LZW Compression Algorithm

Introduction

The LZW Algorithm is a widely used algorithm even today for many different processes. Two of these processes are the Unix file compression utility and what a lot of us have heard of, the GIF image format. The LZW algorithm was created in 1984 by Abraham Lempel, Jacob Ziv, and Terry Welch. It is also called the lossless data compression algorithm. This algorithm is very simple to implement, and it has potential for very high throughput in hardware implementations. The idea of LZW is that instead of using variable length code words to encode single characters, we can just use block code words to encode groups of characters. The more that can be represented by a single code word, the better our compression will be. For example, if we had a 24 bit word, and we were able to encode this entire word in one 12 bit code word, we will cut its size in half, thus using less memory.

Now we turn our focus on how exactly does this algorithm work. The compression (compress()) and decompression (expand()) are both building the exact same (code word, string) dictionary as they proceed. The main difference simply lies in their function. During compression, strings are looked up and code words are returned, so when we run “java LZW - <foo.txt> foo.lzw” we get some arbitrary, weird code word that only our computer understand. During decompression, these weird code words that we get and are stored in the “foo.lzw” file as stated from the example are then looked up by the computer and a string is returned to us. So basically the output of “foo.lzw” after decompression will simply be whatever is in “foo.txt” depending on the implementation of course and whether or not the programmer actually programmed the LZW properly.

It is also important to note that the LZW is an adaptive algorithm. This means that the compression and the decompression adapts to patterns as they are seen. However, we need to note that the decompression is one step behind the compression algorithm in building the dictionary. In this paper, I will discuss my proposed plan of editing and completely modifying our current LZW algorithm to make it so that the LZW algorithm varies the size of the output from 9 to 16 bits, increases the size of a code word when all code words of a previous size has been used up, and to make it so that when the code book is filled up that there are three options: the do nothing mode, the reset mode, and the monitor mode.

Tackling the Problem One Step at a Time

To begin with our modifications, we first had to make the algorithm vary from 9 to 16 bits. In order to do so, we just simply needed to change our code word width to the initial bit width that we wanted which was 9 bits. Since we made a change to the width, we then needed to change the number of code words as well which was simply 2 to the power of the width. Since the current width is 9, the number of code words would then be 512 words. Also, these variables for the number of code words and the width are not constant as since we are varying them, the bit width will increment up to 16 and the number of code words will increase in accordance to the bit width. In addition, I also plan to set constants for the maximum bit width, which is 16 and the max number of code words which is 2 to the power of 16, which then is 65.536 words.

If we wanted to increment the bit width, we would need to check if the current number of code words is equal to 2 to the power of our current bit width. If it is, then we also check if our current bit width is less than 16. If both cases apply and are true, then we increase our bit width by one and set the number of code words in accordance with the formula 2 to the power of the bit width.

Now that we have completed the first two steps: varying the bits from 9 to 16 and handled the incrementing of the bit width when all the code words have been used up, we now move on to the more complicated part. In this, when the code book has been used up (i.e. when the number of code words equals 2 to the power of 16 or 65,536 words), then we have 3 modes to choose from. For the first mode, the Do Nothing mode, in this mode, literally do nothing at all with it. Do not concern yourself with its implementation as all you need to do is literally nothing and to just worry about properly coding the other two modes. Just continue on with the implementation that is already provided in LZW.java.

For the next mode, the reset mode, let us go through the very basics. First of all, let us define the term reset. The term reset is best defined as to revert everything back to its original state. In coding that would mean to revert all variables to its initial value for which it was declared in. That means for this particular program, we need to reset our bit width back to 9, our number of code words back to 2 the power of 9 or 512, and finally our current number of code words to the state for it was initially declared in. In this program it was declared as to be the number of input characters, so 256 input characters plus 1.

For the last mode, the monitor mode, in this mode we had two different ratios, the old ratio and the new ratio. The old ratio was the ratio recorded in the program when we last filled the codebook. This ratio is the ratio that tends to change. The new ratio represents the current compression ratio. This ratio stays constant until the new ratio divided by the old ratio exceeds 1.1 in which case we basically reset. In order to even do any kind of ratio testing, we first need to make two double variables to represent the old ratio and the new ratio and set these variables to zero. Then in order to set the new ratio, my plan is to create a boolean variable that will allow us to set the new ratio and make it constant for a certain period of time. I would want to initially set this variable to true. Now we need to calculate the old ratio. To do this, we simply divide the size of the uncompressed data that has currently been processed by the size of the compressed data that has currently been processed. To calculate the size of the uncompressed data, which is measured in bits, we take the length of the string for which we are processing and multiply that by 8 bits as the length of a string is measured in bytes. To calculate the size of the compressed data, simply add it with the current bit width. Now just divide the uncompressed data and the compressed data together to get the old ratio. Make sure to calculate this entire old ratio on the outside of the monitor loop. Now that we have our old ratio set up, to set our new ratio, since from the first iteration our calculated old compression ratio is the same as our current ratio, we can just set our new ratio to be equal to our old ratio. As such, since we also want our new ratio to remain constant until the division of our new ratio by our old ratio exceeds 1.1, we can use our boolean variable and use that to say that if our variable is true, then we set our variable to false and our new ratio to equal our old ratio. By doing so, only the old ratio will see changes and the new ratio will stay the same. Once the division of our new ratio and our old ratio exceeds 1.1 then we simply reset everything. This is the same as the reset method except for the fact that in this kind of reset we also reset the ratios and our boolean variable as well.

Conclusion

Overall, in my project, I implemented my LZW.java compression in the same style as I had described it above in the previous section. Programming is done in baby steps. I started with creating multiple variables, some of which were constants, most of which were private through the entire class and could be incremented and changed as such. Then I went on to tell the computer to increment the bit width only when all the code words had been used up. Next I implemented each of the three modes: the Do Nothing Mode, the Reset Mode, and the Monitor Mode, where the do nothing mode had zero extra implementation, the reset mode simply involved resetting a few of the initial variables to its initial state, and the monitor mode compared ratios. I feel like the hardest part during implementation was trying to find the placement of where I should put each of my modifications and actually understanding every line of what the original LZW.java code did.

Even though many webpages tell us that the design of the LZW compression algorithm is very simple, and yes looking at just LZW.java, it does not look like much, but when you look over the amount of code that goes into creating the functions, a couple of which include BinaryStdOut.java and BinaryStdIn.java, they are massive. There is still a lot left that I have yet to learn about this algorithm and the entire creation of it, but for now, the entire implementation of this varying 9 bit to 16 bit LZW compression is very useful and an efficient algorithm to use in multiple processes.