Raga Pribadi

CS 1501

Professor Ramirez

29 June 2015

LZW

Lempel–Ziv–Welch (LZW) is a universal lossless data compression algorithm created by Abraham Lempel, Jacob Ziv, and Terry Welch. This compression uses an adaptive algorithm. As new words are read in, the algorithm created new codewords containing longer strings. The algorithm used for compression and decompression are the same in that they both build the same codeword and string dictionary. During compression, the algorithm stores the data as (string,codeword) in the dictionary. Strings are searched and codewords are returned. Inversely, decompression stores data as (codeword,string). Here, codewords are looked up and strings are returned. Another difference is that the decompression algorithm will be a step behind in the building of the dictionary. The basic compression algorithm for LZW is to first initialize the dictionary to single character words by using their ASCII codes. While the end of the input file is not reached, three steps are executed. First, match the longest prefix from the file in the dictionary. Next, output a codeword for the prefix. Then add the longest prefix with the next character to the dictionary using the next codeword. The decompression Algorithm is basically the same. You initialize the dictionary to single character words using ASCII codes. However, this time while you have not reached the end of the file, a codeword is read from the file. The codeword is then looked up in the dictionary and the corresponding string is outputted. The dictionary is then updated with a new codeword and string pair. The goal of this project was modify the LZW algorithm provided by the book and compare the compressed results of the book algorithm, modified algorithm, and compress command in unix.

The following are tables comparing the Original size of the file and the compressed files compressed by each algorithm.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| File | Original(kb) | LZW(kb) | LZWmod(kb) | Compress(kb) |
| all.tar | 2960 | 1804 | 1751 | 1152 |
| assig2.doc | 85 | 73 | 40 | 40 |
| bmps.tar | 1080 | 904 | 80 | 80 |
| code.txt | 71 | 31 | 24 | 24 |
| code2.txt | 57 | 24 | 21 | 20 |
| edit.exe | 231 | 245 | 153 | 148 |
| forsty.jpg | 124 | 174 | 160 | 124 |
| gone\_fishing.bmp | 17 | 10 | 9 | 9 |
| large.txt | 1193 | 591 | 491 | 508 |
| lego-big.gif | 92 | 126 | 120 | 92 |
| medium.txt | 25 | 13 | 13 | 13 |
| texts.tar | 1350 | 989 | 584 | 576 |
| wacky.bmp | 901 | 5 | 4 | 4 |
| winnt256.bmp | 154 | 156 | 62 | 62 |

|  |  |  |  |
| --- | --- | --- | --- |
| File | LZW ratio | LZWmod ratio | compress ratio |
| all.tar | 1.640798 | 1.690462593 | 2.569444444 |
| assig2.doc | 1.164384 | 2.125 | 2.125 |
| bmps.tar | 1.19469 | 13.5 | 13.5 |
| code.txt | 2.290323 | 2.958333333 | 2.958333333 |
| code2.txt | 2.375 | 2.714285714 | 2.85 |
| edit.exe | 0.942857 | 1.509803922 | 1.560810811 |
| forsty.jpg | 0.712644 | 0.775 | 1 |
| gone\_fishing.bmp | 1.7 | 1.888888889 | 1.888888889 |
| large.txt | 2.018613 | 2.429735234 | 2.348425197 |
| lego-big.gif | 0.730159 | 0.766666667 | 1 |
| medium.txt | 1.923077 | 1.923076923 | 1.923076923 |
| texts.tar | 1.365015 | 2.311643836 | 2.34375 |
| wacky.bmp | 180.2 | 225.25 | 225.25 |
| winnt256.bmp | 0.987179 | 2.483870968 | 2.483870968 |

As you can see in the first table, the predefined *compress* Unix provided either the best compression. Compression ratios suggests that the larger file size the larger the difference between LZW and either LZWmod or Unix’s compress. The table also shows that my LZWmod program either gave the same compression results as the Unix compression or it did slightly worse than it. However, the LZW compression algorithm provided by the book gave the worst performance of the three. I believe that the LZW algorithm provided by the book was the worst was because it had a lot of overhead. This was due to the creation of new strings by the substring() method. Furthermore, this program reads in the entire input file into a string. Second of all, the program used a fixed codeword size. This limited the number of codewords that was available. Also, this meant that earlier on it used twelve bits for small codewords. It did now allow for variable length codewords.