CS 1501 Assignment 3: LZW Variable-Bit

By: Andrew Beers

Email: [acb75@pitt.edu](mailto:acb75@pitt.edu)

Modifications for Variable Bit

The first step I took into modifying the given LZW code into variable code width was to change it to statically 9-bit LZW. This was fairly straight forward as there was already a static LZW in place. After changing the variables around I removed the ‘final’ attribute on the variables to be able to change them in the new scheme. I then added the condition when compressing to the existing if structure of i( code < L ), basically if we’ve used up all the codes for this bit yet. If we have then go into an else if branch of if W (number of bits being written out ) is less than 16 (our arbitrary end number of bits ). Upon entering here we know that we are full of W bit codewords, so W is incremented by 1, and L is multiplied by 2 because the number of codewords doubles for each additional bit added. To expand under this coding scheme I similarly check on each time if the codeword we are it is L, if it is and W is less than 16. If it is less then 16 I know that the codeword I have is not correct because it was written with 1 more bit during the compress. Here I add an additional bit to my codeword then recheck to get the string and write it out. I then increase W and L similarly to compression. In both compress and expand, when W gets to 16, the algorithm proceeds to read / write in 16 bit blocks until the end of file, this is the option where the dictionary does not reset.

A particular part here that gave me trouble was realizing I needed to add a bit to the expand codeword when it was time to increase W.

Modifications for Dictionary Reset

The modifications made to implement the scheme for the dictionary reset include everything from the modifications for variable bit LZW implementations above. The difference here is when the dictionary gets completely full (W = 16 and code = L ) and in this case we want to clear out the dictionary. To do this, since we are using a TST of integers for the current dictionary, I instantiated a new TST and initialized it with all the 8 bit ascii characters. I then reset W, L, and code to their initial values as if we are starting over. Lastly I add whatever code word was going to go in next as the first codeword as my dictionary. When expanding, I find out when W = 16 and I = L and instead of reading in 16 bits, I read in 9 this time because I know that the next codeword I wrote in this case was 9 bits because I reset my dictionary. This was the part of this section of the assignment that gave me the most trouble, but once I thought about what was going on, it made a lot of sense to me. Then I reset all my variables (W, L, and code), and this time create a new array and fill the first 256 values with the ascii codes and add the current code word as 257. Then the algorithm continues as if it was at the beginning again.

Modifications for Monitoring

The modifications made here include all the modifications above, including for the dictionary reset and for variable bit LZW. The change here is that after we have filled the dictionary, this time instead of immediately resetting, we get a compression value then monitor. For this I added counting variables to count the bits being read in from the file and the bits being sent out, during both compression and expansion. This ratio is the compression ratio. The algorithm continues writing 16 bit codewords and well continue calculating the new compression ratio at every step. If the old compression ratio / new compression ratio = 1.1 (compression decreases enough) we then through out the dictionary. The issue for me here was making sure the compression ratios I had calculated were correct and that I was throwing away the dictionary at the same time in both compression and expansion.

Results

I have two results tables each with 7 rows and 17 columns, so fitting them into this word document would not have been easy to read, I have turned in the excel file with these graphs, and the data in them is what I base my results on.

Each individual algorithm had files it did better on and worse on. Individually, they are as follows.

1. LZW

While this algorithm was statistically obviously the worst of the five for my tests, it was slightly closer to the rest of the algorithms on text files than on binary files. This is expected because LZW does better with more patterns, and I would expect more patterns in a text document than a binary file. This version of LZW does the worst on images and zips/tars. This is also expected because on zips/tars the files are of a different type. This is bad for LZW because it is looking for patterns and since the files change from text to binary, there are going to be times when the data will have a substantially worse compression ratio, especially because this algorithm does not reset its dictionary. Also having static length code words doesn’t seem to help this algorithm out in any compression way, the only benefit may be that it is easier to code.

1. LZW-variable length code words (no reset)

This scheme seemed to perform well on most of the files. I anticipated this because most of the files retain the same type of data throughout the file so you can expect to get a lot of overlap as you get deeper into the dictionary. This algorithm performs poorly on files where the data in the beginning of the file is very different from the data at the end. This happens because we are not resetting our dictionary so when the dictionary is full and we get data we haven’t seen before, we cant use any of our long prefixes, or any benefit that LZW gives us.

1. LZW-variable length code words (reset immediately)

This scheme performs exactly the same as scheme 2 in many of the files because many of them are small enough that we do not fill up the dictionary. One file that it does do better on was largezzz.txt where the data switches from paragraphs to all letter z’s. In this scheme we don’t keep our text dictionary so that when we get to the z’s we see major compression because we can store many z’s as one codeword where scheme 2 is stuck with it’s original text dictionary.

1. LZW-variable length code words (monitor)

For all but one of the files (largezzz.txt) this scheme is the same as scheme 2 because we do not see the compression ratio drop below the critical value of 1.1 in any of the other files. However in this file I saw a better compression ratio for this scheme than any other scheme. This is expected because unlike scheme 3 this scheme will retain the plain text dictionary for a longer period of time and unlike scheme 2, this scheme will throw out its dictionary when it comes across the x’s both leading to this being the best scheme for this file.

1. Unix compress

This scheme was in my opinion the most dependable. While the LZW files failed on the images, in many cases making the files bigger, it seemed like unix compress somehow got a sense of what kind of file it was compressing so it could decide how it would go about compressing rather than blindly doing the same algorithm for every file type. Because of this it performs the best on already compressed files and stays very close, if not the same to the variable bit LZW algorithm in most text files.

Conclusion

I think the variable bit LZW algorithm would be the best if we added some checking of file extensions to see if files were already compressed. This way we can eliminate compressing the same file multiple times usually resulting in expansion of the file. This is the only area in my opinion where this scheme was worse than any scheme in this assignment.