William Kurek

Professor John Ramirez

CS 1501: Algorithm Implementation

# LZW MODIFICATIONS:

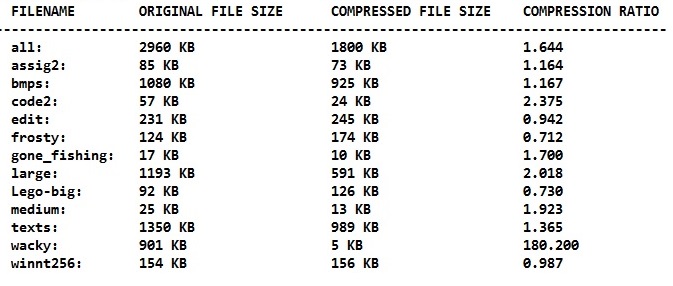
For the LZW algorithm to support streaming character input and variable length code words, various code modifications were necessary. The first modification I performed was removing the ‘final’ declarations from the variables which stored the number of codewords and the codeword width, allowing the variables to be updated throughout execution. Specifically, the variables W(the current variable codeword length) and L(the numbers of codewords) change consistently during runtime. Also, I added variables that store the minimum and maximum codeword widths.

Then, as stated in the assignment details, I replaced the Strings objects with StringBuilder objects, allowing the values to be updated more efficiently. This also required modifying the TST class to support StringBuilder. Also, rather than using the longestPrefixOf() method on a single string, I utilized the existing readChar() method in the BinaryStdIn class to read input a single character at a time, and appended this value to the StringBuilder object. By repeatedly appending characters and searching the prefix in the symbol table, the algorithm can find the longest prefix on its own.

Then, I added multiple conditions to the algorithm. First, in the compress() method, if all the codes were used for a specific bit and if the number of bits being written out is less than the maximum codeword width, the capacity for W bit codewords has been reached. Therefore, W is incremented by one, proceeding to the next codeword length. Also, L is multiplied by two since the number of codewords doubles for each added bit. Next, in the expand() method, I added conditions to check if the current codeword is greater than or equal to L, and if W is less than the maximum codeword length. If this statement is true, this indicates the table is full, and W is incremented by one and L is multiplied by two, as they were in compression.

**COMPARING VARIATIONS:**

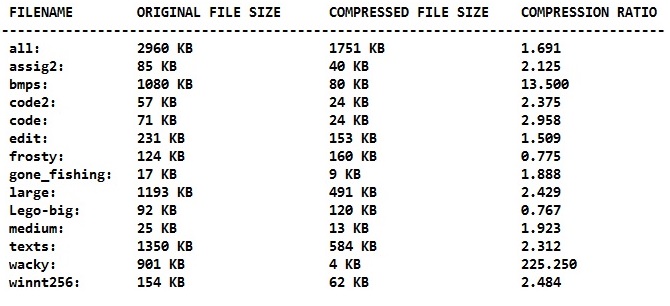
**LZW (unmodified):**



Of all the analyzed LZW variants, the original LZW algorithm performed the worst overall. Particularly, it seemed to perform at its worst on compressed files and images. These file types have a significantly worse compression ratio because they tend to contain less patterns, and LZW achieves optimal compression through searching for patterns. Therefore, there will be very long, repeated entries in the dictionary, which can result in almost no compression.

Also, because fixed-width codewords limits compression later in the process, the compression ratio suffers even further.

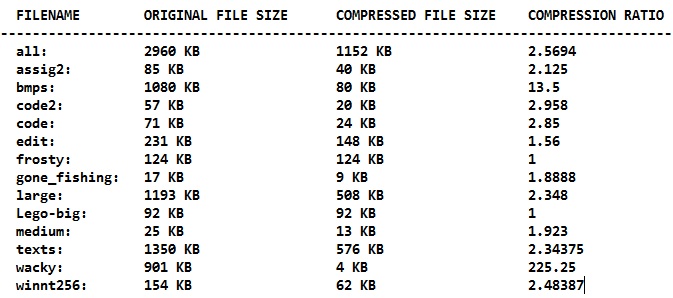
But, since this LZW variant benefits from repeating sequences of data, it performs well on text files.

**LZWmod (modified):** 

My modified version of the LZW algorithm performed well on almost all the test files. With variable-width codewords, we achieve better compression throughout the entire process. Smaller width codewords at the beginning of the process allows for better compression sooner, while larger width codewords later in the process allows better compression later. Since most of the test files are comprised of the same type of data, we can expect a lot of pattern matching as you progress deeper into the dictionary. Particularly, this version further increases the compression ratio of some text files and other uncompressed data types, such as bitmaps.

But, some of the files contain data which differs throughout the file, which can fill the dictionary early in the compression process. Therefore, when the algorithm attempts to process new data at the end of certain files, compression is significantly limited due to an inadequate dictionary. Particularly, this version also struggles with some compressed files, such as jpegs. For these reasons, even with variable length codewords, the compression ratios of files such as jpegs still suffer greatly.

**Compress (Unix):**



While my modified LZW algorithm achieved a better compression ratio on some test files, the predefined Unix compress program appears to be more reliable overall. Specifically, it could compress all the test files, unlike the other analyzed LZW versions. Even the image files, which increased in size with the other algorithms, could be compressed successfully. It would appear this version of the algorithm is able to predetermine or predict data types, avoiding unnecessary repetitions and lookups. For that reason, the Unix compress algorithm seems to perform best on compressed data types, such as jpegs.

But, it performs the same, if not worse, in comparison to my modified LZW algorithm, regarding large text files. This is expected, as the LZW algorithm, in any implementation, performs best on repetitive text files.