Information Theory-Final Project

File Compression

Submitted by: Tamar Yanetz 207139940

The data:

Text file.

For example: ‘dickens.txt’

The purpose:

Compress and decompress the file using algorithms that were learned in class.

The size of the file is initially 30.6 Mo. The compression should get to 12 Mo.

The Method:

Several algorithms and several combinations of algorithms were tested until the purpose was achieved.

First, I tried to compress the file (‘dickens.txt’) using LZ’78. After several tries, the implemented code I tried did not succeed to decompress the file. Indeed, it couldn’t decode the number characters that was mixed with the code of the algorithm. I wasn’t able to fix it.

Then I implemented LZW algorithm. It gave a very good result. I could get with this algorithm a compression to 12.6Mo.

In order to improve it, Huffman coding was added. I implemented a combination the two algorithms.

Huffman did improve the compression, but it was not that significant. Though the combination compressed to 12.5Mo, which is a great compression.

Then, the decompression was implemented. First, I decompressed using Huffman to decode the final file that has initially been compressed by LZW. Then, decompressed this one with LZW decompression. Then, I get the initial text file back (decompressed).

The Algorithms:

LZW:

LZW compression works by reading a sequence of symbols, grouping the symbols into strings, and converting the strings into codes. Because the codes take less space than the strings they replace, we get compression.

LZW compression uses a code table (dictionary). I am working with ascii table. Codes 0- 255 in the code table are always assigned to represent single bytes from the input file.

To begin, the dictionary contains only the first 256 entries of ascii table, with the rest of the table being blanks. Then, LZW identifies repeated sequences in the text, and adds them to the code table. The dictionary keeps a correspondence between the longest encountered words and a list of code values. The words are replaced by their corresponding codes and so the input file is compressed.

(The efficiency of the algorithm increases as the number of long, repetitive words in the input data increases.)

The decoding program that uncompressed the file is able to build the table itself by using the algorithm as it processes the encoded input. It's taking each code from the compressed file and translating it through the code table to find what character or characters it represents.

Huffman coding:

Huffman coding provides an efficient code by analyzing the frequencies that certain symbols appear in a text. Symbols that appear more often will be encoded as a shorter bit string while symbols that aren't used as much will be encoded as longer strings. Huffman coding is working such that the code assigned to one character is not the prefix of code assigned to any other character. Huffman coding works by using a frequency-sorted binary tree to encode symbols.

To do so, we need to build a Huffman Tree from input characters, traverse it and assign characters with codes.

We create a binary tree, then choose the two trees with the smallest frequency and combine them into a new tree with a new parent node with the frequency of both of its children (the addition). We iterate until there are no more trees to merge. Every symbol must have its code. We start at the root node. For each left arc, we add 0 to the end of the code and for each right arc, we add 1 to the end of the code

Technical Instructions:

The code submitted is a Python 3.7 commented code.

In order to run the code, you need to download the files into one folder and run 'main.py' in the 'code' directory.

In the directory 'Output' you will get:
'encode_0.bin' - The first encoding (with LZW)
'encode_1.zip' - The second encoding (with Huffman)
'decode_0.bin' - The first decoding (with Huffman)
'decode_1.txt' - The last decoding (which is the file itself)