Julian Galbraith-Paul

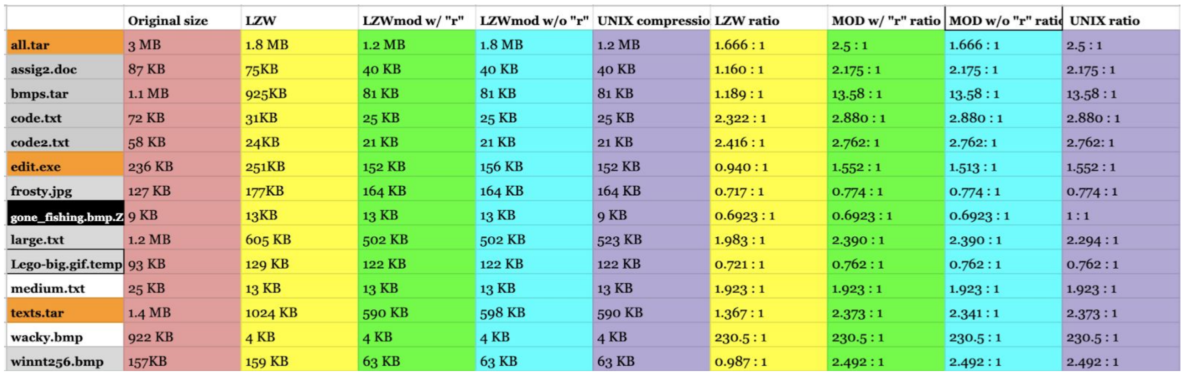
Sherif Khattab

CS 1501

11 / 03 / 20

LZW Compression

The pursuit of optimal runtime and space complexity is a crucial consideration in any application of computer science. It is inevitable that sizable files will engulf large fragments of memory, however, the addition of compression algorithms allows such files to be reduced to a fraction of the size; If a user has limited storage on their native device, such techniques could reduce said storage up to 230.5% (based on the *table 1* below). In this assignment, I implemented a variation of the Lempel-Ziz-Welch algorithm, an infamous execution of the classic compression concept. LZW loosely works by finding recurring sequences within a file and encoding (compressing) these sequences as smaller sets of symbols. These symbols are then decoded (expanded) in order to retrieve the original content. In correspondence with the project guidelines, I varied the bit width from 8-16 and implemented a reset flag(r), allowing for almost optimal performance across a variety of file extensions. Below are detailed comparisons between the file sizes of hybrid LZW and UNIX compression approaches (best and worst compression ratios are at the end).



	Original size	LZW	LZWmod w/ "r"	LZWmod w/o "r"	UNIX compressio	LZW ratio	MOD w/ "r" ratio	MOD w/o "r" ratio	UNIX ratio
all.tar	3 MB	1.8 MB	1.2 MB	1.8 MB	1.2 MB	1.666 : 1	2.5 : 1	1.666 : 1	2.5 : 1
assig2.doc	87 KB	75KB	40 KB	40 KB	40 KB	1.160 : 1	2.175 : 1	2.175 : 1	2.175 : 1
bmps.tar	1.1 MB	925KB	81 KB	81 KB	81 KB	1.189 : 1	13.58 : 1	13.58 : 1	13.58 : 1
code.txt	72 KB	31KB	25 KB	25 KB	25 KB	2.322 : 1	2.880 : 1	2.880 : 1	2.880 : 1
code2.txt	58 KB	24KB	21 KB	21 KB	21 KB	2.416 : 1	2.762: 1	2.762: 1	2.762: 1
edit.exe	236 KB	251KB	152 KB	156 KB	152 KB	0.940 : 1	1.552 : 1	1.513 : 1	1.552 : 1
frosty.jpg	127 KB	177KB	164 KB	164 KB	164 KB	0.717 : 1	0.774 : 1	0.774 : 1	0.774 : 1
gone_fishing.bmp.Z	9 KB	13KB	13 KB	13 KB	9 KB	0.6923 : 1	0.6923 : 1	0.6923 : 1	1 : 1
large.txt	1.2 MB	605 KB	502 KB	502 KB	523 KB	1.983 : 1	2.390 : 1	2.390 : 1	2.294 : 1
Lego-big.gif.temp	93 KB	129 KB	122 KB	122 KB	122 KB	0.721 : 1	0.762 : 1	0.762 : 1	0.762 : 1
medium.txt	25 KB	13 KB	13 KB	13 KB	13 KB	1.923 : 1	1.923 : 1	1.923 : 1	1.923 : 1
texts.tar	1.4 MB	1024 KB	590 KB	598 KB	590 KB	1.367 : 1	2.373 : 1	2.341 : 1	2.373 : 1
wacky.bmp	922 KB	4 KB	4 KB	4 KB	4 KB	230.5 : 1	230.5 : 1	230.5 : 1	230.5 : 1
winnt256.bmp	157KB	159 KB	63 KB	63 KB	63 KB	0.987 : 1	2.492 : 1	2.492 : 1	2.492 : 1

Table 1

Although the algorithm causes a substantial decrease in the file size, the basic LZW implementation underperforms vs. the UNIX compression in almost every case(orange and grey files in *table 1*). When reading each element as a chunk of 16 bits, we waste valuable space when codewords are under this threshold. Once varying the bit length from 8 to 16, based on a variety of different runtime specifications, our compression improves drastically. Even without the addition of the reset-flag, we see a significant enhancement over the baseline implementation derived from the Sedgewick textbook. The files highlighted orange and grey in *table 1* all experienced greater compression when varying the length of the codewords. This idea is best illustrated in the example file, “bumps.tar”. While the LZWmod compresses the file 13.58 : 1, the plain LZW only accomplishes a 1.189 : 1 reduction. This difference is clearly substantial, but we can still do *better*.

Next I added a reset flag, which when set via the command line, reset the width, size and symbol table (among other variables) to their initial size-preserving states. This caused the LZWmod compression to improve in several key cases. The orange highlighted files above in *table 1* denote three of the largest files we tested. In each example, we notice a sizable decrease in the compressed size for implementations with and without the reset flag. For example the file “all.tar” compressed down to a size of 1.8 MB without the r-flag and an impressive 1.2 MB with it; this yields a remarkable 150% decrease in total size. This improvement is due to the clearing of our symbol table as well as the re-initialization of the bit width and number of codewords. Once the bit width reaches 16, we reset our class variables back to their original values to optimize the compression (i.e from width 16bit --> 8bit). *Therefore reducing the size of our compressed file without affecting the actual content.*

Unlike the previous algorithms, the UNIX compression provides the best results in all of the cases except one. While this method uses a variation on LZW (Lempel-Ziv 77), this approach has clearly been optimized even further. This may be due some sort of monitoring of the compression ratio -- i.e strategically “resetting” once a certain ratio is reached.

When comparing the relative performance of each variation, interestingly enough, all four algorithms experienced the worst compression ratio on the *lego-big.gif.temp* and *frosty.jpg*. When attempting to compress these files I received ratios of .774 : 1 and .746 : 1 across all 4 algorithms. These sub-1 values indicate that the compression actually expanded the files rather than make them smaller(yikes!). This is because of large amounts of non-repetitive data within each file. With a greater quantity of unique sequences of data, more codewords are required to represent the information. More codewords means more symbols and space! Naturally, the inverse is true as well -- files containing repetitive data need fewer codewords to represent their contents. In my work, *wacky.bmp* resulted in an astounding 230.5 : 1 compression ratio across each of variations. This can be attributed to less variation in data, then fewer codewords can be used to represent the information -- fewer codewords, less symbols and smaller compression size.