For this assignment I modified Robert Sedgewick’s (the author) implementation of the Lempel–Ziv–Welch (LZW) compression algorithm. While the author’s code used String objects, a fixed length codeword width (12 bytes), and a single dictionary (of 2^12 codewords), my implementation utilized StringBuilder objects, a variable length codeword width (9-16 bytes in length), and a variable size codebook (which could grow beyond 2^12 codewords with an option to reset the codebook once all 65, 536 codewords were used.

For all implementations:

LZW works the best for homogenous or repetitive data. This is because as codewords are added to the codebook, it will match these and replace very long strings, which could take up many bytes, with very short codes which only take a few bytes. LZW doesn’t fair well with random or noisy data. This is mainly due to not being able to match long enough codewords more than a few times if at all. Expansion (larger “compressed” file size) can actually occur in files without much repetitive data due to writing data in byte width sizes greater than 8. All of the implementations write data in either variable byte width (9-16) or a fixed byte-width of 12 bytes.

The best file in all cases was wacky.bmp, which had a compression of 0.4-0.5% of the original size. This is because the file is fairly homogenous and the LZW algorithm can generate very long codewords which can be replaced with very few bytes.

The two worst files in all cases were frosty.jpg (1.292-1.4% of the original) and Lego-big.gif (1.312-1.381% of the original). Both of these files contained very little repetitive data and were the complete opposite to wacky.bmp in terms of noise. Since all have ratios greater than 100%, this means that the file was actually expanded by using LZW compression.

These three files were outliers among the test files, and will not be discussed for individual implementations.

LZW (Sedgewick):

While the author’s code works fine for small files, it is especially slow with large files and may not even work with particularly large files since the file is read in as a single string and may be too big. The other problem is that it calls the substring() method in order to generate new test code strings from the input file. Since JDK 7, this method will generate a new string every time the method is called, which is wasteful and very time consuming. As illustrated in Table 1, The author’s implementation does give a nice compression ratio for most files (x̅ = 0.735), but for the large files (e.g. all.tar) it can take a very long time to run. This excess time is due to the substring generation mentioned before and is a major drawback of using this code for practical situations. Compressing the file all.tar for instance took several minutes. The decompression however was very fast – as it was for all of the implementations.

Of the non-outlier files, code2.txt gave the best compression ratio (41.8% of the original) and edit.exe gave the worst ratio (1.061% of the original). Seeing as human written text usually contains repetitions, this was to be expected. Machine code, which was what the executable file consisted of, is more random and given that the author’s implementation doesn’t use codebook resetting or a variable length code width, I speculate that it simply ran out of codewords and because of writing in 12-byte width, ended up expanding the executable.

Overall this implementation of LZW was the worst of the four.

LZWmod without reset:

Compared to the author’s LZW, my implementation ran many times faster and was able to achieve similar, and in many cases much better compression ratios. The faster compression was due to using StringBuilder objects which are mutable.

Of the non-outlier files, bmp.tar gave the best compression ratio (7.3% of the original) and edit.exe gave the worst ratio (66.2% of the original). In general, this implementation had a compression ratio of around 43% for most files and did especially well with the large .tar files. This makes sense given the fact that there is bound to be repetition in much larger files or archives. This makes LZW an ideal algorithm for compression of such files.

Overall this implementation of LZW was the third best of the four.

LZWmod with reset:

Resetting the codebook in this implementation did have some effect but it was only noticeable on the largest files (e.g. all.tar was compressed by 38.9% vs 59.1% for non-reset). For most of the smaller files it made no difference and gave byte sizes equal to those without the reset. The noticeable exception were frosty.jpg and large.txt which gave slightly bigger ratios than those produced without codebook resetting. There was probably more similarity in the data after the reset point which is why the non-reset version did better. But without doing a second pass of the entire file, on average this version did as good as or slightly better than the non-reset version. The average compression (excluding the outliers) was 41.2%. This is compared with the 43% for the non-reset version.

Of the non-outlier files, bmp.tar gave the best compression ratio (7.3% of the original) and edit.exe gave the worst ratio (64.4% of the original).

Overall this implementation of LZW was the second best of the four.

Unix compress:

This implementation uses intelligent compression ratio monitoring which in some cases achieved even better results than all of the other implementations. In fact, by default, the compress program won’t produce a compressed file if using LZW causes an expansion (e.g. for frosty.jpg). I had to force the program to run and redirect the output to a file in order to see how much it would expand the file to.

Of the non-outlier files, bmp.tar gave the best compression ratio (7.3% of the original) and edit.exe gave the worst ratio (63.9% of the original).

Overall this implementation of LZW was the best of the four (average compression ratio of 41.1%), but not by much compared to my LZWmod with reset which gave identical results on all but three files.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Table 1. Compression Comparison of different LZW Implementations | | | | | | | | | |
| File | Original File | LZW | | LZWmod without reset | | LZWmod with reset | | Unix | |
|  | Size (bytes) | Size (bytes) | Ratio | Size (bytes) | Ratio | Size (bytes) | Ratio | Size (bytes) | Ratio |
| all.tar | 3,031,040 | 1,846,854 | 0.609 | 1,792,784 | 0.591 | 1,178,104 | 0.389 | 1,179,467 | 0.389 |
| assig2.doc | 87,040 | 74,574 | 0.857 | 40,041 | 0.46 | 40,041 | 0.46 | 40,040 | 0.46 |
| bmps.tar | 1,105,920 | 925,079 | 0.836 | 80,914 | 0.073 | 80,914 | 0.073 | 80,913 | 0.073 |
| code.txt | 72,351 | 30,980 | 0.428 | 24,545 | 0.339 | 24,545 | 0.339 | 24,545 | 0.339 |
| code2.txt | 57,701 | 24,138 | 0.418 | 20,517 | 0.356 | 20,517 | 0.356 | 20,516 | 0.356 |
| edit.exe | 236,328 | 250,742 | 1.061 | 156,410 | 0.662 | 152,234 | 0.644 | 151,111 | 0.639 |
| frosty.jpg | 126,748 | 177,453 | 1.4 | 163,790 | 1.292 | 171,173 | 1.35 | 163,789 | 1.292 |
| gone\_fishing.bmp | 17,336 | 9,278 | 0.535 | 8,963 | 0.517 | 8,963 | 0.517 | 8,964 | 0.517 |
| large.txt | 1,220,703 | 605,184 | 0.496 | 501,778 | 0.411 | 527,491 | 0.432 | 522,673 | 0.428 |
| Lego-big.gif | 93,371 | 128,973 | 1.381 | 122,494 | 1.312 | 122,494 | 1.312 | 122,493 | 1.312 |
| medium.txt | 25,407 | 13,197 | 0.519 | 12,531 | 0.493 | 12,531 | 0.493 | 12,531 | 0.493 |
| texts.tar | 1,382,400 | 1,012,179 | 0.732 | 597,846 | 0.432 | 590,317 | 0.427 | 589,697 | 0.427 |
| wacky.bmp | 921,654 | 4,302 | 0.005 | 3,951 | 0.004 | 3,951 | 0.004 | 3,952 | 0.004 |
| winnt256.bmp | 157,044 | 159,050 | 1.013 | 62,932 | 0.401 | 62,932 | 0.401 | 62,931 | 0.401 |