Huffman Coding: An Implementation in Python

1 Executing the Program

1.1 Directory Layout

Unzipping the 'HuffmanCoding.zip' file will create a directory organised like figure 1.1. When the program is run, files for compression are taken from the 'InputFiles/' directory and the results into 'CompressedFiles/'. When decoding, files in the 'CompressedFiles/' directory are available and the original result is placed in 'DecompressedFiles/'.

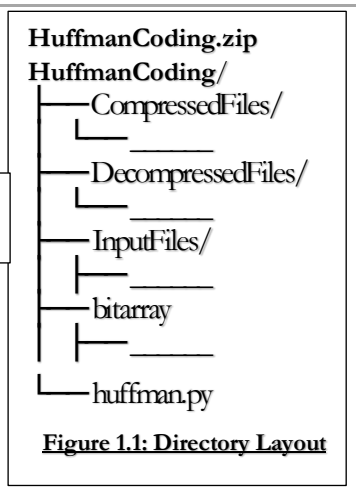


Figure 1.1: Directory Layout

Figure 1.1: The layout of HuffmanCoding

1.2 Requirements

The 'huffman.py' program was written using the 3.6.4 Python Interpreter, which is the most recent version as of 10th February 2018; running the program therefore requires this installation together with the standard library. Modules including time, os, heapq and operator from the standard library are all used.[1]

In addition to this, the 'bitarray' module is also required. This can be downloaded at the command line using:

```
$ pip3 install bitarray
```

Note the specification of 'pip3'. Installation of the module automatically installs to the first library it identifies. So, if earlier versions of Python are available, the module will be installed into those site packages and will not be available to Python 3.6 as it compiles.

This installs the module into the site packages within the python library's framework and therefore makes it available for use with the Huffman program. If unable to download using this method, the module has been included as a directory in the .zip file (see figure 1.1) and can be downloaded from source at https://pypi.python.org/pypi/bitarray/.

1.3 Main Menu: Command Line Interface

To run the program, navigate to the 'HuffmanCoding/' directory and type into the command line:

```
$ python3 huffman.py
```

An intuitive command line interface will then ask for a user input, asking what command should be performed.

1.3.1 Encode and Decode

After choosing either encode or decode, the files available for compression or decompression respectively will be listed. In order to choose a file, simply type out the files name.[2] As explained the outputted file will be placed in the relevant directory. Some example files have been placed in 'InputFiles/' for convenience.

Before being taken back to the main menu, details of the process will be printed. In the case of encoding, this will be the size of the input and output files in bits, the resulting compression ratio achieved by the program and the length of time it took to encode. For decoding, only the time taken is printed.

1.3.2 Testing

A function that takes a specified file from the 'InputFiles/' directory, compresses and the decompresses the file and then compares the resulting decompressed file with the original. An error is thrown if the two files are not identical.

[1] See Section 2 design choices for the justification of using these modules and also for an explanation of why 'bitarray' is used.

[2] It is important to note that the files extension is required in compression but not decompression. The program reads files in binary and therefore has the limited functionality in being able to compress not just .txt files. While, decompressed files as explained will all have a .hc extension by default so inclusion of it is not necessary.

1.3.3 Compress Directory

A function that compresses all of the contents of an individual directory that is in the 'InputFiles' directory using a single command.

2 Implementation description and design choices

2.1 Implementation

When the program is run, the "Implement" function is called. This takes the user to a command line interface, which simply takes a command and then a file to apply it to. The 'os.path' module is used in order to list the files that are available to the user and then subsequently check that the file the user has chosen is available, thereby preventing errors by being unable to locate the file and having to restart the program. The interface will quit the program if there is an unexpected error or if the user inputs 'exit' at any time.

The time taken to run compression and decompression for files is recorded using the 'time' module. The start time is set before a function such as 'encode' is called and is taken away from the time when the function finishes running. This ensures that the time taken for the file to be converted is the only thing being measured by minimizing the amount of other calculations such as the compression ratio.

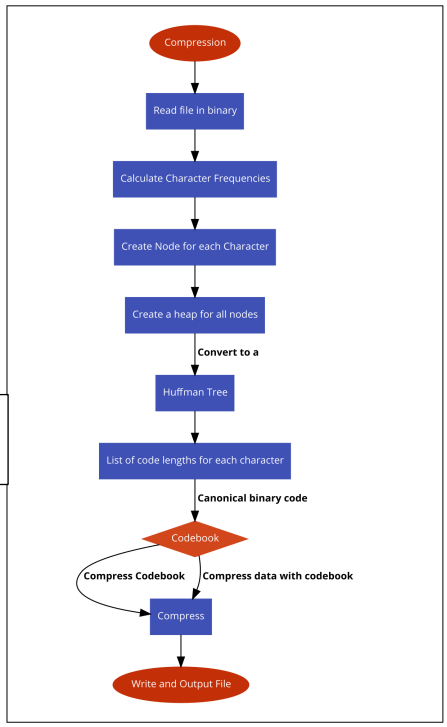


Figure 2.1: Compression/Encoding

2.2 Compression

2.2.1 Character Frequency

After the data is extracted into a bytes object the frequency of each character is calculated by a simple loop through every character. The frequency is then stored as a dictionary in order to easily append the frequency of each character and save on computation when a character that has already been seen reappears. Frequency is used as a representation of the probability that a character will appear.

2.2.2 The Huffman Tree

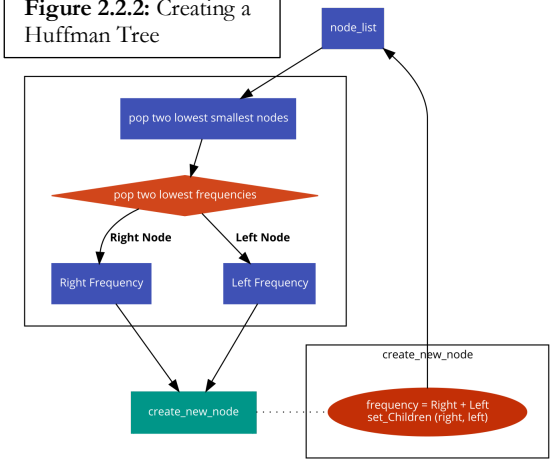


Figure 2.2.2: Creating a Huffman Tree

In order to create a Huffman tree, each node in the tree is created as a separate object. The Node class is used to simulate this, with each node having a left and right child, a value representing the character and the frequency of that character. The class has a function, 'set_children' that accepts a left and right node as parameters and allocates them as children for the current node. The '__lt__' function from the operator module is also present. It sets the frequency as the value to compare between the nodes and is used in the construction of the heap. A list of nodes is created by looping through the frequency dictionary so there exists a node for each character that also has the number of time is occurs in the data.

This list is then converted into a heap using the 'heapify' function from the 'heapq' module. A heap is a binary tree where each of the parents in the tree have a value less than or equal to that of the children. This is the opposite to a Huffman tree, but the heap priority queue is the most efficient way to order the characters for the trees creation.[3]

The Huffman tree can be reverse engineered from this heap by popping two values from the top of the heap, the two currently the smallest in the heap. A new node then is created that has a value attribute 'None' and a frequency equal to that combined of these two nodes. The popped nodes are then set as the new nodes children and then the new node is pushed on to the heap, updating it. Therefore, each time two nodes are popped off the list of nodes, they are made children to a new node which is then added back to the heap with the children behind it; thereby reducing the number of nodes in the list by one. This process is repeated until the list of nodes only has 1 node remaining, which is the root of the Huffman tree with all of the children built below it, ordered by frequency.

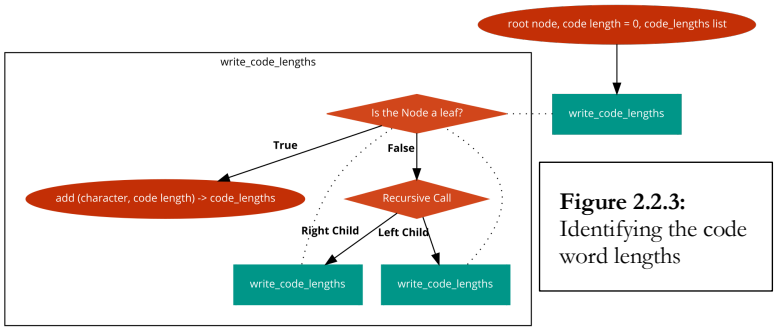


Figure 2.2.3: Identifying the code word lengths

[3] A detailed description of this is avalaible from the python index library at https://docs.python.org/3.5/library/heapq.html

2.3 Generating the codebook

2.3.1 Looping through the Huffman Tree

A node called 'root' now exists which has extending from it every other node ordered by frequency. In order to generate a Huffman code all that is now required is to visit each node and assign a binary code to the leaves at the end of the tree. The codebook is generated and recorded canonically, a design choice justified in section 2.3.2. This means for each leaf it is only necessary to know how many paths there are between it and the root. Generating the code in the traditional manor by adding "0" to the binary code if moving to a left node and "1" to a right node is unnecessary.

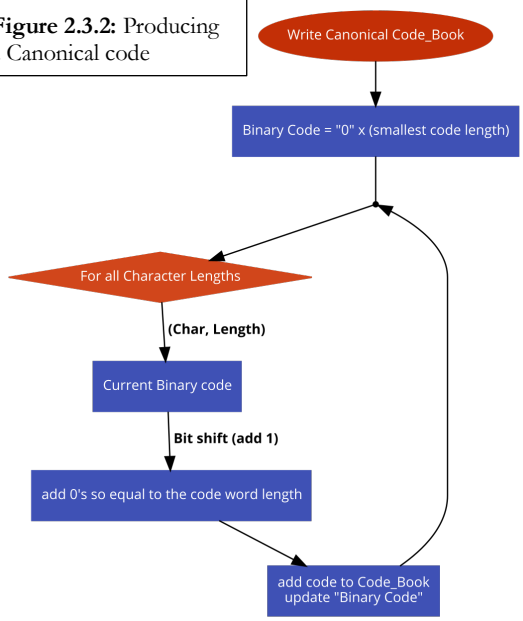
A recursive function called 'write_code_lengths' is used to generate these code lengths. It outputs a list of tuples with each character and the length of its Huffman code. The function takes the node, current code length and the list of tuples as parameters. It is originally called on the root which has length 0. If the value is 'None' and therefore the node is not a leaf, the length of the code is extended by 1 and the function called again for both of the nodes children. If 'None', a leaf has been reached and the current code length is assigned to the list for the character in that node.

2.3.2 Canonical Coding

A canonical code is generated first through Huffman coding. A set of rules are used to describe how the code is generated and it is therefore easy to construct the code given only a bit of information. In this instance, the length of each code word is all that is needed. This means that when storing the codebook in the decompressed file, only the lengths of each character's code word in alphabetical order is needed as opposed to the length, the character itself and the binary code. This saves storage space in the file and makes compression of the codebook significantly smaller.

The list of code lengths is ordered first by the length of the code and then alphabetically. The first character in the list is then assigned a code word with all zeros equal to its length. The next character is then given a code equal to the next in the binary sequence with 0's then appended until the binary code is equal to the length. The current binary code is then updated, and the process repeated for the next character. The length of the previous code word is tracked so the correct number of bit shifts can be achieved. Binary codes for each character are stored in a list of tuples and sorted alphabetically.

Figure 2.3.2: Producing a Canonical code



2.4 Compression

If after compression the resulting output file is in fact larger than the original file, the file is not compressed and the console flags that the file cannot be made any smaller. The user then returns to the main menu.

2.4.1 Codebook

The codebook is stored before the encoded data, so the decoder knows how to interpret it. In the case of some files, canonical storing is not always optimal. Storing canonically means stating the number of bits for each symbol in sequential alphabetical order and replacing 0's when the character does not have a coding. When files use few character types, the number of 0's added to the canonical compression is very large relative to the number of lengths actually stored.

Therefore, two methods of encoding the Codebook are used. The first is simply listing the number of bits in sequential alphabetical order. Before the length of a character is stated, a "1" appears so the decoder knows the next set of digits are the length of the currently counted character. The second states the number of bits for each character followed by the character itself. To encode this, the size of the resulting codebook string is calculated and stated at the start of the string so that the decoder knows what section of the data the codebook is. Both methods are calculated and the one that stores the codebook smallest is used. This is computationally a little less efficient, however since both are calculated simultaneously in linear type by looping through all the character binary codes, minimal time is added. In order to tell which method of encoding has been used, at the start of the encoding, a "1" is recorded to say that data is stored canonically and a "0" if using the alternative method.

2.4.2 Bitarray's and Writing data to file

The bitarray module is used as a way to represent binary data and provide an incredibly efficient way to write to file. The outputted file is opened to writing in 'wb, binary-mode' and a string of binary values is passed into a bitarray. After this, the 'tofile' function economically writes the binary data to the file. The module is written and runs using the c-interpreter and so is computationally incredibly efficient in comparison to manually writing similar functions in python, which dynamically compiles.

Bitarray provides a useful function for encoding and decoding the data according to a codebook using prefix rules, making it perfectly applicable to Huffman coding. When converting the data from the file, bitarray has an 'encode' function, which given a dictionary of characters to their respecting encoding, will compress the data, which can then be written to file.

As writing to a file is in bytes, an 8-bit representation, in order for the decoder to know where the codebook part of the compressed data starts, 0's is appended to the string to ensure the number of bits is a multiple of 8. Bitarray has a function for this, which calculates the number of unused bits. This number of 0's is then added to the start of the output and a numerical value saying how many are there in front of it. At the start of decompression, this number can be read and the correct number of 0's removed to ensure the codebook and resulting data is read correctly.

2.5 Decompression

Decompression is comparatively a simpler process. It involves first extracting the codebook and then decoding the remaining data.

The file is opened in binary format and the bytes converted to bits in a long string. The buffer that ensures 8-bit encoding, is removed and the codebook storing format identified. If encoded canonically, a loop is entered which counts from 0 to 256, each character in the code. If encoded using the alternative method, the buffer at the start is read and the data depicting the codebook is extracted and read as described in 2.4.1. Both of these methods result in a string of each of the characters and the code bit lengths. This list is converted into a canonical codebook using the same method as encoding.

Once the codebook is available, the only remaining binary data from the file is that of the text. This is converted into a bitarray and uses the decode function, which creates decompresses the data from the codebook dictionary, returning a list of all the characters. This list is joined as a string and is subsequently written to the output file.

3 Analysis of results

3.1 Running Time of the Encoder

3.1.1 Time complexity

It can be argued that the Huffman algorithm runs in linear time. The data supports this argument as shown in section 3.1.2, however run time depends on the implementation.

Calculation of character frequencies has time complexity O(n), where n is the total number of characters in the file. However, sorting of the frequencies for the construction of the Huffman tree using the heap requires O(nlog(n)) time to sort, with O(log(n)) required to determine the lowest frequency and then to insert the new one. This is repeated for O(n) iterations. Sorting is also required after the codebook is constructed in order to encode itself in sequential alphabetical order. The size of the alphabet however, of course is fixed and independent of the size of the input file. As a result, Huffman encoding runs in time O(nlog(n)).

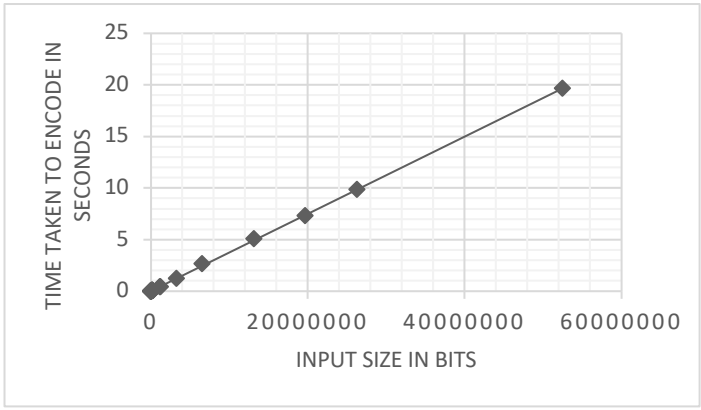


Figure 3.1.2: A Graph to show the encode time in seconds compared to the input size in bits

3.1.2 Run Time

Figure 3.1.2 shows graphically how the time taken for a file to be compressed in seconds changes with the size of the file in bits. A strong linear relationship is shown confirming the time complexity. The only variable for which the running time is dependent upon is the size of the input file and run time seems to be not be significantly affected by the types of files compressed.

3.2 Running Time of Decoder

Running time for the decoder has a larger standard deviation than that of encoding. This might be explained by how decompressing files takes significantly longer than compression so variations in results are more distinct. While time complexity is similar for decompression, it takes significantly longer due to the number of times that the codebook has to check whether current data matches a prefix.

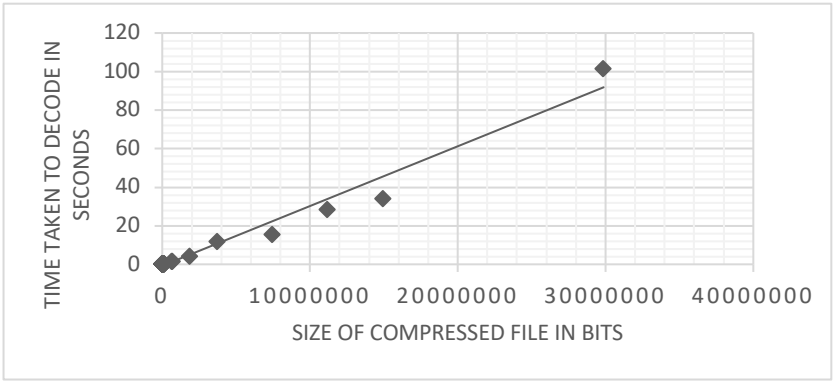


Figure 3.2: A Graph to show the decode time in seconds compared to the input size

3.3 Compression Ratio

The average compression ratio for a group of 30 files is 1.76 with a standard deviation of 0.0873 and therefore does not vary significantly over normal ".txt" files. The ratio is completely independent of all other variables,[4] however it does change dramatically depending on the contents of the file.

[4] The time taken to compute, size of the input file and number of characters when it is sufficiently large.

Taking 50 files which contain 10,000 randomly generated strings of length up to 10, a much poorer compression ratio was produced with a mean ratio of 1.36. However, where the algorithm performs best is when only a few character types are used in the file. This reduces the average code length of each symbol and so the resulting encoding is significantly shorter than the original. When say only 4 characters are used, the maximum code length is 3, which in comparison to 8-bit storage is a significant reduction. Compression ratios therefore exponential improve as the number of encoded characters decrease. If pretty much all characters are encoded in a file in roughly equal frequencies, the advantages of Huffman coding are drawn out and the compression ratio reaches as low 1.05 for files larger than 1mb. This is because the average code length is closer to 8-bit coding.

4 Limitations and their Possible Resolutions

4.1 File Types

Since the files before encoding are read in binary, the possibility for not just '.txt' files to be encoded exists. Compression of other file types is available; however, decompression is not as doing so would require storing the extension of the original file, which has not been implemented to optimize the current version.

For files such as JPEG and PDF which already use a form of Huffman compression to store data anyway there is little point in compression, however the creation of such files from ordered data through this Huffman algorithm is very much possible and could easily be implemented by changing the encoding to match the type of data it is accepting.

4.2 Use of Bitarray

The use of Bitarray is a potential issue. If a user is unable to download the module or operate a c-interpreter within python, this acts a limitation to the current design. There are alternatives to using Bitarray such as using the creation of bytes objects that can be written directly to file. The reason this were not used is due to the obvious efficiency of the Bitarray module.

In addition to this, using the module and its capabilities means working around its requirements. The modules for encoding requires the codebook to be in the form of a dictionary while when it is originally derived it is a list of tuples, ordered for using in canonical code. The conversion of to a dictionary while minimal is a computation inefficiency that would be unnecessary in another implementation or if Bitarray was not used or if the module was altered to meet specific needs before installation.

4.3 Python as a Programming Language

In terms of computation time, python is very inefficient. It is dynamically typed and therefore requires more time to interpret data and allocate it to variables in comparison to C or Java, which compile all code before running it. An implementation in another language could dramatically improve computation time. As already stated, the Bitarray module is written in C and therefore the implementation is partly there. Converting the implementation into C won't alter compression ratio but will improve computation.

4.4 Numbers of characters available

Currently the Huffman algorithm is unable to encode and decode an alphabet larger than 256. This is implemented in order to write and store the codebook canonically. In order to avoid writing how long the codebook is, during decompression 256 characters are cycled through at which point the program know it has read the full code word lengths. Altering this would not be difficult. The additional method used to store the codebook could simply be permanently implemented, allowing for a larger range of potential symbols, including uni-code for example. This is not currently in place in order to utilize the advantages of canonical coding.

4.5 Phrases Codebook

The compression ratio is not particularly effectively for larger files. Due to how the codebook is created canonically, small files of the right size have excellent compression ratios, but larger files let it down. This Huffman implementation does not include any method for aggregating common phrases perceived in a text. This is currently not implemented for two reasons. Firstly, since a canonical codebook is order alphabetically, encoding combinations of different characters and still storing canonically will get incredibly complex. Secondly, identifying common chunks of text for the codebook is very computationally heavy. It does depend on the level of analysis that is implemented but such methods would exponentially increase run time.

However, implementing such a program that identified such patterns would be incredibly useful for certain types of files. If for example, a book is being encoded in English, the word 'the' will appear very frequently and would be easily identifiable as patterns could be checked for by spaces. Executing such techniques would increase the time taken to both encode and decode files however, the compression ratio achieved will be significantly better. Diminishing marginal returns to additional computation time will be experienced as more focus patterns are identified but the correct balance between the two could produce a very effective implementation.

An alternative to searching for phrases could be a Lempel-Ziv implementation, which uses a sliding window to track sequences of characters that have already been encoded and then references to that incursion. Such a method could be implemented to aggregate common patterns before implementing the Huffman Algorithm, therefore maximizing the benefits of both systems. However, this would of course extend the run time of the resulting algorithm.