Literature review

1. Lempel-Ziv-Welch

The first algorithm I will be looking at is Lempel-Ziv-Welch. This algorithm increases the number of bits used to represent characters in order to be able to store more of them. Specifically, the LZW algorithm uses this to store pairs of characters into a dictionary and assign a numerical value to them. This is useful as it means that a given string consisting of repeating substrings will be stored with fewer characters needing to be represented.

Typically, LZW compression increases the number of bits used to store each character to 12 bits, however in my example scenario the string we are encoding is incredible small and so we will only be utilising 9 bits.

In this example, I will be encoding the string ‘Banana’ using the LZW algorithm. Without any sort of compression, this ASCII string would be stored in binary as:

01000010 01100001 01101110 01100001 01101110 01100001

This uses 6 bytes; or 48 bits.

To begin compression, we will look at the first and second character in the string. In this case, ‘B’ and ‘a’. We then pair them up and see if that pair has already been stored in the dictionary. ‘Ba’ is not currently in the dictionary so we will add it and assign the value of 256. We then output the value for ‘B’.

We then repeat this step for the subsequent letters ‘a’ and ‘n’, adding ‘an’ and ‘na’ to the dictionary and assigning them the values of 257 and 258, respectively.

When we hit the second ‘a’ we pair it up with the subsequent letter, which is ‘n.’ We see ‘an’ is already in the dictionary, so we do not do anything.

As we have found a pair in the dictionary that already exists, on the next iteration instead of using ‘n’ we use ‘an’. So, our current character is ‘an’ and our next character is ‘a’. We then concatenate them as we have been previously doing and see if ‘ana’ is in the dictionary. It is not, so we add it to the dictionary, assign it a value and then output the value for ‘an’.

Finally, we move on to the final character; in this instance ‘a’. There is no next character to concatenate, so we simply output the value for ‘a’.

First	Next	Out	Dict
066 B	097 a	066 B	‘Ba’ 256
097 a	110 n	097 a	‘an’ 257
110 n	097 a	110 n	‘na’ 258
097 a	110 n		
257 an	097 a	257 an	‘ana’ 259
097 a	-	097 a	-

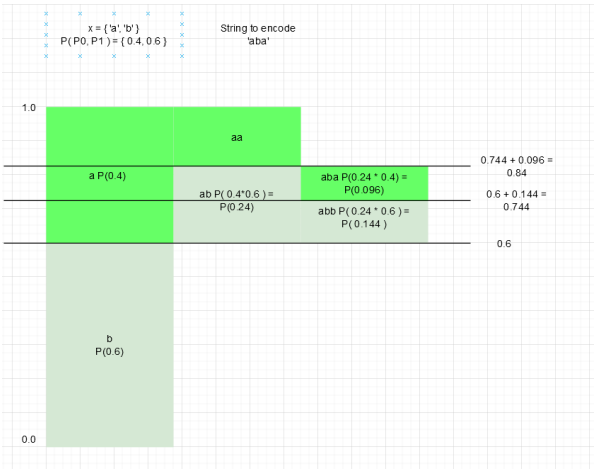
By doing this, we see our output to represent ‘Banana’ is now 066 097 110 257 097, or in binary:

001000010 001100001 001101110 100000001 001100001

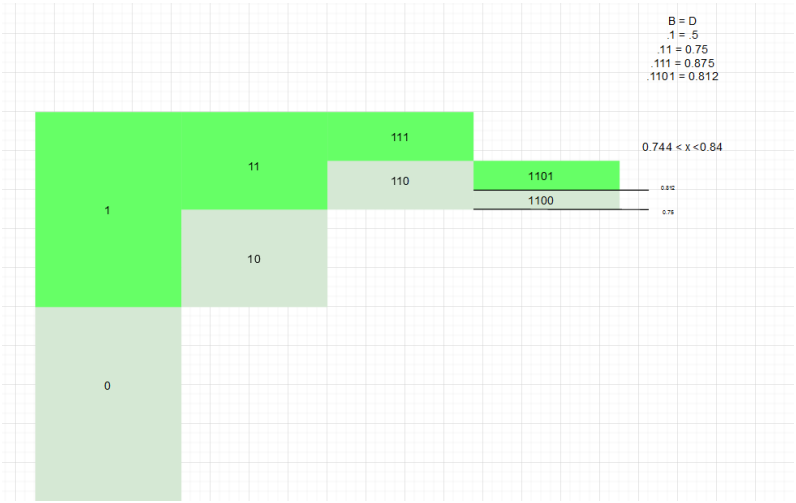
This encoded binary sequence only uses 5 groups of digits, as opposed to the 6 in the unencoded sequence. Each group being 9 bits means that this encoded sequence uses 45 bits compared to the 48 bits that the unencoded sequence used.

2. Arithmetic Coding

In arithmetic coding, we assign our string of values into sub interval of the interval 0 and 1 based on the probabilities of the letters. We call this interval I_0. We then take the interval [0, 1] and divide it into two chunks which we can denote 0 and 1 respectively. We then see which chunk I_0 would fall into and divide that chunk in half again. For example, if I_0 was in the interval $0.5 \leq I_0 \leq 1$, or the chunk we labelled as '1', we would divide this in half again to get '11' and '10'. This process is repeated until we have a chunk in which both sub intervals fit into I_0. This is then the encoded value of our string.



In this example the string to encode is 'aba' and is made up of symbols 'a' and 'b' with probabilities 0.4 and 0.6, respectively. The first character to encode is a so we create a branch off 'a' with all the symbols and their respective probabilities again to get our 'aa' and 'ab' chunks. Our next character to encode is 'b' so we create a branch off the 'ab' chunk with the symbols and their probabilities again. Finally, we want to encode the 'a' again. This means that we can now select the 'aba' chunk. We know the probability of 'ab' is P('a') * P('b'), or P(0.4) * P(0.6) = P(0.24). We can do the same process on the 'aba' chunk to get the probability being 0.096 and the 'abb' chunk to get P(0.144). We can use these values to get the interval that 'aba' occupies being between 0.744 and 0.84.



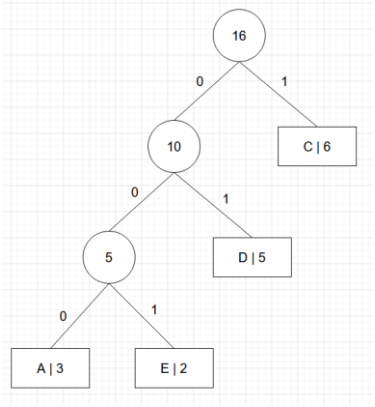
Next, we create a new tree and divide it in half and label each chunk '0' and '1'. We know the lower bound for I_0 in this instance is 0.744, so we know it will be in the '1' chunk. We then create a branch

off of this chunk and once again split it into half, each chunk now being labelled ‘11’ and ‘10’. Once again, we perform the binary expansion of .11 to get a lower bound of .75 for the chunk ‘11’. Therefore, we know the chunk we’re looking for will fit into the ‘11’ chunk. We therefore branch off again to get ‘111’ and ‘110’. The binary expansion of .111 is 0.875 which is outside our upper bound for I_0. Therefore, we will branch off of ‘110’ to create ‘1101’ and ‘1100’. The expansion of ‘1101’ gives us a lower bound of 0.812. This is within out range. As we know that ‘1100’ has a lower bound of 0.75 and an upper bound of 0.812, we can say it is within the interval of I_0.

Therefore, the encoded string for ‘aba’ using Arithmetic Coding is 1100. This is a great increase in saving space from the unencoded ascii which is 01100001 01100010 01100001.

3. Huffman Coding

Huffman Coding is the simplest method we have described. It involves performing a frequency analysis on a string of data and storing each character and its respective frequency as a node. We then pick the lowest two nodes and connect them together under a parent node. This parent node is simply denoted by the value of the combined frequency of it’s two child nodes. This new node is then added back into the priority queue and this process is repeated until there is only one root node left. Each character in the text will be a leaf node connected to the root node, with characters that occur more often being closer to the root node. We can then assign each character a binary code by getting it’s traversal path on the tree.



On this tree, for example, the character code for ‘A’ would be ‘000’. This is an improvement from the unencoded binary value of ‘01000001’.

List of Data Structures and Algorithms

Selection Sort- Selection sort is used twice in this program. At the beginning when the program counts the frequency of each character, it uses a selection sorting algorithm to order the list of frequencies. The other time when this happens is when we have the list of nodes. These nodes need to be sorted by frequency before they can be connected into a tree. Selection sort seems like a controversial choice, as the worst case and average time complexity is $O(N^2)$. However, this is not as much of an issue as it would first seem. When we use this sort, it is not on the plain text itself but instead a list of all the characters used in the text. This means that the time complexity does not necessarily increase depending on the size of the input text but depends more on the diversity of it. Most texts are written in a single language and as such this limits the number of characters being used. The trade off for this is that the space complexity of this algorithm is O(1). This is incredibly useful as it means that will be able to use this algorithm on larger datasets with the same amount of memory available to us than we would if we were using merge sort for instance. It also fits the criteria as it is quick and relatively simple to implement. As it is not the focus of this project and considering the

implications of the time complexity previously talked about, it seemed a good fit for this purpose. If I were to go back and alter this project, I could utilise a different sorting algorithm such as merge sort and speed up performance on extremely diverse data sets.

Tree Traversal- Tree traversal is used in the program to generate a list of all the characters and their respective character codes based on where they are in the tree. It is a recursive function in which each node, if it is not a leaf node, gets all the information about their child nodes. This would include any currently generated character code also their respective child nodes; for this example, we will call the grandchild nodes. We then insert a '1' or a '0' to the beginning of all the grandchild character codes depending on which child node we are on. If it is the first in the child node array then it would be a '0', and if it is the second one it would be a '1'. We then set the character code of the respective child nodes to '1' or '0' depending on the same factor and return this array list. This then returns these values up the tree until we get to the root node and have a list of each character in the tree and their respective character codes.

Hash map- Storing the characters and their character codes.

List- Storing the characters and their frequencies.

Tree- Storing the sorted nodes.

Weekly Development Log

Week Beginning February 1st- Created a node class. Instances can be linked together as parent/child to form a tree. You can call a function to get the child of a node depending on the integer value supplied (Supplying 0 gets the leftmost child, 1 gets the next one, etc). Also added a tree class which keeps track of the current traversed node, and the root node. This means there is now a full tree system implemented.

Week Beginning February 8th- Implemented a file reader class which simply abstracts the Scanner system into one simple function call. This function returns an ArrayList of Strings which each element being a line from the text file.

Week Beginning February 15th- Implemented a CharMap class. This is a class that contains a HashMap of types <char, int>. This class is meant to directly relate a character to an integer value which will be used for storing character frequencies. We can set the numerical value assigned to a character in the map, get the numerical value, and get an 'Iterable Map'. This is simply a an ArrayList of String arrays. The function loops through each element in the hash map and adds a String array to the ArrayList where the first array element is the character cast to a String and the second element is the numerical value also cast to a String. It then sorts the ArrayList by frequency using a selection sort.

Week Beginning February 22nd- Began to implement the function to encode a file. First it creates a CharMap instance and then it takes in a path to read the file from, and then loops through each character in this file. It then increases the frequency of each character it finds in the CharMap. We then turn this frequency table into a tree, get the character codes from the tree, store these characters and codes in a new hash map, and then once again iterate through the input file; this time getting each character, finding it in the hash map and then appending the character code to a StringBuilder. This gets converted to a string and written to a file. The tree also gets written to a separate file.

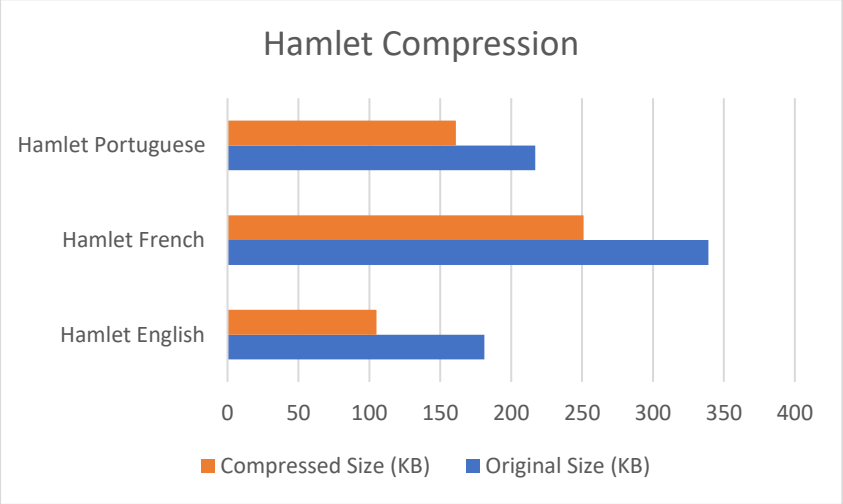
Week Beginning March 1st- Implemented decoding of encoded files. We first take in the path to the encoded file. Then we read the tree file from that directory and append the characters and their character codes to a HashMap. Then we read the encoded file, adding each character to a temporary string variable as we go. We then check if the temporary string plus the next character is in the hash map. If it is, we add this character and loop again. If it is not, we know it is a leaf node, and so append

the respective character to a StringBuilder and then empty the temporary string. We then output this to a text file in the same directory called decoded.txt

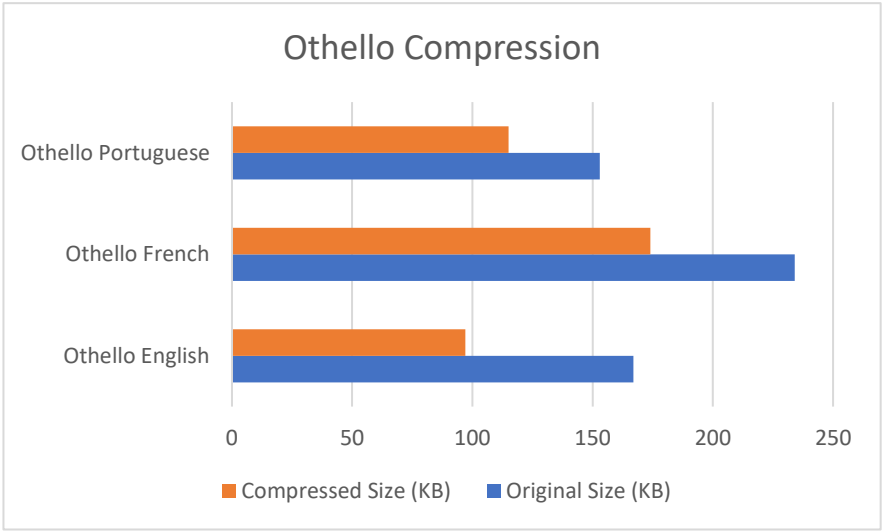
Week Beginning March 8th- Instead of writing the encoded file to a text file, we split the binary string into 8-character chunks, convert these to a character and then store that in a byte array. This byte array then gets written to a binary file instead which means we can compress the file. We also alter the decoding function to reflect this change. Finally, we add a UI to allow people to specify file paths.

Performance Analysis

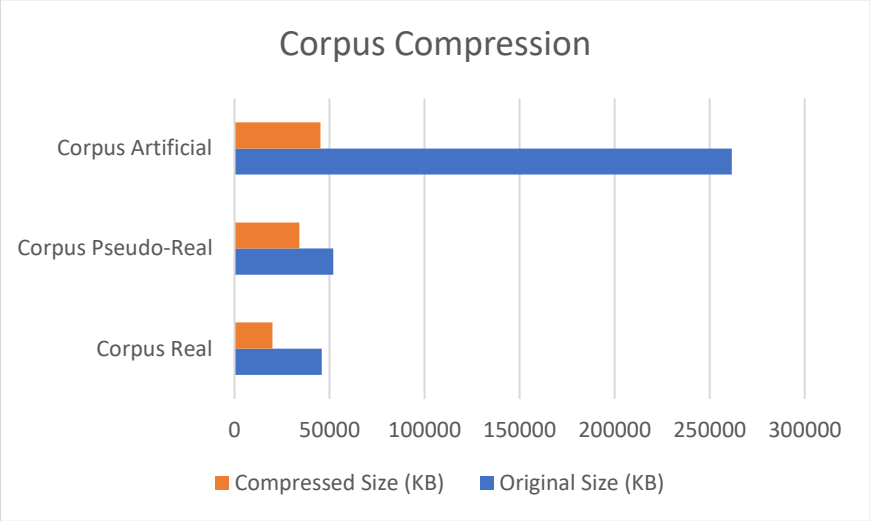
For my performance analysis I used two books in three separate languages and three datasets from the Corpus repetitive data set. For the books, I generated the encoding using the English book and then used this encoding for the French and Portuguese ones to test how well the encoding worked between languages.



The first book I used was Hamlet. English had the highest compression rate, with the file size going from 181KB to 105KB making for a ratio of approximately 1.72:1. French went from 339KB to 251KB for an approximate ratio of 1.35:1 and Portuguese from 217KB to 161KB for an approximate ratio of 1.35:1. This make sense as the encoding was generated with the English file so any extra characters from French or Portuguese would not have been taken into account for this original encoding.



I then performed the same test on Othello. Once again English had the best compresion rate going from 167KB to 97KB for a ratio of 1.72:1. French went from 234KB to 174KB for a ratio of 1.35:1 and Portuguese went from 153KB to 115KB for a ratio of 1.33:1. These values are incredbily similar to the Hamlet ratios. This would once again make sense as the frequency of letters in French and Portuguese are very different to that of English and using the English coding would then create this rift where the characters are misaligned to the frequency in the text.



Finally, I performed the algorith on the Repetitive Corpus Dataset. This is by far where the algorithm performed best. The artificial dataset went from 261636KB to 45198KB for a ratio of approximately 5.8:1. The Pseudo-Real went from 51893KB to 34041KB for a ratio of 1.52:1, and the Real went from 45868KB to 19945KB for a ratio of 2.3:1. This is by far the best set of results. There are many reasons why this would be the case. Huffman tends to perform better on larger datasets with many repeating strings. This is exactly what the Artificial Corpus dataset is. This dataset is the ideal scenario for Huffman coding which explains why it performs so well. The Pseudo-Real was the worst performing. However, it still had a ratio of 1.5:1 which according to DSPGuide is average for Huffman Coding [3].

Bibliography

[1] C. McAnlis and A. Haecky, "Understanding Compression", *Google Books*, 2016. [Online]. Available: https://www.google.co.uk/books/edition/Understanding_Compression/2C2rDAAAQBAJ?hl=en&gbpv=1&dq=history+of+data+compression&printsec=frontcover

[2] "Information theory", *En.wikipedia.org*, 2021. [Online]. Available: https://en.wikipedia.org/wiki/Information_theory

[3] "Huffman Encoding", *Dspguide.com*, 2021. [Online]. Available: https://www.dspguide.com/ch27/3.htm