Writing Assignment 2: Subtitle

Compression and memory management are as crucial today as they were in the early computer age as space utilization has always been a limiting factor when it comes to what a computer can and can’t run. LZW compression developed by Abraham Lempel, Jacob Ziv, and Terry Welch in 1984 has become an extremely powerful and popular tool to make files smaller to the computing world. LZW can compress files to be many magnitudes smaller than their original form. LZW essentially outputs a compressed file using codewords implemented using a symbol table. The author’s version utilizes fixed-length codewords, this is can be optimized to drastically increase performance of our program.

The author’s LZW implementation, as well as the upgraded LZW, both rely on a symbol table to encode the patterns which will be used for both compression and expansion. From here on I will be referring to the authors implementation of LZW as LZW.java. LZW.java is initialized with a 256-character codebook with the ASCII table. LZW.java has a codeword width of 12, and a max codebook limit of 2W = 4096. LZW.java will then scan a file and read in the data while looking for patterns. The file is scanned to find the largest prefix that can be found in the codebook, this prefix is then concatenated with the next free character, this happens each iteration. The discovered patterns are added to the codebook and the codebook index is incremented once. Any matching pattern found later will be encoded to the compressed file with 12 bits. Here we have a problem though, because ASCII characters are 8 bits long and our codewords are encoding to 12 bits long, we are essentially wasting memory space and usage. After, the codeword limit is reached, LZW.java stops encoding patterns and compresses the remaining file with the codebook. In the worst case (where the file has all new prefix patterns), the compression algorithm’s performance will suffer. A big limiting factor of the LZW.java is what occurs when the codeword limit is reached. Variable-width encoding is the solution to this problem. The variable length encoding will encode codewords ranging anywhere from 8 bits to 16 bits. We will call this modified version of the implementation MyLZW.java.

The act of implementing the variable-width encoding required for MYLZW.java presents a few problems regarding the current methods of compression and expansion. The first being what happens once MyLZW.java arrives at the codeword limit during compression. The solution, initialize MyLZW.java using codewords of length 9 bits with a codeword limit of 29 = 512 which will be incremented to 16 bits codeword length with a limit of 216 = 65,536. Once the initial limit is reached, the codebook length can easily be adjusted by incrementing it, then calculating a new codeword limit. That said, once the 16 bits limit is hit, the algorithm will either “do nothing”, “monitor” compression ratios based on user input, or “reset” the codebook. ‘Do nothing” mode entails that the remaining file will be encoded with the current codebook of 16 bit width. This implementation is essentially just LZW.java but with a much larger codebook.

“Monitor” mode causes the algorithm to compress as normal until the limit is reached, at which point it begins monitoring compression ratios. Each iteration from here takes the old and new compression ratios and determines whether or not to reset the codebook or use the current patterns. If the ratio between the old and new ratios is greater than 1.1, the codebook is reset. In “reset” mode, the codebook is rest back to its original settings once the limit is reached and remaining patterns are added to the codebook. This nicely addresses the worst case scenario of needing to encode entirely new patterns. “Monitor” mode functions as a check to “reset” mode, preventing “reset” from potentially clearing common patterns what had already been encoded.

To start, I will abstract out the creation and initialization in the compress and expand methods. My.LZW.java will be initialized in the manner I mention above with a 9 bits codeword length and a codeword limit of 29 = 512. My compress method will constantly check whether or not the codeword limit has been reached. Upon reaching the codeword limit, the mode is set to “reset” mode, the codebook will return to it’s initial configuration of 9 bits and a limit of 29 = 512. Should the mode be set to “monitor” mode, compression ratios will be dynamically calculated. Within this same block, another check will be made to see if the ratio compression ratios exceed 1.1. If exceeded, the codebook is reset, and the ratio is returned to 0. When the user specifies a mode, a single character will be inserted at the top of the compressed file to represent the selected mode. This character is used by the expand method to determine which expansion method to use. The expansion method will begin to decode the compressed file. Once the codeword limit is reached in “reset” mode, a new symbol table is initialized with the same starting configuration we have used before, an initial codeword length of 9 bits and a codeword limit of 29 = 512, from here decoding will continue. In “monitor” mode, the ratios will be calculated similarly to in the compress method. Finally, a simple change to the command line arguments will allow the user to choose the mode of compression.

MyLZW.java shouldn’t require any enormous changes to the current LZW.java to meet the requirements. The overwhelming majority of my implementation remains largely the same as the author’s with the acceptation of, the addition of more modes of compression, the addition of variable-width encoding and an altered initial codebook configuration. MyLZW.java will have significantly improved performance over the now obsolete LZW.java. Thank you for reading my piece.