Jack Anderson

Sherif Khattab

CS 1501

5 November, 2018

LZW Comparisons

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | **all.tar**  **2960KB** | **assig2.doc**  **85KB** | **bmps.tar**  **1080KB** | **code.txt**  **71KB** | **code2.txt**  **57KB** | **edit.exe**  **231KB** | **frosty.jpg**  **124KB** |
| **LZW** | 1804KB, 1.64 | 73KB  1.16 | 904KB  1.19 | 31KB  2.29 | 24KB  2.375 | 245KB  0.94 | 174KB  0.71 |
| **LZWmod** | 1751KB, 1.69 | 40KB  2.125 | 80KB  13.5 | 24KB  2.96 | 21KB  2.71 | 153KB  1.5 | 160KB  0.775 |
| **LZWmod (r)** | 1151KB, 2.57 | 40KB  2.125 | 80KB  13.5 | 24KB  2.96 | 21KB  2.71 | 149KB  1.55 | 168KB  0.74 |
| **Compression.exe** | 1152KB, 2.57 | 40KB  2.125 | 80KB  13.5 | 24KB  2.96 | 21KB  2.71 | 148KB  1.56 | N/A |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | **gone\_fishing.bmp**  **17KB** | **large.txt**  **1193KB** | **Lego-big.gif**  **92KB** | **medium.txt**  **25KB** | **texts.tar**  **1350KB** | **wacky.bmp**  **901KB** | **winnt256.bmp**  **154KB** |
| **LZW** | 10KB  1.7 | 591KB  2.02 | 126KB  0.65 | 13KB  1.92 | 989KB  1.67 | 5KB  180.2 | 156KB  0.99 |
| **LZWmod** | 9KB  1.89 | 491KB  2.43 | 120KB  0.77 | 13KB  1.92 | 584KB  2.82 | 4KB  225.25 | 62KB  2.48 |
| **LZWmod (r)** | 9KB  1.89 | 516KB  2.31 | 120KB  0.77 | 13KB  1.92 | 577KB  2.34 | 4KB  225.25 | 62KB  2.48 |
| **Compression.exe** | 9KB  1.89 | 511KB  2.33 | N/A | 13KB  1.92 | 576KB  2.34 | 4KB  225.25 | 62KB  2.48 |

Generally speaking, all four of the LZW algorithms performed closely to each other, though the differences that did occur are very important and in some cases they become glaringly apparent. Overall though, it can be definitively said that the original LZW code provided by the textbook by far performed the worst in every trial, sometimes by ratios as large as a 12 point difference from the others. For the files of assig2.doc, bmps.tar, code.txt, code2.txt, gone\_fishing.bmp, medium.txt, wacky.bmp and winnt256.bmp, the remaining three algorithms performed identically in terms of the compressed files size, with the textbook’s LZW performing worse (and sometimes far worse). I believe this can be attributed to the fact of the textbook code’s fixed codeword width of 12 bits. As described by the assignment details itself, “with this limit, the program will run out of codewords relatively quickly and will not handle large files (especially archives) well”. Moving on from that algorithm’s disappointing performance, examining the remaining three is much more interesting. For the files edit.exe, and texts.tar we see increased performance down the line of the remaining algorithms. This makes sense, as each version of LZW should have improvements over the others in some cases, as they have additional functionality. The base LZWmod features variable length codewords, LZWmod(r) has the ability to reset the codebook when full, and compress.exe features both of these as well as a monitor mode. However this is not always the case for other files, where sometimes the LZWmod performs better seemingly inexplicably over the others, as is the case with large.txt where it performs with a compression ratio of 2.43 over the LZWmod(r)’s 2.31 and Compression.exe’s 2.32. This could be explained by the fact that in some cases it may be advantageous to not reset the dictionary depending on the file structure. Much of the text may be very similar, leaving the original full codebook effective in continuing to compress the file where a reset offers less optimal results. Another interesting fact to note is the performance of all algorithms on the wacky.bmp file. Here we saw compression ratios as high as 225.25. This is easily explained looking at the file itself, which contains a large amount of white space. Repetitive features in a file allow for LZW in any form to very effectively compress where other compression algorithms may see worse performance.

Speaking of each algorithm individually and moving aside from wacky.bmp, the base LZW performed the best on the code2.txt file with a compression ratio of 2.37. There doesn't seem to be too much of a reason for this file specifically other than the speculation that this file was composed in a way slightly more advantageously towards working only with 12 bit size codewords and a non-resetting dictionary. The other algorithms of course performed better. The worst case for LZW (that wasn't on an already compressed file e.g. jpg or gif) seen was with the file edit.exe. There once again is no necessarily clear reason for this, other than the fact that an executable file may have a larger and more random assortment of bit patterns not conducive to the base LZW way of operating. LZWmod and the following algorithms performed their best runs past wacky.bmp on the code.txt file. This is again most likely due to the fact that code is inherently pattern-based in terms of there being set procedures that coders follow and the abundant reuse of variables and keywords throughout.

What originally may seem as shocking to the casual viewer is the performance of all algorithms, and the outright refusal of compression.exe to work on the frosty.jpg and Lego-big.gif files. In the first three algorithms we saw an increase in file size following the compression. To understand this, we must understand the file types of jpeg and gif to begin with. These two file types are already holding compressed data before they are run and therefore can not be compressed further with any version of LZW. In fact, the .gif file format is compressed with LZW to begin with. Knowing and understanding this, it is clear as to why further compression with the same algorithm would not lead to satisfying or productive results. All in all, the latter three versions of LZW performed about the same in the average case, with minute differences between them in specific instances. Truthfully, it may not be possible to know which version would work best on any given file without actually testing them or monitoring the compression ratio closely.