3. The next matched sequences RARRA can be found in a both side. Because we can also search the next sequences from search buffer.

⟶ (3, 5, X)

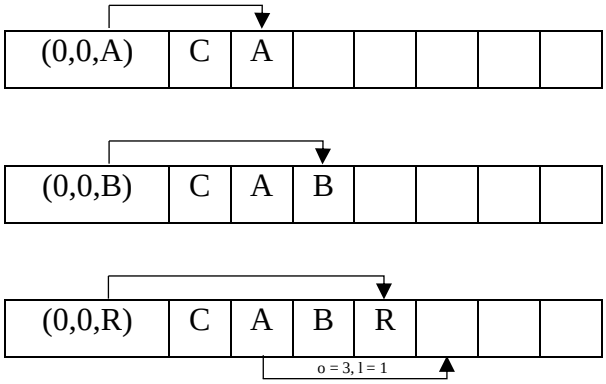
Decoding:

We can start decompression process from the first tuple.

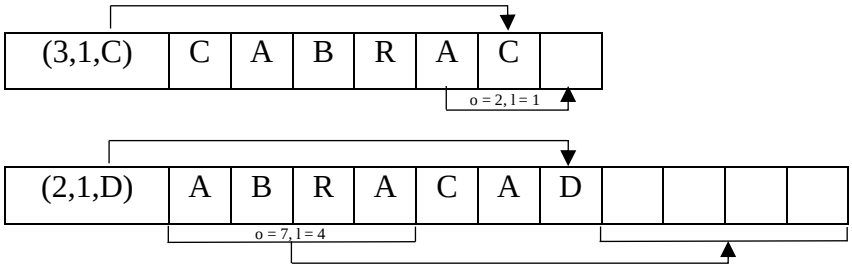
Dictionary Decoding Sequences



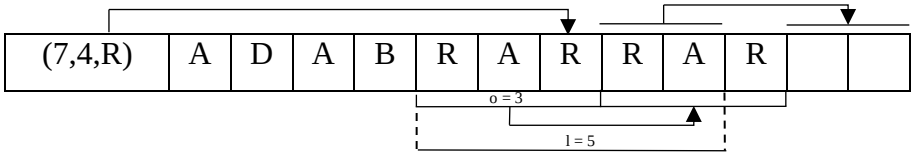
1. Starting with (0,0,C), we have move o=0 position to left and there is no any matched string. So we just put the character ‘C’ into sequence. This can be applied for every tuples where the offset is zero.



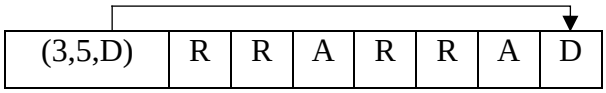
2. The next tuple is (3,1,C) that means we have to move o=2 position left from the last character and take the string which length is equal to 1 (l=1). So we have to take A character and merge it with next character C. Then this merged string ‘AC’ are putted into decoded string.



3. The next tuple is (7,4,R) that means we have to move o=7 position left from the last character and take the string whick length is equal to l=4. Then merge this string with ‘R’ and we get ‘ABRAR’.



4. The next tuple is (3,5,D) and first we have copy the ‘RAR’. Then we have copy the copied character which length (l) is equal to 2. Because we copy the first three characters. First and second character of the copied string is R and A and we have put this characters near to ‘RAR’. So we get the decoded string like ‘RARRA’. After that we have to add a next character ‘D’ into string.



To find a number of bits for each tuple, we can use the following equation [1]-[4].

$$\mathrm{N} = \log_2(S) + \log_2(L) + \log_2(M)$$

S: length of search window size (represents offset)

L: length of lookahead buffer size (represents length of the matching)

M: size of source alphabet (number of bits for coding each triplet)

2.4 LZSS Algorithm

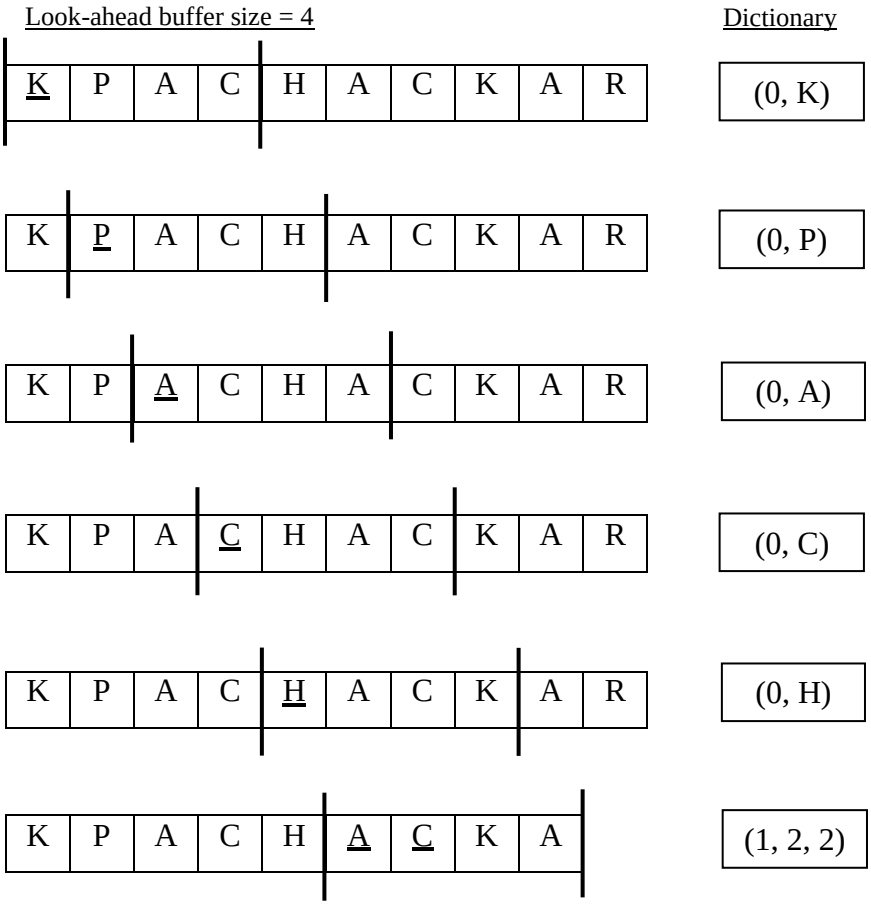
Lempel Ziv Storer-Szymanski (LZSS) is a compression algorithm that was published in 1982 by James A. Storer and Thomas Szymanski. It is a derivative of LZ77 [3].

It is same as LZ77, but we use flag to denote coding of a single character whether the next symbol was matched or not. If there is no match found, store only next symbol into dictionary [3]. Otherwise, store the length of the matched string and its offset, but not the next character.

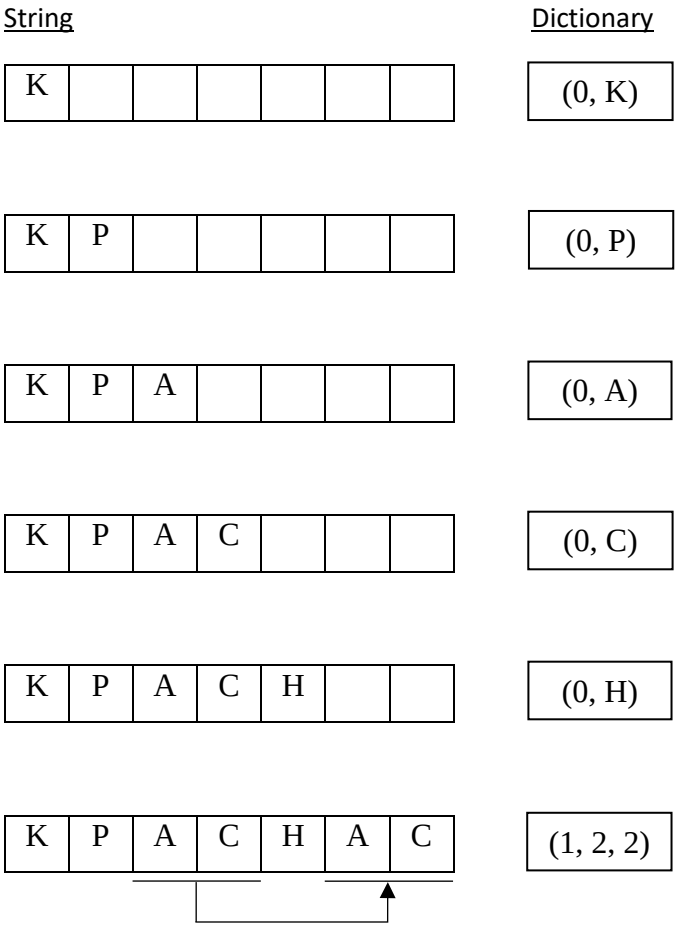
The dictionary [3]-[4] is consist of a set of tuples and each tuple is shown as (0, c) if there is no any match or (1, o, l) if the match is found. As can be seen, if there is no any match, flag=0. Otherwise flag=1. We know that from LZ77, o represents the offset, l represent length of the string, c represents the character.

Let's look at what this algorithm means with an example.

Encoding:



Decoding:



In the encoding process, the dictionary need to be searched for matches to the string to be encoding. However, in decoding process an offset and codeword length enough to find encoded string.

To find a number of bits for each tuple, we can use the following equation [4].

$$N = \begin{cases} 1 + \log_2(M) & if\ no\ match \\ 1 + \log_2(S) + \log_2(L) & otherwise \end{cases}$$

S: length of search window size (represents offset)

L: length of lookahead buffer size (represents length of the matching)

M: size of source alphabet (number of bits for coding each triplet)

1: represents the flag for each triplet

2.5 Matlab Implementation

- **lz77_encode** This function encodes text into the LZ77 algorithm. In this function, the length of the given input is calculated first and an empty dictionary is created for the indices in the dictionary. Then the initial dictionary is created according to the length of the search buffer. The cursor is used which is created for the checking the sequence between the search buffer and look-ahead buffer. If any match is found, its offset is checked. To find a maximum length of the matched string, *findLenght.m* function is used. This process is applied for each string that is found then the longest string of this process put into the dictionary. In LZ77, we put the next character of the matched string, so the next cursor value is checked and the found character is added to the library.
- **findLength.m** This function starts from an offset in search buffer and cursor position. Then it finds the length of the matched string in sequence
- **lz77_encode_img.m** This function implements same algorithm with *lz77_encode.m*. The only difference is *lz77_encode.m* merges the char type data and *lz77_encode_img.m* merges the uint8 type data.
- **lz77_findRatio.m** This function creates binary type data for every source symbol. Then number of bits are compared between input source data and encoded data.
- **lz77_decode.m** This function implements the LZ77 decoding using the values that is found in the dictionary.
- **lz77_main.m** This file contains all method of the LZ77 and implies every function in one file.
- **lzss_encode.m** This function encodes input source data with the LZSS algorithm. In this function, The dictionary and its indices are initialized. The current index position is checking for any matches between the look-ahead buffer and the dictionary. If any match is found, flag is set to 1 and *match.m* is used to find matched indexes and its position. After that flag, offset and length variables are added into dictionary. If match is not found, flag=1 and new index are added into dictionary.
- **lzss_encode_img.m** This function implements same algorithm with *lzs_encode.m*. The difference between these function are: *lzss_encode.m* merges the char type data and *lzss_encode_img.m* merges the uint8 type data.
- **lzss_findRatio.m** This function creates binary type data for every source symbol. Then number of bits are compared between input source data and encoded data.

- **match.m** This function starts from an offset in search buffer and cursor position. For every position it finds the string and the position of the matched string.

- **lzss_decode.m** This function implements the LZSS decoding using the values that is found in the dictionary.

- **lzss_main.m** This file contains all method of the LZSS and implies every function in one file.

3 Comparison Results

3.1 Compression of Text data

Table 1: Text comparison ratio of LZ77/LZSS

	Input	**LZ77**		**LZSS**	
	Input bits (N)	Encoded bits (E)	Comparison Ratio (N/E)	Encoded bits (N)	Comparison Ratio (E)
W=200	693688	361129	1.92	449262	1.5441
W=400	693688	342749	2.0239	398421	1.7411
W=800	693688	319399	2.1719	339474	2.0434
W=1600	693688	291760	2.3776	282100	2.4590
W=3200	693688	264107	2.6265	246288	2.8166
W=6400	693688	250983	2.7639	231260	2.9996
W=12800	693688	252085	2.7518	226454	3.0033

Table 2: Text comparison ratio of other compression techniques

	ZIP	**GZIP**	**XZ**
Comparison Ratio (100kb)	4.0269	4.0331	6.7153

3.2 Compression of Audio data

Table 3: Audio comparison ratio of LZ77/LZSS

	Input	**LZ77**		**LZSS**	
	Input bits (N)	Encoded bits (E)	Comparison Ratio (N/E)	Encoded bits (N)	Comparison Ratio (E)
W=800	7408800	1928942	3.84	1898191	3.90
W=1600	7408800	1899507	3.90	1856449	3.99
W=3200	7408800	1877121	3.94	1833444	4.04
W=6400	7408800	1867918	3.96	1807410	4.09
W=12800	7408800	1873756	3.95	1783219	4.15

Table 4: Audio comparison ratio of other compression techniques

	ZIP	**GZIP**	**XZ**
Comparison Ratio (904kb)	4.52	4.52	5.13

Note: 70mono8bits.wav is used in this lossless data techniques comparison according to sliding window size.

3.3 Compression of Image data

Table 5: Image comparison ratio of LZ77/LZSS

	Input	**LZ77**		**LZSS**	
	Input bits (N)	Encoded bits (E)	Comparison Ratio (N/E)	Encoded bits (N)	Comparison Ratio (E)
W=800	2097152	1886058	1.111	2454254	0.9167
W=1600	2097152	1869703	1.121	2486761	0.9245
W=3200	2097152	1849897	1.133	2427642	0.9520
W=6400	2097152	1835193	1.142	2368179	0.9901
W=12800	2097152	1826188	1.148	2287493	1.0597
W=25600	2097152	1834107	1.143	2205690	1.1238

Table 6: Image comparison ratio of other compression techniques

	ZIP	**GZIP**	**XZ**
Comparison Ratio (256kb)	1.16	1.16	1.30

Note: lena.tiff is used in this lossless data techniques comparison according to sliding window size.

3.4 Observations

The following observations can be made:

- Although the efficiency of the LZ77 is better than the LZSS for small window sizes (see table 1 and 3), we can not say the LZ77 algorithm is better than the LZSS algorithm. In fact, LZSS is much more efficient than the LZ77 for higher window sizes. If we look at it realistically, we can say that the efficiency of the LZ77 algorithm is insufficient because such small data is no longer used.
- We can observe that the LZ77 algorithm loses its efficiency after a certain window size. This is caused by the sliding window size. Because search buffer has so many initial elements and it affects the number of the compressed strings. After the sliding window size exceeds 6400 (see tables 1 and 3), the efficiency of the compression starts to decrease. We cannot say the same for the LZSS algorithm (see table 1 and 3). Because there was no loss in efficiency according to the search window sizes used.
- Other compression techniques (zip, gzip, xz) are more efficient (see table 2 ,4, and 6) than dictionary-based compression techniques. We can see that zip and gzip are 15% more efficient than the LZ77 algorithm on efficiency. If we would like to compare the LZSS algorithm with other compression techniques, we can see that zip and gzip are 8% more efficient (see tables 3 and 4) than the LZSS algorithm.
- Dictionary-based algorithms have failed to compress (see tables 5 and 6) the image data. Because LZ77 and LZSS, use the fact that contiguous data sequences are correlated. These dictionary-based algorithms mostly used to compress text because they reduce one-dimensional data efficiently. I linearize the image data to use LZ77 and LZSS algorithms on the image data. However, results showed that linearization does not work for image datas.

4 Conclusions

In this report a short introduction to the lossless data compression is given and then Lempel Ziv 77, Lempel Ziv (LZ77), Lempel Ziv Storer–Szymanski (LZSS) are represented and the performances of this algorithm are analyzed. This algorithms are a kind of dictionary based compression algorithms and generally used for compressible data. For LZ77, The compressed data occurs from triplets: Each triplets contains the offset, the codeword length of the matched sequence, and the next character after the matched sequences. If the most of strings did not match anything in the dictionary, you can see that LZ77 may not be efficient algorithm except for compressible data. LZSS uses the flag which are prefix signal bits. As we can see LZSS can be compress the data more compactly than the LZ77 depending on window size. In fact, we can say LZSS is much more efficiently than the LZ77. As a result, comparing LZ77 and LZSS with other compression techniques, we cannot say that they are a very efficient compression method for these days.

4.1 Lessons learned

This project provides me a theoretical and coding experiences on lossless data compression.

4.2 References

[1] Sayood, K., 2000. *Introduction to data compression*. 4th ed. San Francisco: Morgan Kaufmann Publishers, pp.4-5,139-143.

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

[3] Storer, J. and Szymanski, T., 1982. Data compression via textual substitution. *Journal of the ACM*, 29(4), pp.928-951.

[4] Bell, T., Cleary, J. and Witten, I., 1990. *Text compression*. Englewood Cliffs, N.J.: Prentice Hall, pp.218-222.

[5] Kwon, B., Gong, M. and Lee, S., 2017. Novel Error Detection Algorithm for LZSS Compressed Data. *IEEE Access*, 5, pp.8940-8947.