Project 2 Design Document

LZW is a simple compression algorithm designed by Abraham Lempel, Jacob Ziv, and Terry Welch. LZW achieves compression by building up large patterns of characters, such as words or the encoding of white in an image, and representing them as numbers with a smaller number of bits. For example, the word “word” is 4 characters, which could range between 8 and 16 bits to represent. LZW would take those characters and represent them with a number, such as 633, as the “code” for those 4 characters. This would allow “word” to be stored using only as few as 10 bits, in comparison to the minimum of 32 to store them without compression. In its most basic form, LZW picks a set number of bits to compress each character pattern down to. My changes to LZW allow the number of bits to vary, allowing for more patterns to be stored if needed. More patterns allow for longer patterns to be created, which leads to better compression. Also, adding the ability to reset the codebook when it fills and to reset if compression is not efficient allows for even better compression than fixed bits and not resetting.

First, it is necessary to vary the width of the bits representing the pattern. Varying the pattern width allows for improvements on instances where LZW is inefficient. The first case is when the algorithm has built few patterns. When this occurs, many small patterns are present in the data structure storing the patterns, called the codebook. If the number of bits used to represent patterns is small, the codebook will quickly fill up with small patterns. No new patterns can be added once the data structure fills up, so longer, better, more efficient patterns are unable to be created. This causes compression to still occur, but not very well. The second case is if the number of bits is large. The data structure will be able to store very large patterns, and the potential exists for very good compression. However, that is only on files large enough to reach that point. If the number of bits is picked to be 32, the beginning of the algorithm, when small patterns are being created, will cause small patterns to be represented as more bits than they were before. If larger patterns are later built and used, this is okay, but if the file being compressed is small, that point is not reached, and compression will not be good.

In order to add variable width patterns, multiple changes are needed to the basic LZW code. If characters are being represented with 8 bits, such as with ASCII, then LZW should start with a pattern width of 9 bits. This allows for the minimum amount of space to be used at the beginning, causing the number of occurrences of representing 1 character with more than 8 bits to be at a minimum. When 512 patterns have been created and when the next pattern attempts to add to the codebook, it is time to increase the number of bits by 1. This increase of 1 bit doubles the number of patterns which can be stored and keeps the number of wasted bits to a minimum. After 1028 patterns have been added, upsize again. Each upsize adds 1 bit to the number of bits and doubles the number of patterns. Once 16 bits are used, it starts to get inefficient to store more patterns, so no longer upsize when the codebook is filled. This leads to a few options on what to do next.

Once the codebook fills, there are 3 options for solutions. Option 1 is to do nothing, and let the algorithm use the patterns it has already found to compress the rest of the file. This would most likely still achieve pretty good compression, provided that the file does not change the frequency of characters from earlier in the file. If, for example, the file switches languages, or switches to be a representation of an image, compression would no longer be good. It is now required to remedy the situation by resetting the codebook. The codebook can be reset in two ways: either instantly when it fills or when compression drops below a certain threshold.

Resetting the codebook instantly when full has a few pros and cons. After reaching the maximum number of patterns using 16 bits, a lot of patterns have been created. Most of the time, these patterns would be good enough to use on the rest of the file. However, in situations where the file type switches at an unknown time, these patterns would perform much worse. Resetting the codebook instantly when full allows this case to be more efficiently handled. However, it is not always useful to reset instantly. When compressing large text files, many large patterns have been created and will continue to be used. Resetting the codebook instantly causes these long patterns to need to be regenerated. Regenerating the patterns is a waste of time and space, and this case should be avoided. In addition, anytime the codebook is reset, the code for each letter in the alphabet must be regenerated.

The other option, monitor mode, allows for the case of long text files to be handled more efficiently. Monitor mode gets its name from the way it operates: it monitors the rate of compression. If the rate of compression falls below a set threshold, reset the codebook. The rate of compression is found by tracking a few things. Monitor mode tracks the number of bits compressed, the number of compressed bits created, and the ratio of the two. When the codebook fills, begin save the current ratio as the “previous” ratio, and keep this value saved until a reset is needed. Then, for each new pattern, compare the previous ratio to the current ratio, and if compression has degraded, reset the codebook and try to reach a higher ratio. If a reset is needed, preform the same reset operation as before, but also reset the previous ratio to 0. Using monitor mode like this keeps the compression rate from falling to subpar levels. However, the compression rate might not be as good as it can be. If the file type switches from text to an image, the compression rate might still be above accepted levels, but not at the best it can be.

After these additions to compression have been made, the expansion algorithm must also be adapted. The compression algorithm needs to do 2 extra things: increase the width of the pattern and reset the codebook at the same time as compression. To achieve this, it is a very similar process to compression. Track the number of patterns currently made, and if the number of patterns is at maximum, add 1 bit and increase the maximum. Once 16 bits have been used and filled, expansion needs to know whether to do nothing, reset instantly, or begin monitoring. To do this, compression will place a single character at the beginning of the compressed file, either ‘n’, ‘r’, or ‘m’. Expansion will read the first character separately from the algorithm and save which mode to use as a variable. Expansion must reset the codebook at the same time compression did, and to do this, it will monitor the same variables as compression. Expansion will reset at the same time, because the same number of patterns have been created, so checking to see if the codebook is full will sync compression with expansion. For monitor mode, it is important to update the number of bits processed and outputted at the same part of the algorithm as compression, such as right after retrieving and writing. Keeping these variables in sync between compression and expansion is crucial to create the same file after both algorithms are ran.

In summary, variable width patterns and resetting the pattern data structure will allow for better compression to be achieved. Varying the number of bits creates more efficient codebooks and resetting the codebook at different times allows for continued good compression. Implementing these changes in LZW will be beneficial to the algorithm.