DS&A Specification

Literature review

The Huffman algorithm is a form of lossless compression that assigns characters a binary code based off their frequency. It works by getting the frequency of every character in a given file and storing each character as a single node tree in a "forest" of single nodes, this forest can be represented as a sorted list. The algorithm then takes the least 2 items of the list and creates a new tree with 2 sub-nodes of the removed trees and a frequency equal to the sum of the 2 sub-nodes frequencies. This then repeats until only 1 node remains, this is the Huffman Tree which is the core of this compression algorithm. To then encode each letter of the file, you start at the root of the tree and as you navigate the tree, add 1 to the encoding when you go to a right sub-node and 0 when you go to a left sub-node until you reach a node with no sub-nodes, a leaf node. You then assign the character in the leaf node the encoding generated and write this encoding to the output. This is done for every character until the file is fully compressed.

Lempel–Ziv–Welch Compression, or LZW, is a universal lossless compression algorithm which achieves high levels of compression by use of a table lookup algorithm. First the algorithm creates a dictionary of each character, giving each character a different index in the dictionary. Then starting from the first character, read the next character c in the file, given an empty string n, if c + n is in the dictionary then add it to the string n and move to the next character. If c + n is not present in the dictionary however, output the index n as the encoding and add p + c to the dictionary with a new index and set n = c. This means that as string n gets longer and longer, more characters are stored under 1 index making this algorithm very powerful at compressing lots of data into a small file and will produce better results than Huffman encoding, which stores 1 encoding for each character.

JPEG compression is a lossy compression algorithm which is typically used to compress pictures and it works by taking groups of similar coloured pixels and grouping them into 1 colour, losing some resolution but greatly reducing the file size. First the image must be converted to a YCbCr colour space to separate the luminosity of the image (as a greyscale) from the colours so that the colours in the image can be down sampled but still appear as bright. A mathematical function called the Discrete Cosine Transformation (DCT) is then applied to the file. This splits the image into blocks of 8 by 8 pixels and each block is assigned a cosine wave based off the luminance and all the cosine waves are averaged in the image to produce what seems like an 8 by 8 checkerboard pattern. By assigning different cosine waves different weightings based of the luminance the checkerboard is coloured to look like the image in grayscale. After this the file will be quantised, which is where the weights of each cosine will be divided by some constant, determined by how pre-set conditions of how much data can be lost in the compression, and rounded to the nearest integer. This means the blocks with the largest amount of data are set to 0, so they add the least to the image and the smallest blocks are provided the greatest weightings, so that less data is used in general. The remaining blocks can then be compressed via Huffman encoding which is what is stored in the jpeg file. This means the image loses data overall that for most purposes won't be easily perceived by people in order to have large amounts of compression. This is one of the most commonly used photo compression formats due to the effectiveness of the algorithm.

References used in literature review:

- https://en.wikipedia.org/wiki/Data_compression#Outlook_and_currently_unused_potential
- https://en.wikipedia.org/wiki/Huffman_coding
- https://en.wikipedia.org/wiki/Lempel%E2%80%93Ziv%E2%80%93Welch
- https://www.youtube.com/watch?v=rZ-JRCPv_O8

- https://www.youtube.com/watch?v=Q2aEzeMDHMA
- https://en.wikipedia.org/wiki/JPEG

Data Structures and Algorithms used in my project

- Hash map – I used a hash map when storing the frequencies of each character in a text file as its simple to pair up a character and a number. I also used one to pair each character to a binary encoding after I had generated the Huffman Tree.
- Priority queue – a priority queue is a queue which dequeues the smallest item first. I used this when generating my Huffman tree so that the lowest items were always together so I could pair the least 2 into 1 node combining the two until only 1 node remained.
- Tree – the tree is how I stored the Huffman Tree, each node having 2 sub-nodes unless it's a leaf node so that each node can have a binary character associated with it.
- Pre-order tree traversal – this algorithm is how I assigned each character at the tree leaf nodes a binary encoding, I visit each node recursively starting at the root I then visit the left sub node then the right, adding a binary digit each time I visit a node, each sub-node becoming the new root until a leaf is reached where that character is assigned the binary string generated.

Work Log

A log of what work I did each session of coding:

1. Learn about Huffman algorithm and how it works.
2. Create node class and add functions to manipulate each node and its sub-nodes.
3. Create compression class and add functions to merge "forest" of nodes into 1 tree.
4. Create functions in compression to traverse tree and encode each letter into binary.
5. Driver class with main function, add function to create tree by reading each letter of input text file and adding each letter to the tree.
6. Compression function reads character and writes binary equivalent to a byte array.
7. Byte array writes to .bin file, prints file size and compression time on console, saves tree as a .ser object.
8. Basic cmd interface and ability to load trees to compress files, start on decompression function.
9. Decompression Loops through the binary in compressed folder and adds char to char array when series of bits match.
10. Decompression successfully creates new txt file with correct output.
11. Create submission folder, jar file of code and run test cases. Write up specification and hand in.

Performance analysis

File	Language	tree	Initial Size (no tree)	Compressed Size (inc. tree)	% Diff
fib41 (repcorpus/artificial)	English	self	267914296	33489287	87.5
dna.001.1 (repcorpus/pseudo-real)	English	self	104857600	29278612	72.07774
world_leaders (repcorpus/real)	English	self	46968181	20419872	56.524031
Origin Of Species	Finnish	self	1335247	706054	47.121843
Origin Of Species	French	self	1475237	813716	44.841676
Complete Works Of H.P. Lovecraft	English	self	2718928	1512769	44.361565
Origin Of Species	English	self	971669	543613	44.053685
Macbeth	Finnish	self	129727	76539	40.999946
Macbeth	French	self	201651	120658	40.164938
Macbeth	Finnish	English	129727	78274	39.662522
Macbeth	English	self	130740	79388	39.277956
Origin Of Species	Finnish	English	1335247	826971	38.066066
Steamed Hams	English	self	2448	1540	37.091503

Here my data is sorted from greatest to least levels of compression by percentage difference in order to easily display which files had the largest compressions, with greater compression rates coloured darker green and less compression darker red. File size however is coloured the opposite, with greater sizes corresponding to red and lesser sizes green. This is to help differentiate the data when looking at the chart.

For my implementation of the Huffman compression, I tested it on 11 different text files, as seen above. From the repetitive corpus data set I used: fib41 as the artificial data set; dna.001.1 as the pseudo-real data set and world_leaders as the real data set. These were good files to test the program on because they feature different levels of repetitive data to show how well my program deals with different types of character when compressing the file.

As these were the largest files, they achieved the highest levels of compression as there were more characters to compress. As expected, the artificial data achieved the highest levels of compression, being about 88% smaller than the original file. This will be because it only has 2 characters repeated, “a” and “b”. This means it will have the smallest tree so each character will take up 1 bit which is the smallest dataset possible with Huffman encoding. The pseudo-real data set achieved second most compression with 72% compression compared to the original file. This too is expected as it has a larger data set of 4 characters so depending on character frequencies, up to 3 bits may be needed per character producing a larger tree resulting in a larger final file size. Finally, the real data set from the repcorpus data sets produced the lowest compression rate from the repcorpus data sets with 57% reduction in file size. This is significantly lower than the other 2 data sets, likely due to the vast character set it would be required to compress and the large tree structure generated in the encoding.

I also tested my implementation on 3 different books: Macbeth by William Shakespeare; On The Origin Of Species by Charles Darwin and the Complete Works of H.P. Lovecraft – a collection of his books in one. Macbeth and Origin Of species I tested in 3 languages: English, French and Finnish in order to test how well the program compresses different language character sets as well as to test how well how well one language encodes another. For both books Finnish was the most compressible, followed by French then English. This is not what I expected as I predicted having a larger character set, such as French having â, would result in a larger tree structure thus more bytes required to store the book, however this was not the case. This could be because French and Finnish use characters more evenly than English, so the longest character representations are shorter overall even if there are more stored, resulting in a more compressed file. However, I also tested the finish copies of the books by using the English Huffman tree to compress them instead of generating a Finnish tree. For Origin

of Species the compressed file from this has much less compression than any of the other files with only 38% compression from the original text and when decompressed, characters in Finnish absent from the English character set are decoded as the letter “h”. I am unsure why this is however It is likely “h” in English coding will have a different binary encoding than the special characters or h from Finnish which is why the resulting file is longer than the Finnish encoded version. However, with Macbeth surprisingly the regular English encoding of Macbeth compresses by 39.3% whereas the Finnish copy of Macbeth, when compressed by the English encoding is 39.7% smaller. This means English naturally is worse at compressing Macbeth by the Huffman encoding than Finnish so much that even when characters are missing from Finnish encoding, it’s still smaller. The complete works of H.P. Lovecraft compressed by 44%, which was better than all Macbeth compressions and the English and English encoded Finnish compressions of Origin of Species. Steamed Hams, a script from the Simpsons, had the worst performance of only compressing by 37%. This is likely due to its small size, there is only so much that can be compressed, and the tree size will take up a greater amount of space relative to the file than larger files so the percentage change is affected more.

Below is a table of the same files and their compression, but now measuring the time taken to compress the file and comparing it to the final size of the file. The File size: time ratio can be used to asses which files compressed the quickest compared to their final size. This table has been sorted worst to best file size/time ratio to easily see and compare which files performed badly.

File	Language	tree	Compressed Size (inc. tree)	Avg Time taken (µs)	File size/time ratio
Steamed Hams	English	self	1540	2276.2	0.676566207
Macbeth	Finnish	English	78274	63564.7	1.23140674
Macbeth	French	self	120658	65944	1.829703991
Macbeth	Finnish	self	76539	40473.4	1.891093904
Macbeth	English	self	79388	39295.7	2.020271938
fib41 (repcorpus/artificial)	English	self	33489287	15854947	2.112229514
Origin Of Species	Finnish	English	826971	255994	3.230431182
dna.001.1 (repcorpus/pseudo-real)	English	self	29278612	8089013.4	3.619552911
Origin Of Species	Finnish	self	706054	192525	3.66733671
Origin Of Species	English	self	543613	147349	3.689288696
Origin Of Species	French	self	813716	212553	3.828296942
Complete Works Of H.P. Lovecraft	English	self	1512769	298665.2	5.06509965
world_leaders (repcorpus/real)	English	self	20419872	3932534.8	5.192547056

From the table it is clear to see the smallest files, Macbeth and Steamed Hams, compressed the fastest. This will be because they had the least data to be encoded thus less time was required to compress them. After that the next fastest file for its size was fib41 despite it being the largest file to be compressed during the tests. This could be caused by the fact it only contained 2 characters, so the speed to create and read the tree would’ve been the fastest possible for the Huffman encoding. Similarly, dna.001.1 was very quick for its size, likely again due to the small character set within the file. Origin of species was the next slowest at ratios between 3 and 4, with the fastest among them being the Finnish compressed with English encoding and the slowest being the French compression. It could be that because not all Finnish characters are present in the English encoding of the file, it takes less time to search the tree because t is smaller and when those characters are required, it writes the letter h to fill in, possibly reducing the time required. Finally, the complete works of H.P. Lovecraft and world_leaders had the worst time performance both with a ratio of about 5. This could be due to the fact these texts have a wider variety of characters as well as being lengthy thus more time is required to read the Huffman trees and a lower level of compression being possible from the bigger trees. Overall my program functions well, compressing small and large text files quickly to a smaller size that can be more easily manipulated and transferred.

References used to create my Huffman Encoding and decoding program:

https://vle.exeter.ac.uk/pluginfile.php/2414085/mod_resource/content/4/description.pdf

https://vle.exeter.ac.uk/course/view.php?id=10890&section=10

https://www.codejava.net/java-se/file-io/how-to-read-and-write-binary-files-in-java

https://cseweb.ucsd.edu/~kube/cls/100/Lectures/lec8/lec8-15.html

https://www.geeksforgeeks.org/huffman-coding-greedy-algo-3/

https://www.geeksforgeeks.org/huffman-decoding/

https://www.geeksforgeeks.org/tree-traversals-inorder-preorder-and-postorder/

https://www.w3schools.com/java/