1 Execution

Please use Windows on one of the Network connected Durham machines. Through the app hub, please run the latest version of Python 3 (3.6.3). Then, before running my programs, please install the *chardet* package by running the command:

```
J:\> pip install chardet
```

If no (output) is specified in the following commands, the default "encode.hc" and "decode.txt" will be used instead. Please ensure that the *.txt* and *.hc* file extensions **are** included, and the square brackets and parentheses **are not**. If the output file already exists within the directory specified, the user will be prompted before the file is overwritten. If an incorrect number of arguments are specified, a 'usage' message will be displayed. Please use absolute file paths, unless the file to be encoded or decoded is located within the same directory as the *encode.py* or *decode.py* programs. As the the two programs share functions (but not data), please ensure they are located in the same directory.

1.1 encode.py

To encode a file, please issue the following command:

```
J:\> python encode.py [input] (output)
[input]: The path to the input .txt file to be encoded
(output): The optional filename of .hc file to write the encoded output to.
```

1.2 decode.py

To decode a file, please issue the following command:

```
J:\> python decode.py [input] (output)
[input]: The path to the input .hc file to be decoded
(output): The optional filename of .txt file to write the decoded output to.
```

2 Implementation

2.1 encode.py

I focused on using a 'defensive' coding style for this project, ensuring all possible errors are caught and appropriate messages are displayed back to the user. For example, my programs ensure files with the correct extension are inputted, and only 'known' encoding types are handled. The program begins by detecting the encoding type of the the file, using the *chardet* library. If a non-text file is inputted, the file does not exist, or the character encoding cannot be detected, an error message will be displayed to the user and the program will exit gracefully. The *chardet* library uses multiple detection methods to determine the encoding of the source, including: BOM detection, escaped encoding detection and single/multi-byte encoding detection. During my analysis, I realised that this library adds a considerable amount of delay to the execution time of my program for specific files, which I will discuss in greater detail in the following section. The ENCODING global variable is set if the encoding is detected successfully, which is then used throughout the rest of the program. The input file is then opened with the correct encoding type, which is vital for the program to run correctly. Currently only ascii, UTF-8, UTF-8-SIG and UTF-16 encoding are handled, but the program is designed with modularity to ensure further encoding types can be added in the future.

Once the file has been read as a single string, the frequencies of the characters within the file are calculated by the *countFrequency*( ) function, which uses the *Counter* library's *most_frequent*( ) function to efficiently calculate the frequencies of the characters within the document. A character/frequency dictionary is returned. The *createNodes*( ) function is then called, which loops through every character in the character/frequencies dictionary and creates a *Node* object for each one. The *Node* class has six instance variables: *frequency* - the frequency of the character within the document (initially 0), *lNode* and *rNode* - the node's two children nodes within the binary tree (initially 'None'), *char* - the character the node represents (initially '-1', representing an internal node of the tree rather than a leaf), *depth* - the depth of the node within the tree (initially 0), and finally *code* - the Huffman code assigned to the node (initially ' '). Each node is added to two lists: *nodes* - a binary-heap priority queue used to construct the tree, and *nodeList* - a list of all the nodes within the tree, including the internal nodes.

The *buildTree*( ) function is then called, which constructs the tree. With each iteration, the function pops the two nodes with the smallest frequency off of the priority queue, and creates a new node whose *lNode* and *rNode* instance variables are set to the two nodes just popped off the queue, and whose frequency is set to the sum of the two frequencies. This new node is then pushed back on to the queue, and also added to *nodeList* (the list of all the nodes). With each iteration the structure of the tree is 'implicitly' created ('implicitly' as the tree structure itself is not stored explicitly), by linking pairs of nodes together. The function returns a list of all the nodes, and the root node. For the priority queue implementation I used Python's *heapq* library, which uses highly efficient binary trees in its implementation. The time complexity of heapq's *heappush* and *heappop* functions is O(log(n)), where n is the number of nodes in tree, which is highly efficient. I also had to define a *__lt( )__* function within my Node class, to instruct the *heapq* module on how to compare two Node objects when maintaining the heap invariant.

The algorithm I used to generate the (canonical) Huffman codes later on in the program requires only the leaves of the tree (as leaves represent characters), and the depths these leaves are at within the tree. However, during the creation of the tree in the previous step, I was not able to design an efficient system that could update the depth of the existing nodes in the tree when a new node is created (without performing a recursive 'depth first update' down the tree every time a new node is created). Therefore after creating the tree, I call the recursive function *findDepths*( ) once, which sets the *depth* instance variable of each node within the tree. As generating the Huffman codes requires only the leaves of tree, I then call the *removeInternalNodes*( ) function on my list of nodes.

As it can be proven that the Huffman codes themselves are optimal, the main source of further compression arises from how the decode table is stored at the start of document, which enables the decoder to decode the document. The first significant increase in storage efficiency can be realised by generating *canonical* codes, as opposed to 'normal' ones. When canonical codes are generated, all characters with codes of a given length (which equates to the depth of the leaf within the tree corresponding to that character) are assigned their codes sequentially. When combined with the representation of the tree that I used (and will discuss shortly), this allows for an extremely efficient representation of the decode table, as we no longer have to explicitly store the tree shape, but instead the number of nodes and leaves at every level, as the 'shape' can then be inferred from this. In fact, only the number of leaves at each level needs to be stored, as the total number of nodes at any level can be inferred from the total number of internal nodes in the level above. To ensure canonical codes are generated, my *sortLeaves*( ) function uses a lambda function to sort the list of leaves first by depth, and then by alphabetical character value, which has the effect of 'canonicalising' the tree, as shown in Figure 1. As the order of the leaves within the canonical tree are sorted in a predictable manner, the decoder is able to re-assign characters to the leaves of the tree correctly.

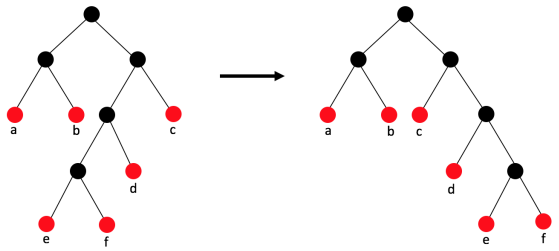

Figure 1: Converting a normal tree to a canonical tree

Once the tree is in canonical form, I allocated the codes to each character/leaf with the *generateCodes*( ) function. This function avoids using recursion and iterates through the list of leaves, allocating the binary codes in sequence, of length equal to the depth of the node, to each leaf at a given depth. When the next level is reached, the binary code is appended with zeros until its code length is equal to its depth, and then binary codes are again allocated sequentially, starting from this new binary value. The function assigns these codes by updating the *code* instance variable of the nodes (leaves). Once all codes have been allocated, the *findLevelsAndLeaves*( ) function returns a list of the total number of nodes at each level, and a list of lists of leaves at every level, which are used in the creation of the decode table.

As mentioned earlier, *any* correct implementation of Huffman's algorithm should produce prefix-free optimal length codes, and so I focused on how efficiently I could represent the decode table which is prepended to the output *.hc* file. Historically, a dictionary of characters to codes was prepended. However, this method is extremely space inefficient, particularly for relatively small text documents. A more modern approach is to represent the tree shape itself using a combination of zeros and ones, inserting the character codes when a leaf is reached in the tree, and then have the decoder reconstruct the character/code dictionary itself. This can be further optimised by using a canonical tree - here, only the number of leaves at each level has to be stored, followed by an ordered list of characters that are allocated, left to right, to the leaves of the tree by the decoder. I implemented an even more efficient version of this algorithm, which uses a base-2 encoding of the number of leaves at each level, followed by an ordered list of characters. *An Efficient Implementation of Huffman Decode Table* (D. R. McIntyre, F. G. Wolff,) details this algorithm, which I implemented in my *buildDecoderTable*( ) function.

For every level within the decode tree, the function performs the following steps to generate the *binary code* for that level:

1. Find the total number of nodes and number of leaves for the current level, from the function arguments.
   (a) If the number of nodes is a power of two, **and** if the number of leaves equals the number of nodes, the *binary code* for that level is *binary(*number of leaves - 1*)*, followed by a **1**.
   (b) Else, if the number of nodes is a power of two, **and** if the number of leaves equals the number of nodes **- 1**, the *binary code* for that level is *binary(*number of leaves*)*, followed by a **0**.
   (c) Else, the *binary code* for that level is *binary(*number of leaves*)*.
2. If the length of the *binary code* is **less than** the log_2(number of nodes), prepend the *binary code* for that level with the required number of zeros.
3. Append this *binary code* to the *tree shape* string, and append the characters relating to the leaves at this level **in order** to the *character list*.

Thus, the *binary code* for each level in the tree is a binary representation of the *number of leaves* at that level, padded with zeros to ensure that its length is equal to the base-2 logarithm of the *total number of nodes* at that level. This string contains all the information required for the decoder to be able to deduce the number of leaves and internal nodes at the current level, and from this deduce the total number of nodes at the level below, and thus the number of bits it should read to find the *binary code* for the following level. Perhaps the only non-trivial parts of this algorithm are stages 1.a) and 1.b), which handle the problem that an n-bit binary number can only hold the decimal numbers 0 to 2^n - 1, and not 0 to 2^n . For example, a tree level with 16 nodes usually requires a $log_2(16) = 4$ bit code, but should the level have 16 leaves, 5 bits would be required to represent 16 in binary (10000). The algorithm in parts 1.a) and 1.b) handles this situation by encoding both 15 and 16 as 1111, and then adding another 1 if the number of leaves is 16, and a 0

if the number of leaves is 15. Although somewhat convoluted and complicated, this method is more efficient than using an extra bit for every level. The function returns the *tree shape* string and *character list* array, the latter of which will be encoded to a string too, which combine to form the decode table.

Penultimately, the *encodeText*( ) function iterates through every character in the text file, and appends to a string its Huffman code. The output file is then written by the *writeToFile*( ) function, which first encodes in the *character list* list into binary, using the encoding detected by *chardet*. Currently, the output string to be written to the file is:

`[Tree Shape String] [Character List String] [Encoded Text String]`

However, this string of binary digits may not in fact be a multiple of 8, and so will not be correctly converted to an array of *bytes*, Python 3's native primitive type for handling binary outputs. Therefore, we need to pad our string with 0s to ensure its length is a multiple of 8. We then need to store a binary representation of the amount of padding used in our header, to ensure the decoder knows when it has reached the end of the file. The maximum number of 0s required to bad the binary string to a multiple of 8 is 7, so we need three binary digits (000 - 111) to represent the amount of padding required. However, this presents another problem - we now have a spare 5 bits in the header. I considered simply padding this with 0s too, but instead I used the spare bits to represent the encoding type of the inputted text file. This allows my program to handle 32 different encoding types, only four of which are currently used (00000: ascii, 00001: UTF-8, 00010: UTF-16, 00011: UTF-8-SIG). In the future, I will extend my program to handle all the major encoding schemes. Our final output string is now as follows:

`[Encoding Type] [Padding Size] [Tree Shape] [Character List] [Encoded Text] [Padding]`

which is converted into an array of *bytes*, and written to the output file specified by the user, or the default output file.

2.2 decode.py

The program begins by opening the file in binary mode, ensuring the file is a *.hc* file. The *bytes* are converted into a consecutive string of bits, giving the entire binary string that was written to the output file by the encode program. By looking at the first eight bits, the encoding type and padding size are detected by the *detectEncodingAndPadding*( ) functions. This string is then passed to the *buildTree*( ) function, which decodes the table and rebuilds the tree simultaneously.

Initially, we start at level 1 (level 0 is the root node), as we know there will always be two nodes at this level. We create a root Node (using the same Node class as in our encoding program), and two child nodes, assigning the *lNode* and *rNode* instance variables of the root node to these two nodes. We then add these two nodes to a *nodes at this level* list, and whilst the variable *number of nodes at this level* > 0, we perform the following steps:

1. Set the number of bits to read as log_2(number of nodes at this level), and 'read' this many bits of our *binary string*.
2. If the bits read are a sequence of repeated 1s, we are in the undetermined state detailed above, so we read the next bit.
   (a) If the following bit is a **1**, then the number of leaves at this level equals the number of nodes at this level.
   (b) If the following bit is a **0**, then the number of leaves at this level equals the number of nodes at this level **- 1**.
   (c) Otherwise, the number of leaves at this level equals the decimal representation of the binary number read.
3. The first 'number of leaves at this level' nodes of our 'nodes at this level' list are the leaves at this level (i.e the red nodes at a given level in our canonical Huffman tree in Figure 1 above), and so are added to a separate list 'leaves'.
4. The remaining nodes in the 'nodes at this level' are therefore the internal nodes for that level. These are iterated through, and for each internal node two children nodes are created and linked. These children nodes are all added to a final list, 'nodes at the next level' - the length of which is held in the variable 'number of nodes at next level'.
5. Finally, the 'nodes at this level' is equated to the 'nodes at the next level', and the 'number of nodes at this level' is equated to 'the number of nodes at next level', and the algorithm performs the next iteration, unless we have reached the bottom of the tree.

Once the tree has been created, I simply needed to allocate the characters in the order they appeared in the character string to the leaves in the order they appear in the list of leaves. As I encoded the characters based on the detected encoding type, I had to allow for the various bit length encoding schemes when decoding the binary back into the correct character. Finally, I reused my *sortLeaves*( ) and *generateCodes*( ) functions from my encode.py program to regenerate the canonical Huffman codes for the leaves, thus reducing the code redundancy and repetition significantly. It is worth noting here that the *generateCodes*( ) function returns two dictionaries: a character to code 'codes' map, and a code to character 'decodes' map. The encode and decode programs use either the 'codes' and 'decodes' maps respectively, but in using the same function to output both I reduced code redundancy and did not have to iterate through the 'codes' dictionary, swapping keys with values, to find the 'decodes' dictionary. Once the data has been decoded using this dictionary, the output text is then written to a user specified *.txt* file in the encoding type of the original document, or to the default output file if no *.txt* file is specified by the user.

3 Analysis

3.1 encode.py

I began my analysis by rigorously testing the time efficiency of my encode program as a whole, and then continued by testing logical groups of functions, to establish where the main sources of time inefficieny. I tested my encoding program on several text files of different sizes and encoding types using the highly accurate *timeit* library - the results of which can be seen below in Figure 2.

Filename	Encoding Type	Original Size	Read & Detect Speed	Count Frequencies Speed	Build Tree and Generate Codes Speed	Build Decode Table Speed	Encode Speed	Write Speed	Total Encode Speed	TES - Detect Speed
yw50.txt	UTF-8	51,185	0.268409916	0.010296201	0.001506202	0.000978799	0.009930301	0.01785649	0.308977908	0.040567993
ow140.txt	UTF-8-SIG	142,384	5.31052E-05	0.0094944	0.000668079	0.000436132	0.023227561	0.068928546	0.102807824	0.102754718
aw170.txt	UTF-8-SIG	173,595	0.000605302	0.011389382	0.00063999	0.000305301	0.000745459	0.10099965	0.114685084	0.114079782
sh560.txt	UTF-8	581,878	3.229389215	0.009436867	0.019619318	0.009863816	0.020337853	0.231043845	3.519690914	0.290301699
pp700.txt	UTF-8-SIG	726,223	0.00222725	0.044495471	0.001656303	0.00111586	0.125885812	0.298359012	0.473739708	0.471512458
illiad1100.txt	UTF-8-SIG	1,201,891	0.003679419	0.073203938	0.004591604	0.004016336	0.217269709	0.502433803	0.805194808	0.801515389
md1200.txt	UTF-8	1,276,200	7.150002768	0.152530551	0.100156267	0.042959704	0.268924365	0.476851737	8.191425392	1.041422624
uly1500.txt	UTF-8-SIG	1,580,890	0.00513727	0.099613227	0.001685537	0.005091759	0.283320157	0.714405245	1.109253196	1.104115926
bk1900.txt	UTF-8	2,044,191	11.82255681	0.131507923	0.017261434	0.133752315	0.36207821	1.000024318	13.46718101	1.644624201
lm3200.txt	UTF-8-SIG	3,324,334	0.008103932	0.210359344	0.005014538	0.002295975	0.589766351	1.442232614	2.257772753	2.249668821
wp3400.txt	UTF-8	3,351,415	20.00997645	0.227755148	0.001564536	0.005889081	0.577743641	1.426914983	22.24984383	2.239867388
wp3400.txt	UTF-16	3,351,415	-	-	-	-	-	-	2.257361413	-
bk1900.txt	ascii	2,044,191	-	-	-	-	-	-	13.24453702	-
chinese4800.txt	UTF-16	4,790,018	-	-	-	-	-	-	2.737599469	-
cryllic2000.txt	UTF-8	2,018,559	-	-	-	-	-	-	25.97357924	-

Figure 2: Each timing was performed 20 times and averaged

Initially, I believed the main sources of added time would be in counting the frequencies of the characters within the document, and then encoding the text itself. However, it soon became apparent that it was in fact detecting the encoding type of the document, to ensure the correct encoding is used, that was significantly delaying my program. Specifically, the *chardet* library seemed unable to efficiently detect the encoding of standard UTF-8 documents. For the large *wp3400.txt* file (which is 3351415 bytes in size) this was particularly apparent, with the library taking over 20 seconds on average to detect the encoding. However, when I tested the program on UTF-8-SIG encoded files, which unlike (most) standard UTF-8 documents are prepended with a Byte Order Mark (BOM), the library was able to detect the encoding rapidly, even for large files. Therefore I believed a comparison of the total encode speed *without* the detection time and the file size would be more informative than if the detection times were included. Figure 3 shows this linear relationship, which I emphasised by superimposing a line of regression in blue over the top.

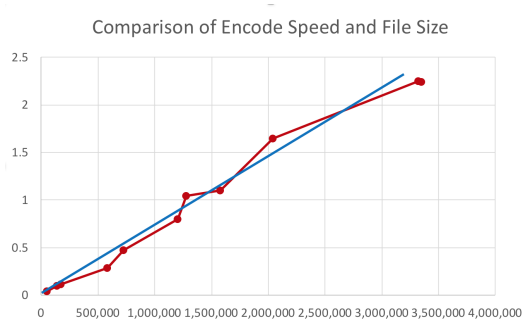

Figure 3: Total encode speed (seconds) plotted against file size (bytes)

From the timing results, we can also see that the time taken for my code to build the tree, generate the canonical codes and build the decode table is effectively constant. The linear relation between file size and encoding time arises from the counting the frequencies of the characters, encoding the characters with the Huffman codes, and then writing the output to a file. This seems intuitive, as one would expect the time taken to perform these operations to increase marginally (and linearly) as the file size increases. I would like to emphasise here that it is the extremely slow *chardet* library, and not my own code, that causes the significant delay for large UTF-8 (and ascii) files. After realising this, I was impressed with the performance of my code, which was able to encode the 3.3 megabyte *lm3200.txt* text file in just 2.25 seconds.

3.2 decode.py

I began timing my decode program with similar precision as I did for my encode program, but soon realised that this was not necessary as the decode program is much simpler and the source of delay was quickly identifiable. Instead, I simply took multiple timings to decode the *.hc* files back to text files, and took an average - these results can be seen in the final column of figure 4. As no character detection is required by the decoder (besides reading the first 5 bits), I did not need to use the slow *chardet* library. Although the algorithm to rebuild the tree is complicated, it only requires l iterations, where l is the number of levels in the tree. It is therefore largely independent of the file size, and depends only on the number of distinct characters in the original text document, which will usually be relatively small. Sorting through the leaves and regenerating the Huffman codes reuses the functions from encoder, which I have previously demonstrated to run with fairly constant time too. Therefore the only source of increased delay must be from actually decoding the binary-represented message back into characters. This appears reasonable too, as to decode the bits to characters I have to read the string bit by bit, until a bit pattern matching a Huffman code is read. When this occurs the corresponding character code is added to the message string, and we begin reading again at the next bit. Consequently, the time taken to perform this process will increase linearly as the file size increases. This 'greedy' method of decoding the file is guaranteed to work due to the prefix-free nature of the Huffman codes.

3.3 Compression Ratio

I also systematically calculated data about the compression ratio for each inputted text file. The *File Compression Ratio* column is simply the *Compressed Size / Original Size*, and represents how much smaller the compressed file is compared to the original file. As shown in Figure 4, the compressed file is typically 55% the size of the original file, and slightly higher for the smallest text documents. This slight increase for smaller text documents can be attributed to the header itself taking up a significant amount of the total space

required for the encoded file.

Filename	Encoding Type	Original Size	Compressed Size	File Compression Ratio	Entropy	Average Length	Maximum Compression Ratio	Compression Ratio	Relative Difference (%)	Decode Time
yw50.txt	UTF-8	51,185	29,495	0.57624304	4.593682593	4.630693791	0.574210324	0.578836724	0.805697764	0.125153767
ow140.txt	UTF-8-SIG	142,384	81,178	0.570134285	4.630269284	4.670636349	0.578783661	0.583829544	0.87180815	0.340448695
aw170.txt	UTF-8-SIG	173,595	95,314	0.549059593	4.60808873	4.649698442	0.576011091	0.581212305	0.902971166	0.39325517
sh560.txt	UTF-8	581,878	329,839	0.566852502	4.493134324	4.533546556	0.561641791	0.56669332	0.899421857	1.434704411
pp700.txt	UTF-8-SIG	726,223	397,267	0.547031697	4.480440804	4.512003136	0.560055101	0.564000392	0.704447013	1.725225578
illiad1100.txt	UTF-8-SIG	1,201,891	668,944	0.556576262	4.501938305	4.551565303	0.562742288	0.568945663	1.102347358	2.948757021
md1200.txt	UTF-8	1,276,200	700,798	0.549128663	4.497529053	4.526810548	0.562191132	0.565851318	0.651057377	2.996775808
uly1500.txt	UTF-8-SIG	1,580,890	886,128	0.560524768	4.584747881	4.619932744	0.573093485	0.577491593	0.767432894	3.899225559
bk1900.txt	UTF-8	2,044,191	1,112,156	0.544056793	4.508280136	4.544386119	0.563535017	0.568048265	0.800881527	4.981435522
lm3200.txt	UTF-8-SIG	3,324,334	1,853,639	0.557597101	4.526740972	4.556369479	0.565842621	0.569546185	0.654521823	8.520231948
wp3400.txt	UTF-8	3,351,415	1,822,841	0.543901904	4.499327261	4.528012841	0.562415908	0.566001605	0.637552655	8.275891181
wp3400.txt	UTF-16	3,351,415	1,823,129	0.543987838	4.499327261	4.528012841	0.140603977	0.141500401	0.637552655	8.373679851
bk1900.txt	ascii	2,044,191	1,112,156	0.544056793	4.508280136	4.544386119	0.563535017	0.568048265	0.800881527	4.765642742
chinese4800.txt	UTF-16	4,790,018	2,205,952	0.460531046	7.317085436	7.365079365	0.22865892	0.23015873	0.655915931	8.310517589

Figure 4: 'Compression Ratio' is the compression ratio of the Huffman codes, not the actual file

For each text file, I also calculated the entropy - the sum over all the characters of (the probability of the character appearing within the document multipled by the base-2 logarithm of said probability) - and the average length - the probability of a character appearing in a document multiplied by the length of its allocated Huffman code - of the generated Huffman codes. The maximum compression ratio is the entropy divided by the (average) number of bits used per character in the input document , and it is evident that the actual compression ratio, (which is the average length divided by the (average) number of bits used per character in the input document) is almost always within 1% of the maximum compression ratio. The maximum compression ratio is given by Shannon's noiseless coding theorem, and as is evident, the Huffman codes generated by my code are almost as small as theoretically possible. It is worth noting here the extremely impressive compressive ratio for the UTF-16 files, which were compressed to 14% and 23% of their original size. This is due to the substantial amount of unused bits in the UTF-16 encoding scheme, which requires at least 16 bits, and often 32 bits, to represent a single character. Although still impressive, the file compression ratio was not as small as the compression ratio of the actual text for these two files, which is due to the header prepended to the file. I discuss this in further detail in the following section.

4 Improvements

As discussed in the previous section, the main source of time inefficiency comes from the extremely inefficient *chardet* library, which takes a considerable amount of time to detect UTF-8 encoding. Unfortunately, I only discovered this during the testing phase of the project, and so could not reimplement my code using a faster library, such as *cChardet*. It appears the *BeautifulSoup* and *UnicodeDammit* libraries may also provide a possible alternative, by converting any inputted text into UTF-8. However, I am unsure if these libraries can indeed handle the opening of files with the correct encoding codec. Another small inefficiency arises from the *chardet* library, which interprets ASCII documents as UTF-8. This leads to 8 bits, as opposed to 7 bits, being used per character in the *character string* in the encoded file header. Although a relatively small inefficiency, a few bytes would certainly be saved for text files with many distinct characters.

One other improvement that I could make would be to find a method to record the depth of every node within the tree during creation, as opposed to setting the depths by recursively searching through the tree after creation. This is a challenging task, however, as Huffman Trees are built from the bottom-up, as opposed to top-down (as in other algorithms such as Shannon-Fano coding). The optimality of the Huffman Codes is a consequence of this bottum-up procedure, however, and so the cost of iterating through the tree after creation is probably outweighed by the guarantee of optimal code generation. Furthermore, as can be seen from my analysis tables, the time taken to build the tree is almost constant and independent of the size and encoding of the document being encoded.
Another area of improvement can be found within the *buildTree*( ) function. This function iterates n times, calling the O(log(n)) *heapq* functions at each iteration, and so its time complexity is therefore O(nlog(n)). To improve the time complexity of this function, an alternative implementation involving two priority queues could be used, which has been proven to run with O(n) (linear) time complexity. However, as n also represents the number of distinct characters in the inputted file, this value should be relatively small for any normal text document, as traditional alphabets usually have a relatively small number of characters and symbols. Therefore, the time complexity of this function is a relatively negligible factor in the efficiency of the program. However, if the document does contain a mixture of languages or symbols, the O(nlog(n)) time complexity of this function may indeed have a negative impact on the encoding speed.

I believe my representation of the decode tree is extremely efficient, and have not been able to find a smaller method. This led me to consider if I could represent the *character string* in a more efficient manner. Currently, I encode the characters in the encoding of the original document. However, as UTF-8 can handle the entire Unicode character set, using UTF-8 for UTF-16 (and UTF-32) text files will reduce the number of bytes needed per character in the character string, as UTF-8 requires a minimum of 8 bits (as opposed to UTF-16's 16, and UTF-32's 32). As shown in Figure 4, the compression ratio of the encoded text is considerably smaller than the actual compression ratio of the file, which is due to the size of the header at the beginning of the file. Had I used UTF-8 instead of UTF-16, I could have reduced the file size further and pushed the file compression ratio down further towards the compression ratio of the text. One final consideration would be to use a precomputed, agreed-on tree, that is based on the expected frequencies of characters within texts in certain languages. For example, in English the most common character is the letter **e**, and so this could always be assigned the shortest character code. However, this method will only provide optimal codes in the simplest of cases, and in most cases would lead to some loss in compression efficiency.