# Compression Report

Benjamin Russell, fdmw97

January 16, 2021

## Idea 1 - Using Lempel–Ziv–Welch (LZW) Compression

I am using LZW compression as the backbone of my compression algorithm for a number of reasons. The main reason is that I wanted to use a form of dictionary compression due to the repetitive nature of LaTeXfiles. By this I mean the use tags in latex files that are often repeated as well as if the LaTeXfile is focused on a specific topic then it is likely that there are words often repeated throughout the work. Another reason for my choice of LZW is the ease of implementation as well as the run-time performance. LZW is symmetric in the sense that the encoder and decoder both follow the same process (with some very minor alterations) and will build the same dictionary meaning the dictionary does not have to be written to the compressed file. The size of the proposed file to compress (approximately 200 pages) also makes LZW a suitable compression algorithm as it has been shown that the compression ratio asymptotically tends to the maximum as the source length increases[1].

## Idea 2 - The Dictionary

In my compression algorithm the dictionary is stored using a hash-table as this allows fast ($\mathcal{O}(1)$ on average) look-ups to the dictionary, so it is fast to check whether a string has already been added. I have also initialised the dictionary with the 256 different bytes as well as a list of common LaTeXcommands and all their prefixes - this is to avoid having to add all the prefixes of the commands during compression as this would result in worse compression due to having to encode all prefixes of *"\documentclass"* before being able to use the predefined code in the dictionary. Instead whenever a common LaTeXcommand is found in the source code it can be concisely represented as a single code straight away.

## Idea 3 - Compression Monitoring

One issue of using LZW compression is the fixed size of the dictionary and what to do when said dictionary eventually becomes full. The three main ways of dealing with this problem are to either continue using the current dictionary for the remainder of the compression, to reset the dictionary immediately, or to monitor the compression ratio after the dictionary has

filled up and reset if the ratio becomes to low. I chose to use the compression monitoring solution as it has the benefits of the other two solutions without the drawbacks of either. The benefits of compression monitoring being that if the source does not dramatically change then using the existing dictionary will still provide good compression performance when compared with immediately resetting the dictionary and having to build it again, and if the source does dramatically change then using the existing dictionary will reduce compression performance when compared with recreating a new dictionary. For my compressor I monitor the compression ratio until the dictionary is full and then monitor the new compression ratio, if $\frac{oldRatio}{newRatio} > 1.1$ then the dictionary is reset with the encoder outputting a clear code so the decoder knows when to reset its dictionary as well. This allows the encoder and decoder to adapt to a rapidly changing source file.

## Idea 4 - Variable length LZW Codes

The original paper for LZW compression[2] suggested using fixed width 12 bit codes for each phrase in the dictionary due to the added complexity of varying the length of the codes. I decided in my compressor to vary the length of the LZW codes as they are generated. This means that since the first 255 codes (byte values) can be stored as 8 bits that the next set of codes must use 9 bits etc. I have used variable length codes from 9-16 bits in my compressor as when testing it I found going above 16 bits harmed compression performance. It is obvious that variable length codes will increase compression performance as using a fixed 12 bit code for values that could be represented in less than 12 bits clearly introduces redundant bits into the encoding, I have therefore avoided this issue.

## Idea 5 - Huffman Coding

In order to further compress the encoding produced by my LZW implementation I then perform Huffman Coding on the bytes in the encoding. In testing this did produce an increase in compression ratio albeit small as it removed some redundancy from the encoding the increase wasn't large due to the bytes of the encoding having no real structure anymore. I did test performing Huffman Coding on the individual symbols produced by the LZW compression this produced very high compression ratios if the Huffman Tree was not being stored in the file. However since storing the Huffman Tree in the compressed file was required and would be infeasible to do due to the size of such a tree I was forced to only Huffman Code the bytes. I also experimented with adaptive Huffman Coding however this did not always improve compression ratio (sometimes it had a negative effect) so the added complexity of the encoder and decoder was not worth it.

## Idea 6 - Storing the Huffman Tree

In order to use the aforementioned Huffman Coding I had to somehow write the Huffman Tree to the compressed file in an efficient manner that didn't negate the compression gains from using Huffman Coding. In order to address this problem I first generated the Huffman Tree from the LZW encoding, then I sorted the generated codes in order of their bit-lengths and lexicographical order. This allowed for the codes to be regenerated as canonical Huffman Codes, this has the advantage that since producing them follows a standard algorithm based only on sorted bit-lengths it is possible to encode the entire Huffman Tree as a list of bit-lengths in lexicographical order of the alphabet (byte values). Since there are 256 different byte values the maximum length of any one Huffman Code can be 255 bits (due to it being a binary tree with 256 leaves, the most unbalanced tree has depth 256-1) this means that each code word length can be encoded as a single byte. This would add 256 bytes to the compressed file size. However due to many of the codes having the same bit-length it is possible to further encode this using Run Length Encoding. For this I followed a simple scheme I designed such that if a run is of length one the symbol is encoded as itself preceded by a 0. For runs greater than one the run is encoded as a 1 followed by the run length as a byte (maximum run length of 255) and the byte being repeated. This encoding was finally prepended with a 9 bit unsigned integer representing how many runs to decode and a byte representing how many trailing bits were added to the encoded message to keep it divisible into bytes.

## Required Packages

My encoder and decoder were both tested on Windows 10 using Python 3.7.4
The bitstring package is required to use them. It can be installed using the command:
pip install bitstring

## References

[1] J. Ziv and A. Lempel, "Compression of individual sequences via variable-rate coding," in IEEE Transactions on Information Theory, vol. 24, no. 5, pp. 530-536, September 1978, doi: 10.1109/TIT.1978.1055934.

[2] Welch, "A Technique for High-Performance Data Compression," in Computer, vol. 17, no. 6, pp. 8-19, June 1984, doi: 10.1109/MC.1984.1659158.