# EE 565 Final Project: Lossless Text Compression

Yihao Xia (yihaoxia@usc.edu)

Apr. 2021

## 1 Introduction

In this project, both Huffman coding and Lempel-Ziv-Welch(LZW) compression algorithm are implemented to handle the lossless text compression task. For LZW comporession algorithm, I explored the effects the size of dictionary on the compression performance. Besides, I also implemented LZW + Huffman Coding which compresses the original text by using LZW and then further compresses the successive LZW codewords by Huffman Coding. The compression performance in terms of both compression rate and compression time are compared for both LZW and LZW+Huffman compression. One finding is that the LZW+Huffman have better compression performance when the size of LZW dictionary is relevantly small ( $< 2^{13}$) Specifically for the compression rate, the best ones get from LZW and LZW+Huffman are 0.410 and 0.513.

## 2 Implementation details

### 2.1 Huffman coding

The Huffman code is an optimal prefix code The random variables with lower probability would be encoded by a longer codeword [1]. In this text compression task, the characters in the text is treated as random variables. To obtain the characters' binary codewords, we first generate a Huffman tree, where the characters are assigned according to their frequencies in the text. Specifically, in the binary coding case, a binary tree would be generated. The two characters with lowest frequencies are firstly selected and assigned to two nodes at the lowest level of the tree. Then, we can combine the two nodes by connecting them with a parent node. The frequency associated with the parent node is the sum of its children's. By recursively combining nodes, a Huffman tree would be constructed in a bottom-up way. Within the Huffman tree, the Huffman code of a node is the concatenation of parent node and alphabet corresponding to the child node. The binary Huffman codewords of each character in the text are concatenated and then encoded as ASCII characters. The Huffman coding dictionary containing the pairs of character and codeword is text specific. The characters together ASCII encoded codewords are also saved in the compressed file. In the decompression phase, we first readout the dictionary part and then the compression codewords sequence of the and convert back to binary code. Then the original text can be recovered by scanning through the compression codewords, finding matched binary codewords in the Huffman dictionary, and concatenate the corresponding characters to be the original text.

### 2.2 Lempel-Ziv-Welch(LZW) Compression

The repetitive elements of a text include not only characters but words, phrases, or even sentences. The Lempel-Ziv-Welch compression algorithm [2] could recognizes the general repetitive elements and adaptive constructs the dictionary. In this project, I also used the LZW method for text compression. The LZW is one of the most popular version of LZ78 [3]. In my implementation the LZW dictionary is initialized by the extended by the extended ASCII table (256 code pairs). With the initial dictionary the whole text could be compressed in a greedy way. As scanning through the text, the longest string ($s$) held in the dictionary would be encoded by the corresponding codeword. The longest string indicates that the string s is in the dictionary while the combination of string and the next character ($s + c$) is not in the dictionary. Then dictionary would adaptively extended by including the combination of string and the next character($s + c$). Theoretically, in the compression phase, we can add new string to the LZW dictionary until the text is fully scanned. However, in practice, the size of LZW dictionary is usually less than the maximum size of LZW dictionary. The size of LZW dictionary would affect both the compression time and the compression rate. A small dictionary takes less time for searching but may not convert the enough repetitive phrase. A large dictionary results in longer compression time but not always results in a better compression rate. The codewords for large dictionary require more bit number to represent, while long codewords are less likely to be repetitive results in codeword redundancy. In this project, the size of dictionary is set to be $2^m$. The successive LZW codewords is converted to $m$ bits binary codes. The concatenation of binary codes is then converted to ASCII characters and saved as the compressed file.

Different from the Huffman coding, the LZW dictionary would be generated in an adaptively way in the decompression phase as well. Thus the compression file only contains the successive LZW codewords. In the LZW decompression phase, we need to scan through the successive LZW codewords only once. Let $s$ denote the strings corresponding to the current codeword. Note that at the beginning of the decompression we can always found the string and codeword pairs according to the initial dictionary. Then as we move on to the next codeword, the codeword may or may not in the LZW dictionary. If it is not in the codeword we will add the $s' = s + s[0]$ in the dictionary and concatenate $s'$ the with the decompressed text, where $s[0]$ is the first character of string $s$. Otherwise $s' = s + c[0]$ where the $c$ is the string corresponding to the next codeword. Again we will add $s'$ into the LZW dictionary but concatenate $c$ with the decompression text. By repeating this process, we can get the decompressed text when the scanning is over.

### 2.3 LZW Compression with Huffman Coding

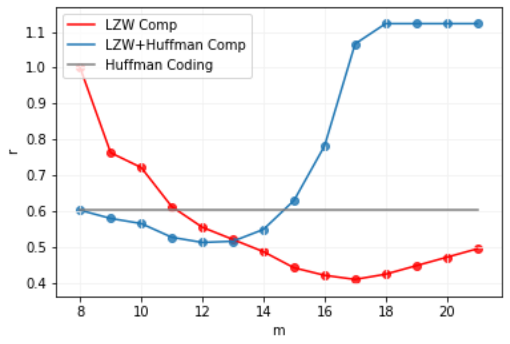
In a text, the frequencies of repetitive strings are usually different. Thus compressing the successive LZW codewords by Huffman Coding has the potential to further improve the compression rate. In this project, I also implemented this idea which is first use the LZW algorithm to encode the original text and then use the Huffman coding to encode the LZW codewords according to their frequencies.

## 3 Experiments and Results

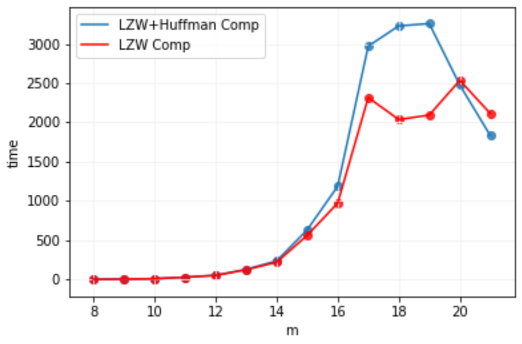
In the first experiment, I compared the efficiency of compression for Huffman Coding and LZW Compression. The efficiency of compression is indicated by the compression rate:

$$r = \frac{number\ of\ characters\ in\ compression\ text}{number\ of\ characters\ in\ original\ text}.$$

In Fig.1a the red curve shows relationship between the compression rate and different dictionary size. We can see that the compression rate first reduce, then reach its minimum 0.410 when $m = 17$, and then increase, as the $m$ increase from 0 to 20. In fact, the maximum size of LZW dictionary for *principia.txt* is $161019 \in [2^{17}, 2^{18}]$. When $m \geq 18$, the redundancy of binary codes would be the main reason leads to the decrease of the compression rate. The grey line in Fig. 1a indicates the compression rate of the Huffman Coding which is 0.593. We can see with a proper choice of dictionary size, the LZW compression would generally outperform Huffman Coding in terms of compression rate. While, the high compression rate of LZW compression is at the cost of high computation cost. The compression time is theoretically proportion to the dictionary size. In Fig. 1b, we can see that the compression time of LZW algorithm (red curve) is increase exponentially over $m$.



(a) Compression rate



(b) Compression time

Figure 1: Compression performance for Huffman Coding, LZW, and LZW+Huffman with different dictionary size

Then, I examined the performance of LZW compression with Huffman coding in terms of both compression time and compression rate. From Fig. 1(a), it is observable that the LZW + Huffman coding has a lower compression rate when $m \leq 13$. The lowest compression rate 0.513 is reached when $m = 12$. When $m > 13$, the larger LZW dictionary results in a larger the Huffman dictionary which would take large amount of space in the compression file. Thus the performance of the LZW+Huffman coding is worse. In terms of compression time, LZW compression and LZW+Huffman coding are generally the same. So here is the potential trade-off we can play. If the compression time is important we could choose LZW+Huffman coding, otherwise we use the LZW compression with a turned dictionary size.

## 4 Usage

All the compression and decompression code is tested under python2. Users can conduct compression and decompression by running following command.

**Huffman Coding**

```
python Huffman_encoding.py --input principia.txt --output compressed.txt
python Huffman_decoding.py --input compressed.txt --output recon.txt
```

**LZW Compression**

```
python LZW_encoding.py --input principia.txt --output compressed.txt
python LZW_decoding.py --input compressed.txt --output recon.txt
```

**LZW+Huffman Compression**

```
python LZW+Huffman_encoding.py --input principia.txt --output compressed.txt
python LZW+Huffman_decoding.py --input compressed.txt --output recon.txt
```

For LZW and LZW+Huffman compression, users can specify the dictionary size by add **--codeword_bit m** in the command. It sets the upper-bound of the dictionary size to be $2^m$.

## 5 Reference

[1] Huffman, David A. "A method for the construction of minimum-redundancy codes." Proceedings of the IRE 40.9 (1952): 1098-1101.
[2] Welch, Terry A. "Technique for high-performance data compression." Computer 52 (1984).
[3] Ziv, Jacob, and Abraham Lempel. "Compression of individual sequences via variable-rate coding." IEEE transactions on Information Theory 24.5 (1978): 530-536.