Overview

Explored in this project was applying different compressions techniques to a text file. Huffman, Shannon-Fano-Elias, adaptive huffman, and Lempel-Ziv codes were used. The text file analyzed was the US constitution, representable using a fixed width 8-bit encoding with 360952 total bits. The entropy of the distribution of the characters for the file was 4.4652 bits.

Each code will have its process described and results given. The results and algorithms for each will then be compared.

Huffman

A huffman code uses the frequency distribution of the characters in the file. A tree is constructed by successively merging the two lowest weighted trees and assigning the merged tree's weight as the sum of the two combined trees. A character's encoding is then the binary string representing the path from root to character in the final tree.

Below are a few of the character-encoding-length tuples. The encodings with minimum length were chosen. Interesting the short encodings correspond to frequently used English characters.

[('e', '100', 3), (' ', '001', 3), ('t', '0001', 4), ('n', '0111', 4), ('s', '1101', 4), ('a', '1010', 4)]

The length of the message encoded using a huffman encoding was 202686 bits, indicating a compressions ratio of 1.7808. Not included was the storage necessary for the code.

Shannon-Fano-Elias

A Shannon-Fano-Elias encoding also makes use of the frequency distribution of the characters in the message. Using a modified cumulative distribution for each character, the encoding of the character is set to the binary representation of the modified cmf value. The number of digits used is determined by the ceiling of the log based 2 of the character's probability. The encoding produced in this manner is prefix free.

Below are a few of the character-encoding-length tuples. Once again the frequently occurring characters are given short encodings.

[(' ', '0001', 4), ('t', '11101', 5), ('e', '01101', 5), ('n', '101011', 6), ('s', '110110', 6), ('a', '010001', 6)]

The length of the message encoded using Shannon-Fano-Elias was 272904 bits, indicating a compression ratio of 1.3226. Not included was the storage necessary for the code.

Adaptive Huffman

A huffman encoding relies on two passes through the message: one to calculate the frequency distribution of the characters in the message, and another to encode it. An adaptive huffman encoding encodes the message as it is being read. A tree with each character and frequency of its occurrence is constructed during the reading of the message. The character in the message is encoded based on its current location in the tree if the character has already been seen, or the location of the NYT node followed by the binary representation of the character's ascii value. New characters are added to the tree by expanding the NYT node and updating the tree.

The encodings for characters change as the message is being encoded. Below are the first 10 characters from the message, along with their encodings.

'Provided b'

['01010000', '001110010', '0001101111', '10001110110', '00001101001', '110001100100', '100001100101', '011', '010000100000', '1110001100010']

The length of the message encoded using the adaptive huffman encoding was 203502 bits, indicating a compressions ratio of 1.7737.

Tree based Lempel-Ziv

The tree based lempel ziv iteratively encodes text segments in the message as either a prefix already seen plus a new character, or a single character. Output is a list of character-integer tuples representing the index into the list representing the prefix and the next character, where an index of 0 indicates that the tuple refers to a single character (the empty string implicitly occupies index 0).

The first 10 tuples of the encoding are provided below.

[(0, 'P'), (0, 'r'), (0, 'o'), (0, 'v'), (0, 'i'), (0, 'd'), (0, 'e'), (6, ' '), (0, 'b'), (0, 'y')]

After converting the tuple to bits (16 bits for the index and 8 bits for the character), the length of the encoded message was 226864 bits, indicating a compression ratio of 1.59105.

Comparison

For a symbol by symbol encoding, one cannot expected to do better than entropy with respect to expected code length. Therefore the maximum compression ratio for codes of this form is 1.79162. This would not change had the order of the characters in the file been shuffled.

The huffman and adaptive huffman encodings come close to this. The Shannon-Fano-Elias encoding is quite far behind.

When taking into account the order of the characters in the file, a much tighter on compression can be obtained. The limit here is the entropy rate of the English language, an unknown quantity. The Lempel-Ziv encoding relies on the order of the file. Interestingly, the Lempel-Ziv encoding did not perform as well as either of the Huffman variations (with respect to the length of the encoding). This could be explored more with a larger message or a message with more structure (a song perhaps).

The standard huffman encoding relies on constructing a tree from each character's probability after estimating the probabilities from the read file. This requires two scans through the message, plus a number of heap operations linear in the size of the character set.

The Shannon-Fano-Elias encoding requires a linear scan through the message to estimate the character probabilities, followed by a linear scan through the characters to assign codewords, and finally another linear scan through the message to perform the encoding.

The adaptive huffman encoding relies on tree operations. Many of these required a linear scan through the tree, with a size bounded by 512. The message must be scanned once as well. There are almost surely structures of the tree that can be exploited to reduce the running time that I did not take advantage of.

The tree based Lempel-Ziv encoding can be constructed with as little as one pass through the message and constant calculation per character. This implementation relies on suffix trees. My implementation did not do nearly this well. I relied on a trie library to find the longest prefix in the message after adding each entry to the Lempel-Ziv tuple list. I did this by keeping a pointer in the message with my current location. However, I accessed the string using Python's slice notation, which creates a new string. This means my algorithm was at least quadratic in the size of the message.