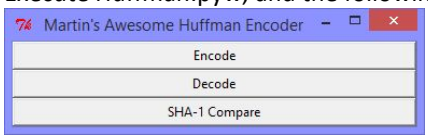
Huffman Encoding Assignment

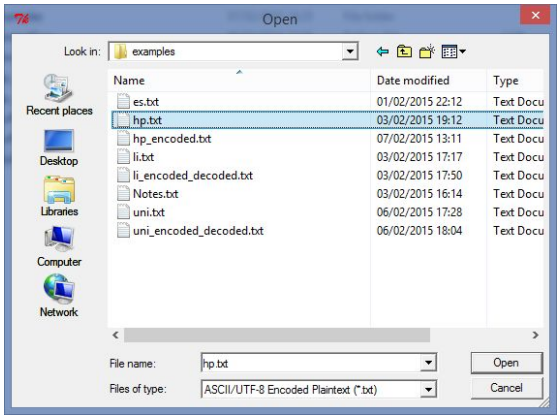
Instructions on how to use the Huffman encoder

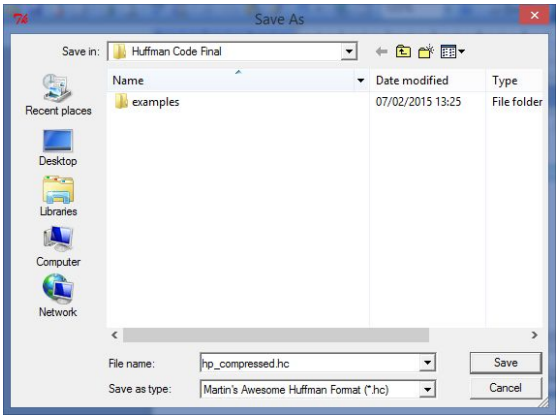
Execute Huffman.pyw, and the following window will appear:



Select Encode or Decode depending on how you wish to use the encoder. (The third option will be discussed later).

To encode, select the file you wish to encode, then enter a file name to save the compressed file as.





In this case, "hp.txt" is the file I am encoding, and "hp_compressed.hc" is the name of the output file. The decoding procedure is similar, selecting the encoded file and choosing a decoded filename. After a short time (depending on the file size) a dialogue box should appear, telling you that the process has completed.

The program can also be executed via the command line, by the calls of the form:

python Huffman.pyw **ENC INPUT OUTPUT**, where **ENC** is either 'encode' or 'decode', **INPUT** is the path of the input file to compress/decompress, and **OUTPUT** is the path of file to create.

For example, the equivalent command line representation of the above process would be:

python Huffman.pyw encode examples\hp.txt hp_compressed.hc

Description of the program

While *Huffman.pyw* is the main executable, it mostly deals with managing input parameters and the GUI. I used Tkinter to generate the GUI.

Regardless of the way the program is executed, the function calls encode() or decode() from the file *huffenc.py*. I will explain encoding and decoding individually. Both functions take an input and output path as parameters.

Encoding

Generating a Huffman Tree

The encoding function reads the entire text file as a string and generates a frequency table by iterating through each of the characters in the input and incrementing a dictionary by 1 each time a character is seen. I use this frequency table instead of a probability table, since integer operations are more efficient than floating point operations, and the character frequencies are proportional to their probability density. I call this table S_0. The function frequencies() must iterate over each of the n characters exactly once, and each incrementing operation is O(1), so frequencies() is O(n).

S_0 is then passed to increment_table(), and the first operation is to obtain the two symbols with lowest frequency (running on O(1), since the size of the dictionary cannot exceed 256). It then calculates the combined frequency of these two Symbols, and maps a combined array of the two symbol to this value in the dictionary, and removes the original two pairs from the dictionary. All of these operations run in O(1), so increment_table() is O(1). I call the modified table S_1. This process occurs a total of k-2 times, producing S_{k-2} (k is the alphabet size $\Rightarrow k \leq 256$). Since each iteration reduces the number of elements by 1, this will produce a table of two elements. Increment_table() is executed a maximum of 254 times, so the loop producing S_{n-2} is O(1). (Note: each S_i is a modification of S_{i-1}, a new dictionary is *not* generated in memory on each iteration.)

S_{n-2} is a dictionary whose values are the total frequencies of each of the elements in the key, and additionally they are stored as Arrays within Arrays corresponding to how they were combined, and so the two keys of the set are an equivalent representation of the two initial branches of the Huffman tree. I take take the keys of the dictionary as a list, forming variable 'tree'. This operation is O(1).

Generating a Canonical Huffman Code from tree

A standard way of generating a Huffman Code from a tree is to traverse each node in the tree, and have each branch of a node represent the addition of either a 0 or a 1, depending on the direction chosen. However, when it later comes to storing the codebook to disk, this requires that the entire code dictionary must be stored to disk since the traversal pattern is dependent on the input instance. In order to reduce the amount of disk space used, I decided to generate a 'Canonical Huffman Code'.

The canonical code is generated as follows. First, the 'depth' of each character in the binary tree is calculated for each character, and a mapping is made from character to 'depth', ignoring the actual route taken to reach that character. This is the purpose of the generatelengths() function, and is O(1), since worse case complexity is bounded by the size of the alphabet, which in this case is fixed at 256.Note that the binary tree depth is equivalent to code length.

The characters must be sorted first by code length, and then by their alphabet position. Since dictionaries are unordered, I convert it to an array of 2-tuples. This conversion and sorting is, again, O(1) for the reasons mentioned above. The first element , of length x, is assigned the code of x 0's. For the remaining elements, I start a counter and iterate over the remaining elements, incrementing the counter by 1 at the beginning of each code, and performing a left-shift of order d on the number, where d is the difference in bit-length between the current and previous code (e.g. code lengths of x_i=5 and x_{i+1}=5 would result in no shift, whereas x_i=5 and x_{i+1}=7 would result in two shifts). This results in a Huffman Code that is generated from only the lengths of the codes, and this process runs in O(1). I call this mapping from character to code f_0.

Using this particular type of code means that only the lengths of each code need to be stored. Since the number of possible characters encoded is 256, the longest possible code would result from following the longest path on a maximally unbalanced binary tree with 256 end nodes, and is therefore 255. This means that 1 byte will be required to store the length of the code of each character, and the disk space used for storing the mapping is precisely 1 byte per character, or 256 bytes per file. Characters that occur with 0 frequency have a code length of 0, and the bytes are stored in the file in alphabetical position.

Mapping the String

A useful property of f_0 is that the values of the dictionaries are actually strings of the binary code they represent. This provides two benefits, firstly that storing the mapping in the program is simpler, since storing data of size less than one byte in a program results in technical issues. The second is that it makes mapping the output much easier, since it allows for concatenation of bits sequences that are not multiples of 8, and incomplete bytes can 'overlap' much more easily. This means that generating the encoded output is simply a matter of iterating over the input String and for each character x_i in the input append to the output $f_0(x_i)$ (the corresponding code in f_0).

Storing the output

The output string is converted into an array of bytes, by splitting the string into multiples of 8 and converting each part into a byte. However, since the encoded string is not necessarily a multiple of 8, the final byte may not contain 8 bits. Therefore, a final additional bit is appended, which denotes the actual length of the penultimate bit.

As explained above the first 256 bytes of the file will be used to store the lengths of the codes as a byte array, followed by the byte array of the encoded output.

This means that the total overhead for encoding a file is 257 bytes (256 for the lengths, plus the final bit). As a result, encoding a 0 byte file will produce a 257 byte encoded output.

This combined byte array is then stored to file, with file extension ".hc" (or a file extension of the user's choice if used in command line mode).

Decoding

Decoding requires somewhat less explanation than encoding, due to many of the functions used for encoding being used to decode also.

The encoded file is read in as a byte array and divided into the key and data sections. The key sections is then use to create an mapping from character to code length. Using the method described above, this is used to derive the mapping f_0^{-1}, which is the inverse of the mapping f_0 , taking time O(1).

The data section is used to recreate the original encoded string, using the final bit to determine the length of the last encoded bit.

A sliding window of two indices is then used to traverse the encoded string and append the decoded characters to decoded output string, taking time O(n). This decoded string is then stored to file, with file extension ".txt" (or a file extension of the user's choice if used in command line mode).

Other Features

Dealing with multiple encodings

In order to deal with the variety of encodings that a text file could be encoded in, I decided that I would utilise what I initially saw as a limitation of Python, which is its low compatibility with non ASCII characters, such as Unicode.

When iterating over a string, Python recognises each byte as a character (as in ASCII). This is actually useful, since n-byte characters can simply be represented as n single-byte ASCII characters within the program, and as such the size of the Huffman character alphabet is bounded by 2^8=256, regardless of the encoding. This means that the Huffman encoder actually encodes the bytes of the input, rather than the characters (though for ASCII these are equivalent).

Non-text binary Encoding

An interesting side-effect of compressing data byte-wise is that the encoder is essentially blind to the type of file it is encoding, and so any kind of file can be encoded using the encoder. However, the compression ratio for non-text files is usually less effective, since unlike natural language, it is less likely that there will be more frequent bytes.

SHA-1 Hashing check

I decided to implement a function that allow the user to compare an original file with it's encoded-decoded counterpart to check for equality. I do this by providing a GUI window for opening both files. The program then reads both files in binary mode, generating a SHA-1 hash of both files, and compares these hash outputs for equality. The result is then reported to the user in a dialogue box.

Time Complexity and Compression Ratio

Time Complexity of the Encoding/Decoding Process

Both encoding and decoding take time O(n), since each element of the main encoding and decoding function is O(n) (the code itself has more thorough commenting on this). Since any implementation of a Huffman Encoder has to calculate the frequencies of each character regardless of the implementation of other aspects of the Encoding, and this alone runs in O(n) then my program deviates from optimal only by a linear factor.

Limitations of the Design

One major current limitation is that the program loads the entire text document to memory as the form of a string when calculating the frequencies of each character. This problem could be averted by reading the string in line-wise and adding the frequencies one line at a time. This means that there may be memory issues for extremely large text files. Similarly, this is an issue when mapping the input string to the output.

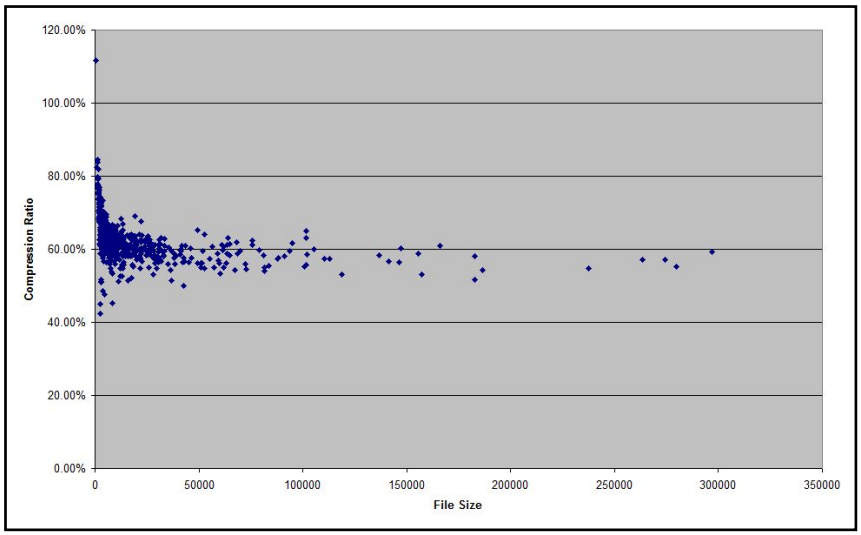
Also, the encoded binary is stored in memory as a string of 0's and 1's, and later converted to binary. Since the characters "0" and "1" use 1 byte each, a string 8 times the size of the encoded string is stored in memory. However, since the program is designed for dealing with text files of natural language, which rarely exceed several megabytes , I decided it was not necessary to potentially introduce more design errors. While the in-program memory of the program is not as efficient as it could be, the program has a strong focus on reducing time complexity and improving the compression ratio.

In order to resolve these problems, I would most likely use a more low-level language, such as C (or create wrappers for C code that could be called within Python) and this would result in a much faster encoding process overall, as well as better memory management.

Experimental Analysis of the Compression Ratio

Since the compression ratio for each file is dependent on the entropy of each individual file, it varies significantly, especially when compressing files that are not simply encoded text. I therefore compressed range of files that I downloaded and their results have been graphed below:

Plain Text Files

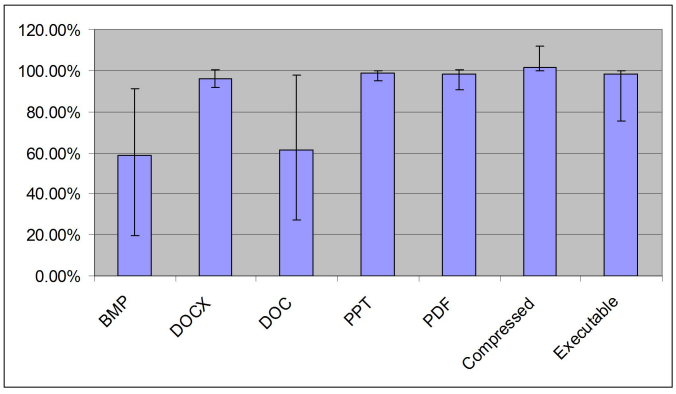


This graph shows that for very small files the overhead of the mapping actually exceeds the disk space saved, and as such, the compression ratio is actually above 100%. However, as the file size increases an initial overview of the graph suggests the compression ratio appears to remain around 60%, with few outliers. Analysis of the data shows that the average is in fact 63.13% (and by ignoring those files of file size less than 2kb, the average is 62.50%). I also performed the analysis on text files of several megabytes, and these were around 60-65% ratio also.

Experimental Analysis of Non Plain Text Files

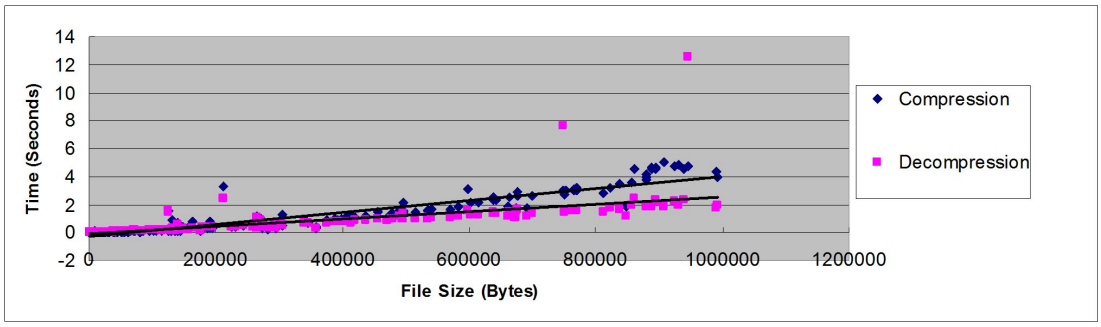
I chose to do a relative comparison of the average compression ratio of a variety of file formats, by compressing a range of files found on my personal machine, and produced this bar chart.

I found that BMP images and DOC documents and have a wide range of compression ratios, ranging from 19.41% to 91.26% and 27.17% to 97.97% respectively. Interestingly, I found that my DOCX documents had a much higher average compression ratio, with little deviation. This suggests that this new format is compressed internally already, and so the encoder has almost no effect. For the compressed files (I grouped together .zip, .7z .rar etc), every single file increased in size, since the compression algorithms used to encode those files are more sophisticated, so my Huffman Encoder actually increased the file size.



The error bars represent the minimum and maximum compression ratios encountered for those file types.

Experimental Analysis of Running Time



I decided to investigate my proposed time linearity of my encoder, by timing the same selection of text files in my first graph, and the results I found seemed to support that proposal (with a few extreme outliers). The amount of time taken for encoding and decoding seemed to be proportional, and both appeared to be linear. The equations generated for the trend lines are:

$$t = 4F - 0.2632$$

$$t = 3F - 0.0654$$

Where t is the time taken and F is the size in megabytes. Ignoring the constants would suggest that the compression rate is around 250kb/s, and that the decompression rate is around 333kb/s.

External Materials Used:

The text files that I used in my analysis can be found at: http://www.textfiles.com/computers/
The other files were files I took from my computer.