Jack Anderson

Sherif Khattab

CS 1501

5 November, 2018

Creating LZWmod

For Assignment 3, we were tasked with modifying Sedgewick’s LZW algorithm source code in a few key ways to strongly increase the algorithm’s performance. To start, the initial and first important change was modifying the algorithm to read in the input file as a stream of bytes one at a time as opposed to the author’s version of reading in the file as one large string. To accomplish this I utilized two different working attempts. Initially I attempted to use the java.util classes of InputStream and InputStreamReader, utilizing its ready() function, which tells whether the stream is still ready to be read as the condition for the while loop, and the read() function, casting it as a char. This was simple enough and worked efficiently, however ready() in practice would cut the loop prematurely by one character each time, and to solve this it was necessary to repeat the interior loop processes once more after the loop. This should be entirely unnecessary and wastes an enormous amount of space in the source code. Therefore, I opted to using the author’s BinaryStdIn with the readChar() function and isEmpty() function for the while loop instead, however the same issue occurs. While this may seem rather small and insignificant for some text files, this error could prove fatal for many other types of files, especially source code. I speculated that it may not be necessary to have the identical loop code run after the while loop for the last time, but without extensive testing I opted to keep it there in case of edge cases with the codebook. This workaround was fine, and allowed the algorithm to work flawlessly, however it was not ideal in terms of style or readability. However it was after completing the assignment that I realized that at this point in the algorithm, only one character would be left in the StringBuilder, and therefore the code for that character could just merely be printed out to the compressed file, simplifying things greatly.

Following this it was necessary to rework the entire rest of the compression algorithm to utilize StringBuilders instead of using Strings and the substring() method which incurred massive amounts of overhead and would slow the execution of compression by a large factor. To do this I first made a modified version of the TST class called TSTsb which accepted and utilized StringBuilders instead of Strings. From there, I was able to populate the dictionary with a StringBuilder for each of the solitary characters as per LZW. It is inside the while loop that the most change occurs. Here, instead of utilizing the aforementioned long string as the entire input and continously making substrings to move to the next location, I used one StringBuilder (“w”) where at the beginning of the loop the next character in the input stream is appended to it. Following this is a check for whether or not the dictionary contains the current state of the StringBuilder. If it does, the loop simply jumps back to the start again with a “continue;” and the next character is appended with the same check being made. If at some point the StringBuilder reaches a state that is not yet contained within the dictionary, the just-appended character is removed and the code for what remains in the StringBuilder is printed to the output file. Then the character is re-appended and added to the codebook as a new code word. Finally the StringBuilder is changed using the replace() method to only contain the appended character, and the loop is reset once more. This was all accomplished rather simply for the most part, with the only real questionable moments with the class used to read in the input as previously discussed.

Next was creating the ability of variable length codes, allowing for the program to run significantly faster than the base LZW provided. This proved to be a bit more of a challenge as both the compress() and expand() functions had to be modified so that they would work with increasing code lengths in sync. To do this with compress(), a check is made to see whether or not the current code word number in the dictionary exceeds the current maximum code words allowed. If it does, and the current length does not exceed the pre-defined maximum of 16, the size is simply incremented and a new maximum amount is calculated while the word is placed in the dictionary. The same is done for the compress() function at the same point in its algorithm to make sure that both work in tandem to ensure the correct data is maintained.

Lastly the problem of resetting the dictionary via command line prompt was tackled. To track this, I created a boolean variable that would be set to true if the second argument was “r”. When set to true, and utilizing the previous variable length code words, when the dictionary reached maximum capacity at max code word length 16, a new TSTsb is created and initialized as before, the current code number is reset, the code word length is set back to the minimum, and the max number of codes is recalculated. This is done in such a way in expand() as well so that both functions would reset the codebooks at the exact same time to preserve the integrity of the data. For the compress function specifically, the TSTsb class is not used, and instead an array of strings is used. It should be noted that in order to maintain this “reset” approach when expanding as well, a byte is written at the beginning of any compressed file. The char is either “n” if not using dictionary resets, or “r” if it is. Therefore in the beginning of the expand() function, the first character is read in and tested to see if it is “r”. If it is, then dictionary resets are used in keeping with how the file was originally compressed to maintain identical data.