Joanne Chen

Professor John C. Ramirez

CS1501

27th June 2018

Initially, the LZW compression code reads the entire file as a single string, then compresses group of characters in this string. But if we have a very large file, it is possible that this large file could not fit into a single Java string. And the code uses substring(), which would generate a new String object every time looping through and cause lots of overhead. To solve this problem, I try to modify the code so that during compress the input file is read character by character and apply StringBuilder to add codeword to dictionary. Since I want to read the input file in a single character at a time, I create a while loop to check if I am reaching the end of the input file. First, I create a while loop to repeatedly reading and appending characters to StringBuilder until the current StringBuilder is not in the dictionary, to replace the method longestPrefixOf(). If the dictionary is not full, I add this StringBuilder that I just found to dictionary. Then I save the last character in the StringBuilder somewhere and remove it, as it is one that makes the codeword new to the dictionary. After that, I output codeword that is already in the dictionary and let the StringBuilder be the last character that I just remove. If there is a next character existed in the input file, we would keep going from the first step. Otherwise, we leave the loop and output the last character in the StringBuilder. In order to make the dictionary in which the StringBuilders are looked up, I also modify author’s TST so that they work for StringBuilder.

Next, I modify the code so that the LZW algorithm has a varying number of bits. It starts out using 9 bits for codewords, adding an extra bit when all codewords for previous size are used. For example, we use 9 bits for codewords 0-511, 10 bits for codewords 512-1023, etc. By doing so, we would get compression sooner as fewer bits are used earlier in the process and get greater compression later in the file as more bits are used later in the process. First, I change the number of codewords L to 512 and let the codeword width W begin with 9. Every time the dictionary is full, that is all the codewords have been used, I increment the codeword width by 1 and double the number of codeword. If the dictionary is full and the codeword width is already 16, we would not do anything but just keep the compressing as before, while simply stop adding new entries to the dictionary. And I would need to check if dictionary is full every time before finding the prefix that is not in the dictionary so that we have space to add this new entry to the dictionary. Like compression, decompression would do the same thing. But since decompression is one step behind compression, the simple table index actually be maximal L-1 rather than L.

And I give user the option to reset the dictionary via command line argument as a partial solution to the issue of the dictionary filling. The first command argument would tell if it is to compress or decompress. If there is a second command argument, which is “r”, it means we would like to reset the dictionary once all (16-bit) code words have been used. Then we would let the number of codewords L be 512 and the codeword width W be 9 again. Since this option will erase the entire dictionary and start rebuilding it from scratch, I need to re-initialize dictionary to single character words using their ASCII codes. To sync both the compress and decompress to reset the dictionary at the same point, I write the first bit in the output file as a flag to see if we need reset during decompressing. So before decompression, my program will read this flag and determine whether or not to reset the dictionary when running out of codewords. During decompression, the program would check if the dictionary is full and there is a need to reset every time before reading codeword in.