LZW Compression is a lossless compression algorithm used widely for compression even today. Most applications today use a form of LZW compression because of its reliability and impressive compression ratios. In relation to other compression algorithms, in particular Huffman, LZW outperforms because it allows for the build up of prefix's in a dictionary so patterns in files can be condensed easily.

LZW Compression uses a simple technique of building a codebook, for compression and expansion, to keep track of patterns to help compress files more effectively. The initial task in using LZW is to create a data structure to hold all single character values at first. Each character in the data structure would simply map to its corresponding ASCII value to help keep track of location. From here, while the file is not empty, elements are read in from the file and used to match the longest preexisting prefix in the codebook. Then the matching prefix is outputted to the compressed file and it's size is expanded or compressed to match a certain bit length best for compression. For example, if "the" had already been entered into the codebook, instead of reading in each letter and outputting a 24 bit value, a 12 or 16 bit value can be used to output the same string. On the other hand if a single character is read in, it will still output a 12 bit value where as only 8 bits were read in. This is what allows for compression, but in certain cases where the codebook does not fill up as fast can lead to problems and actually cause expansion of the file. Once the prefix is outputted, the next character after it is read in from the file, added to the prefix and this new codeword is added to the dictionary. LZW compression is very good when long patterns build up in files and are reused, but when this doesn't happen it can lead to poor compression ratios and in some cases even expansion.

One way to fix this problem is to use variable length codewords. Instead of using a mandatory 12 or 16 bit codeword, we can start at 9 bits and work our way up when the dictionary becomes filled. The extended ASCII table will fill up a 9 bit data structure half way which will allow for 256 other codewords. When the dictionary is filled up, the bit size will increase by 1 and then therefore double the size of the dictionary. This allows for smaller outputs initially to help compression rates and as patterns build up, the dictionary size can increase as needed. To implement this in the project, first the final type of the width of the words needs to be removed so it can start at 9 and change as needed. Along with the width of the word size being changed, the dictionary size will have to be set to 2^W, where W is the width, to allow for resizing. For compression this is easily implemented by adding a check to see if the codeword being added is at the size of the dictionary, and if so the width of the codewords is incremented, the size of the dictionary is changed, and then the new codeword is added to the dictionary. For expansion this is a little trickier, since expansion is always a step behind because it reads in the codewords, the check to change the size will need to be when the last codeword is being added. When this is the case, the width of the codeword is incremented, size of the dictionary increased, and the new codeword is added. This variable length codeword allows for better compression times, but we want to be able to cut the increment of the width of the word off, otherwise it will continue to increase and become detrimental to the compression ratio. This can be solved by adding a check along with the codeword size, to see how big the current codeword size is. For this program we max our size out at 16 bits so as long as the codeword length stays below that no problems arise. However the case of what to do arises when the codeword size is 16 and a new codeword is trying to be added. In this situation we have three options; do nothing, reset, monitor.

In our do nothing mode we will simply, do nothing when the codebook fills up. This means when the codeword size is 16, and the dictionary is filled up we will just simply not add any more codewords to the book and continue to compress with what we have. This is easily implemented by just having the one check for dictionary size and width size. If both are at their capacities it simply outputs the codeword and continues with compression reading in more values. This same implementation is used for expansion so at capacity both compression and expansion are using the same codebook to encode and decode.

The next option we have is a reset case where when the codebook fills up it is simply reset so new codewords can be added. For this case a separate check will need to be added to the compression and expansion methods. A check for when the codebook is trying to add a new word and is currently filled, and the codeword size equals the cut off, so 16. When this check returns true the entire codebook and its relative variables are reset along with it. In this case W is set back to 9, the dictionary size is reset and the codebook is wiped. The now empty dictionary must be filled back up with the initial ASCII values and then the codeword trying to be entered before is entered as the first new codeword in this dictionary. For expansion the same thing is done, except again it is done a step earlier. Once the codebook fills up, it is reset and then compressed data is read in creating new codewords for the new codebook.

Our final option is a monitor option. In this case instead of immediately reseting the codebook, a ratio of how the compression is working will be kept and when it exceeds a certain threshold, the codebook will be reset. The way to calculate this ratio is by using the size of uncompressed data and dividing by the size of the compressed data. The size of the uncompressed data is tracked by using a variable and incremented by the length of the codeword times 8. The size of the compressed data is much easier because it is mandated so every time we write out a codeword of length W, the variable is simply incremented by W. After every read of the file, and write, and increment of the variables, the ratio is taken and used for comparison. When the codebook is filled the current ratio is stored in a new variable called oldRatio and then used to determine when to reset the codebook. If the oldRatio divided by the current ratio of compressing exceeds 1.1 in this situation, the codebook is reset to its initial state. This allows for the good compression because old codewords that are used often can stay in the codebook until they become too irrelevant and in that case the codebook can reset to store more common patterns being used. This same ratio is tracked through expansion to ensure the resetting of the codebook at the same time in compression and expansion.

While the LZW compression alone is very good, the enhancements made in this project will definitely help improve it. The addition of variable length codewords allows for smaller output at first so compression ratios don’t get out of hand in the beginning and then resizing of the dictionary as needed. The requirement of a simple if statement to check when the width of the word should be increased is very minor in the overall run time and implemented very easily. This also allows for the saving of space because the size of the codebook only gets as large as it needs. The 3 cases for what to do when the codebook fills up also only requires minimal code and maximizes space efficiency. In the do nothing case, no extra code is needed after implementing the variable length codeword code. In the reset mode, all that is required is another check for when the codebook is filled and the code width is at its max. At this point the codebook is reset, along with the width size and length of the codebook and the compression then continues as usual. Finally for the monitor mode, all that is added is a check for the reset code that will check the compression ratio at each step in the process. If it exceeds its limit, the reset code will be executed and the codebook and codeword length will be reset.