18310726

COMP20290

Algorithms

Assignment 1

Huffman Compression

Task 1 Huffman Tree of a given phrase drawn by hand

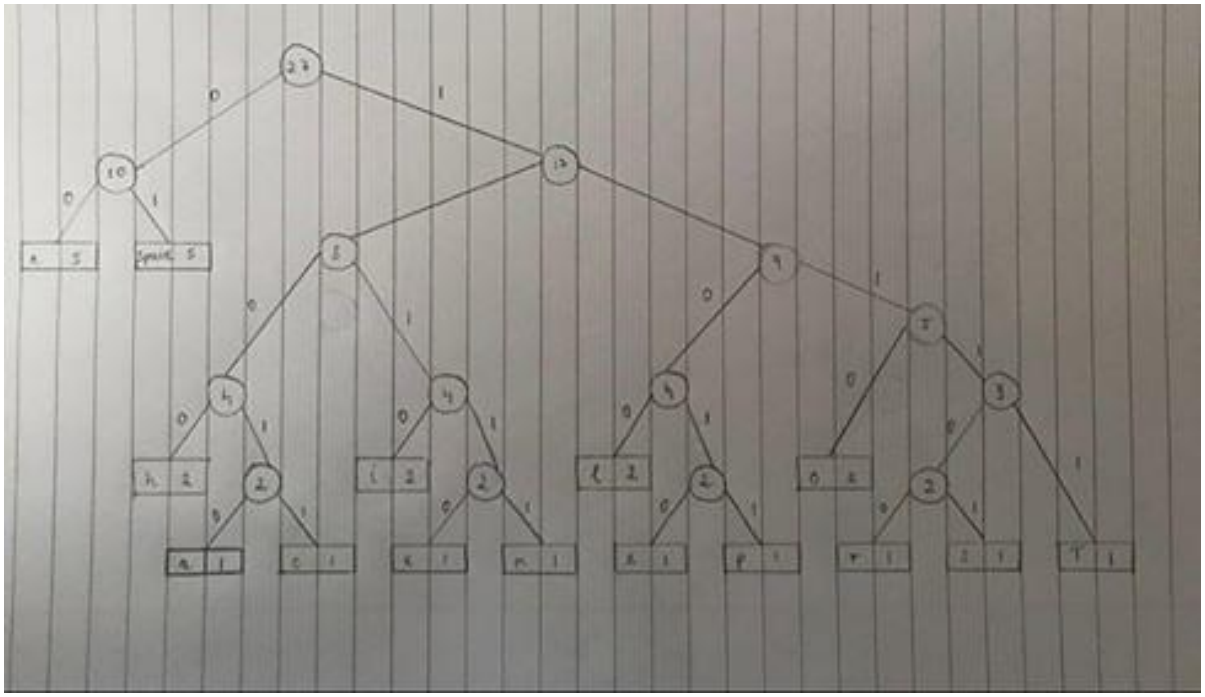
Phrase used for encoding: “There is no place like home”

The code word table

char	**frequency**	**encoding**
a	1	10010
c	1	10011
e	5	00
h	2	1000
i	2	1010
k	1	10110
l	2	1100
m	1	10111
n	1	11010
o	2	1110
p	1	11011
r	1	111100
s	1	111101
T	1	11111
space	5	01

The hand drawn Huffman tree is included in the picture below.

18310726



Task 2 Huffman Compression Suite

Some notes on my implementation of the Huffman compression algorithm

The file HuffmanCompression.java implements the Huffman compression algorithm

Please ensure that all the necessary files are compiled before execution.

-to count the number of bits in a particular file use the command:

java BinaryDump < filename.file_extension

-to count the number of bits produced upon compression of a file

java HuffmanCompression - < file.file_extension | java BinaryDump

-to count the number of bits produced upon decompression of a file

java HuffmanCompression + < file.file_extension | java BinaryDump

-to output the compressed version of a file into a new file

java HuffmanCompression - < file.file_extension (the file to be compressed) newFile.file_extension (the new file to be created with the compressed version of the file as its contents)

-to output the decompressed version of a file into a new file

java HuffmanCompression + < file.file_extension (the file to be compressed) newFile.file_extension (the new file to be created with the compressed version of the file as its contents)

As well as implementing the Huffman algorithm within the HuffmanCompression.java file, I have also altered the RunLength.java file so as to be able to output the time it takes to execute the compression using that file. Please ensure that you use the updated version of that file while trying to time the RunLength compression.

Task 3 Compression Analysis

Overall performance of the algorithm

I believe my implementation of the Huffman Compression algorithm is quite efficient which can be seen when examining the results in the table provided below. The times taken to compress and decompress the files used are quite small, even in the case of the files of significant size e.g. mobydick.txt. All of the compression ratios indicate a significant amount of compression (note that since the ratios are taken as compressed bits/ original bits, therefore the smaller the percentage, the more compressed the file is. In the table below it can be seen that all of the ratios are lower than or equal to around 50% indicating that the files were reduced in size by almost a half or more).

Upon benchmarking the Huffman compression with RunLength compression (a table with the results for RunLength has been included below), we can also notice a significant difference in the compression ratios between the two, indicating that the Huffman Compression performs better for the files used.

Q1: Compression ratios as well as the times recorded for each file are included in the table below.

Q2: Upon execution, it can be seen that all of the files that have been compressed return to its original size (number of bits), as they were before compression was completed, i.e. decompressed bits = original bits for each of the files. Those can be seen in the table below as the number given for “original bits”. The reason why this happens is due to the fact that Huffman Compression is a type of lossless data compression, meaning that it allows the original data to be perfectly reconstructed from the compressed data, hence why the size is also the same.

Q3: Trying to compress a file that already has been compressed doesn’t yield beneficial results. Although there is nothing stopping us from compressing a file multiple times, once a file has been compressed all the further compressions will only produce a file that is larger in size than the previous one. Since we’re using lossless compression the file won’t become corrupt when we perform multiple compressions but there won’t be any real benefit in doing so.

Huffman encoding works by assigning codes to all of the symbols in the file on the basis of their frequency. A symbol that appears more frequently is assigned a shorter-bit string than one that appears less frequently. Although there might be multiple ways to assign the bit-strings to the symbols in the file, since the frequencies of some symbols might be the same, all of those ways will be equally efficient, meaning that once a file has been encoded, its size can’t be reduced any further through further compression.

I have executed multiple compressions on an already compressed file as an example to demonstrate the point made above. The results yielded are as follows:

Compressed file to begin with: medTaleCompressed.txt, size: 24664 bits

Upon further compressions: medTaleDoubleCompressed.txt, size: 26568 bits

medTaleTripleCompressed.txt, size: 29000 bits

medTaleQuadrupleCompressed.txt, size: 31 488 bits

This demonstrates the point made. The files listed are included as part of the submission for reference.

Q4: Upon compression of the bitmap file using the two different compression algorithms, the Huffman compression and the RunLength compression we can notice a significant difference in the efficiency of the compressions, as shown in the tables below. We can see that the Huffman compression is better suited for this type of file since the size of the compressed file is smaller (816 bits) than the one when RunLength compression is used (1144 bits). This is because RunLength compression is better suited for the case where we have long runs (sequences in which the same data value occurs in many consecutive data elements) of 1's and 0's and the bitmap file provided contains runs of 1's and 0's which are relatively short. With RunLength runs of data are stored as a single data value and count, therefore if the file to be compressed does not consist of many runs, the size of the file could be significantly increased rather than decreased through the use of this type of compression.

Huffman Compression

File Used	Time to Compress	Time to Decompress	Compression Ratio
	in Milliseconds		*compressed bits/original bits*
genomeVirus.txt	15	11	25% (12576/50008)
medTale.txt	23	28	54% (24664/45872)
mobydick.txt	154	120	57%(5505432/9708968)
randomText.txt	16	14	57% (17968/31616)
q32x48.bin	9	7	53% (816/1536)

RunLength Compression

File Used	Time to Compress *in Milliseconds*	Time to Decompress	Compression Ratio *compressed bits/original bits*
genomeVirus.txt	19	19	447% (223632/50008)
medTale.txt	16	16	405% (185784/45872)
mobydick.txt	198	232	406%(39407992/9708968)
randomText.txt	13	15	411% (130040/31616)
q32x48.bin	4	3	75% (1144/1536)