In the world of computers, smaller is better. The need for compression and memory management was crucial in the early computer age as space utilization of even several kilobytes determined whether a computer could even run. LZW compression – a popular compression algorithm developed by Abraham Lempel, Jacob Ziv, and Terry Welch in 1984 – brought to the computing world an extremely powerful tool to make files smaller. It essentially outputs a compressed file, sometimes magnitudes smaller, as a series of codewords implemented via symbol table. The authors implementation uses fixed-length codewords. This presents an opportunity for optimization and performance increase that will be discussed in this software design document.

The current LZW implementation, as well as the enhanced LZW implementation, rely on a symbol table that encodes the patterns to be used for compression and expansion. The current implementation referred to hereafter as LZW.java is initialized with a codebook of 256 characters represented in the ASCII table, a codeword width of 12, and a maximum codeword limit of 2W = 4096. LZW.java will scan a file and start reading in data while looking for new patterns. Every iteration, the file is scanned to find the largest prefix that can be found in the current codebook and then concatenated with the next available character. The new pattern is then added to the codebook with the current codebook index incremented by one. Any matching pattern will be encoded to the compressed file with 12-bits. As ASCII characters are only 8-bits long but the codewords are encoded as 12-bits, this algorithm essentially sacrifices initial memory usage for future efficiency. Once the limit is reached, LZW.java will stop encoding new patterns and compress the remaining file with the current codebook. In the worst case where at that point the file has completely new prefix patterns, the compression algorithm will start to suffer in its performance. Already it can be seen that a limitation of this implementation would be what happens after the codeword reaches its limit. A solution to this problem would be to implement variable-width encoding where the algorithm can range anywhere from 8-bit codewords to 16-bit codewords. This implementation will be referred to hereafter as MyLZW.java.

Implementing variable-width encoding, while not requiring a lot of code itself, presents numerous challenges to the current compression and expansion methods. The first challenge is what to do once MyLZW.java reaches the current codeword limit in compression. MyLZW.java will be initialized using 9-bits codeword length with a codeword limit of 29 = 512 and incremented ultimately to 16-bits codeword length with a codeword limit of 216 = 65,536. Once the codebook reaches its initial limit, the codebook length can be easily resized by incrementing the codeword length, calculating the new codeword limit, and proceeding from there. Once the codeword length of 16-bits limit is reached however, the algorithm will either “do nothing”, “reset” the codebook, or “monitor” the compression ratios based on user input. In “do nothing” mode, the remaining file will be encoded using the current codebook of 16-bit width. This implementation is not difficult as it is essentially LZW.java but with a larger codebook. In “reset” mode, the codebook is reset to its initial configuration once the limit is reached and any subsequent patterns from that point are added to the codebook. This clearly addresses the worst-case scenario of having to encode completely new patterns but with the drawback of potentially clearing common patterns that were used and encoded earlier. In “monitor” mode, the algorithm will compress normally until the limit is reached and begin monitoring the compression ratios. Every iteration from that point will take the ratio of the old and new compression ratios to determine whether to reset the codebook or use the current patterns in the codebook. If the ratio passes the threshold ratio of 1.1, the codebook is reset. This addresses the problem that the “reset” mode could potentially present.

First, the symbol table creation and initialization in both the compress and expand method will be abstracted out. MyLZW.java will be initialized using the initial configuration of 9-bits codeword length and a codeword limit of 29 = 512. The compress method will continually check whether the codeword limit has been reached. When the codeword limit is reached and the mode is set to “reset”, the codebook will be set the initial configuration mentioned above. If the mode is set to “monitor”, it will start dynamically calculating the compression ratios. Within that block, another conditional will check if the ratio of compression ratios exceeds the max compression ratio of 1.1. If it does exceed 1.1, the codebook will be reset and the ratio will be reinitialized to zero. When the user specifies a mode, the algorithm will insert a single character representation of that mode at the top of the compressed file. The expand method will read in a single character to determine which mode to use. The expansion method is essentially whatever mode that was chosen to compress but in reverse. The expand method will take the compressed file and start decoding. Once the codeword limit is reached in “reset” mode, a new symbol table will be initialized with its starting configurations just as it was in the compress method and will continue to decode from there. In “monitor” mode, the ratios will be dynamically calculated similar to the compress method. The calculations of these ratios are coordinated in the same exact way as the compress method and so these two methods should work if these modes are implemented in the same logical order. Finally, a change in the command line arguments will allow the user to specify the mode for compression.

MyLZW.java will not require massive changes to the current logic of LZW.java. Most of the implementation remains largely the same with the exception of the additional modes, variable-width encoding, and the initial codebook configuration. In short, these changes will significantly improve the compression performance over LZW.java.