

Project Report

INF224 – Data Structures and Algorithms

Lempel-Ziv-Welch Compression Algorithm

Hasan Kayra Mike

20401878

Lempel–Ziv–Welch (LZW) is a universal lossless data compression algorithm created by Abraham Lempel, Jacob Ziv, and Terry Welch. It was published by Welch in 1984 as an improved implementation of the LZ78 algorithm published by Lempel and Ziv in 1978. The algorithm is simple to implement and has the potential for very high throughput in hardware implementations. It is the algorithm of the Unix file compression utility `compress` and is used in the GIF image format, PDF, and TIFF files.^[1] The algorithm was patented in 1985 and the patent was owned by Unisys Corporation until 2003. The algorithm remains in the public domain ever since.^[2] LZW is very fast, but achieves poor compression compared to most newer algorithms and some algorithms are both faster and achieve better compression.^[3]

The LZW algorithm is a greedy algorithm in that it tries to recognize increasingly longer and longer phrases that are repetitive and encode them. Each phrase is defined to have a prefix that is equal to a previously encoded phrase plus one additional character in the alphabet. Note “alphabet” means the set of legal characters in the file. For a normal text file, this is the ASCII character set.^[4]

The algorithm makes use of a dictionary that stores character sequences chosen dynamically from the text. With each character sequence the dictionary associates a number; if s is a character sequence, we use `codeword(s)` to denote the number assigned to s by the dictionary. The number `codeword(s)` is called the code or code number of s . All codes have the same length in bits; a typical code size is twelve bits, which permits a maximum dictionary size of $2^{12} = 4096$ character sequences.^[5]

The dictionary is initialized with all possible one-character sequences, that is, the elements of the text alphabet (assume N symbols in the alphabet) are assigned the code numbers 0 through $N-1$ and all other code numbers are initially unassigned. The text w is encoded using a greedy heuristic: at each step, determine the longest prefix p of w that is in the dictionary, output the code number of p , and remove p from the front of w ; call p the current match. At each step we also modify the dictionary by adding a new string and assigning it the next unused code number. The string to be added consists of the current match concatenated to the first character of the remainder of w . It turns out to be simpler to wait until the next step to add this string; that is, at each step we determine the current match, then add to the dictionary the match from the previous step concatenated to the first character of the current match. No string is added to the dictionary in the very first step.^[6]

Since codes are added in a manner determined by the data, the decoder mimics building the table as it sees the resulting codes. It is critical that the encoder and decoder agree on the variety of LZW used: the size of the alphabet, the maximum table size (and code width) and initial code size. Most formats that employ LZW build this information into the format specification or provide explicit fields for them in a compression header for the data.^[7]

How the codes are packed into bytes must be agreed upon between the encoder and the decoder. GIF uses little-endian representation (least significant bit first) while TIFF and PDF use big-endian representation (most significant bit first).^[8]

Encoding algorithm is the following:

1. Initialize the dictionary to contain all strings of length one.
2. Find the longest string W in the dictionary that matches the current input.
3. Emit the dictionary index for W to output and remove W from the input.
4. Add W followed by the next symbol in the input to the dictionary.
5. Go to Step 2.

Decoding algorithm is the following:

1. Initialize the dictionary to contain all strings of length one.
2. Read the next encoded symbol: Is it encoded in the dictionary?
 1. Yes:
 1. Emit the corresponding string W to output.
 2. Concatenate the previous string emitted to output with the first symbol of W. Add this to the dictionary.
 2. No:
 1. Concatenate the previous string emitted to output with its first symbol. Call this string V.
 2. Add V to the dictionary and emit V to output.
3. Repeat Step 2 until end of input string

Example:

Let's take the string: "TOBEORNOTTOBEORTOBEORNOT#". The character '#' represents the stop code which is a special code or a character that indicates the end of the string is reached. The LZW algorithm iterates over the character array just once. For this example, let's assume that only the English alphabet is used and we won't be needing any other strings of length one of the ASCII.

Algorithm starts by initializing a dictionary which contains the stop code and the letters of the English alphabet. In order to iterate over the string just once, LZW stores both the current substring and the character after that. After having initialized the dictionary, it now assigns the first character of the string to the "next character" variable and accepts the current substring to be NULL. Then it checks if the concatenation of the current substring and the next character is in the dictionary. In this case, it would be NULL + 'T', which is 'T'. The dictionary was initialized with the letters of the English alphabet, so the character 'T' is in fact contained in the dictionary. Once the algorithm decides on that, it assigns the letter after that, which is 'T', to the current substring and assigns the character after that, which is 'O', to the "next character" variable. Now the concatenation of the two variables, "current substring" and "next character" is calculated: 'T' + 'O' = "TO". Then it checks if the concatenation is contained in the dictionary, which is not. The dictionary was initialized with a total of 27 codes from 0 through 26, so the next available code is 27. Since the concatenation is not found in the dictionary, the algorithm assigns the code 27 to it and adds it to the dictionary and increments the next available code by one to be 28.

If the concatenation of the “current substring” and the “next character” variables is contained in the dictionary, the algorithm assigns the concatenation to be the current substring and moves on with checking the concatenation of that with the new next character. Once the algorithm assigns ‘#’ to the “next character” variable, it decides that the end of the string is reached and checks if the current substring is in the dictionary, executes the operations accordingly, and stops.

Current Sequence	Next Char	Output		Extended Dictionary	Comments
		Code	Bits		
NULL	T				
T	O	20	10100	27: TO	27 = first available code after 0 through 26
O	B	15	01111	28: OB	
B	E	2	00010	29: BE	
E	O	5	00101	30: EO	
O	R	15	01111	31: OR	
R	N	18	10010	32: RN	32 requires 6 bits, so for next output use 6 bits
N	O	14	001110	33: NO	
O	T	15	001111	34: OT	
T	T	20	010100	35: TT	
TO	B	27	011011	36: TOB	
BE	O	29	011101	37: BEO	
OR	T	31	011111	38: ORT	
TOB	E	36	100100	39: TOBE	
EO	R	30	011110	40: EOR	
RN	O	32	100000	41: RNO	
OT	#	34	100010		# stops the algorithm; send the cur seq
		0	000000		and the stop code

The string is now encoded as: 20 15 2 5 15 18 14 15 20 27 29 31 36 30 32 34 0

The bit representation: 10100 01111 00010 00101 01111 10010 001110 001111 010100 011011 011101 011111 100100 011110 100000 100010 000000

Before the encoding, the string had 25 characters which were 5 bits each. So the total space required for the string:

$$25 \text{ characters} * 5 \text{ bits/character} = 125 \text{ bits.}$$

After the encoding, the string now has 6 substrings with 5 bits per substring and 11 substrings with 6 bits per substring. So the total amount of space that the string uses is reduced to:

$$6 \text{ substrings} * 5 \text{ bits/substring} + 11 \text{ substring} * 6 \text{ bits/substring} = 96 \text{ bits.}$$

Having used the LZW algorithm, the string now requires 23.2% less space. As seen in the example, length and the repetitiveness of the string is directly proportional to the percentage of compression. If the input string were “TOBEORNOT#”, the LZW algorithm wouldn’t have saved any space, for there wouldn’t be any repetitions of character sequences.

In order to decode the output of an LZW algorithm, the initial dictionary must be available for the decoder. In this case, it is the dictionary with the stop code ‘#’ and the letters of the English alphabet.

The algorithm starts with the first code of the encoded string, which is 20, then looks up that code in the dictionary and finds that it’s the code for ‘T’. Then takes the next code, which is 15, while the current sequence is still ‘T’ and looks the next code up in the dictionary, which returns ‘O’. The algorithm now gets the next available code, 27, to encode the concatenation of the current sequence and the first letter of the next code’s value in the dictionary. So, the new code 27 is now added to the dictionary as the encoding of “TO” to decode sequences of “TO” in the future. Once the algorithm reaches the stop code, which is 0 and encodes ‘#’, it decides that the decoding is completed and stops.

Input		Output Sequence	New Dictionary Entry		Comments
Bits	Code		Full	Conjecture	
10100	20	T		27: T?	
01111	15	O	27: TO	28: O?	
00010	2	B	28: OB	29: B?	
00101	5	E	29: BE	30: E?	
01111	15	O	30: EO	31: O?	
10010	18	R	31: OR	32: R?	created code 31 (last to fit in 5 bits)
001110	14	N	32: RN	33: N?	so start reading input at 6 bits
001111	15	O	33: NO	34: O?	
010100	20	T	34: OT	35: T?	
011011	27	TO	35: TT	36: TO?	
011101	29	BE	36: TOB	37: BE?	36 = TO + 1st symbol (B) of
011111	31	OR	37: BEO	38: OR?	next coded sequence received (BE)
100100	36	TOB	38: ORT	39: TOB?	
011110	30	EO	39: TOBE	40: EO?	
100000	32	RN	40: EOR	41: RN?	
100010	34	OT	41: RNO	42: OT?	
000000	0	#			

Complexity Analysis of the LZW Algorithm:

The compression algorithm takes a data input and iterates over that data only once; therefore, the time complexity for the LZW compression algorithm is $O(n)$, where n is the number of characters in the input data.

The algorithm stores a dictionary of unique substrings which is linearly proportional to the number of characters in the input data, so its space complexity is $O(n)$, where n is the number of characters in the input data. The input data contains repetitive characters and the dictionary stores only the unique substrings; therefore, at all times:

$$X \leq n$$


where n is the number of characters in the input data and X is the number of substrings in the dictionary.

When decompressing, the decompression algorithm iterates over the compressed data just once, so the time complexity for the LZW decompression algorithm is $O(n)$, where n is the number of characters in the compressed data.

The decompression algorithm reconstructs the original data from the compressed data with the knowledge of the initial dictionary, so the space complexity for the algorithm is $O(n)$, where n is the number of characters in the original data. Note that the decompression algorithm creates the final dictionary in less space than the original data but also stores the decompressed data which is typically the same size as the original data.

Definition of the Problem:

A book editorial company that receives hundreds of thousand-pager books every day mandates that each book needs to be verified by 5 different editors and if an editor catches errors, they fix them and forward the book to the next editor, meaning, each book needs to be forwarded 4 times. The company then realizes that this rule is costly, but doesn't want to reduce the quality of their brand. The solution for this problem would be to use the LZW compression algorithm before forwarding the documents and the editor who receives the compressed file would decompress it to edit. The example input ebook will be [Emma by Jane Austen](#) in .txt filetype, which has a size of 863KB. The program takes one input file which is and creates one output file, where both files need to be of .txt filetype.

Name	Date modified	Type	Size
 book	27.12.2022 09:00	Text Document	863 KB

An editor has the book.txt file, in order to be forwarded efficiently to another editor, the book file needs to be compressed.

1- Compilation

```
C:\Users\Hasan Kayra Mike\Documents\Everything\3.Donem\LZW>gcc lzw.c -o lzw
```



2- Program Call

```
C:\Users\Hasan Kayra Mike\Documents\Everything\3.Donem\LZW>lzw book.txt
```

3- Command Selection

```
Operation      Command
Compress       c
Decompress     d
>>> c
```

4-Output File

Name	Date modified	Type	Size
 book	27.12.2022 09:00	Text Document	863 KB
 compressed_book	15.01.2023 02:37	Text Document	398 KB

compressed_book.txt is the compressed file that the editor is going to forward. Let's take a look at the path that the editor, to whom the compressed_book.txt file is forwarded.

The editor must hold or have access to the original dictionary to be able to decompress the file. So the compilation part doesn't change.



1- Compilation and the Program Call

```
C:\Users\Hasan Kayra Mike\Documents\Everything\3.Donem\LZW>gcc lzw.c -o lzw
C:\Users\Hasan Kayra Mike\Documents\Everything\3.Donem\LZW>lzw compressed_book.txt
```

2- Command Selection

```
Operation      Command
Compress       c
Decompress     d
>>> d
```

3- Output File

Name	Date modified	Type	Size
 compressed_book	15.01.2023 02:37	Text Document	398 KB
 decompressed_book	15.01.2023 02:46	Text Document	879 KB

The LZW algorithm reduced the size of the book before forwarding by 53.88%. This will allow the company to cut costs without having to sacrifice the quality of work.

Project Video:

- 1- https://youtu.be/RkvzTjN-w_s

References:

- 1- <https://en.wikipedia.org/wiki/Lempel%E2%80%93Ziv%E2%80%93Welch>
- 2- <https://cs.stanford.edu/people/eroberts/cs201/projects/1999-00/software-patents/lzw.html>
- 3- https://ethw.org/History_of_Lossless_Data_Compression_Algorithms#LZW
- 4- <https://www.cs.columbia.edu/~allen/S14/NOTES/lzw.pdf>
- 5- <https://www.cs.columbia.edu/~allen/S14/NOTES/lzw.pdf>
- 6- <https://www.cs.columbia.edu/~allen/S14/NOTES/lzw.pdf>
- 7- <https://en.wikipedia.org/wiki/Lempel%E2%80%93Ziv%E2%80%93Welch>
- 8- https://rosettacode.org/wiki/Talk:LZW_compression

External Links:

- 1- https://courses.cs.duke.edu/spring03/cps296.5/papers/welch_1984_technique_for.pdf
- 2- https://courses.cs.duke.edu/spring03/cps296.5/papers/ziv_lempel_1978_variable-rate.pdf
- 3- <https://youtu.be/FvWQsuwbraA>
- 4- <https://youtu.be/KJBZyPPTwo0>
- 5- <https://github.com/jefftime/lzw/tree/master/src>
- 6- <https://patents.google.com/patent/US4558302>