Data Compression course work write up

Hugo Hewitt - 700021852

Compression algorithms:

Huffman Encoding:

Huffman encoding on text files works by taking an input text file and then splitting it into each character. Working out how many times each character appears in the file next create a node and in that node is a character and the frequency of that character. After this Huffman encoding needs to create a tree, to do this it will find the two nodes with the smallest (and second smallest) frequencies (x and y nodes). After it has found the two smallest nodes, it creates a new node with the frequency as the sum of the two smallest nodes and no char, then make the x and y nodes children of the new node. As it is a binary tree to code to get to x and y would be 0 and 1, respectively. These steps are repeated until there is one node left, this is the root and has the frequency of all the characters summed together. To then to find the binary codes that represent each character (leaf of the tree) traverse the tree to get to each node. When traversing the tree to get to a node it must always start at the root, and if it goes left or right to get to the node add 0 or 1 (respectively) to the binary code and then when a leaf is reached the binary code will be the code that represents that char in the coded file. After this it will create a new file that in place of the chars will be the binary bits that represent them. It must also create a file that stores the tree so that the file can be decoded

To decode the encoded file that Huffman encoding creates a program must load in the tree from the file that is associated with the encoded file. Once that tree is loaded into the program, it will need to go through the encoded file bit by bit, and (starting from the root of the tree) if a 0 is there go left in the tree and 1 go right in the tree. Do this until a leaf is reached in the tree and that character is the character that is stored in the leaf node, then go back to the root of the tree and carry on reading the file bit by bit until the end of the file. Once the end of the file is reached a list of characters will have been created, it will then write these characters to a new file. This will create the file that was originally encoded.

Arithmetic Encoding:

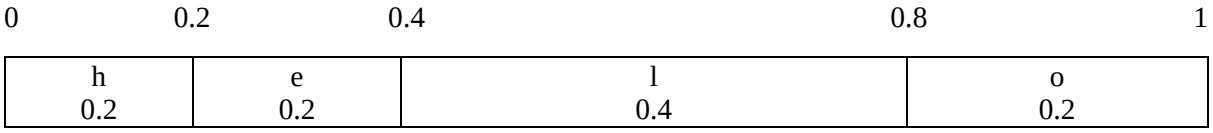
Arithmetic Encoding is a floating point and lossless compression algorithm. The algorithm has two parts, first translating the string into a floating-point range and the second is translating this into a binary sequence.

Frist part: Arithmetic encoding starts like Huffman encoding, where it reads the characters from a file, and assigns each individual character a frequency (the number of times it appears in the string of letters in the file). However, unlike Huffman, these characters and frequencies are stored in a frequency table, with the character, the frequency and the percentage of that character in the file (worked out by the frequency and the total number of characters.

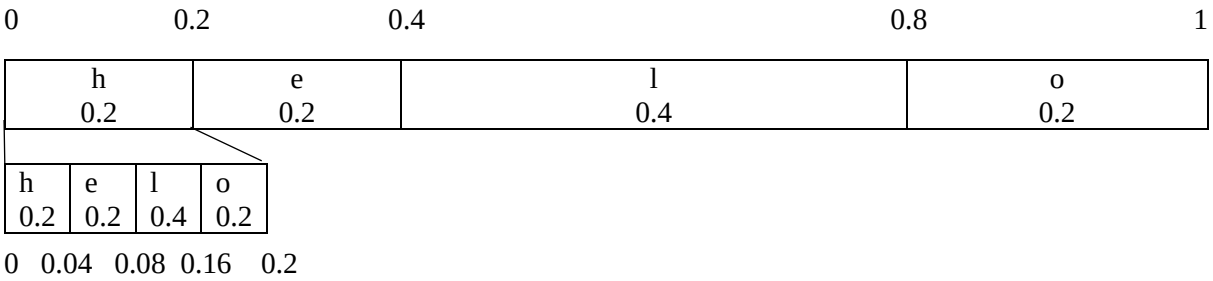
So for the word “hello”.

Character	Frequency	Probability
h	1	0.2
e	1	0.2
l	2	0.4
o	1	0.2

This is table is then used to map the frequency and characters to a number line:



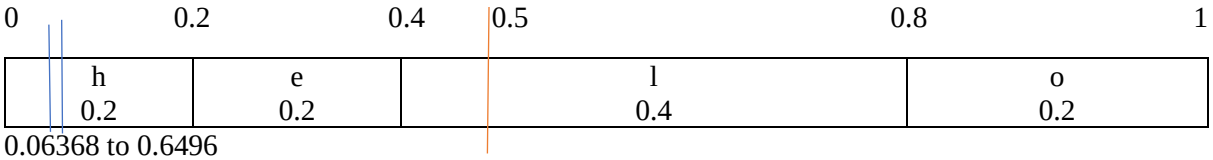
Next to do the encoding, a floating-point range representing our encoded string is needed. By starting with encoding h, would give the range 0 to 0.2. Then to encode E take the range from encoding h, 0 to 0.2 and apply the same frequency table to it.



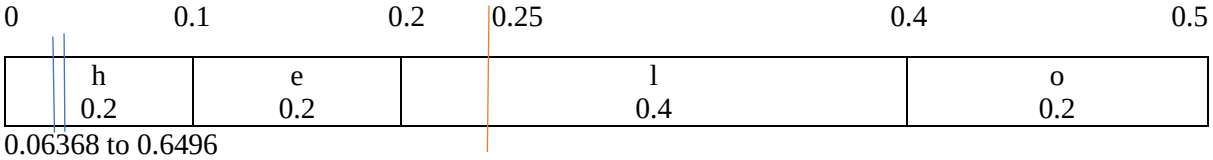
To encode e now, it has the range 0.04 to 0.08.

This process is repeated, copying the number line the fitting it in the previous number range until the entire file (or string) is encoded. In this case it continuous to a range from 0.06368 to 0.6496.

Second part: The second part is to run a binary search algorithm over the table to find a binary range that lays within the range from the first part (0.06368 to 0.6496). To do this it needs to plot the range on the original number line, then place the target range in the middle (0.5). Then see if the range is left or right of the middle, if left it is 0, and on the right it is 1.



Then repeat the process but change the range from 0 - 1 to 0 - 0.5.



In this example the range is still below the middle so output a 0, and if it was higher output a 1. Repeat this process until the middle lays between the range, the binary number that is outputted from the encoding is the encoded version of the file (or string). It can also be used to get back to the original with the frequency table.

Dictionary-based Compression:

Dictionary coding is a very simple way of encoding, the system will store a dictionary of words, and when encoding a file, it will create a dictionary (if there is not already one) with the words in the file an index assigned to it. So, for example.

“Hello my name is Hugo. Hello.” The encoder would then store these in a dictionary:

1-‘Hello’ | 2-‘ ‘ | 3-‘my’ | 4-‘name’ | 5-‘is’ | 6-‘.’ | 7-‘Hugo’

So, the encoded file would look like. “123242527621”. And could be easily decoded with the dictionary.

Data structures used:

My Huffman encoding and decoding system is implemented in Java.

-An array list is used to initially store the characters, and initial nodes.

-A priority queue is used to help with the creation of the tree

-A binary tree is then used to store the characters, frequencies and left/right node, this is implemented with objects. A huffnode object is used to represent a node which holds a character and the frequency of that character in the file, along with the children of that node (children are null if it is a leaf).

-An array of bytes to store the binary bits after traversing the tree from the root to that leaf

-A Bubble sort that sorts the short array list of nodes according to length of the binary string (shortest first)

Progress:

Week 1:

-Classes defined and interface in console working.

-Reading from a file (David Copperfield) and printing character by character to console

-The interface clears the screen after selecting an option

-Started creating an object-based tree system with nodes

Week 2:

-Creating the nodes for the tree with the characters extracted from the file

-Creating the tree (assigning the children nodes to the nodes) and creating head nodes with frequencies

-Traversing the tree to get the binary codes of each character (leaf)

-Creating a new folder

Week 3:

-Creating the Huffman tree file and the encoded file with the binary codes (which are converted to binary bits) representing the characters

-Creating the decoding algorithm

-Extracting the tree from the file

-Then using this tree to decode the compressed file

-Putting the decoded text in a new file

Week 4:

-Performance analysis and write up of other compression algorithms

-Finishing touches to the code

-Finishing the write up document, screen recording using the system.

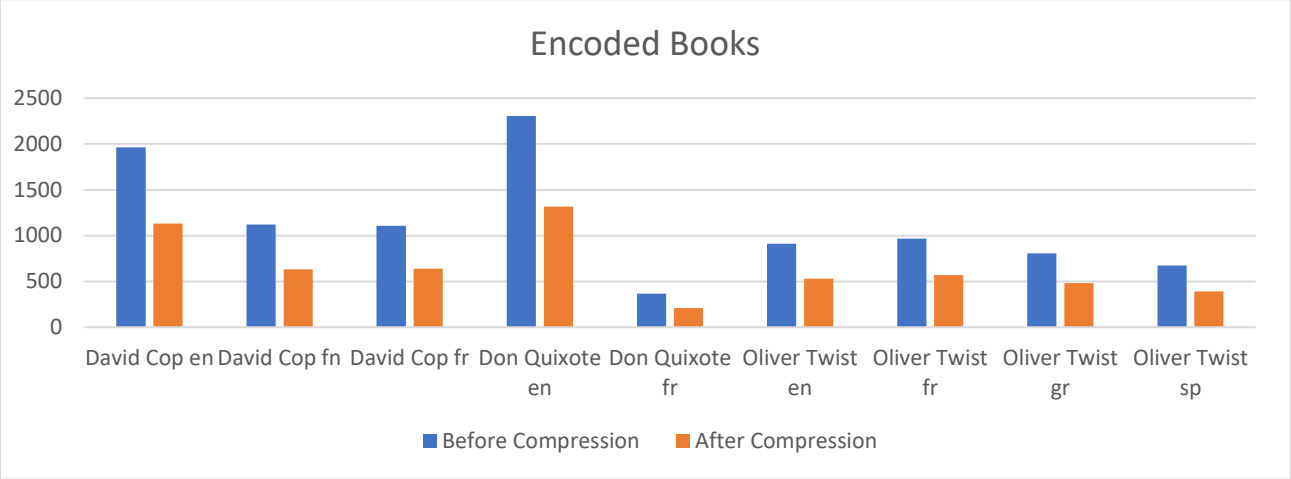
Performance analysis:

Books:

On average with realistic data sets (books) the file size was reduced by about 40%, and with datasets that had few different characters the files were reduced by more than 40%.

Note: the after encoding size includes the tree file that also needs to be stored.

Size in KB --- Key: en=English; fn=Finnish; fr=French; gr=German; sp=Spanish.

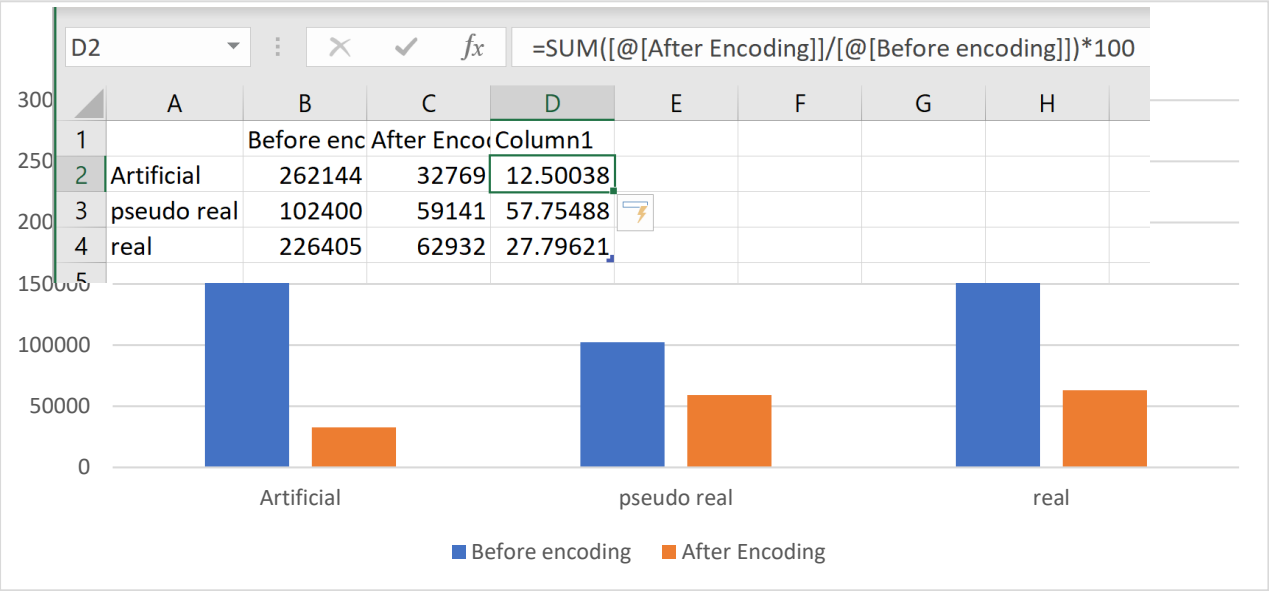


D2 =SUM([@[After Compression]]/[@[Before Compression]])*100

	A	B	C	D	E	F
1		Before Compression	After Compression	Column1		
2	David Cop en	1964	1132	57.63747		
3	David Cop fn	1121	633	56.46744		
4	David Cop fr	1107	639	57.72358		
5	Don Quixote en	2306	1317	57.11188		
6	Don Quixote fr	367	208	56.67575		
7	Oliver Twist en	911	532	58.39737		
8	Oliver Twist fr	968	568	58.67769		
9	Oliver Twist gr	806	481	59.67742		
10	Oliver Twist sp	672	391	58.18452		

As shown above, the size after compression is on average just under 60% of the original file size, meaning the compression reduces the size by 40%.

Data Sets:



As shown above, with data sets that have few repeating characters (such as the artificial and real data sets) Huffman encoding is very efficient and makes the files much smaller. But with the pseudo real data set, it is closer to a book with lots of different characters, so it achieves a size similar to the books above, around 40% smaller.

References:

https://www.candidjava.com/tutorial/program-to-read-a-file-character-by-character/

Free eBooks | Project Gutenberg

https://www.thejavaprogrammer.com/char-cannot-be-dereferenced/

https://beginnersbook.com/2013/12/how-to-empty-an-arraylist-in-java/

https://www.javatpoint.com/how-to-create-a-new-folder-in-java#:~:text=In%20Java%2C%20we%20can%20use,to%20create%20a%20new%20folder.

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

https://www.geeksforgeeks.org/priority-queue-class-in-java-2/

Text Book: Java A beginner's Guide, Herbert Schildt.

Arithmetic Coding - The Hitchhiker's Guide to Compression (go-compression.github.io)