Data Structures and Algorithms Literature Review

Compression in terms of computers and files on the computers, is the action of condensing the size a file so that it is smaller to be able to store and travel across a network. In other words, compression is the process of reducing the size of a file in bytes by either removing unneeded bits of data or from grouping patterns of data together. Compression is commonly used in digital music recordings to be able to transport large files faster. There are two types of compression, lossy and lossless compression. A data compression algorithm converts data from a ready to use format into a format optimised for compactness. A decompression algorithm turns the compressed data back into its original form so that it can be used again.

Types of compression

Lossless: An advantage if lossless compression is that there is no loss of information or quality; thus, you can recreate the original data from the compressed data. An example of lossless compression is GIF (Graphics Interchange Format) the widely used lossless compression format for images which is more often used for simple drawn images such as logos that have large areas of a single colour. Lossless compression is essential for textual data such as a document or a computer program, otherwise, the decompressed document will be unreadable.

Lossy: A disadvantage of lossy compression is that there is always some loss of information or quality and therefore you cannot recreate the original data from the compressed data. The data lost in lossy compression however is never essential and is usually redundant data. Nonetheless, the sound quality of a song for example under lossy compression will lose some of its sound quality. However, for most lossy sound compressions, the lost data, as mentioned previously is not at all important, it is usually sounds over 20,000 Hz which is also known as ultrasound that is removed as this cannot even be heard by humans anyways.

Compression is often described as a trade-off between process time and storage space. If a file is stored compressed, it takes up a lot less storage space, naturally, however, the time it takes for the processor to retrieve that file is longer due to it having to decompress the file before the contents of the file can be accessed. However, if the file is stored uncompressed then it takes up a lot more storage space, so either option has its advantages and disadvantages. A rare example where the trade-off is very weighted towards compressed files being more efficient, however, is the instance in which it is possible to directly work with compressed data, such is in the case of when using compressed bitmap indices, where it is faster and therefore more efficient to work with compressed files than without.

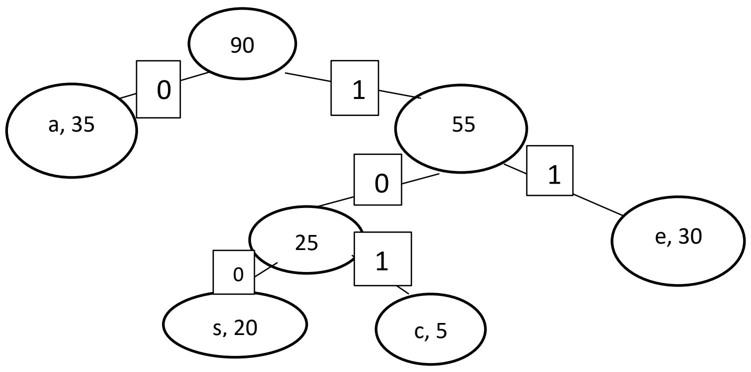
Compression is important for several reasons. For example, when transferring files across a network such as the Internet, transferring a smaller, compressed file is a lot faster than transferring a larger file that has not been compressed because smaller files are split into less packets that travel across a network, which means when the file is being received there is less packets for the recipient device to wait for before the packets can assemble and represent the file once more.

Huffman's algorithm

The principle of the Huffman coding algorithm is based on the fact that some data is used more frequently such as the word "the" in a text file. In Huffman compression, the length of the code multiplied by the frequency quates to the total bits, the Huffman coding algorithm is used to reduce the number of bits used without losing any of the information.

The Huffman coding algorithm is a compression algorithm that uses lossless compression so that the file does not lose any data. It is based on the lengths of assigned codes that are based on frequencies. The characters that are being compressed are first sorted by frequency and then put into a binary tree with the characters in nodes at the top of the tree having the highest frequency, and as you traverse down the tree, the frequencies decrease so that the characters of the nodes at the bottom of the tree have the lowest frequency. A binary digit is added to the code each time you go down the tree for example, if you go left then a 0 is added, and if you go right then a 1 is added. This means that the most frequent characters have the shortest codes, and the least frequent characters have the largest codes; this lowers the number of bits in the compressed file.

Example of what a Huffman tree could look like:



Huffman pseudocode

```
create a priority queue consisting of each unique character in the
file.
get the frequency of each unique character.
sort the unique characters in ascending order of their frequencies.
for all the unique characters:
     create a new node in  the binary tree.
     extract minimum value frequency from the priority queue
     and assign it to left child of the new node.
    extract minimum value from the priority
    queue and assign it to right child of the new node.
    calculate the sum of these two minimum values and assign it to
    the value of the next new node.
    insert this new node into the tree.
```

This is the pseudocode that I will implement in my program.

Other compression algorithms:
1: Lempel-Ziv W Data Compression Algorithm
The Lempel-Ziv W algorithm is more efficient and simpler to implement which is why it is now the standard algorithm for file compression, it is used in ZIP compressions for example.

The main idea behind the Lempel Ziv W algorithm is that the source data stream is parsed into segments that are the shortest data redundancies not encountered previously.
For example: the input data stream: 01000101110010100101

The encoding process begins from the left; therefore, the first subsequence is simple just 0, and this subsequence will have a numerical position index of 1. The next subsequence will be 1, and this will have the numerical position index of 2. The next bit in the sequence is a 0, if you compare this with the previous bits, 0 has already been encountered. Therefore, the next subsequence is 00, as 00 has not yet been encountered, and has a numerical position index of 3. This method repeats for the entire data stream. The full list of sub sequences for this input data stream would be: 0,1,00,01,011,10,010,100,101. The last bit of each subsequence is called the innovation symbol. Each subsequence has a numerical representation which is given by, the last digit being the numerical position of the subsequence's innovation symbol, and the first digit being the numerical position of the rest of the sub sequences bits for example the subsequence at the end of the list of subsequences in the example above was 101, the numerical representation of this subsequence would be 62 because when you remove the innovation symbol from the subsequence you're left with 10 and the numerical index position of 10 in the example was 6.

After calculating the numerical representation of each subsequence, one is then able to calculate the binary encoded block of each subsequence. This is calculated by the last bit just being the innovation symbol, so for subsequence 101 it would be 1. Then, the algorithm takes the first digit of each numerical representation and converts it into binary. So, for subsequence 101, the first digit of the numerical representation is 6, and the binary representation of 6 is 110, therefore the binary encoded block for the subsequence 101 in this example would be 1001. In summary, the algorithm calculates the numerical representation of each subsequence, which is then used to calculate the binary encoded block of each subsequence. The binary encoded blocks then make up the encoded string. In this case, as for all small strings passed into the Lempel Ziv W algorithm, the encoded string is longer than the input/source string. This bis because there were no obvious redundancies in the source. In practical implications of this algorithm however, where the strings may be millions of bits long, the data is compressed to a smaller file size, where we would have lots of redundancies. [1]

<u>2: Run Length Encoding Data Compression Algorithm:</u>

The essence of run length encoding is to scan the data that you want to compress and for each item , report the run length, that is the number of times it occurs (the frequency) followed by the item itself.
Example 1;
For example, consider this block of data: aaaaabbbbbbbbbbbbccccdddddddddeeeeeeeeee
Compressed: 5a12b4c9d10e
Here, the compressed data set is roughly a quarter of the size of the original data set.
Run Length Encoding is a lossless form of compression because the compressed data set contains everything necessary to re-create the original data.

Example 2:
Now consider this block of data: aabccdeefgijjjklmnoqrrstttuvvwwxyyyz
Compressed: 2a1b2c1d2e1f1g1i3j1k1l1m1n1o1p1q2r1s3t1u2v2w1x3y1z

In this case, the size compressed version of the data is larger than the size original data; therefore, compression has made the data larger and made the situation worse; this is what is known as negative compression.

To summarise run length encoding is a lossless compression algorithm that works best on data which contains long runs of the same value (it works well for scanned images of text that contains long runs of white, for example). Furthermore, some advantages of run length encoding are that it is easy to implement and has many variations; the algorithm also has fast execution, as it is a relatively simple process. Due to this algorithm being fast, it generally does not require a lot of CPU power. A disadvantage of run length encoding however, is that if the run length encoding algorithm is run on complex data, the algorithm can result in negative compression; unfortunately, the algorithm does not suit all types of data.

<u>A list of all the data structures and algorithms I used in my implementation of the Huffman Algorithm:</u>

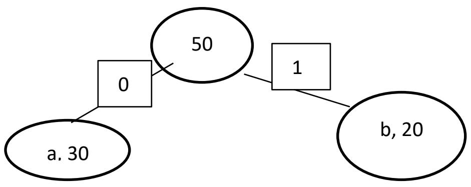
- In my implementation of the Huffman algorithm, I used arrays to store the characters read in from each file.
- My compression algorithm implements a Huffman tree, which I stored as an array so that I was able to easily add nodes of new characters in the file.
- I was unable to implement this in my code due to timing constraints but if I was able to write a separate decompression algorithm. I would use a breath-first search over a depth-first search because the more common characters are higher up the tree so to go right first from the base node of the tree then all the way down would be inefficient if, for example, you go left first from the base node of the tree and the first character you come across is the one the being searched after.

<u>Weekly log</u>

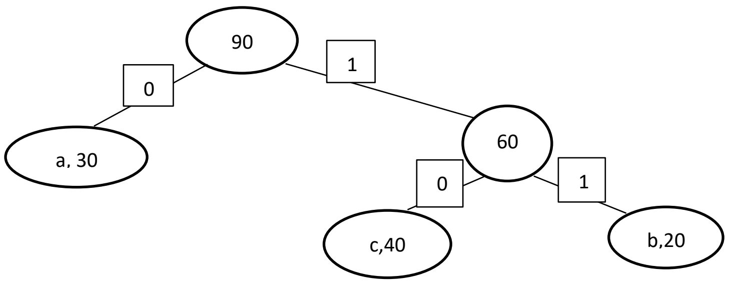
Week 1	I researched compression, the history of compression, the two different types of compression (lossless and lossy) and wrote about what I found.
Week 2	I went on to research and write about the Huffman compression algorithm, how it worked, why it is so efficient, et cetera.
Week 3.1	I wrote some pseudocode for the Huffman algorithm.
Week 3.2	I tried to implement the Huffman algorithm as an Object-Oriented Java application, as I am currently learning Java and wanted to get better at the language and be more familiar with implementing problems in it. However, I really struggled with the syntax and could not get the algorithm to work.
Week 4	Due to this, I decided to implement the algorithm procedurally in Python (I have not given up with the Java project and will one day have that version working too, but as this project is part of my course I would rather submit a working version and then carry on with the Java in my own time which is what I am doing).
Week 5	I tested the algorithm with books in English, French and Portuguese and compared the compression ratios and speeds of these (also known as their efficiencies). I also implemented a decompress function so that I could retrieve the original file again.
Week 6	I wrote up these comparisons in tables and graphs to make it easier for others to understand and visualise my comparisons and the differences between the algorithms.
Week 7	I researched two other compression algorithms and wrote about them.

Discussion

Compression algorithms tend to do well in certain scenarios and not so well in others. If an algorithm had for an example, a tree for a string of as and bs:



If the next letter in that string is a c, a character not already in the tree, there needs to be a design choice of whether the tree should be reconstructed so that c can be included in the tree or just left as the character c which is 8 bits. Another issue arises of where the new node for the c character should be placed in the tree, because if the c character is added as a leaf on the right-hand side like this:



..and as shown above, the frequency of c ends up being higher than a, then this is an inefficient reconstruction of the tree as the most frequent character has a larger binary code (10) than less frequent characters in the dataset (such as a who's binary code is just 0).

This sort of issue is highlighting why the Huffman compression algorithm is so preferable because it orders characters in terms of their frequencies' first, before putting them into the tree.

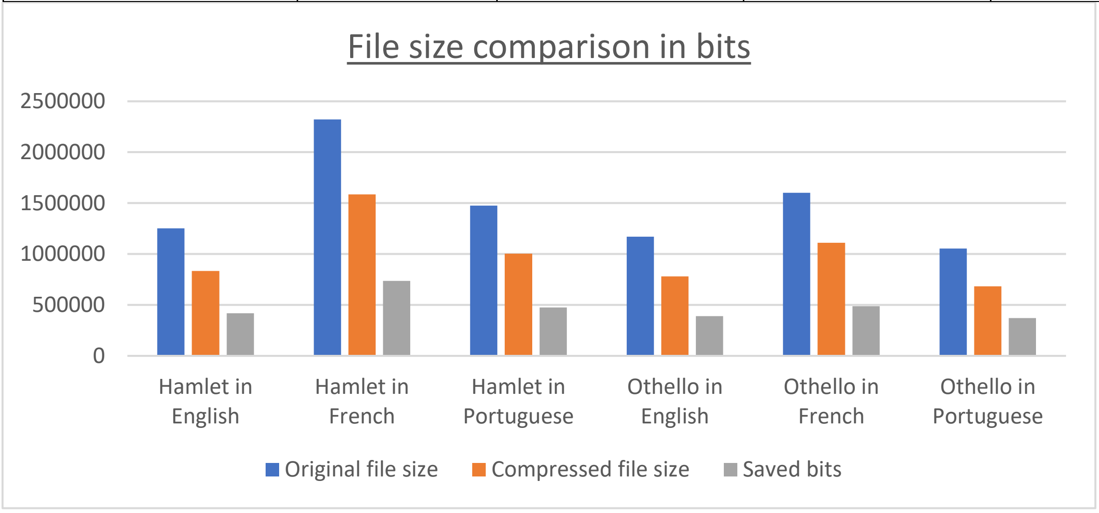
Performance Analysis:

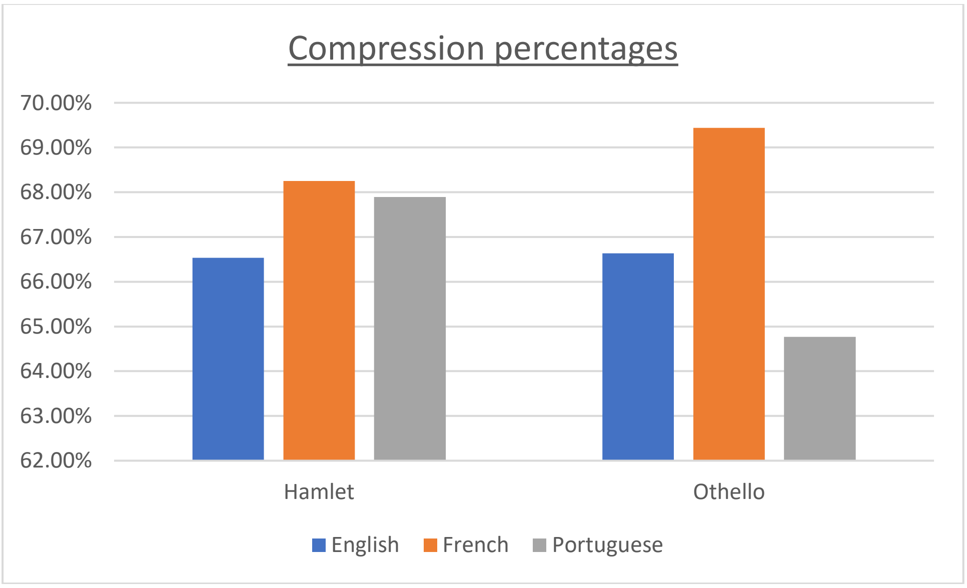
Claims

I achieved a 66.54% compression of Hamlet in English with my Huffman compression algorithm implementation. I achieved a 68.25% compressed of Hamlet in French with my Huffman compression algorithm implementation. I achieved a 67.89% compressed of Hamlet in Portuguese with my Huffman compression algorithm implementation. I achieved a 66.63% compressed of Othello in English with my Huffman compression algorithm implementation. I achieved a 69.44% compressed of Othello in French with my Huffman compression algorithm implementation. I achieved a 64.77% compressed of Othello in Portuguese with my Huffman compression algorithm implementation.

The 'saved bits' referred to below are the extra bits removed from the files that are no longer needed under compression, the compression algorithm is lossless, and when decompressed, the file size equates to the original file size.

Book:	Original file size in bits	Compressed file size in bits	Number of bits therefore saved.	Resulting compression
Hamlet in English	1253672 bits	834167 bits.	419505 bits,	66.54%
Hamlet in French	2322929 bits	1585398 bits	737531 bits	68.25%
Hamlet in Portuguese	1476986 bits	1002767 bits	474219 bits	67.89%
Othello in English	1170715 bits	780100 bits	390615 bits	66.63%
Othello in French	1600578 bits	1111390 bits	489188 bits,	69.44%
Othello in Portuguese	1054571 bits	683005 bits.	371566 bits	64.77%





A link to my online repository GitHub https://github.com/NatalieCole where I uploaded my project.

References

[1] Dheemanth H N, Dept of Computer Science, National Institute of Engineering, Karnataka, India, *LZW Data Compression*, American Journal of Engineering Research (AJER), 2014.