140012065 Tom Kelsey 08/10/15

Adaptive Huffman Coding Algorithm

Overview:

For this practical we were asked to program the Adaptive Huffman Coding algorithm with whichever language we decided to choose. The language I chose for this practical was Java. Adaptive Huffman coding is a more efficient form of Huffman coding. Instead of waiting for the tree to be made. One creates the tree and restructures the tree depending on how the source symbols change as one reads the file. The code is also made on the fly instead of waiting. The particular Huffman Code I have implemented has been developed by Vitter.

Design:

When planning the design for this algorithm there were several things I had thought about. Namely, how I would update the tree to make it adaptive, how the encoding and decoding would work and finally how I would do the experimentation. These four broad questions helped me plan the structure of the program I have made. The basic structure of my program is built across 4 main classes. Huffman.java, EncodingDecoding.java, Tree.java, Node.java. After making these four classes I did some further reading and research to find out if there were any other things I needed to think about.

The following describes how the classes and methods work on an higher abstract level.

Tree.java contains the methods needed to update the tree which also makes sure that the tree remains adaptive to any new symbols that are entered. Tree.java contains a very important method called treeUpdate. This method calls on many other methods such as insert(). The point of treeUpdate is that it allows the frequencies of the symbols to be updated. It also allows the swapping of the nodes when a highest index of the same weight class of the current node is seen.

Node.java makes up the leaves and nodes of the tree. This is a helper class to the Tree.java class. This basically contains all the information a node in the tree might have.

EncodingDecoding.java. This is where the compression and decompression occurs. The compression occurs when the encode method is invoked. This means that the tree is made and updated as the symbols are being fed in. This also creates the codes of the message.

The way Encoding works in this class is that symbols are read from the file. This is then input to the encode function through a buffer in order to quicken the program. At first I had tried to read and encode an entire string. I realised this would be too slow and hence decided to use the buffer which is much faster. The encoding basically goes through the tree as it is updating it then finds the right code depending on where it is in the tree. This then gets appended to the string. This string being the encoding of the file.

The way Decoding works:
This works in a similar way to that of the encoder. It goes through encoded string. As it gets passed down the algorithm the symbols are recognised through the tree which is being rebuilt. This rebuilt tree then converts the codes back to the original characters.

Huffman.java is the main class and allows user input to be made. This is also where the experimentation of the code occurs. A csv file involving the files to be compressed and decompressed are taken in. These files are then encoded and decoded. Whilst this happens the time is taken to see how long it takes the decoder and encoder to finish processing. This data is then used in the analysis part of the report.

I heavily used the resource that was given to us in the practical specification.

Testing:

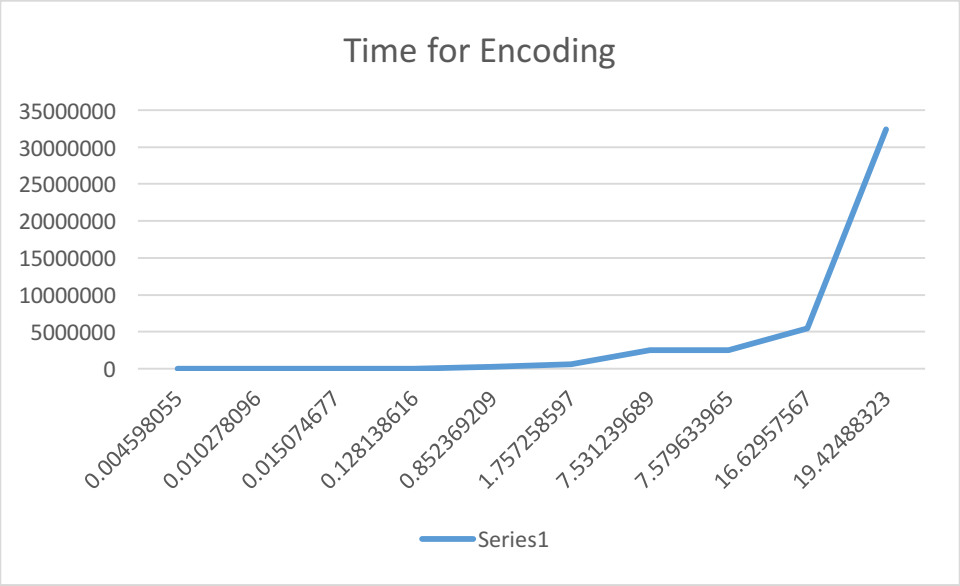
I have created unit tests to make sure that my smaller methods are in working order. I used three units tests in order to see whether or not my tree was being updated correctly after inputting symbols which would alter the weight hence forcing a swap. The swap method has also been tested to check that it works. I've also tested to see if the largest index with the same weight can be found correctly as well. These seemed to be the most important steps when updating the tree.

I've also manually checked that the encoding and decoding work. If the decoding did not accomplish its task it could mean that the encoder was not working as well as the decoder. In my case the decoder was able to decode the encoding which suggests that both parts of the program work. After looking at compression rates online I've seen that the compression rate I usually get which is 40% is fairly normal. [2]

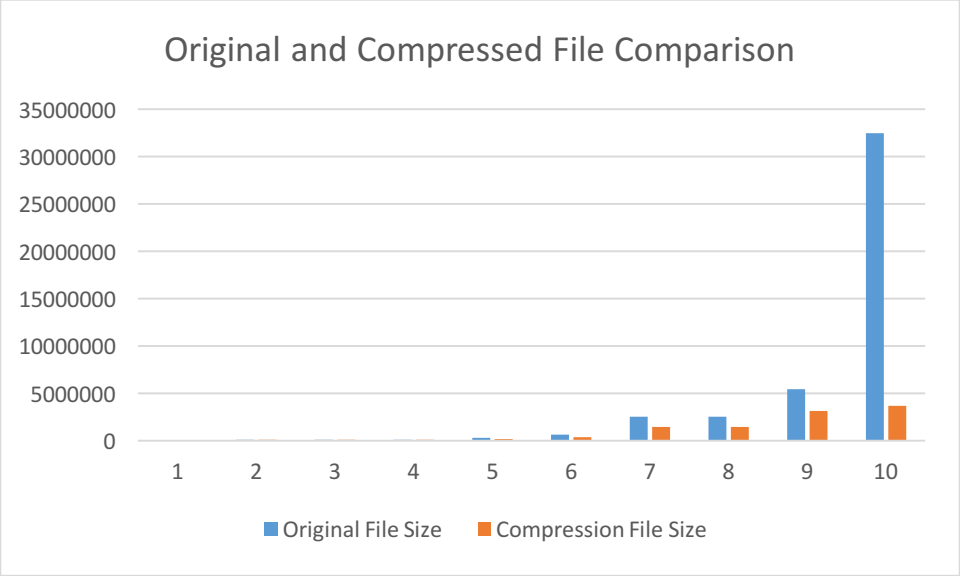
Analysis:

Whilst running the experiment I was able to get a significant sum of data. I analyse the results in the section below.

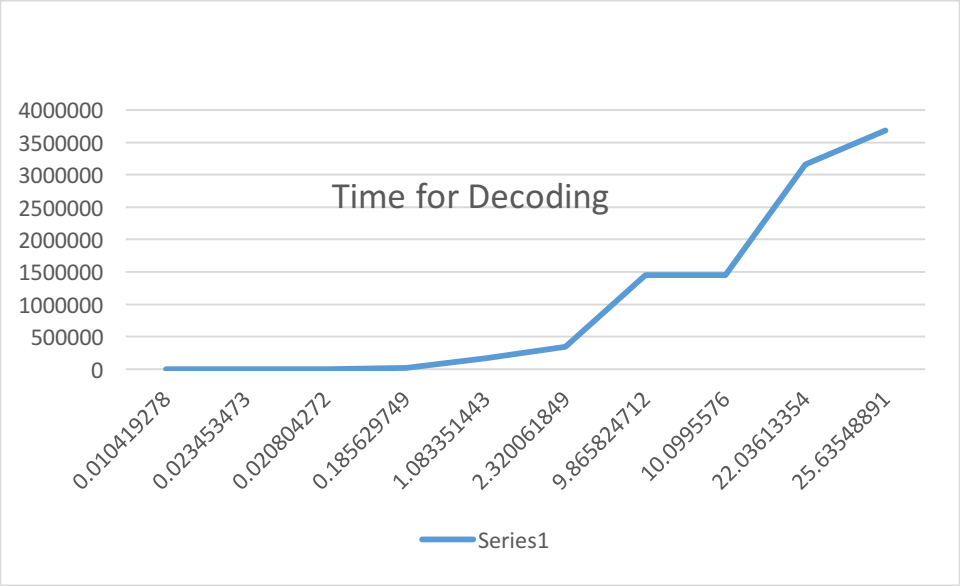
The below graph shows the time it took to encode the .txt files. One can see from this that it follows a fairly obvious pattern with the fact that the time taken to encode goes up as the size file goes up. Time taken seems to rise exponentially however.



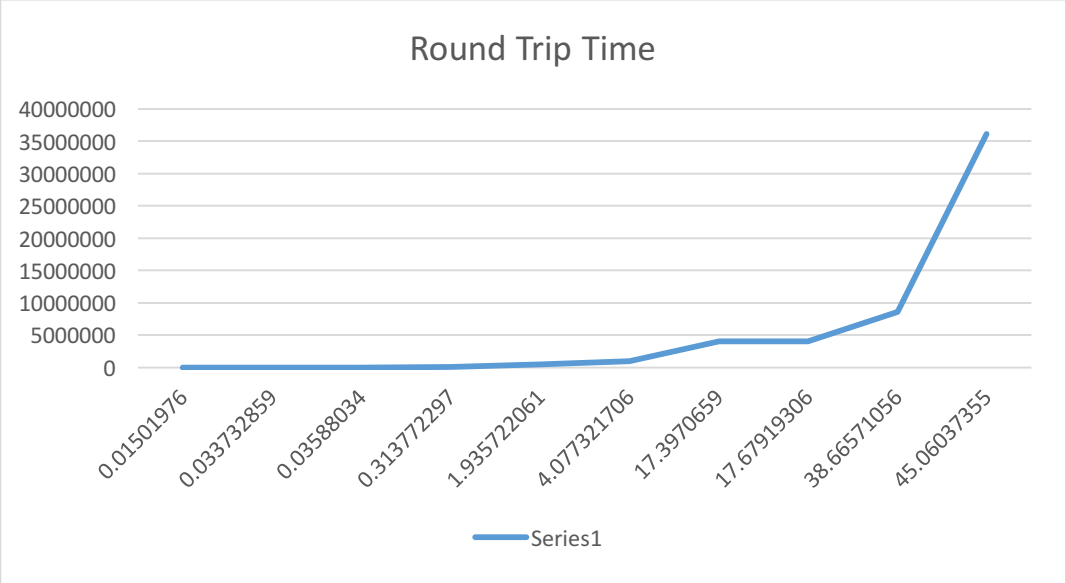
The below graph shows the difference between the compressed file sizes to there uncompressed counter parts. It seems that the compression size gradually increases as the file size increases.



The below graph shows the time it took to decode the .txt files. Again the time taken goes up as the file gets bigger. This graph seems to be different compared to the time taken encoding graph. There are more gradual steps that take place as the file size increases.



The graph below shows the time it took for the entire encoding and decoding to take place. This graph is far more similar to the encoding graph and is very different to how the decoding graph looks like.



From what I've seen from these graphs it seems as though the decoder is slower then the encoder. This could be due to the fact that the decoder does slightly more work by having to traverse the tree. Also in the code it self there are more next loops meaning a higher complexity. This would make the decoder much slower then the encoder as it doing more loops.

The table below shows the compressions rates for each of the .txt files that have been compressed. All of the below values are averages taken during the experiment.

Original File Name	Original File Size	Compression File Size	Compression ratio (%)
TestFiles/Text/6b.txt	6	7	-17
TestFiles/Text/500b.txt	541	325	40
TestFiles/Text/1.4kb.txt	1402	801	43
TestFiles/Text/38kb.txt	38147	21502	44
TestFiles/Text/300kb.txt	301627	169249	44
TestFiles/Text/500kb.txt	611529	343008	44
TestFiles/Text/1mb.txt	2535874	1449946	43
TestFiles/Text/2.5mb.txt	2535874	1449946	43
TestFiles/Text/5mb.txt	5458199	3157453	42
TestFiles/Text/6.5mb.txt	32443330	3683304	43

I have also included the excel file that contains the graphs I have used in this report.

Conclusion:

There were some minor hiccups that came while trying to code this assignment. The main hiccup was trying to convert strings to bits and storing them as bytes followed by decoding the bytes and converting them into a string.

Another hiccup I had was when trying to update trees so that the right Huffman coding could be made from them.

If I had more time I think I would have done my encoding and decoding slightly differently. This would've allowed me to test the compression's for two slightly different ways of encoding and decoding. I would have gone through the tree once, followed by encoding message. I would then decompress the tree. I think the codes for this compression might be slightly more different as you are not encoding whilst symbols are coming through. It might be slower however it could mean for a better compression.

I had a difficult time trying to get the images and files other than .txt to work. I believe that the problem arises due to the way I am handling the way I store the incoming bytes. Instead of keeping them as bytes I convert them into string (numbers). This makes it harder to convert things to bits. If I had more time to do the practical I would make sure to try and solve the bug I currently have.

I have enjoyed planning and programming this practical. I've learnt how to experiment and analyse different kinds of curiosities one might have about compression algorithms. I am far more comfortable with manipulating the abstract data structure types of trees. Also If I had more time I feel like I could have been able to do some more analysis such as comparing the compression rates for different file types and their sizes. I would also try and find out which file types were the fastest to encode/decode as well as the slowest.

Sources:

[1] https://www.cs.duke.edu/csed/curious/compression/adaptivehuff.html#implementations

[2] https://quixdb.github.io/squash-benchmark/