Describing MyLZW Optimizations

The LZW (Lempel-Ziv-Welch) compression algorithm is a hallmark in the study of data compression algorithms. It is a universal lossless data compression algorithm that was published in 1984. A lossless data compression algorithm is one that can take any file input and reduce the file's contents to a version that takes up less memory, while fully retrieving all the original information of the original file upon expansion.

Alas, to accomplish this feat of lossless compression, the algorithm makes and updates a codebook of codewords that serve as the actual message upon expansion. The codewords are of fixed length in terms of bits (0's and 1's) but can encode variable sized input words. The idea is that as we find and build more patterns in our codebook, we can compress large patterns into much smaller lengths of bits giving us a pruned file. Let's dig deeper into how it really works.

Primarily, we must start off by creating a codebook (the data structure to be used as the codebook is at the discretion of the implementer). We initialize the codebook to contain all possible single character values (ASCII) with the value of them being their ASCII integer. Until we reach the end of our input file, we do the following things.

First, we find the longest prefix of the input characters in our codebook. Then, we output the associated codeword to our compressed file. Finally, we take the largest prefix we found previously and add the next character in the file (input file) to it. This concatenated result is then added to the dictionary as a new codeword. The important part to highlight is that as we keep concatenating and building up these codebook strings, it is not the case that they take up the same amount of space. By giving larger prefixes a simpler/smaller codeword to represent them, we cut down on space significantly.

Furthermore, now that we have this compressed file, it’s time to prove it is lossless. Upon expansion, we expect that the expanded file is the same as our original uncompressed file at the binary level. This is true and the expansion algorithm details are explained in the subsequent paragraph.

We initialize, just as before, a codebook of all ASCII characters. Until the end of our compressed file, we do the ensuing steps.   We read the next codeword from the file and try to find the corresponding pattern in our codebook. We output this pattern to our expanded file. We then add this previous pattern plus the first character of our current pattern to our codebook.

Moreover, we can improve upon this algorithm which was the motivation behind optimizing the LZW algorithm for our second project. One of the major problems with this algorithm is that it uses a fixed width codeword size. This limits all the codewords in the codebook to be equal to 2 raised to the power of W, which is the length in bits of each codeword.

So, this means that we can only add so many patterns to our codebook. When dealing with large files, this codebook can fill up early on and make compression later in the file very poor. This is because, to efficiently compress, we ideally need more bits to store the information that does not have any similar patterns in our codebook. Remember once again, that there are no similar patterns in the codebook because we cannot add anymore, due to the codebook being full already. However, we worry not about fixing this issue but rather hope that compression will still be efficient. This is how “Do Nothing” mode works in MyLZW.java.

Likewise, another way to deal with the codebook filling up is to decide, at that very moment when it is filled, to reset the codebook back to the original table of ASCII characters and values. This is another optimization that we make so that we can go back to finding and adding new patterns to improve our compression. This is the thought process behind “Reset” mode in MyLZW.java

In contrast, what if we decided to reset the codebook only if the compression ratio is worse than a set threshold? The compression ratio is the size of the uncompressed data that has been processed/generated so far divided by the size of the compressed data generated/processed so far. Let’s decide that if our uncompressed data is more than 1.1 times the size of our compressed data while compressing, we will decide to reset the codebook. This is the way “Monitor” mode works in MyLZW.java.

Accordingly, let’s consider the implementation details for MyLZW.java that allowed the optimizations to come to fruition. I worked on the expand method and compress method separately instead of updating them concurrently. So, from here on out, we shall delve into the details explaining what I did in each method.

To allow for the mode to be chosen in my main method, I decided to use a for loop that converted the second argument in the args array to be a char. Formerly, the requirements for this project wanted us to change the fixed-width codewords that were size 12 to be size 9. I updated this bit-length variable W as such. Then, since L represented the number of possible code words (2 to the power of W), I updated that to be 512. The variables were originally final variables since these numbers were not supposed to change. Now that it’s a variable-width W and therefore L, we must allow their values to be updated. These changes affect the entire file, however, the next few paragraphs are changes that occurred within the compress method of MyLZW.java.

Of utmost importance, we must write the mode that was used when compressing directly into our compressed file at the beginning of the method. Then, we go into the loop that checks if our input string's length is greater than 0. Since we need to check for compression ratios later on, we must maintain a counter of compressed vs uncompressed bytes. We set our uncompressed bytes to be the length of the longest prefix of the input string we have in our codebook. The compressed bytes are equal to the bits we are writing into our compressed file (W) divided by 8 so that we get bytes.

Next, we check whether the longest prefix of the input word is less than the actual input word. This is where we make new codewords. This is also where most of the code for this method must be updated and where our different modes come into account.

When the aforementioned clause is triggered, we will first check whether we have enough space to add more possible codewords. If this is the case then we have a simple solution: just add the codeword to our codebook and update our codeword counter!

Instead, if the codebook is full and the bit length is less than the maximum of 16, we resize the codebook and allow the next codeword to be one bit longer. This means we have to multiply our counter for the maximum number of codewords that we can make by a factor of 2.

After, we check if reset mode was chosen and so this means that if we reach this statement that the codebook is full and the user wants to reset the codebook so we just reinitialize the entire codebook.

Lastly, if we go past the last if statement and we are in monitor mode, we trigger a whole bunch of code. Within this if clause, the first time that it is ever triggered we calculate the original compression ratio and store that. On subsequent iterations (remember this is all inside a while loop) we check the current compression ratio and compare it against the original. If the original compression ratio is 1.1 times bigger than the current one, we reset the codebook. The manner of resetting it is the exact same as it was before when reset mode was triggered. This is the last bit of optimized code added to the compress method.

In contrast, let’s look at the optimization implementation details of the expand method in MyLZW.java. The first thing to change in this method is the size of the array that holds the codebook which must be equal to 2 to the power of 16 (maximum number of codewords we can have).

After this, we need to know which mode was used to compress the file in the first place. So, the first thing we do is read the first character of the file which is the mode used, since the mode is the first thing written out to the compressed file in our compressed file.

Surprisingly, all the major optimization work happens in this method within the while loop. We start off by keeping variables to count the number of bytes that were compressed and uncompressed. After this we check when our codebook to expand our file is full. If this is the case, then do the same things as in the previous method. We either expand our bit length that we are writing out, we enter reset mode, or we are in monitor mode. The code that goes into these different if statements is virtually the same as in the compress method. One important thing to mention is that when resetting the codebook, we must use an array for our codebook and keep it’s size at 2 to the power of 16. The final important thing to mention is that all of this code must be written before we write out anything to our decompressed file.

To conclude, the optimizations that I made for this project involved being able to resize the codebook, have variable-width codewords, and allow to monitor the compression ratio so we know when to restart. The motivations for these optimizations were to see whether we could reduce the size of our compressed output files which based on further testing, is reasonably true.