Ronnye Alvarado - L20348093

November 13th, 2019

Multimedia Processing - CPSC 4430

Instructor: Prof. Jiangjiang (Jane) Liu

LZW Algorithm Project Report

Introduction

The LZW(Lempel-Ziv-Welch) algorithm is a lossless data compression algorithm which is a type of dictionary coding. It does not use statistical knowledge of the data compared to other algorithms such as Huffman and Arithmetic coding. The encoder develops a dictionary as the input is processed and transmits the index of strings found in the dictionary. The decoder inverts the process of encoding to reconstruct the dictionary and source file. It can be found in multiple applications such as Unix Compress, gzip and GIF.

Encoder

The encoder scans the source file and stores non-duplicated symbols which creates the base dictionary. It then goes through the source file and finds the longest pattern match that is in the dictionary. Afterwards, it will output the index of the pattern and concatenate the pattern and the following unmatched symbol in the sequence to an index. Doing this creates more entries in the dictionary that the encoder can later use when/if it comes across the same pattern. Once it is complete, the encoded file will contain the base dictionary and the indices that associate to specific patterns.

Decoder

The decoder emulates the encoder in building the dictionary but is slightly behind. Since it can’t see the upcoming symbol in the sequence due to being encoded, it has to work slightly different. The decoder has to decode the first symbol and add the symbol along with a question mark to a dictionary index. The question mark is a placeholder since it does not know what the next symbol is. However, once it decodes the next symbol, it will go back and update the question mark with the first character of the now decoded symbol. Once it is complete, the decoded file will be the same as the source file before it was encoded.

Methodology Used

Encoder.java opens a source file, scans all the symbols in the file and stores non-duplicated symbols into a HashMap. This creates our base dictionary. It also creates an inverse of the base dictionary where <key, value> pair are inversed to <value, key>. The file also creates a HashMap temp that only stores the base dictionary. This is so we can send the base dictionary, without the updated entries, in the encoded file along with the encoded sequence to the decoder.

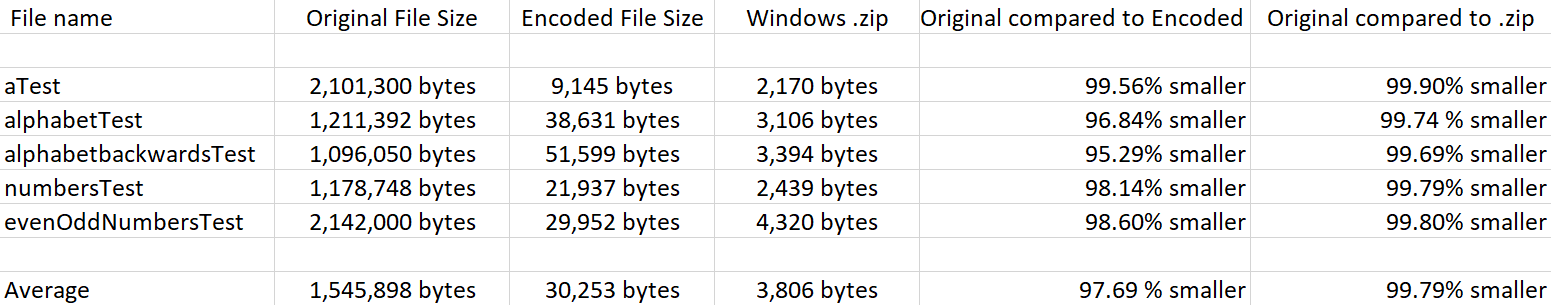
ArrayList stringArray will hold all the characters in the source file and ArrayList integerArray will hold the encoded sequence and will be sent to the encoded file along with HashMap temp.

The encoder will loop while it is not at the end of stringArray. If the dictionary containsValue(n), it will put(key, n + n+1) into the dictionary and puts the key associated with value n into the encodedSequence. I created the inverse dictionary because calling a key(index) returns the associated value(symbol) stored in the dictionary but not vice versa. This allows me to easily get the index that holds the pattern. After all of this is complete, the encoded file will contain the encoded sequence and the base dictionary. The decoder will use this information and return the original source file.

Decoder.java uses a lot of the same methods as Encoder.java. It reads the encoded sequence created by the encoder and stores the base dictionary. It uses the base dictionary to decode the first symbols and will concatenate the symbols with a question mark. Once it knows what the following pattern/symbol is, it will update the question mark with the first symbol. This way it builds the dictionary similarly to the encoder.

The files I used had repeated, recurring patterns since this algorithm works the best under those conditions. I had five files of at least 1MB: aTest, alphabetTest, alphabetbackwardsTest, numbersTest and evenOddNumbersTest. aTest contained only the symbol ‘a’ and was used to see how the encoded file size would compare under the absolute best conditions. alphabetTest contained the alphabet repeatedly while alphabetbackwardsTest contained the alphabet forward then backwards repeatedly. I used those two to see how the encoded file size would vary since the latter contained a longer pattern. numbersTest and evenOddNumbersTest were very similar since they had the same number of unique symbols and pattern length, but the former was only half the file size.

Results



Compared to the original files, the encoded files were, on average, 97.69% smaller when using my encoder. Compressing the files using ZIP had better results averaging 99.79% smaller files. Although these percentages are extremely close, ZIP files were approximately 87.22% smaller than the encoded file provided by my encoder.

Conclusion

In conclusion, LZW is a great way to compress and decompress files that contain recurring patterns. It is not as efficient if it doesn’t contain many recurring patterns since the encoded file might be just as large along with a very large dictionary. Using my tests, I was able to compress the source file size approximately 98% smaller.