Jack Purkiss

Specification

Literature review

Huffman coding

Huffman encoding is a lossless data compression algorithm which involves a variable length code table with codes for each character in a file and can typically save between 20% and 90% of data. This is based on the frequency that this character appears, with the most frequently appearing characters being encoded with a different number of bits. For example, if the character e showed up in a file a significant amount more than z then it would be inefficient for them to both be stored with the same amount of bits as they would take up an equal amount of space when it is unnecessary. Instead, e may be stored with two bits while z could take up four bits as it would show up less frequently. Once expanded, the savings can be far more effective as there may be hundreds of thousands of one character and only a few hundred of another which overall results in large amounts of savings.

Huffman is a greedy algorithm, meaning a rule is set for elements for each step of the algorithm. In this case, it is seen with the creation of the huffman tree. The algorithm to create the huffman tree involves nodes with the character and the frequency of said character. This is important to be a binary tree as each node can only have one ‘left’ child and one ‘right’ child. The two lowest frequencies are then taken and added together with the sum of their frequencies creating a new node. These two lower frequency nodes are then made to be children of their sum in the tree, with the sum being reinserted into the list, or whichever data structure is being used, at its new location. This is repeated with the current two lowest frequencies until there is only one root node left and the rest are all children. As this just involves looking at the two lowest frequency nodes, it is a greedy algorithm. Once this tree is created, we are able to traverse it, providing a value of 1 or 0 depending on whether the child is to the left or right of its parent node. Then once we reach a leaf node, it has to contain a character, meaning we can encode it with the series of bits that lead to that point on the graph. This is how higher frequency characters are able to be stored with less bits as they will appear higher in the tree, therefore meaning there is less to be assigned to them.

Additionally, Huffman encoding uses prefix codes. This means that it can always achieve data compression at an optimal rate with character code. The use of prefix codes in huffman provides much simpler decoding due to how it is presented. With one long string of binary, we are able to traverse the tree going left or right down it depending on whether the current bit is a 1 or 0. We can then continue to do this until we reach a leaf node which is a character as no code is a prefix of another, meaning the code for that character is unambiguous. Once we’ve reached a leafnode, we can then translate it back to that character and restart from the root. Ideally, this would be a full binary tree, as this would mean that each non-leaf node has two children. Therefore, the efficiency of huffman coding is largely dependant on the distribution of character frequencies from its input alphabet.

Run Length Encoding

Run Length Encoding is another example of a lossless data compression algorithm, but is more simple than the previously mentioned huffman algorithm. RLE is used by looking at a stream of data, for example “ggggggguuusssss”, and providing the output of the amount of each consecutive data value present “6g3u5s”. In this example, it can be seen that 14 characters has been reduced down to only 6. This therefore is only beneficial for specific applications and types of data as if it was unlikely for consecutive characters to occur such as with a larger variety of characters there would be a very limited amount of compression. Additionally, if there is mostly the use of single characters, the reverse effect is observed as one character turns into two, for example “Y” becoming “1Y”.

A large benefit to RLE is its simplicity as it is very easy to implement. With a string as an example, to encode all that is required is to loop over the string, keeping a count of the current letter. Then when a different letter is seen, the current count is appended to another string with that letter and the count is reset. Decoding is also simple, as you can go through the encoded string and when there is a number in the string that value can be added to a count variable which would then be used to append the following character to a string that iteration of times.

RLE is more effectively used for any data that contains many of the same values in a run. This can be seen in graphic images that may contain a lot of pixels that are the same colour, especially in simpler designs with less colours being used.

Lempel-Ziv-Welch (LZW)

LZW is a lossless compression algorithm based off of a dictionary system. The idea behind this algorithm that strings of characters are replaced with individual codes in a dictionary that are smaller in size than the strings themselves. Here, the first 256 codes are assigned to the regular set of characters as bytes with any additions to the codes being the strings that are added further along as substrings. Throughout the compression, a sequence is formed by the input bytes until a character is added to create a sequence that isn’t present in the dictionary, which then gets added. Additionally, the bit size of the codes are variable width so that it can start off smaller and increase up to a maximum, which when reached means that new codes are no longer added to the table. Similar to RLE, this algorithm is most effective with files with repetitive data such as text. Also similar to RLE, it can mean that if there isn’t much repetition, the file size could potentially get larger.

In order to encode, the dictionary is originally set up with all of the single length strings. The input is then scanned to find strings with longer substrings that aren’t in the dictionary. When this is found, the index for the longest substring in the dictionary is outputted and this new string that has been found is then added with it being the starting point for substrings.

When it comes to decoding, it is important for both the encoder and decoder to change the width size at the same points. Because of this, the decoder remains one code behind the encoder when building the table as the decoder has to increase the width with the encoder when the largest code generated can fit.

Data Structures and algorithms used

Priority Queue

The priority queue was used in order to create the queue of nodes to be used in the tree. By using a priority queue, it meant that I was able to add the node objects into it and have them organised from lowest to highest frequency. This meant I could utilise the .poll() function to find the two lowest frequency nodes and add them to the tree. Additionally it would allow me to insert the combined frequency node into the queue and still have it sorted from lowest to highest. With the .poll() function, the nodes were removed from the queue so ultimately at the end I was left with a single node left which was the root of the tree, with everything else being linked to it through parent/children relationships.

Tree

The tree is the fundamental aspect of the Huffman algorithm as it is directly used in order to create the codes. Once created, the tree consists of a root node with the left and right child nodes that have a sum of their frequencies equalling the current node.

Huffman tree creation algorithm

This algorithm is how the tree is created, utilising the priority queue and tree data types. It involves collecting the two lowest frequency nodes to create a combined node which has them both linked to a left and right attribute for the combined node to then be added back into the priority queue.

Recursion

Recursion was used in the getCodes method in the program. This works by checking if the current node is a leaf node and if it's not then calling the method again with its left node with an additional "0" added to its current code and then to its right node with an increment "1". Then once it reaches a leaf node, that is then added to the codes hashmap with its character.

Array

A byte array is used to store the converted binary string. This is important for the compression as it is the main feature that reduces the overall file size by using significantly less bits.

Hashmap

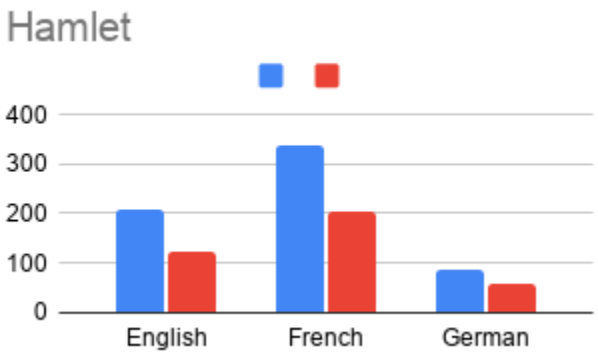
Two hashmaps are used throughout the program. One is used to store the frequency of each character by using the character as the key and an integer for the frequency of the character. Another hashmap is used for the codes so that the binary code for the character is stored, providing the opportunity to iterate over the text and easily convert to its binary code.

Performance Analysis

The process behind this huffman code involves storing the converted bitstring in a byte array as well as the tree with the codes. These are then put into a separate wrapper object with these as its attributes which is then serialized into a .bin file so that on decompression, everything required for decompression is present within this object. For each of the charts shown below, the size in kilobytes is seen on the y axis with the blue representing the original text file size and red representing the compression .bin file with the language or dataset seen on the x axis depending on area being tested.

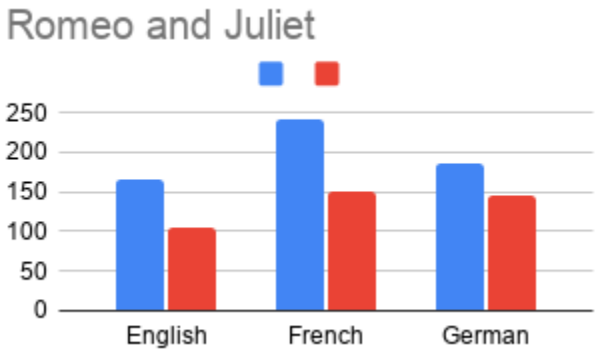
Hamlet

In general here, we are able to observe roughly a 60% compression ratio for each language with English being 59.4%, French at 60.4% and German being highest at 64%. The slight variation between compression ratios may be explained by some languages featuring the same characters less frequently than others, resulting in a larger bit size for individual characters.



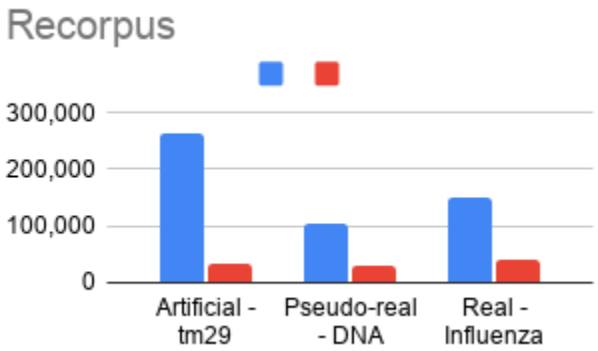
Romeo and Juliet

The Romeo and Juliet dataset shows smaller file sizes but slightly higher compression ratios with English being 63.6%, French at 61.8% and German being 77.4%. This follows the same pattern as Hamlet with the file size and compression ratio with relation of the size compared to the other languages between them.



Recorpus

Due to the repetitive nature of these datasets, the Recorpus data shows significantly lower compression ratios. The artificial set tm29 shows the lowest ratio of 12.5% with the pseudo-real and real dataset having similar ratios of 27.9% and 27.4% accordingly. This is related to the idea that with less characters present, the amount of bits required for each character is significantly lower, therefore creating a much smaller compressed size.



Weekly log

Week	Implementations	Explanation
Week 1	Initial ideas and early development	During my first week I took the time to fully understand the concepts behind huffman encoding so that when it came to development I would be ready. I also developed the early stages of nodes, exploring hashmaps first, then creating a class and eventually using a mix of the two
Week 2	Tree development and beginning to encode	Once the nodes had been created I worked on finding a method to create the tree required. This initially involved a sorting algorithm to sort the HashMap by frequency, which then changed to a priority queue based on the node object frequency due to the simplicity of the .poll() function easily finding the lowest frequencies. Once the tree was built, I explored how to traverse it in order to find the nodes.
Week 3	Encoding and decoding	I found a recursive method for finding the codes that was effective so utilised that and implemented the methods to be able to read and write to the file required by serializing an object with all the required information. I found that by having the tree and byte array stored within a wrapper object to be serialized, it was then very simple when decoding as the tree was in the file already alongside the byte array so it could then immediately be used. When it came to decoding, it was a matter of traversing the tree by moving to the left or right node depending on the value in the bitstring and restarting once finding a leaf node after writing to a string.