Jordan Carr

CS 1501

6/30/17

File Compression (and How to Do It Efficiently)

LZW is one of the most widely used algorithms to compress data. Its use of finding and replacing patterns with smaller numbers of bits makes it efficient for a variety of file types. However, implementation details can make its compression ratios and runtime vary greatly. Whether this be I/O details or code word size, these details make radical differences when choosing how to implement these algorithms. To explore how files compress, a presented LZW program, a modified LZW program, and Unix Compress were studied on a variety of files to explore how file size and type along with algorithm implementation can make critical differences in data compression.

Modifications to the LZW algorithm were primarily made to increase efficiency in runtime and compression. The main drawback for the presented solution was its means of taking in input: reading in the entire file as an immutable object with slices of it being analyzed at a time to find patterns. After each pattern was found it would be saved to the dictionary. To scan ahead to the next part of the file a new object would be created, the analyzed portion being thrown away while saving the rest of the file. As long as the size of the object was greater than zero patterns would continue to be found. However, the main issue with this implementation was its use of an immutable object to store the entire input file. That meant the entirety of the file was recopied to a new object every time the algorithm found a new pattern. This was extremely inefficient in terms of overhead and caused very poor runtime, especially for larger files.

To fix this two problems had to be solved: how to read in the file char by char in order to analyze chars and patterns one at a time and how to store patterns so they could be looked up in constant runtime. That being said, the implementation of the compression method had to be completely redesigned to accommodate these changes. The author’s input class was used to read in the file byte by byte, with each byte being converted into a char. Each byte was stored in a mutable object which could have chars appended to it as the patterns were built. Once a pattern was built it would be saved to a modified DLB, which has average constant runtime for searches and additions. In order to be able to store the patterns and the code words the use of terminator nodes was abandoned in favor of code words being stored in the nodes themselves, indicating a complete pattern if that field was not null. To look up the pattern the code would be returned if it was stored or null would be returned if it was not. To write the code to the file, the previous code that was searched would be stored in a temporary variable. The program would then append the next char and check if the new pattern existed in the dictionary. If it did not the previous code would be written to the file and the new pattern would be added to the dictionary until the end of the file was reached. Chars would be appended to the pattern until a new one was recognized. This change made a remarkable difference in terms of runtime and overhead, with the compression of larger files being reduced from several minutes to just a few seconds.

Variable length code words were also implemented in order to improve compression ratios. Previously the program only used fixed length code words of twelve bits. To improve compression code words that ranged from nine to sixteen bits were used in favor of this approach. Implementing this required reading in patterns until the maximum number of code words possible with that number of bits was reached, upon which the number of bits per code word was increased by one. This implementation had two advantages: fewer bits could be written for smaller patterns early in the file and more patterns could be saved later in the process, allowing for greater efficiency throughout the compression process. That being said, the expansion algorithm needed this modification as well. However, since finding patterns during decompression is one step behind that of compression, when checking when to expand the number of bits to read in from the compressed file, it would be checked if the latest possible code word plus one had surpassed the number of allowable code words. This allowed the change in reading in the correct number of bits to happen in sync with when they were written.

With these modifications in mind it is interesting to note the changes in compression ratios between the original LZW program, the modified LZW program, and to have a control program Unix Compress. A variety of types and sizes of files were used when testing the programs. In nearly all cases the modified LZW program and Unix Compress outperformed the original program. This is due to the varying number of bits per code word used to compress the file. In most cases the difference in ratios is clear. However, while saving quite a bit of memory for larger files it only saved a few kilobytes for smaller ones. In the case of assig2.doc and bmps.tar the amount of space saved is much more noticeable than some of the others. The ratio expanded from 1.1:1 to 2:1 and 1.1:1 to 13:1 respectively. In the case of assig2.doc, a medium sized file, this is probably due to the fact that fewer bits (9-11) were used when forming the earlier code words. Due to its size, it is unlikely that many as many 16bit code words were used, if at all. On the other hand, bmps.tar is a much larger file. It is very likely that it was able to take advantage of the greater number of code words later on in compression, saving a much larger amount of space. This same case holds for texts.tar.

In the case of winnt256.jpg, the original LZW program did not provide any compression at all. This is likely due to a large amount entropy and a lack of patterns for 12 bits to adequately compress it. This may explain why the file slightly expanded since more bits were used to write some of the smaller patterns. However, this problem was corrected when using the other two programs. The same can be said for the compression of the LZWmod file.

Two interesting cases to note are frosty.jpg and lego-big.gif. None of the programs were able to compress them at all. On the contrary, when the files were compressed the resulting files were larger than the originals. This is likely due to the amount of entropy being too great to be adequately compressed by the LZW algorithm. Varying the number of bits were only able to reduce the amount of expansion the compression algorithm did.

The most notable compression achieved by any of the algorithms was that achieved on wacky.bmp. It is a comparatively large file (901 KB) but was able to achieve a compression ratio of 180:1. These results can be explained by a low amount of entropy in the file. That being said, it is likely that there were a large amount of long patterns found within file. This made the compression ratio much better than that of any other test file. Consequently, the varying number of bits made only a slight improvement: a difference of 1KB. This likely means that the patterns found within the file were large enough to not use all of the code words available, more than likely not going far past the 16bit range if at all.

In general, the larger the file was the greater amount of compression was achieved using the varying number of bits. For smaller files, there was not nearly as much improvement. This could be explained by entropy levels or even file sizes not being large enough to see a real effect from it. It is likely the case that they did not use all of the code words but had enough to be similar to that of the original program.

As the size of the file got larger the differences between the modified LZW program and Unix Compress became more apparent. This is demonstrated by the largest of the test files, all.tar. The modified LWZ shows a slight improvement over the results of the original, but Unix Compress far out performs both. This is likely due to the implementation differences between them. The file is large enough for it to be likely to use all of the available code words provided by the first two programs. Once this limit has been reached patterns are no longer added to the dictionary and only what has already been recorded is used for the remainder of the file. Depending on the file, this may worsen compression, especially if there are differences between the beginning and end of the file. This could likely be the case for all.tar. Unix Compress may use larger code words (greater than 16 bits) than the modified LZW program. It may also throw away the dictionary once it’s full, allowing for new patterns to be found later in the file and improving the case in which there are larger differences in the beginning and end of the file. Both of these reasons may explain why all.tar compresses the best with Unix Compress. It is also worthy to note that all.tar is itself a compressed file. This means that this file is more likely to have more entropy and have less compression in general.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Compression Implementation Comparisons | | | | |
| File | **Original Size** | **LZW** | **LZWmod** | **Unix Compress** |
| all.tar | 2,960 KB | 1,804 KB (1.6:1) | 1,741 KB (1.7:1) | 1,152 KB  (2.6:1) |
| assig2.doc | 85 KB | 73 KB  (1.2:1) | 40 KB  (2:1) | 40 KB  (2:1) |
| bmps.tar | 1,080 KB | 904 KB  (1.1:1) | 79 KB  (13:1) | 80 KB  (13:1) |
| code.txt | 71 KB | 31 KB  (2.3:1) | 24 KB  (3:1) | 24 KB  (3:1) |
| code2.txt | 57 KB | 24 KB  (2.4:1) | 21 KB  (2.7:1) | 21KB  (2.7:1) |
| edit.exe | 231 KB | 245 KB  (N/A) | 153 KB  (1.5:1) | 148 KB  (1.5:1) |
| frosty.jpg | 124 KB | 174 KB  (N/A) | 160 KB  (N/A) | N/A |
| gone\_fishing.bmp | 17 KB | 10 KB  (1.7:1) | 9 KB  (1.8:1) | 9 KB  (1.8:1) |
| large.txt | 1,193 KB | 591 KB  (2:1) | 491 KB  (2.4:1) | 511 KB  (2.4:1) |
| Lego-big.gif | 92 KB | 126 KB  (N/A) | 120 KB  (N/A) | N/A |
| medium.txt | 25 KB | 13 KB  (2:1) | 13 KB  (2:1) | 13 KB  (2:1) |
| texts.tar | 1,350 KB | 989 KB  (1.4:1) | 584 KB  (2.3:1) | 576 KB  (2.3:1) |
| wacky.bmp | 901 KB | 5 KB  (180:1) | 4 KB  (180:1) | 4 KB  (180:1) |
| winnt256.bmp | 154 KB | 156 KB  (N/A) | 62 KB  (2.5:1) | 62 KB  (2.5:1) |
| LZWmod.java | 4KB | 4 KB  (1:1) | 2 KB  (2:1) | 2 KB  (2:1 |