Compression

Contents

[Homework Exercise 2](#_Toc183780852)

[Task 1 2](#_Toc183780853)

[Task 2 2](#_Toc183780854)

[Task 3 5](#_Toc183780855)

[Compression Ratio 5](#_Toc183780856)

[Fixed-Length Encodings 5](#_Toc183780857)

[Task 4 5](#_Toc183780858)

[Extension Exercise 6](#_Toc183780859)

[Introduction 6](#_Toc183780860)

[8](#_Toc183780861)

[Challenges 8](#_Toc183780862)

[Results 8](#_Toc183780863)

[Time 8](#_Toc183780864)

[Small Files 9](#_Toc183780865)

[Pre-compressed files 10](#_Toc183780866)

[Executables 11](#_Toc183780867)

[Text 12](#_Toc183780868)

[12](#_Toc183780869)

[Conclusion 13](#_Toc183780870)

[Bibliography 13](#_Toc183780871)

# Homework Exercise

## Task 1

A screenshot of a computer code

Description automatically generated

Figure 1. Presented problem (included due to being incorrect).

001001011011110111 = 00 100 101 101 111 01 11 = ACDDFB 11

The last 2 digits cannot be converted because there is no code for 11. If we reverse the characters, we can fully decode it getting:

111011110110100100 = 111 01 111 01 101 00 100 = FBFBDAC

However, in both cases the message is un-readable so without more context it is impossible to tell if this was a mistake or reversed to further obfuscate the message.

## Task 2

The first step is to count the occurrence of each character in the sentence, (32) is used to represent a space character. The output from the program written to do this is shown in Figure 2. We then put this data into some diagram software and build the Huffman tree as shown in Figure 3. I have also provided Table 1, which is easier to use for practical purposes. Finally, we use the table to produce the following encoded binary message (hyphen delimited for easier reading):

01000-0010-0001-10-11001-01001-011010-010110-011100-10-010101-0011-0000-11100-011111-10-011000-0000-11101-10-011011-01001-011110-11000-11010-10-0000-11011-0001-0011-10-01000-0010-0001-10-011101-010100-11111-11110-10-010111-0000-011001

A screenshot of a computer

Description automatically generated

Figure 2. Output of character counting application.

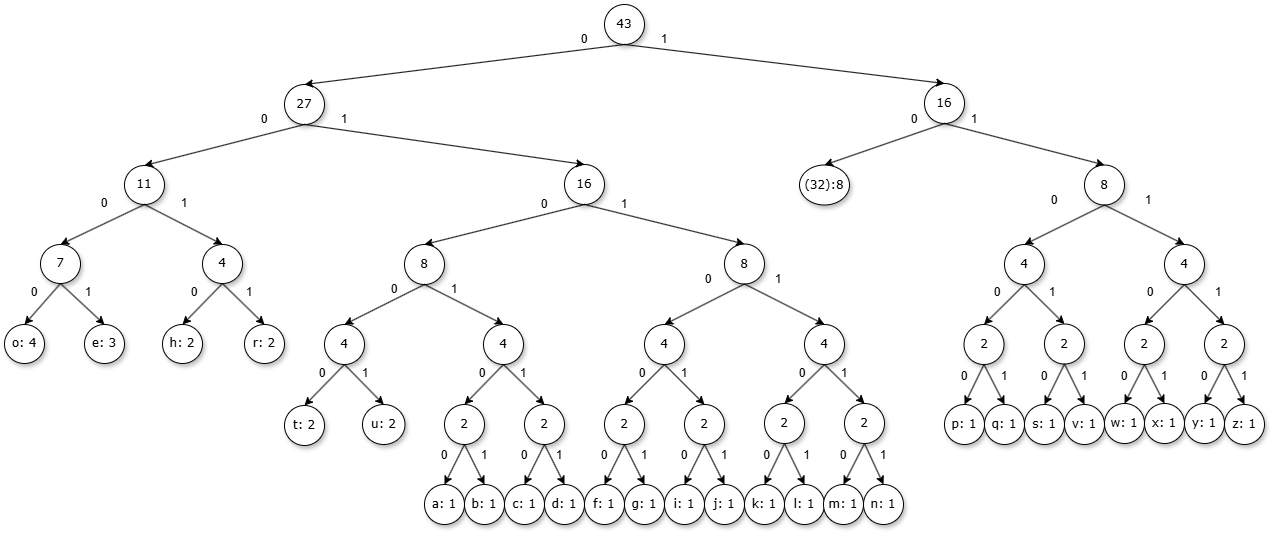


Figure 3. Hand-generated Huffman tree.

A white sheet with black lines and letters

Description automatically generated with medium confidence

Table 1. Huffman codes derived from Figure 3.

## Task 3

### Compression Ratio

Using an 8-bit ASCII encoding for this message the number of bits used would be 43 characters x 8 bits = 344 bits.

As there are 192 characters in the encoded message and as each character represents a single bit the Huffman encoded message uses 192 bits.

The compression ratio achieved using Huffman encoding would be 192 / 344 = 0.558. This means that the encoded message contains the same information in 55.8% of the storage space provided the recipient knows or can derive the dictionary.

### Fixed-Length Encodings

The string only uses the 26 lowercase alphabetic characters and the space character and thus could be represented by any number of bits n where 2n >= 27. The shortest of these is 5 bits which if used would produce an encoded message with a compression ratio of 5/8 or 0.625, significantly less efficient that Huffman encoding.

## Task 4

The way a Huffman tree is constructed indirectly compares the sum of bits saved by encoding a character as a reduced length encoding to the sum of bits cost by doing so and this means by its very structure it produces the most efficient binary encoding possible and cannot be beaten by a fixed-length encoding. Importantly this ignores the cost of communicating the encoding and so in some cases where the message is small, or the Huffman encoding is identical to the fixed-length encoding a fixed-length encoding may be more efficient.

Shannon Entropy in this context is a way to measure the amount of entropy in a string using the formula in Figure 4 (Karaca & Moonis, 2022, pp. 236-238). If the string contained only one type of character, then entropy would be 0 and we would theoretically require 0 bits per character. In the worst case all characters have an equal probability, and we get an entropy of log2(n) where n is the number of unique characters in the string. I have created a program to calculate the Shannon Entropy and the average code length of a given Huffman encoding with the output for my encoding shown in Figure 5. Shannon Entropy for the string is 4.385 bits/character and our encoding has achieved 4.465 bits/character making it a reasonable approximation given that bits are discrete.

A mathematical equation with numbers and symbols

Description automatically generated

Figure 4. Shannon Entropy formula.

A screenshot of a computer program

Description automatically generated

Figure 5. Shannon entropy compared to average code length.

# Extension Exercise

## Introduction

The concept of Huffman coding is simple to understand and well outlined in the original paper (Huffman, 2006), with a good grasp of fundamental computer science, the application was developed from first principles while trying to avoid haphazardly wasting CPU cycles and memory. An output of the usage message from the final program is shown in Figure 6, showing most of the features.

A computer screen shot of text

Description automatically generated

Figure 6. Final program usage.

Initially it was intended for the application to support any input, which would mean supporting up to 255-bit long Huffman codes (Figure 7), however, this was deemed unnecessary due to the added complexity, but especially the very low chances of these being required (Figure 8).

A screenshot of a computer code

Description automatically generated

Figure 7. Maximum code depth demonstration.

## A white text with black text Description automatically generated

Figure 8. n code generation requirement.

## Challenges

Since Huffman codes can be more than a byte long a naïve solution when encoding is to use a pointer to a type that can store the maximum bit depth supported, then copy as many whole bytes as possible before using bit operations to encode the remainder and move the pointer as needed. This however becomes much more complex if your machine, like most, is little-endian. As little-endian machines store the least significant byte first your encoding will not appear in memory as you intend it to, the solution chosen was the simplest, to process one bit at a time.

The Huffman tree can be printed by the application, this is done recursively as a scheme for doing it linearly could not be found through research, nor conceived of in a reasonable time. This is generally not a concern as the algorithm is limited to 32 bits of depth, although it is possible to overflow the stack with excessive recursion.

## Results

### Time

As can be seen in Figures 9 and 10 the time taken to both encode and decode files scales linearly, decoding is slower than ideal due to implementation details. Given additional development time this could be improved, however, decompression will always be slower than compression since compression can be a single lookup per byte whereas multiple tests may be required per byte decompressed due to the fact the code length is not known.

Figure 9. Linear growth of time to encode files with size.

Figure 10. Linear growth of time to decode files with size.

### Small Files

Small files, and especially small files with many unique byte values show very poor performance (Figure 11). This is not surprising, the first byte will always contain 5 bits used as flags and 3 bits to indicate the number of bits used in the last byte of the file, the next 32 bytes are a 256-bit bitmap indicating whether a byte value is used, then the variable length dictionary is encoded with only the byte values used, order by value. It was envisioned that the first 5 bits could be used to define different dictionary encoding schemes (such as fixed length) to overcome this issue, however, this was not achieved due to time constraints.

Figure 11. Space required to store small files with all unique characters.

### Pre-compressed files

Pre-compressed files typically come out larger when run through the algorithm than their original size, since we are not only compressing data with a very high Shannon-Entropy (Figure 12), but also compressing the dictionary used to encode that data this is the expected outcome, and any improvement would show a flaw in the compression used on the file originally.

A screenshot of a computer program

Description automatically generated

Figure 12. Shannon entropy of pre compressed file.

### Executables

Executables compress well getting compression ratios as low as 67% (Figure 13) interestingly looking at a typical Huffman tree generated we see a very significant occurrence of 0 value bytes (Figure 14), furthermore, looking at the encoded files in a hexadecimal viewer these tend to be in large blocks (Figure 15), suggesting specific optimizations could be made with this knowledge.

Figure 13. Executable compression performance.

A black background with white text

Description automatically generated

Figure 14. 0 value node with high occurrences

A blue and white background with numbers

Description automatically generated

Figure 15. Block of encoded 0 value bytes.

### Text

Once text files are large enough to overcome the overhead of the dictionary they show significant reductions in data size, typically reaching 50% - 60% of their original size (Figure 16). This is what we would expect as text files only use the printable subset of ASCII, tend to only use symbols from a single language, and have a high range of byte frequencies due to language making irregular use of different symbols.

Figure 16. Plain English text performance.

## Conclusion

Huffman coding is computationally inexpensive providing fast data transformations. Huffman coding allows the compression of arbitrary data but works best where a subset of byte values are used and a high disparity in byte value frequencies is observed. This makes unilingual text encoded in ASCII an ideal target, which is well supported by the collected data, however, any data that supports these criteria will perform well as shown with executables (Figure 13).

# Bibliography

Huffman, D. A. (2006, Feburary). A Method for the Construction of Minimum-Redundancy Codes. *Resonance, 11*(2), 91-99. Retrieved November 08, 2024, from https://www.researchgate.net/publication/225338470\_A\_method\_for\_the\_construction\_of\_minimum-redundancy\_codes

Karaca, Y., & Moonis, M. (2022). Chapter 14 - Shannon entropy-based complexity quantification of nonlinear stochastic process: diagnostic and predictive spatiotemporal uncertainty of multiple sclerosis subgroups. In Y. Karaca, Y.-D. Zhang, M. Moonis, D. Baleanu, & O. Gervasi, *Multi-Chaos, Fractal and Multi-Fractional Artificial Intelligence of Different Complex Systems* (pp. 231-245). New York: Academic Press. Retrieved November 09, 2024, from https://www.sciencedirect.com/science/article/pii/B9780323900324000183