Huffman coding compression algorithm in Python

Introduction:

Modern day data compression began in the late 1940's. Claude Shannon and Robert Fano found a systematic way to assign code words based on probabilities of blocks. A more optimized method was then found by David Huffman in 1951, and then in the late 1970's Lempel and Ziv suggested "pointer based" encoding. Today, there are now plenty of different data compression algorithms which still use encoding determined by Huffman and Lempel/Ziv. So how does compression work? It is often performed by a program that uses an algorithm to reduce the size of the data. For example, we could reduce the number of bits needed to represent different characters based on their frequency or removing all unneeded characters. For music, we can remove certain frequencies the human ear cannot hear. And so why is data compression so important? It can dramatically decrease the amount of storage needed to store a file or transmit a file. For example, an algorithm that has a compression ration of 5:1 means, it can reduce the size of the original file by 50%! So, a file that was originally 1gb can be reduced to 500mb. One of the disadvantages of data compression can be the performance impact from the used processing power to compress a file, but today having enough processing power is hardly a problem and so, the reduction in storage and transmission time tends the trump the required processing power. [6]

Lossless compression algorithms:

There are two types of compression algorithms, lossless and lossy. Lossless enables us to completely restore our compressed file to its original state, without any loss of data. This tends to be used for text files or source code, where any loss might alter the file. Lossy can permanently delete data that is unimportant or unused and won't impact the file. I will cover three different lossless compression algorithms, Huffman, Arithmetic Coding and Lempel-Ziv.

First, let us look at how a Huffman lossless compression is meant to work. We consider the data we want to compress as a sequence of characters, we then build a frequency table, where we analyse how often each character occurs. For example, let us consider a sequence of letters where the letters f, e, c, b, d and a appeared 5, 9, 12, 13, 16 and 45 times, respectively. We can build a table that looks like this:

f:5	e:9	c:12	b:13	d:16	a:45

Figure 1: Character Frequency [1]

We can then use a binary tree, to associate each character with a binary code (I will go into detail later, about how I built my tree), so we could end up with something like this: (Figure 2)

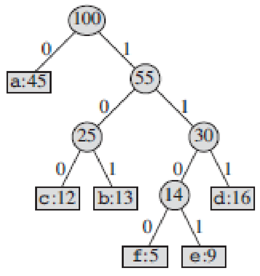
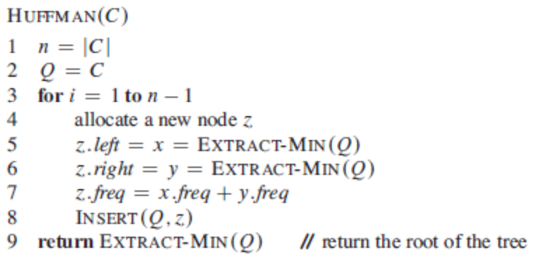


Figure 2: Character tree [1]

We then associate each left branch with a 0 and each right branch with a 1. From the root, we then go down the tree adding 0's or 1's to our binary code until we cannot anymore. Here the codes we would get :
a : 0, b : 101, c : 100, d : 111, e : 1101 and f : 1100
Why is this a good idea? Because each character is encoded using 8 bits, with this compression algorithm we can significantly reduce that number. In this case, a normal encoding would have used :

(5 + 9 + 12 + 13 + 16 + 45)*8 = 800 bits. With our encoding we would only use (45*1 + 16*3 + 13*3 + 12*3 + 9*4 + 5*4) = 224 bits.
That is a 72% reduction in size! (Example taken from *Introduction to Algorithms by Cormen et Al. p.441*)
So how do we build our tree? We assume C is a set of n characters with each c ∈ C an object with attribute c.freq, which gives its frequency. The algorithm then builds the tree from the bottom-up. We start with a set the size of C containing "leaves" and perform a number of operations until there is only one element left in our list. First, our list must be sorted by the value of c.freq in ascending order. We then take the two lowest frequency objects in our list and merge them together to form another object. This new object frequency is the sum of the objects we have just merged frequencies. We then remove the two objects we have just merged from the list and add this new object. In pseudocode, our algorithm would look something like this:



```
HUFFMAN(C)
n = |C|
Q = C
for i = 1 to n − 1
    allocate a new node z
    z.left = x = EXTRACT-MIN(Q)
    z.right = y = EXTRACT-MIN(Q)
    z.freq = x.freq + y.freq
    INSERT(Q, z)
return EXTRACT-MIN(Q)      // return the root of the tree
```

Figure 3: Huffman Pseudocode[1]

We will look at two other data compression algorithms. The first one is known as Arithmetic Coding. The idea is to have a message represented by an interval of real numbers between 0 and 1, the longer the message, the smaller and more precise the interval becomes, hence increasing the number of bits needed to define it. Each character we want to encode has a given probability between 0 and 1, and so the more frequent a character is, the less the interval range is changed compared to less frequent characters. So how does the encoding work? The first range we start with is between 0 and 1, with each character being given a range, let us say [a,b]. We then take the first character of our sequence we want encoded, and our new range becomes [a,b] where we must then redistribute all of our characters in accordance with their probability, we then repeat this until our message is fully encoded. To help visualize this, I will use an example given in *Arithmetic Coding For Data Compression, by Ian H. WITTEN, Radford M. NEAL and John G. CLEARLY* :

Let us say we want to encode the message "eaii!" with probability table :

Figure 3: Probability table[2]

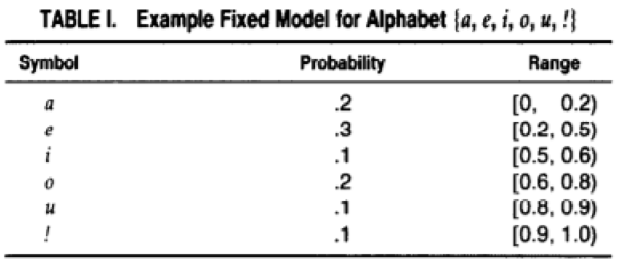


TABLE I. Example Fixed Model for Alphabet {*a*, *e*, *i*, *o*, *u*, *!*}

Symbol	Probability	Range
a	.2	[0, 0.2)
e	.3	[0.2, 0.5)
i	.1	[0.5, 0.6)
o	.2	[0.6, 0.8)
u	.1	[0.8, 0.9)
!	.1	[0.9, 1.0)

We first map out all our characters on the range [0,1], we then see the range for "e" is [0.2,0.5], so [0.2, 0.5] becomes our new range, "a" range then becomes [0.2, 26] using the formula, [lower limit, lower limit + d*(probability of symbol], where d = upper bound – lower bound. We then repeat this for "i", "i" and "!" and we end up with something looking like this:

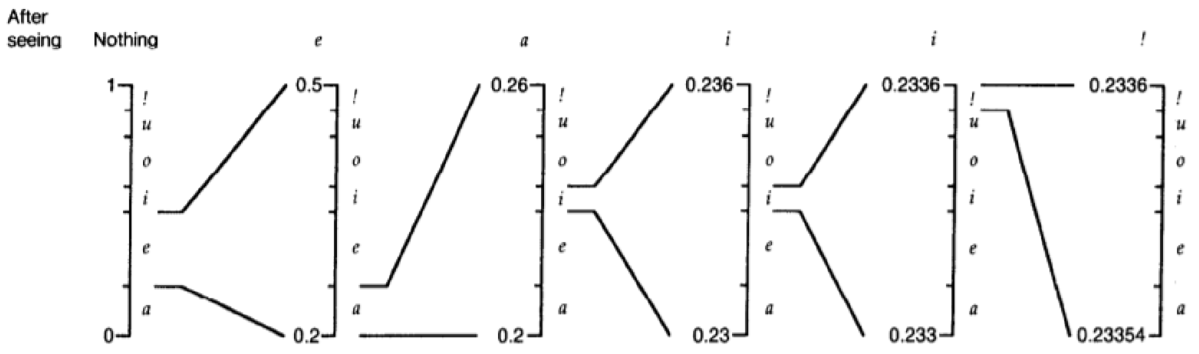


Figure 4 : Representation of Arithmetic Coding[2]

And so, our final range is [0.23354, 0.2336]. To decode the message, we can take a number within this range, for example 0.23357 ((u.b + l.b)/2) the decoder knows the first letter was "e" as our 0.2 lies within the first given range for "e" [0.2, 0.5], our next letter is "a" as 0.23 lies within the range space of "a" on the second branch (we don't see the individual ranges for each character on each branch, however we can calculate them using the formula above), next 0.233 gives us "i", 0.2335 give us "i", and finally 0.23357 give us "!". So that the decoder knows when to stop, a symbol will often be used to tell the decoder when to stop, here we could use "!".

The second data compression algorithm we will look at is the Lempel-Ziv algorithm. This algorithm is quite common today and is mostly used in GIF and sometimes PDF compression. The way it works, is relying on recurring patterns to save space. The idea is, "it parses source sequence such that each phrase is the shortest phrase not seen earlier (incremental parsing) and then describe (encode) each new phrase by describing the index of the phrase from the past that forms its prefix and the new symbol at the end of the phrase"[4].

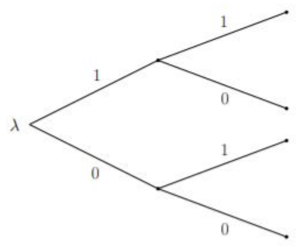
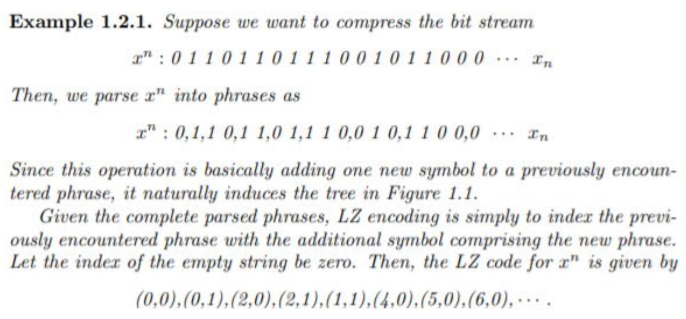


Figure 1.1: LZ tree associated with x^n



Example 1.2.1. *Suppose we want to compress the bit stream*

$$x^n : 0\ 1\ 1\ 0\ 1\ 1\ 0\ 1\ 1\ 1\ 0\ 0\ 1\ 0\ 1\ 1\ 0\ 0\ 0\ \cdots\ x_n$$

Then, we parse x^n into phrases as

$$x^n : 0,1,1\ 0,1\ 1,0\ 1,1\ 1\ 0,0\ 1\ 0,1\ 1\ 0\ 0,0\ \cdots\ x_n$$

Since this operation is basically adding one new symbol to a previously encountered phrase, it naturally induces the tree in Figure 1.1.

Given the complete parsed phrases, LZ encoding is simply to index the previously encountered phrase with the additional symbol comprising the new phrase. Let the index of the empty string be zero. Then, the LZ code for x^n is given by

$$(0,0),(0,1),(2,0),(2,1),(1,1),(4,0),(5,0),(6,0),\cdots.$$

Figure 4: Example of Lempel-Ziv [4]

LZW builds on that and works by again reading the sequence and grouping recurring symbols into strings, and then converting them into codes, these codes then take up less space than the strings, hence we achieve compression. LZW uses a code table where codes 0-255 are used to represent single bytes from the input sequence. When the encoding begins, only the first 256 entries are in the table, with the rest being used to represent sequences of bytes. As it continues, repeated data sequences are identified and added to the code table. Decoding is then done by translating the compressed file with the code table.

Data structures, algorithms, and log:

To implement the Huffman coding, I used multiple data structures. First, I defined a “Node”. A Node contains a frequency, a character, a binary character, a left Node, and a right Node. I wanted to create my tree using linked lists, which are elements containing data that point to each other, and so using “Nodes” here seemed appropriate. I then defined a function to analyse the frequency of each character in a sequence. This function returned a dictionary. Dictionaries are used to store data values in key:value pairs, so in this case, my key was a character, and the value was the frequency. To then sort my dictionary by value, I used Python’s built-in sort function. The reason I used Python’s built-in function, is because it collects all the elements of my dictionary, puts them into a list and uses Timsort to sort the list.
To start my tree, I then create a list of Nodes using the elements of my dictionary (the Node frequency will be the value and the node character will be the key), since my dictionary was already sorted, my list of Nodes will also be sorted by frequency.
To build my tree, I take the two lowest frequency Nodes and define the first Node’s binary character to be 0 and the second’s to be 1. I then add them together to make another node (adding the frequency here is what is important), I then assign this new Nodes left and right to be the two lowest frequency Nodes. I then remove the two first Nodes from my list, add the new Node and sort my list again by frequency. I then repeat this step until there is only one Node left.
Once I have my tree, I make another dictionary where my key is a character, and the value is the binary code. To then encode a sequence, I just read through it and compare each character with the keys in my dictionary, and when a character equals a key, write that character’s binary code to a new list. When I was first testing my algorithm, I found that books worked fine but large data sets (e.g 200mb) would never finish. This is because I was concatenating a string in a for loop. I did not realise, when you concatenate strings, you are basically creating a new data structure after each loop, which explains why for large data, it would take so long. My solution to this, was to instead create a list and then append the characters to that list, and then use the .join() function to create my string at the end. I found this increased performance significantly. According to one article I found, using strings, we can achieve 17,400 concatenations/sec whereas with list.append() we can achieve 94,000. [3] And so throughout my code, I use this method when concatenating strings. I then just read the binary string and convert it to a bytearray to save it as a .bin file. To decompress, I just read the bin file and reconvert the binary into a string. I then call my decode function recursively, where after each loop, I read my binary string going left or right depending on if a read 0 or 1, keeping track of the index of my binary string, until I reach a Node where left and right return None. I then add the character from that Node to a list and continue until I reach the end of my binary string. I also encountered some performance issues using this method with the large data files, my function being recursive is not optimal when dealing with large files, and so, most of the time consumed was when decompressing those files. And finally, when having to compress a book in another language using the tree of a different language, for example, encoding a book in Portuguese using the English encoding, the way I deal with characters missing was to read through the text and check what characters were missing from my tree, I would then remake a new tree having added the characters to my frequency dictionary. This was not optimal, and I will discuss the results in the next section.

Performance analysis:

Original File			Run length		
File	File size	Nb. Of characters	Compressed File size	Compression ratio	Compression/Decompression time
1	70 kb	69,206	40kb	57.14%	0.349sec
2	79kb	76,758	46kb	58.22%	0.349sec
3	73kb	72,277	42kb	57.53%	0.349sec
4	73kb	72, 277	62kb	84.93%	0.530sec
5	261,636kb	300000+	32,705 kb	12.5%	5min32sec
6	91,117 kb	300000+	52, 276 kb	57.37%	4min50sec
7	102,400 kb	300000+	67, 168 kb	65.59%	5min10sec
8	79 kb	76, 758	66 kb	83.54%	0.606sec
9	40 kb	39,127	23 kb	57.5%	0.217sec
10	40 kb	39, 127	48 kb	120%	0.314sec
11	45 kb	42, 995	48 kb	106.7%	0.320sec

In my table, file 1 was my book "Moon of Treason", having normally compressed it, I achieved a compression ratio of 57.14%, files 2 and 3 were the same book but in French and Portuguese, and we see that we achieved similar compression ratios, most likely as all three languages use a similar alphabet although letter frequency differs. However, file 4 is when I tried to compress my book "Moon of Treason" in Portuguese using the English version tree. We see here that the compression ratio is relatively bad compared to those previous at 84.93%, as I stated earlier, my way of dealing with missing characters from the Portuguese alphabet was far from optimal and so we can see that the performance is equally not as optimised, having also taken longer to compress/decompress. File 5 was my file "fib41.txt" which was a large txt only containing a's and b's. Here we see that although compression/decompression took a long time, we achieve a good compression ratio. This is because we only have 2 characters to encode, and so each character only takes 1 bit to encode (0 or 1). File 6 was my file "Einstein.de.txt" which was a text file containing German words and other symbols. Here, we again notice a compression ratio of 57.37% which is like the book above. File 7 was my file "dblp.xml.0001.1" which contained references to multiple books author, date, pages, ect… which meant there were many repeating characters. This explains a slightly better compression ration of 65.59% which is significantly higher than the previous books. File 8 is where I compressed the French version of "Moons of Treason" using the English encoding, again we see a relatively bad compression ration at 84.93%, again, because the French alphabet contains characters which the English encoding did not have, which meant I had to rebuild my tree in a non-optimised way. File 9 is my text "Some Trees", which I compressed normally and achieved 57.5% compression ratio. The other languages also achieved similar ratios. What is more interesting is that with this text file, when I encoded it in English using the Portuguese encoding (File 10), the compression actually increase the size of the file, achieving a compression ration of 120%, this is most likely due to the fact the Portuguese tree contains many characters not used in the English language, added to the fact the frequencies of different characters most likely differ. I found similar results when encoding the French version using the Portuguese encoding as well (File 11).

List of references:

[1] *Introduction to Algorithms* by Cormen et Al.

[2] *Arithmetic Coding for Data Compression*, by Ian H. WITTEN, Radford M. NEAL and John G. CLEARLY

[3] https://waymoot.org/home/python_string/

[4] https://web.stanford.edu/class/ee376a/files/EE376C_lecture_LZ.pdf

[5] https://www.geeksforgeeks.org/lzw-lempel-ziv-welch-compression-technique/

[6] *A New Kind of Science*, Stephen Wolfram,