Digital Communication Coursework

Kbzh45

Execution Instructions

First, use pip or another python module installer in order to '*pip3 install bitstring*'. To run the Huffman encoder on myfile.txt or any text file, navigate to the appropriate directory and type in console: '*python3 encodeDecode.py encode myfile.txt*'. To run the decoder, repeat but with 'decode' replacing 'encode' in the command, and run with a .hc file extension: '*python3 encodeDecode.py decode myfile.hc*'. It is also possible to run both the encoder and decoder one after another, by simply removing the encode or decode commands. This is only possible with an input file extension '.txt'.

Program Description

Encoder

The encoder begins by reading through each character in the input data and adding to a dictionary the character frequencies. A default dictionary is used so that ascii characters can be referenced as a key without needing to first be added to the dictionary. This improves efficiency as the dictionary did not need to be checked for each character in the data.

The frequencies dictionary is used to create a Huffman Tree for encoding in *createTree()*. At each iteration of the while loop the function combines the two least frequent characters, and simultaneously adds either a '0' or '1' to their code words depending on whether they're considered the left or right child. The function returns the newly created tree of code words, sorted first by code word length and second by alphabet for canonical coding. Separate functions for creating the tree and traversing it to build the code words was less efficient, as doing both simultaneously saves time in twice traversing the tree, and follows the same principles of Huffman encoding, creating the same code words as more generic methods, in shorter time.

Canonical encoding

Because Huffman encoding requires the encoding and decoding processes to be separate, so that the program can be deployed at different times and in different places, information to build the tree must also be in the encoded file. Storing the entire tree in the encoded file is possible but inefficient, especially when dealing with files where large numbers of characters in the text are unique, and therefore storing

the entire tree takes up a large portion of the total size of the file. Canonical encoding requires only the length of each character's code word be stored in the file.

The code words are recreated using canonical Huffman encoding, meaning they keep their length but are redefined so that each code word of the same length has a binary value one greater than the alphabetically previous character. This allows for the decoder to recreate the tree for decoding with the minimum information stored in the encoded file, making for better compression ratios.

My canonical method has a 'for' loop to work through each code word and writes in the binary value of the previous code word plus one, and pads the code word with '0's until it's same length as it was previously. Canonical encoding adds a step, so is slower, but more space efficient. The result of using canonical encoding is relatively slim for large files, however for smaller files it can be very effective - a file of 21 bytes could be expected to increase in size when the entire tree is stored in the compressed file, however in my program, the compressed size was also 21 bytes, and all files larger had a positive compression ratio.

Once canonically reconstructed, each character-length pair is converted to binary and appended to the front of the binary encoded string. The built in bin() function is used, converting a decimal to its binary, along with ord(), which converts a character to its ascii numerical value. Characters and number are stored this way so they can be decoded without a key - using only ascii.

After the character-length pairs, a '}' bracket is added, which marks the end of the tree section. Because it is possible to have a '}' appear in the tree section with an associated length, a '}' must not appear before the end of the tree and stop it prematurely. To solve this, the character-length pairs are stored by length first, then character, and the decoder checks 16 bits at a time. The first 8 bits will always represent the length of a string as a number, and when a curly bracket is found in the decoded first 8 bits the tree section has been fully covered.

The function *createCode* is called. It consists of a loop which, for each character in the original text, adds the code word corresponding to that character to the string. The list of character-code word tuples is first made into a dictionary, allowing for each character to be used as the key to quickly find the corresponding code word more efficiently than traversing a list at each iteration.

When converting the binary string to a bitArray to be saved to a binary file, additional zeros are added to the bitArray to make it a multiple of 8 (stored as bytes). This could cause the decoder to add additional characters to the end of the decoded text. To avoid this, the number of zeros to be added is calculated using the modulo function and added to the encoded file. The file is saved in binary for

maximum efficiency - converting 8 bits to an ascii character and writing a new text file had a compression ratio of on average 0.85, whereas in binary it is ~0.5.

Decoder

The binary file read in to the decoder must first be converted to a string of 0s and 1s, which is done using python's 'format' and 'str' functions. The byte representing the number of zeros to be removed from the encoded text is also separated and converted to ascii.

Before decoding, the character lengths must be separated. This is implemented with a recursive function, which continues to decode 16 bits and append them as tuples to a list of lengths, while checking for the end condition - that the first 8 bits is decoded as the symbol '}'.

This part of the program, while dealing with only a small amount of the data, took a large portion of the total time for the algorithm when it was first implemented. While testing, it appeared that this was because of the code 'binary = binary[16:]'. Intended to remove the first 16 bits, this code rewrote the entire string each time, taking far longer than was needed. Instead, what is implemented is a variable 'bitsToGo' which represents how many bits must be removed from the front of the string after the lengths have been generated in order to leave only the encoded text. This only requires the binary string be rewritten once after the recursive process is complete, greatly improving efficiency.

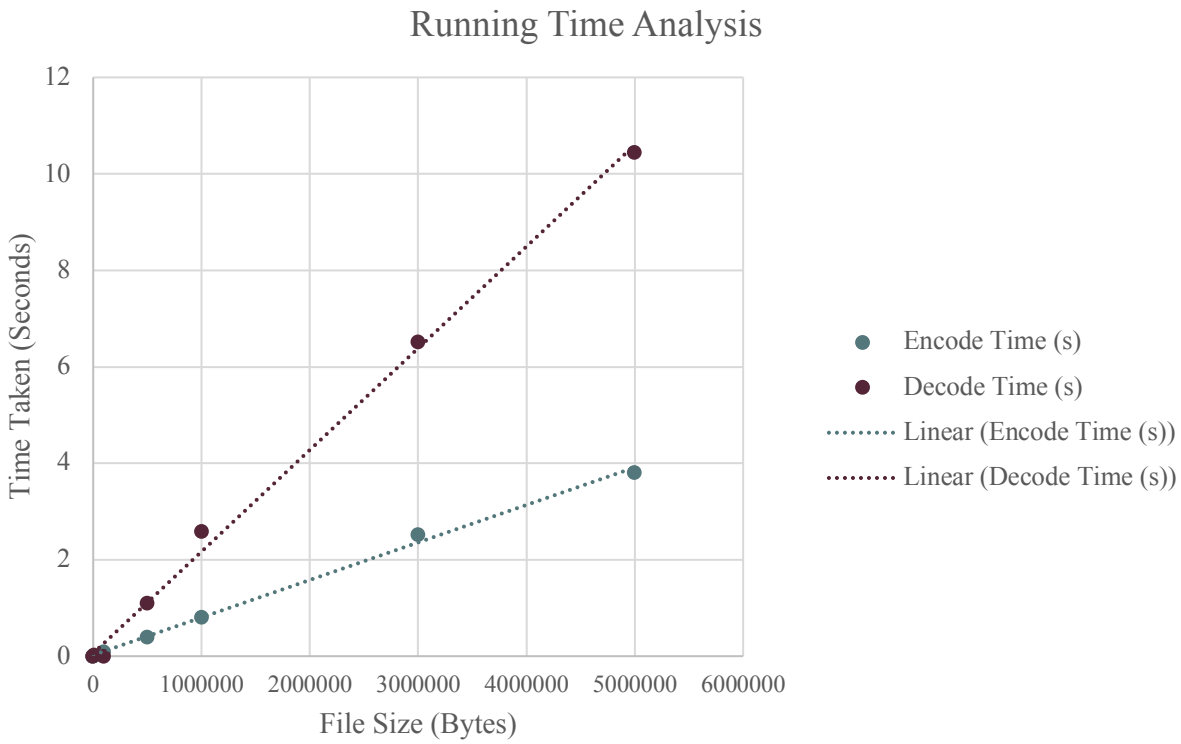
The additional zeros are then removed from the string, using the variable saved. The lengths of code words can now be used to recreate the full code words using the canonical method; the shortest code word is all zeros, and the following code words are created by adding 1 to the binary value, then padding with zeros. The code words are stored as a dictionary, allowing the decoder to look up code words as keys and find the corresponding characters.

Cycling through the binary with while loop checking the length was not equal to zeros seemed logical, however encountered efficiency issues as it required rewriting the string each time a code word was found in exponential time. Therefore instead, a 'for' loop adds the current bit to a variable called 'letter'. At each iteration the dictionary keys are checked to see if 'letter' matches one, which would mean a code word has been found. In this case, 'letter' is reset to an empty string and the corresponding character is added to the new text string.

Once the 'for' loop has ended, the full text has been decoded as a string and is written to a .txt file.

Running Time Analysis

The graph below demonstrates the running times of both the encoder and decoder on a variety of file sizes up to 5MB. The trend lines show a linear correlation between file size and time taken, suggesting that the program runs in complexity O(n) for a file size n. Individual encoding and decoding times deviate from the trend lines due to the nature of the files themselves; files may contain very few or many repeated



characters, and therefore time taken creating more code words or searching larger trees could affect the total time taken for the encoder and decoder.

Encoder

To encode a file of size n, the data must be traversed entirely twice, once to calculate the character frequencies, and once to convert each character to a code. Therefore there are 2n operations involved. The other operations are either done in constant time, or O(f), where f is the character set of the input. As f is finite (f<256 for ascii) and by definition f < n (cannot be more different characters than total characters), f does not affect the time complexity of the encoder.
Therefore the encoder is O(n), which is corroborated by the results.

Decoder

To decode a file of original size n, first each 8 bits of the encoded file are looped through in s/8 steps. s [size of encoded file] is O(n) as each of n characters are encoded to binary. Operations relating to the list of lengths are all O(f) for the character set of the input, and therefore n operations are only required to convert the

binary string back to n characters once the correct decoding dictionary has been created.

Therefore the decoder is also O(n), which the results corroborate. The decoder takes more time than the encoder because the length of the binary string > n, but still linear, and therefore instead of requiring n operations, n * (average code word length) operations are required.

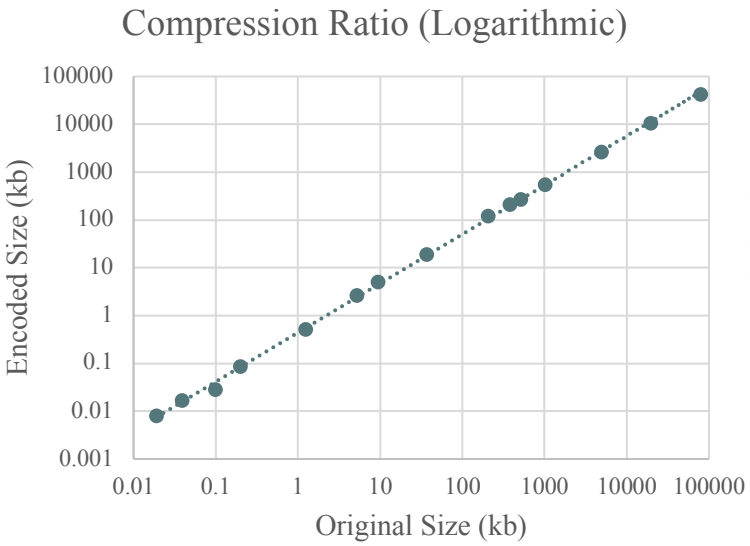
Compression Ratio

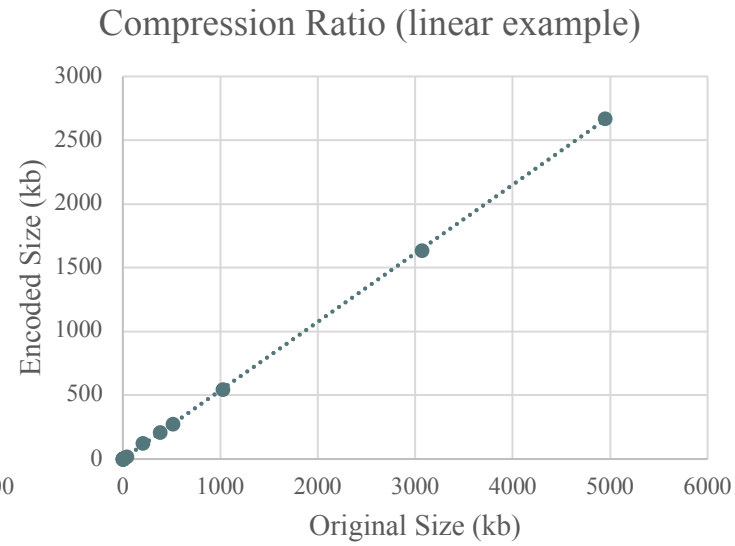
The graphs demonstrate the compression ratio of the program for a range of file sizes. The largest file to be tested was ~80MB, and due to the spread of results, the compression ratio is presented both with logarithmic axis and linear axis (the linear graph excludes the larger file sizes). The average compression ratio is calculated from the entire set of raw data.

Deviations from the trend lines of the graphs are due to the variation of file types, with randomly generated files having lots of similar character frequencies, reducing efficiency, while text files of literature have large numbers of characters such as spaces and vowels which increase the efficiency of the compression.

Average Compression Ratio : 0.533

The average compression ratio of 0.533 means that on average, for an ascii character of 8 bits, the binary string length is 4.





Limitations and Additional Features

Using character-length pairs with Canonical Huffman Encoding

Using Canonical Huffman encoding, maximum efficiency and the lowest compression ratio is achieved by storing as little information as possible about the encoding method in the compressed file. At its most efficient it is possible to only store the length of each code word and reconstruct the tree from that information alone. However to do this, the entire alphabet of characters to be encoded must be known, and a length provided for each character's code word. For larger files with small alphabets this is more efficient. A limitation of my design is that it does not store only the lengths, but also includes the symbol itself encoded using ascii and stored as a binary byte. This is because many text files use only a small number of the available characters, and so storing the code word lengths for characters that aren't used would be a waste of storage space.

The method implemented is more efficient while less than half of the possible ascii characters are used in the input text (less than 128). When more than 128 characters feature in the input text it would be more efficient to store only the length of each code word and list them in the order they appear in the ascii table, which could be traced back, and would allow one bite to represent each character in the table. Meanwhile, with less characters this method would simply lead to a large number of zeros to represent missing characters, which is space wasted. Storing character (1 byte) and length (1 byte) is half as efficient but avoid the kind of redundancy caused by alternative methods.

ASCII encoding fails for large alphabets (Unicode)

The code does not run for texts that include symbols other than those in the ascii alphabet. This is because ascii includes 256 symbols, which is all permutations of 8 bits. Therefore, the encoder changes each ascii character into 8 bits along with an 8 bit string for code word length. Running the code with non-ascii symbols (Unicode) can result in an number greater than 256 being converted to a binary number greater than 8 bits long. This is then stored in the binary file and throws off the decoder as the length difference is not taken into account. A way to factor in larger libraries could be to increase the bit length of stored characters to 16 bits each or more, however for reasons outlined below, large alphabets are less effectively compressed and so is not necessarily a good solution.

Effectiveness when used with diverse texts

Huffman encoding relies on some characters appearing regularly, allowing shorter code words for common characters to reduce the total file size. However, when dealing with diverse alphabets like Chinese, or with files randomly generated which don't share

common language rule like the frequency of vowels, Huffman encoding proves far less efficient. In the Chinese case, because so many words are unique symbols and not made up of letters, the number of words which each would need to be encoded would be great, and the average code word length would also be huge. It is probable that compressing Chinese text would have very little effect due to the diversity of the language.

Heuristic to reduce comparisons during decoding

It is possible to reduce the time taken in decoding by avoiding unnecessary comparisons where a match is not possible, by looking at length of smallest code word, which is minimum word length in encoded string. Time is wasted when checking a single bit in the string to see if it matches a code word when the known minimum length is, for example 4. It would be more efficient if, each time the search for a new code word match commenced, the length immediately jumped to 4 (in this example), and checked for matches from that length onwards, not checking if the first three bits matched because they could not when the shortest code word is of length 4. At its most efficient (where the shortest code word length != 1, and the longest is not significantly longer) this method could significantly reduce the time taken to decode the string, which is the most significant factor in the efficiency of the algorithm.