Nicholas Tillmann

CS1501- Algorithmic Implementation

Sherif Khattab

Assignment 3 Write-Up

When first learning about Assignment 3, I knew from the start that there were a vast array of ways to approach this problem. We had the option of using multiple symbol table representations, including TrieST, DLB, or TST. With this in mind, I thought that it would be in my best interest to use TST, because we had previously used this file in a lab and the file had lots of pre-written methods that could help us tackle this assignment. Our first task in the assignment was to read the input file now as a stream of bytes instead of all at once. We were then also prompted to implement a reset feature, giving the user the option to reset the codebook. Overall, this project was very interesting and gave me a very good understanding of LZW and the difference compression efficiencies it has on a vast array of files due to entropy or other underlying factors.

The focus of project 3 was to more or less modify the LZW compression file so that it would also have a reset implemented along with being able to work on varying codeword lengths. By doing this we were expected to increase the efficiency of the algorithm. As discussed in lecture, the less entropy a file contains the better compression expected for that file. Both frosty.jpg and the lego.gif file gave some of the worst compression ratios for all of the files. This was most likely because both of these files have a higher prevalence of entropy, hence the reason for less efficient compression.

The reset implementation helped specifically with increasing the compression rate of larger files. Files which would fill the dictionary completely and require reset could see significant increases in the overall performance of compression. For smaller files on the other hand, I noticed that they were not affected. This was mainly because, if a small file did not fully fill the dictionary, a reset was not required hence the reason that both files were affected the same as the files despite the fact that they had a reset implementation or not.

LZWmod.java both with and without the reset had better compression ratios than LZW.java for almost all of the test files. This was most likely because the wa the file reads the input file now as a stream of bytes instead of all at once along with also having a dictionary reset implementation added.

It seemed that both the .gif files and the .jpg files had some of the worst compression as the original file ended up having a smaller size than the compressed file. Like mentioned earlier, this was most likely in part due to entropy. The Unix compression had no improvement over the original size of both the frosty.jpg and gif files as it had larger compressed files afterwards when compared to the original sizes. These file types were the only that clearly did not show any improvement when comparing LZWmod.java to LZW.java.

For the small files in general the LZWmod compression algorithm resulted in very similar compression ratios. Most small files had a compression of about 2-3x, which was overall pretty significant. However, nothing performed as exceptionally as the .bmp files.

In conclusion, the LZWmod files out performed the normal authors LZW file. For every file the compressions were more efficient than the normal LZW file, sometimes by extreme margins. This was in part because of having varying variable code length and also implementing a reset functionality. As expected, the dictionary reset helped files of large sizes sometimes, while we noticed it did not affect smaller sized files as much. When comparing the reset with no reset this is the information we could gage from the two columns. We noticed that .tar files seemed to compress a bit smaller with the reset functionality implemented. For the other files, more specifically the smaller files, rese didn’t influence overall compression ratios much, especially since most smaller files did not require dictionary resets.