CS 1501 – Essay 2

Dr. Nicholas Farnan

10/8/2017

LZW compression is a universal lossless data compression algorithm invented by Abraham Lempel, Jacob Ziv, and Terry Welch. ‘LZW’ was named after the initial of last name of these three creators. The algorithm was published in 1980s, but it still impacts contemporary technology through the GIF, TIFF, and PDF file formats. Simplicity, lossless format, and straightforward understanding of LZW have supported its success throughout years.

First of all, LZW compression algorithm relies heavily on a symbol table. This table is initialized with all 256 ASCII characters and these string keys associate with 8-bit codewords obtained by value defining each character. We select one example file which we want to compress. To compress the file, we first find longest string in the codebook or symbol table; that is longest prefix of unscanned input. Then, that longest prefix is concatenated with neighbor character at the right side and the prefix is stored into the symbol table with associated the value or index. After previous step, we shorten input string and it moves past the encoded codeword by using Java substring method. We keep adding codewords to the symbol table until there are no more codeword values. As the process goes, we are actually writing the compressed data to a file. In a fixed-length encoding, the codebook size never exceeds 2n where n is the number of encoded bits and every codeword is only N-bits. But, we can start with 9 bit codewords and move up to 16 bit codewords, allowing 65,536 (216) codewords in variable-length encoding,

Thanks to LZW’s lossless quality, we can also expand the compressed data to produce the file which has same size and content with original file. This feature is very useful for GIF, TIFF, and PDF format. To expand, we first build a symbol table that associates character strings with codeword values and this is the inverse of the symbol table used for compression. For example, if we read the value 81, then we search for the string (character) in the symbol table associated with 81, write that string of character to the newly expanded file, and concatenate the first character of that string to the previous string read before. This new appended string is inserted into the symbol table. Now, another symbol table can be used for the expansion of the file. For project 2, some changes are made in compression and expansion.

Textbook author’s LZW.java uses fixed codeword width of 12-bit. To enhance this, MyLZW.java will start with 9-bits up to 16-bit codewords. Variable-length encoding is very effective in smaller codebook, allowing for not only fewer bits written and efficient compression but also larger capacity to hold more codewords which can help boosting compression later. Some other changes to LZW.java are reset and monitor modes. In reset mode, once codebook is filled up, the codebook (symbol table) should be reset back to the initial state so that new codewords can be added. This allows new codewords to be more stored which can potentially lead to compressing the file further. Also, reset mode can delete already added codewords and it may remove most common words or phrases in the file. Monitor mode initially compress as normal as ‘Do nothing’ mode but it also begins monitoring or recording the compression ratio of two ratios (old compression ratio and new ratio) of the file. Compression ratio is generally calculated by dividing size of the uncompressed data that has been processed by the size of the newly compressed data generated. If the ratio of two ratios exceeds a specific threshold (1.1) or the old compression method result in a larger-sized compression file, we should just reset the codebook. These proposed changes look simple, but can lead to some improvements over the author’s LZW java.

Implementing these modifications is not easy. First, we should aim for the implementing variable-length encoding and then finish up ‘do nothing’ (already implemented by the author), reset and monitor modes. To implement variable-length encoding, we need to first get rid of the ‘final’ keywords for both variables *W* (codeword width) and *L* (number of codewords) in LZW.java so that number of bits and size of the codebook can change. The program checks if codeword value is less than the size of codebook which means it checks whether there is some space left to add more codewords. Since MyLZW.java uses variable-length encoding, it has to check if *W* (codeword width) is lower than maximum bit width (16 bits). If W is smaller, size of the codebook can be expanded so we increment W, calculate L again based on W, and place new codeword into the symbol table. However, monitor and reset mode should be triggered if *W* is equal to the maximum bit width. If the codebook size reaches absolute full capacity and *W* being equal to the maximum bit width, we check which mode user has requested.

If reset mode is on, MyLZW.java initialize new symbol table (object), remake the codebook, and reset some necessary values such as *W* to 9, *L* to 28 and code value to 257 (R+1). Then, we put new codeword into new codebook. For monitor mode, we calculate current compression ratio (old ratio) of the file by dividing uncompressed data size by compressed data size when codebook is very full with 216 codewords. Once monitor mode is on, we calculate new compression ratio, continuously updating the number of bits read and written. Then, we can divide the old compression ratio by the new compression ratio to get ratio of both ratios. If that ratio is larger than 1.1(set threshold), it shows that old codebook will make larger compressed file size than newly formed codebook will so we reset all necessary variables and symbol table and add new codewords to the table. Then we keep going through the loop with a newly created codebook. To calculate uncompressed data size, multiply longest prefix string length by 8 and add that onto the variable of uncompressed data size. For compressed data size, keep adding code width (*W*) which is number of bits written to the file.

There are some significant changes in the expression method of LZW.java. Because compression function stores user’s requested mode as the first character (bit) of the compressed file, expansion function has to read in character (bit) to determine which mode it has to perform. Stated by the textbook author, expansion is very similar to compression in modified program. If the codeword value is within the range of codebook size (maximum of 216 codewords), we read next codeword value, search for associated string of characters in the codebook, and append first character of the string to previous gathered string. Then, we add this concatenated string to the symbol table in similar way as we did in compression. For reset mode, it follows similar steps with one in compression method. On reset mode, we initialize new string array (symbol table) with all of the ASCII characters, assign *W* to minimum bit width (9 bits) and *L* to 29. On monitor mode, we calculate old compression ratio if the codebook is not full (codeword value is within bounds of codebook size) for every iteration in loop. If not, we calculate new compression ratio. Then, we calculate ratio of two ratios and check if it is larger than set threshold. If it is larger, we reinitialize new codebook and follow same procedures as compression.

Finally, we can modify the main function. To allow user to run the program with one of three options : ‘do noting’, ‘reset’, and ‘ monitor mode’, we have to check which character (‘m’, ‘n’, ‘r’) is used in command-line argument by using if statement. Then, we can store that character in global variable to be used by compression function. The modified LZW.java actually shows better results than non-modified LZW.java. According to specification of changes that need to be applied, I enable codewidth and codebook size to vary by changing variable into not constant, reset codebook and necessary variables for reset mode and calculate ratio of two ratios (old and new compression ratio) for monitor mode. With the proposed changes, this modified version of LZW.java will allow for greater flexibility in codewords through variable-length encoding. More effective compression and expansion options given by reset and monitor mode will guarantee the compressed data size to be smallest it can be.