CS EE World
https://cseeworld.wixsite.com/home
May 2024
15/34
C
Submitter Info:
akidoo (Discord)

A Comparative Analysis of the Huffman Coding and Shannon-Fano Coding Compression Algorithms

Which Lossless algorithm between Huffman Coding and Shannon-Fano Coding is more efficient in terms of their Compression Ratio?

Subject: Computer Science

Candidate Code: jpk578

Word count: 3810

Contents

Introduction………………………………………………………………………………… 2

Background Information……………………………………………………………… 5

What is a Tree?.. 5

What is a Binary Search Tree?.. 6

What is Entropy?.. 7

What is Variable Length Coding (VLC)?.. 8

How do Lossless Compression Algorithms work?..................................... 8

Huffman Coding Algorithm……………………………………………………………… 12

Shannon-Fano Coding Algorithm…………………………………………………… 17

Comparing the two Algorithms……………………………………………………… 19

Methodology……………………………………………………………………………….. 20

The Programs…………………………………………………………………………… 21

Test Data………………………………………………………………………………… 22

Results + Analysis……………………………………………………………………... 23

Conclusion……………………………………………………………………………… 26

Limitations……………………………………………………………………………… 27

Works Cited……………………………………………………………………………... 28

Appendix………………………………………………………………………………… 31

Introduction

Data Compression, the process through which information can be represented in a more compact form (Sayood) can work through various different types of algorithms but can largely be categorized into two “umbrella terms”, so to speak, those terms being lossy compression, and lossless compression. According to Ng et al., Lossless compression is used where perfect reproduction is required while lossy compression is used where perfect reproduction is not possible or requires too many bits. This means, hence their names, lossy results in data loss, while lossless does not. Compression is a process that is quite commonly performed to shrink very large files into much smaller sizes in order to be able to, for example, save storage, or to be sent over the internet much faster than if they were not compressed (Chung). Uncompressed files also run the risk of getting corrupted. There are a variety of different algorithms that file compressing software use, with each one of them having varying strengths and weaknesses from each other. Due to the fact that lossless compression algorithms preserve the quality of files, they are often preferred over lossy compression algorithms, hence they will be the focus of my EE.

There are a multitude of criteria that can be used to evaluate the effectiveness of both lossy and lossless compression algorithms; those being the time complexity as well as space complexity (Davies), compression ratio, which is the number of bits required to represent the data before compression to the number of bits required to represent the data after compression (Sayood). The most important one out of the three would be the compression ratio, as the whole point of a compression algorithm is to reduce the size of the file as much as possible. Two extremely popular lossless compression techniques are Huffman Coding, and Shannon-Fano coding.

I chose these two algorithms in particular despite them being relatively old, first and foremost due to the fact that they are lossless compression techniques, making them more ideal for practical applications in the real world. These algorithms are also still very widely used within a multitude of compression formats, such as MP3, to name one. These two are also good to test as they are quite similar in nature to each other; with both of them making similar techniques to compress files.

This research could be beneficial to businesses looking to store large amounts of data; with an ever increasing number of companies opting to store their data digitally, the need for more storage space is increasing rapidly. Furthermore, compression improves the security of data by a significant margin, for companies storing sensitive data such as banks and hospitals (Saunders) and also greatly reduces the cost of storage, as companies don't need to invest as much money into storage space.

Background Information

What is a Tree?

Before delving into the algorithms themselves, it is important to understand a type of data structure known as a tree; this is due to the fact that both algorithms make use of Binary Search Trees, which are a subset of trees themselves.

By definition, a tree is a hierarchical data structure which represents data in such a way that it can be traversed through with relative ease (GeeksforGeeks). It is made up of nodes that are connected to each other via an “edge” - essentially just a line that links the nodes up together. A diagram of this is shown below.

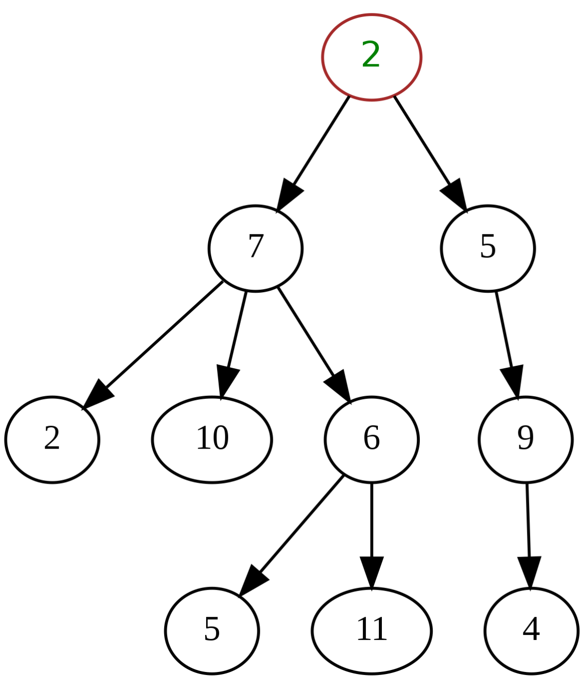


Figure 1. Representation of a Tree (Wikipedia Contributors)

As we can see in Fig 1 above, each parent node can have any number of child, or leaf nodes. Such is not the case with binary search trees, however, which are discussed in the next subsection. It is also important to note that trees are unordered, and do not follow a sequence.

What is a Binary Search Tree?

A binary search tree is a subset of the tree abstract data structure, and follows the same principles, however, in a binary search tree, all the nodes are ordered sequentially, and each parent node can only have 2 child, or leaf nodes the lower valued nodes are always to the left side of the parent, and the higher ones go to the right. An example of this is shown below.

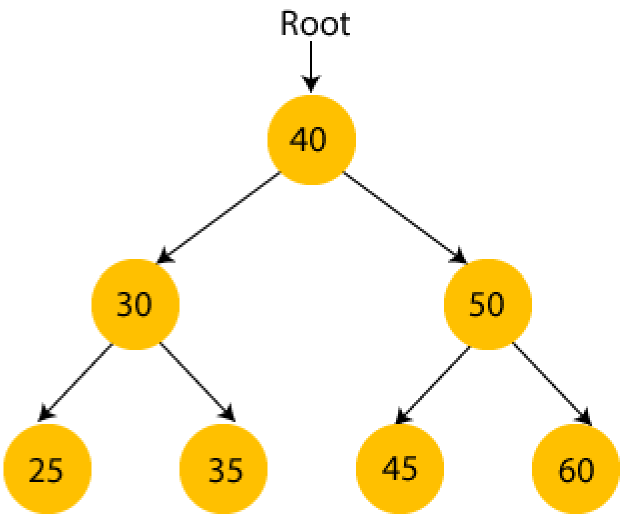


Figure 2. Diagram displaying a Binary Search Tree (Java Tutorials Point)

What is Entropy?

Entropy is defined as the smallest number of bits needed to represent a value. Shannon extended this idea and applied it to larger datasets, where the entropy is the minimum number of bits needed to represent the entire dataset (McAnlis and Haecky), essentially meaning that the entropy of a set is the smallest size a set of data can be compressed to. The formula for this is shown below.

$$H(s) = -\sum_{i=1}^{n} p_i \log_2 p_i$$

Where H is Entropy, p_i is the probability of an element occurring, and ∑ is the number of occurrences of the element.

Most developers of compression algorithms look to disprove this formula; but as a general rule of thumb, this is the smallest size the algorithms look to achieve and this is no exception for Huffman Coding and Shannon-Fano coding.

What is Variable Length Coding (VLC)?

A variable length code is something that represents symbols in a certain number of bits. They allow for the lossless compression of data (Wikipedia Contributors). It is a concept that is important to know in order to understand the theory behind lossless compression algorithms. In essence, the probabilities of the occurrence for each symbol is calculated, and then the variable length code is assigned to them. VLC's are the core principle behind a large number of lossless compression algorithms, including Huffman Coding and Shannon-Fano coding, as both these compression algorithms make use of it in the process of creating a probability model for the Binary Search Tree generated by them. More on that later.

In a nutshell, VLC's use 3 steps to encode data. The algorithm first goes through the string, or whichever data set it is given. Codewords are then assigned to each symbol within the data, depending on their probability of occurring. Lastly, the algorithm once again goes through the data, and outputs the codeword to the compressed bitstream (McAnlis and Haecky).

How do lossless compression algorithms work?

Generally speaking, lossless compression algorithms make use of statistical modeling techniques in order to limit repeated data within a file (Chung). The algorithms work differently when it comes to text files, for example, versus audio, however it is fundamentally the same. A simplified example is given below.

The following is a representation of recorded data in 1 byte sample:

00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 63 63 57 56 60 59 67 67 71 71 71 71 71 71 71 71 6A 7A 66 86 86 83 83 82 81 6B 6B 72 72 72 76 75 75 75 9E 9E 9E

As we can see in this data, there are a significant number of repetitions of certain sounds. These are highlighted below.

00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 63 63 57 56 60 59 67 67 71 71 71 71 71 71 71 71 6A 7A 66 86 86 83 83 82 81 6B 6B 72 72 72 76 75 75 75 9E 9E 9E

These repetitions arise from patterns, such as a certain note being played for several seconds at a time, or in the case of “00”, prolonged periods of silence.

The algorithm, however, may not only look for repetitions of patterns, for example, guitar riffs or drum beats. We will be using a new sample to display this, seen below:

00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 67 66 65 78 79 67 66 65 77 77 77 75 76 80 55 51 67 66 65 69 6E 73 75 76 80 9E 8A 67 66 65 75 76 80

The patterns are highlighted below:

00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 67 66 65 78 79 67 66 65 77 77 77 75 76 80 55 51 67 66 65 69 6E 73 75 76 80 9E 8A 67 66 65 75 76 80

The algorithm can then significantly shrink the file size by representing the repeated values and/or patterns using a single hexadecimal value, where the first letter can be any letter that was not present in the input data, the second letter represents the number of repetitions in hexadecimal, and the third part is what was repeated.

For example:

00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 = CF 00

C is a randomly chosen letter, F is 15 in hexadecimal, and "00" is the repeated data.

Repeating this with the rest of the data, the compressed sample is shown below.

CF 00 C2 63 57 56 60 59 C2 67 C8 71 6A 7A 66 C2 86 A2 83 82 81 C2 6B C3 72 76 C3 75 C3 9E

For the second sample, the patterns are compressed in a similar fashion. The first letter is a randomly chosen letter not present in the sample, and the second letter is a number to represent what pattern it is. For example:

67 66 65 = F0

Where F is a randomly chosen letter, and 0 represents the fact that “67 66 65” is pattern number “0”

Repeating this with the other patterns, we can now compress the second sample.

00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 F0 78 79 F0 77 77 77 F1 55 51 F0 69 6E 73 F1 9E 8A F0 F1

We can now go a step further and compress the repeated values as well, like so:

CF 00 F0 78 79 F0 C3 77 F1 55 51 F0 69 6E 73 F1 9E 8A F0 F1

As can be observed, the sample has been compressed significantly.

Going more in depth into the process, It involves two main steps, the first of which being the generation of the statistical model, the second step being the algorithm using said model to “predict” the next bit sequences. As far as the statistical models go, there are two ways in which they are generated, known as static models, and adaptive models.

The main difference between static and adaptive models is that in static, the model is created and stored based on the input data, however in adaptive, the model adapts and changes based on new data, hence the name. Adaptive models are typically preferred over static models for streaming data, for the reason that they adapt as they are fed more data. (Wikipedia Contributors)

In a nutshell, lossless algorithms create a set of data that can be uncompressed into what is essentially a duplicate of the original file.

Huffman Coding Algorithm

One of the most famous lossless compression algorithms of all time, Huffman coding follows the same principle as any other lossless compression algorithm. David Huffman developed this lossless compression algorithm which is generally most effective with samples with large volumes of recurring data or patterns (Geekific). As a result of its popularity as well as simplicity, it is an extremely commonly used algorithm. The pseudocode explaining how the algorithm creates a binary search tree is shown below:

Huffman Coding 1

Algorithm 1 – Compute Huffman codeword lengths, textbook version.

0: **function** *CalcHuffLens*(W, n)
1: // initialize a priority queue, create and add all leaf nodes
2: set $Q \leftarrow [\,]$
3: **for** each symbol $s \in \langle 0 \ldots n-1 \rangle$ **do**
4: set $node \leftarrow new(leaf)$
5: set $node.symb \leftarrow s$
6: set $node.wght \leftarrow W[s]$
7: $Insert(Q, node)$
8: // iteratively perform greedy node-merging step
9: **while** $|Q| > 1$ **do**
10: set $node_0 \leftarrow ExtractMin(Q)$
11: set $node_1 \leftarrow ExtractMin(Q)$
12: set $node \leftarrow new(internal)$
13: set $node.left \leftarrow node_0$
14: set $node.rght \leftarrow node_1$
15: set $node.wght \leftarrow node_0.wght + node_1.wght$
16: $Insert(Q, node)$
17: // extract final internal node, encapsulating the complete hierarchy of mergings
18: set $node \leftarrow ExtractMin(Q)$
19: **return** *node*, as the root of the constructed Huffman tree

Figure 3. Huffman Coding Pseudocode (Moffat)

What this code is essentially doing is creating a tree using the frequency of characters or values, and creating a “prefix code” for each instance. The idea behind the whole algorithm, mathematically, follows relatively simplistic mathematical principles, which is what makes it such a popular algorithm to implement. The n symbols in the input alphabet make up the weights of “leaf nodes”. The two leaf nodes with the lowest weight are then found and removed from the set and then merged together to make a new node. This node is then re-added to the initial set, and then the whole process gets repeated for n-1 iterations.

For discussion’s sake, we may take the string below as an example:

B	B	B	B	A	A	A	C	C	D

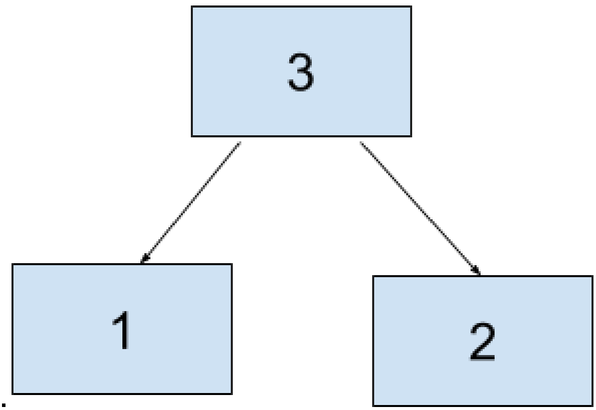
The program first calculates the frequency of of each character in the string, like shown below:

Letter	B	A	C	D
Frequency	4	3	2	1

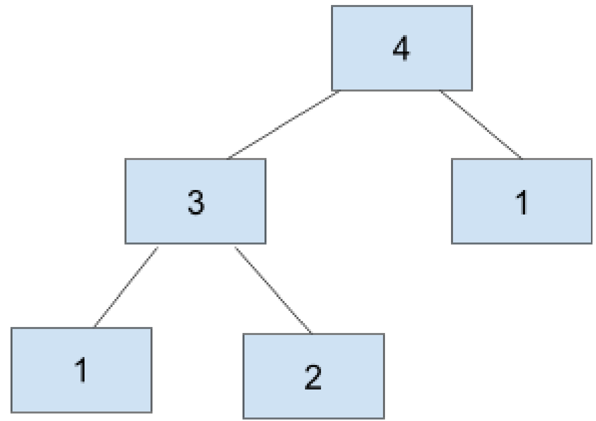
The frequencies are subsequently sorted in ascending order, and then stored in a data structure called a “Priority queue”.

Letter	D	C	A	B
Frequency	1	2	3	4

Following this, an empty node is created. The two lowest values are assigned to the left and right of the parent node respectively as leaf nodes. The parent node is assigned the sum of the two leaf nodes.



The process is then repeated with all the characters in the string.



In the event that a tie was to occur between any of the leaf nodes, the one with higher weight is kept over the other.

Shannon-Fano Coding

Shannon-Fano Coding is a form of lossless compression used for various file formats, and was created in 1949 by Claude Elwood Shannon and Robert Fano. Fundamentally, replaces all the characters in a string with binary code, the length of which is based on how frequently each character occurs in the string. It has several differences when compared to Huffman Coding, with examples of this being that rather than creating the tree from the bottom to the top, the tree is created from the top down. The pseudocode is shown below:

```
 1:  begin
 2:     count source units
 3:     sort source units to non-decreasing order
 4:     SF-SplitS
 5:     output(count of symbols, encoded tree, symbols)
 6:     write output
 7:   end
 8:
 9:  procedure SF-Split(S)
10:  begin
11:     if (|S|>1) then
12:      begin
13:        divide S to S1 and S2 with about same count of units
14:        add 1 to codes in S1
15:        add 0 to codes in S2
16:        SF-Split(S1)
17:        SF-Split(S2)
18:      end
19:  end
```

Figure 4. Pseudocode for Shannon-Fano Coding (Ahuja)

The way this algorithm works is it first creates a list of probabilities, or frequency in order to represent the number of times the characters in the string (for example) occur. This is done in order to determine the relative frequency of occurrence for each character (Lamorahan et al.). This is shown in the diagram below:

SYMBOL	A	B	C	D	E
PROBABILITY OR FRQUENCY	0.22	0.28	0.15	0.30	0.05

THE SYMBOLS AND THEIR PROBABILITY / FREQUENCY ARE TAKEN AS INPUTS.
(In case of Frequency, the values can be any number)

Figure 5. Table displaying Shannon-Fano Probability (Ahuja)

The probability of each character is then sorted by descending order as can be observed in the diagram below:

SYMBOL	D	B	A	C	E
PROBABILITY OR FRQUENCY	0.30	0.28	0.22	0.15	0.05

Figure 6. Probability Table Splitting (Ahuja)

The list is then split in halves, ideally having the total probabilities of both halves being relatively close to each other. The value “0” is then assigned to the left half, and “1” is assigned to the right half. This is then repeated until all characters are in sub groups (Ahuja).

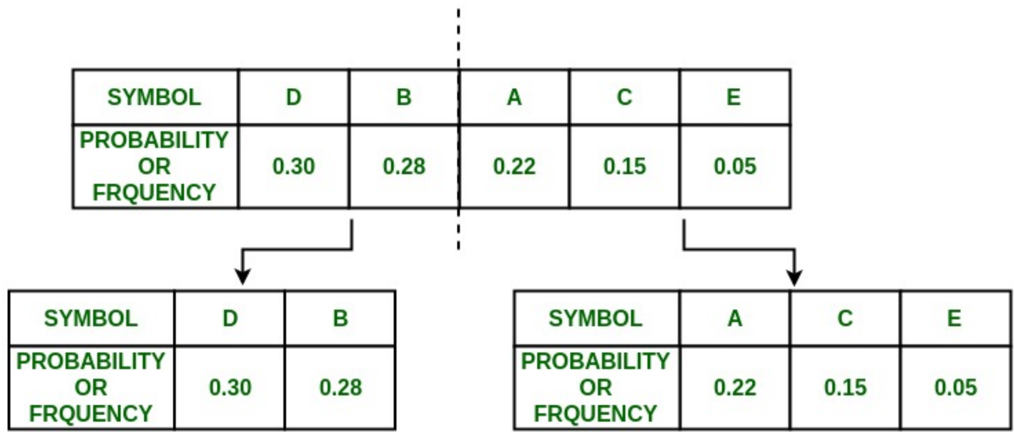


Figure 7. Shannon-Fano Probability Tree (GeeksforGeeks)

A tree is then created, with the condition that the character on the left side is given a value of 0 and the character on the right is given a value of 1 (Lamorahan et al.).

The third and fourth step of the process are then implemented recursively to both halves of the probability/frequency table (Lamorahan et al.).

Comparing the two algorithms

There are several key differences between Huffman coding and Shannon-Fano coding. Huffman coding is often considered more “optimal” than Shannon-Fano coding (TechDifferences). This is due to the fact that Huffman coding is based on a prefixed value, whereas Shannon-Fano coding makes use of a cumulative distribution function. The reason Huffman coding is considered more optimal is due to the fact that Shannon-Fano coding does not always manage to get the smallest file size possible, as a result of the way its binary search tree is made, whereas Huffman coding succeeds with this with a higher frequency (OpenGenus).

A summary is shown below.

BASIS FOR COMPARISON	HUFFMAN CODING	SHANNON FANO CODING
Basic	Based on source symbol probabilities	Based on the cumulative distribution function
Efficiency	Better	Moderate
Developed by	David Huffman	Claude Shannon and Robert Fano
Invented in the year	1952	1949
Optimization provided	High	Low

Figure 8. Algorithm Comparison Table (Tech Differences)

These algorithms have key differences when it comes to their mathematics; they follow similar principles, with slight differences, and it is these slight differences that cause them to largely different results when it comes to the compression of files.

Theoretically, due to Huffman Coding's use of prefix codes rather than the cumulative distribution function, it should theoretically outperform the Shannon-Fano algorithm.

Methodology

In order to test the compression ratio of these two algorithms, A program for both Huffman coding and Shannon-Fano coding has been written; The Huffman one in Python, and the Shannon-Fano one in Go. The investigation data will also be collected from these programs. They will both take the exact same text files as input and compress the file, outputting a compressed version of the file. The compression ratio will then be calculated, and this process will be repeated with different input files.

The files to be tested would be a sample text file containing random text, the entire Bee Movie script, and lastly the script to the movie "Shrek"; The reason I have chosen these three text files is due to the fact that the first sample text contains repeated phrases and the bee movie script contains lots of varied characters within. As aforementioned, these files will be put into the compression algorithms, and a compressed version of them will be outputted. Following this, I will calculate the compression ratio for each of the files, comparing the original file size with the new compressed size of the file.

In the analysis of results, I will then calculate the mean of the ratios, and find the percentage of itself the file gets compressed to.

My hypothesis is that the Huffman Coding algorithm should have a better compression ratio due to the above comparison, and due to the files not having that much repetition within them, barring the “Crazy? I was crazy once” copypasta text file; The copypasta text file is also being tested due to the fact that the sheer amount of repetition within that file should have an effect on the experiment.

The Programs

Two programs have been written; one in python and the other in go, in order to conduct the experiment, and are listed in the appendix. The idea is that these programs will take in the input files and run them through their respective algorithms. The programs are written so that they should output compressed versions of the input files. I will then be able to compare the compressed file sizes to the original ones and calculate the compression ratio each time. I will then be able to find the average compression ratio for each of the algorithms, and evaluate which algorithm is more effective based on the comparison. After undergoing compression, I should have two files like shown below.

the-full-bee-movie-script.bin
the-full-bee-movie-script

The Huffman Coding algorithm utilizes 2 different functions; the first one containing the actual compression algorithm itself, and the second program being used to assign the test file to the function.

The Shannon-Fano program works in a similar manner, in that it also outputs a compressed version of the input file. The compressor itself is split into 4 separate code files; one containing the actual compression function itself, another one containing the decompression function, which is redundant for my experiment, the root file, which executes the functions, and lastly, the util file, which assigns the new file to its directory.

Test Data

The text files will be tested using the two algorithms to test the difference in the compression ratio. I will first simply run the files through the compression algorithms and obtain their compressed sizes. After compiling these results into a table, I will then create a new table in order to calculate the compression ratio for each of the files.

Results + Analysis

For each of the files, I have calculated the average compression ratios and will be comparing these to each other in order to determine which algorithm is more efficient. The raw data of my experiment is shown below:

		Algorithm	
Sample	**Original File Size (KB)**	Huffman	Shannon-Fano
Sample Text	699	385	398
Bee Movie Script	49	29	30
“Crazy? I was crazy once” copypasta	52	27	28
Shrek Script	38	22	24

Figure 9. Raw Data Table

It can clearly be seen that in every single sample fed into these algorithms, the Huffman Coding algorithm performed better than the Shannon-Fano Algorithm every single time. However, In order to correctly evaluate this, and to see the extent to which the Huffman Coding Algorithm performed better, I would have to calculate the compression ratio of each one. This is done in the table below, with the values being calculated by simply comparing the uncompressed file size to the compressed file size. The values are shown in the processed data table below.

Sample	Compression ratio (Huffman)	Compression ratio (Shannon-Fano)
Sample Text	699:385	699:398
Bee Movie Script	49:29	49:30
“Crazy? I was crazy once” copypasta	52:27	52:28
Shrek Script	38:22	38:24
Average	196177:109620 Kilobytes ≈ 196:110 Megabytes = 98:55 Megabytes	570859:334320 Kilobytes ≈ 571:334 Megabytes

Figure 10. Processed Data Table

After calculating the compression ratio, I can now more confidently infer that the Huffman Coding algorithm performed better than the Shannon-Fano one, as we can see a significant difference in their respective compression ratios. It can be observed that For Huffman, on average the algorithm compresses the file to around 56.1% of its original size, while for the Shannon-Fano algorithm, the file gets compressed to around 58.5% of its original size.

While this may not seem like a very large difference on a surface level, the 2.4% difference ends up becoming quite significant when it comes to larger files than the ones used for the test cases. This is in line with my initial hypothesis about these algorithms, as according to my research, Huffman coding always produces optimal results; while Shannon-Fano does not due to its dependance on probability models.

This means that it would require data to follow certain types of pattern for the algorithm to be optimal, which is why Huffman coding is often preferred over it. Inefficient compression may arise from issues such as inefficient probability distributions, or when there is a very limited number of symbols within a file; due to the low number of occurrences for each symbol, it makes it difficult for the algorithm to produce an efficient probability model for the text file. Huffman, on the other hand, does not suffer from such an issue due to the fact that it makes use of the prefix code instead.

A factor that may have impacted the results is that the type of Huffman coding was not considered, as in whether the algorithm was adaptive or not; If one of the two in particular were looked into, it is possible that different conclusions may have been reached.

Conclusion

In conclusion, the answer to my research question, “Investigation into Huffman Coding and Shannon-Fano Coding: Which algorithm is more efficient?” can be sufficiently answered through the tests I have conducted for a general answer. However, if we were to go more in depth, there are specific cases for which Shannon-Fano coding would undoubtedly be more optimal than Huffman coding, due to the values within the files being favorable for the probability model generated by Shannon-Fano. While Huffman coding is a more widely used algorithm due to it being able to always generate the optimal result for itself, both algorithms have their merits, and which algorithm is objectively better depends highly on the use case; like aforementioned, Shannon-Fano would more effectively be able to produce its probability model with files containing many repeated characters within.

Compression ratio, however, is not the sole way to evaluate the effectiveness of a compression algorithm, however. Although Huffman coding had the better compression ratio in the tests I conducted, I did not deduce which algorithm had the better runtime, which is another factor used in determining the efficiency of an algorithm. That being said, the data collected throughout this essay could be useful for firms looking for more efficient data storage solutions and help in evaluating the benefits and drawbacks of both algorithms.

Limitations

Testing for anything other than the compression ratio for my case would not have not been possible with the programs I was running. Go is a much faster language than python, resulting in a test for runtime being unfair, as the Shannon-Fano algorithm would have completely swept every test. That being said however, real world users may prioritize the speed of compression over the actual compression ratio and it is therefore an important consideration when evaluating the efficiency of algorithms.

Using adaptive Shannon-Fano Coding could have also yielded different results; this is due to the fact that the adaptive versions of compression algorithms almost always outperform the arithmetic versions.

Works Cited

Ahuja, Shaleen. “Shannon-Fano Algorithm for Data Compression.” *GeeksforGeeks*, 5 Dec. 2018, www.geeksforgeeks.org/shannon-fano-algorithm-for-data-compression/.

Chung, Conrad. “The Basic Principles of Data Compression.” *Www.2brightsparks.com*, www.2brightsparks.com/resources/articles/data-compression.html.

Coding Coach. “Data Compression Explained: Lossless and Lossy Encoding.” *Www.youtube.com*, 30 Nov. 2020, www.youtube.com/watch?v=gmHoX_EqWF4.

Davies, Nahla. “How to Calculate Algorithm Efficiency - KDnuggets.” *KDnuggets*, 20 Sept. 2022, www.kdnuggets.com/2022/09/calculate-algorithm-efficiency.html.

Geekific. “Huffman Coding Algorithm Explained and Implemented in Java | Data Compression | Geekific.” *Www.youtube.com*, 30 Oct. 2021, www.youtube.com/watch?v=21_bJLB7gyU.

GeeksforGeeks. “Introduction to Tree - Data Structure and Algorithm Tutorials.” *GeeksforGeeks*, 21 June 2021, www.geeksforgeeks.org/introduction-to-tree-data-structure-and-algorithm-tutorials/.

Java Tutorials Point. “Binary Search Tree - Javatpoint.” *Www.javatpoint.com*, www.javatpoint.com/binary-search-tree.

Khalid Sayood. *Introduction to Data Compression*. San Francisco, Morgan Kaufmann Publishers, 2000.

Lamorahan, Christine, et al. “Data Compression Using Shannon-Fano Algorithm.” *D’CARTESIAN*, vol. 2, no. 2, 1 Oct. 2013, p. 10, https://doi.org/10.35799/dc.2.2.2013.3207. Accessed 12 July 2020.

Lelewer, Debra A., and Daniel S. Hirschberg. “Data Compression.” *ACM Computing Surveys*, vol. 19, no. 3, Sept. 1987, pp. 261–296, https://doi.org/10.1145/45072.45074.

McAnlis, Colt, and Aleks Haecky. *Understanding Compression : Data Compression for Modern Developers*. Sebastopol, CA, O’Reilly Media, 2016.

Moffat, Alistair. “Huffman Coding.” *ACM Comput. Surv*, June 2019, people.eng.unimelb.edu.au/ammoffat/abstracts/compsurv19moffat.pdf.

Programiz. “Huffman Coding Algorithm.” *Www.programiz.com*, www.programiz.com/dsa/huffman-coding.

Saunders, Michelle. “How File Compression Benefits Business.” *InStream*, 17 Feb. 2020, instreamllc.com/what-does-file-compression-accomplish/.

Sheremeta, Roman. “Archiver.” *GitHub*, 20 June 2022, github.com/RSheremeta/archiver.

Srivastava, Bhrigu. “Bhrigu123/Huffman-Coding.” *GitHub*, 12 Mar. 2021, github.com/bhrigu123/huffman-coding.

TechDifferences. “Difference between Huffman Coding and Shannon Fano Coding (with Comparison Chart).” *Tech Differences*, 12 Feb. 2019, techdifferences.com/difference-between-huffman-coding-and-shannon-fano-coding.html#:~:text=Key%20Differences%20Between%20Huffman%20Coding.

Wayner, Peter. “Compression Algorithm - an Overview | ScienceDirect Topics.” *Sciencedirect.com*, 2011, www.sciencedirect.com/topics/computer-science/compression-algorithm.

Wikipedia Contributors. “Shannon–Fano Coding.” *Wikipedia*, Wikimedia Foundation, 13 July 2021, en.wikipedia.org/wiki/Shannon%E2%80%93Fano_coding.

---. “Tree (Data Structure).” *Wikipedia*, Wikimedia Foundation, 20 Oct. 2019, en.wikipedia.org/wiki/Tree_(data_structure).

---. “Variable-Length Code - Wikipedia.” *En.wikipedia.org*, en.wikipedia.org/wiki/Variable-length_code.

Appendix

Huffman Coding algorithm

```
import heapq
import os

"""
author: Bhrigu Srivastava
website: https:bhrigu.me
"""

class HuffmanCoding:
	def __init__(self, path):
		self.path = path
		self.heap = []
		self.codes = {}
		self.reverse_mapping = {}

	class HeapNode:
		def __init__(self, char, freq):
			self.char = char
			self.freq = freq
			self.left = None
			self.right = None

		# defining comparators less_than and equals
		def __lt__(self, other):
			return self.freq < other.freq

		def __eq__(self, other):
			if(other == None):
				return False
			if(not isinstance(other, HeapNode)):
				return False
			return self.freq == other.freq

	# functions for compression:

	def make_frequency_dict(self, text):
		frequency = {}
		for character in text:
			if not character in frequency:
				frequency[character] = 0
			frequency[character] += 1
		return frequency

	def make_heap(self, frequency):
		for key in frequency:
			node = self.HeapNode(key, frequency[key])
			heapq.heappush(self.heap, node)

	def merge_nodes(self):
		while(len(self.heap)>1):
			node1 = heapq.heappop(self.heap)
			node2 = heapq.heappop(self.heap)

			merged = self.HeapNode(None, node1.freq + node2.freq)
			merged.left = node1
			merged.right = node2

			heapq.heappush(self.heap, merged)
```

```
def make_codes_helper(self, root, current_code):
        if(root == None):
                return

        if(root.char != None):
                self.codes[root.char] = current_code
                self.reverse_mapping[current_code] = root.char
                return

        self.make_codes_helper(root.left, current_code + "0")
        self.make_codes_helper(root.right, current_code + "1")


def make_codes(self):
        root = heapq.heappop(self.heap)
        current_code = ""
        self.make_codes_helper(root, current_code)


def get_encoded_text(self, text):
        encoded_text = ""
        for character in text:
                encoded_text += self.codes[character]
        return encoded_text


def pad_encoded_text(self, encoded_text):
        extra_padding = 8 - len(encoded_text) % 8
        for i in range(extra_padding):
                encoded_text += "0"

        padded_info = "{0:08b}".format(extra_padding)
        encoded_text = padded_info + encoded_text
        return encoded_text


def get_byte_array(self, padded_encoded_text):
        if(len(padded_encoded_text) % 8 != 0):
                print("Encoded text not padded properly")
                exit(0)

        b = bytearray()
        for i in range(0, len(padded_encoded_text), 8):
                byte = padded_encoded_text[i:i+8]
                b.append(int(byte, 2))
        return b


def compress(self):
        filename, file_extension = os.path.splitext(self.path)
        output_path = filename + ".bin"

        with open(self.path, 'r+') as file, open(output_path, 'wb') as output:
                text = file.read()
                text = text.rstrip()

                frequency = self.make_frequency_dict(text)
                self.make_heap(frequency)
                self.merge_nodes()
                self.make_codes()
```

```
                encoded_text = self.get_encoded_text(text)
                padded_encoded_text = self.pad_encoded_text(encoded_text)

                b = self.get_byte_array(padded_encoded_text)
                output.write(bytes(b))

        print("Compressed")
        return output_path


""" functions for decompression: """


def remove_padding(self, padded_encoded_text):
        padded_info = padded_encoded_text[:8]
        extra_padding = int(padded_info, 2)

        padded_encoded_text = padded_encoded_text[8:]
        encoded_text = padded_encoded_text[:-1*extra_padding]

        return encoded_text

def decode_text(self, encoded_text):
        current_code = ""
        decoded_text = ""

        for bit in encoded_text:
                current_code += bit
                if(current_code in self.reverse_mapping):
                        character = self.reverse_mapping[current_code]
                        decoded_text += character
                        current_code = ""

        return decoded_text


def decompress(self, input_path):
        filename, file_extension = os.path.splitext(self.path)
        output_path = filename + "_decompressed" + ".txt"

        with open(input_path, 'rb') as file, open(output_path, 'w') as output:
                bit_string = ""

                byte = file.read(1)
                while(len(byte) > 0):
                        byte = ord(byte)
                        bits = bin(byte)[2:].rjust(8, '0')
                        bit_string += bits
                        byte = file.read(1)

                encoded_text = self.remove_padding(bit_string)

                decompressed_text = self.decode_text(encoded_text)

                output.write(decompressed_text)

        print("Decompressed")
        return output_path
```

```
from huffman import HuffmanCoding
import sys

path = "shrek_script .txt"

h = HuffmanCoding(path)

output_path = h.compress()
print("Compressed file path: " + output_path)

decom_path = h.decompress(output_path)
print("Decompressed file path: " + decom_path)
```

Shannon-Fano Algorithm:

```go
package cmd

import (
	"fmt"
	"github.com/RSheremeta/archiver/pkg/compression"
	"github.com/RSheremeta/archiver/pkg/compression/table/shannon_fano"
	"github.com/spf13/cobra"
	"io"
	"os"
)

var packCmd = &cobra.Command{
	Use:   "pack",
	Short: "Compress target file",
	Run:   pack,
}

func init() {
	rootCmd.AddCommand(packCmd)

	packCmd.Flags().StringP(method, methodShort, "",
		fmt.Sprintf("compression method available values: %q, %q", actionMethodShort, actionMethodFull))

	if err := packCmd.MarkFlagRequired(method); err != nil {
		fmt.Printf("Error: Flag --%q (or -%q) is required\n", method, methodShort)
		panic(err)
	}
}

// pack is intended to compress raw file
// available flags and opts:
// "--method", "-m":
// values: "sf", "shannon-fano"
// supported raw file extensions: "rtf", "txt"
func pack(cmd *cobra.Command, args []string) {
	fmt.Println("Start compressing your file...")

	var encoder compression.Encoder

	if len(args) == 0 || args[0] == "" {
		panic(errEmptyPath)
	}

	methodVal := cmd.Flag(method).Value.String()

	switch methodVal {
	case actionMethodShort, actionMethodFull:
		encoder = compression.New(shannon_fano.NewGenerator())
```

```
49      default:
50          cmd.PrintErrf("Error: unknown method, cannot recognize %q", methodVal)
51      }
52
53      filePath := args[0]
54
55      file, err := os.Open(filePath)
56      if err != nil {
57          fmt.Println("Error while opening target file:", file.Name())
58          panic(err)
59      }
60      defer file.Close()
61
62      data, err := io.ReadAll(file)
63      if err != nil {
64          fmt.Println("Error while reading target file:", file.Name())
65          panic(err)
66      }
67
68      packed := encoder.Encode(string(data))
69
70      err = os.WriteFile(packedFileName(filePath), packed, 0644)
71      if err != nil {
72          fmt.Println("Error while creating a result file")
73          panic(err)
74      }
75  }
76
```

```go
package cmd

import (
	"fmt"
	"github.com/spf13/cobra"
	"os"
)

var rootCmd = &cobra.Command{
	Short: "Tiny Archiver for compression/decompression files",
}

// Execute - main command invoked whenever executing the built binary.
// eg - "./archiver" in Terminal
func Execute() {
	if err := rootCmd.Execute(); err != nil {
		extCode, _ := fmt.Fprintln(os.Stderr, "ARCHIVER ERROR: ", err)
		os.Exit(extCode)
	}
}
```

```go
package cmd

import (
	"fmt"
	"github.com/RSheremeta/archiver/pkg/compression"
	"github.com/RSheremeta/archiver/pkg/compression/table/shannon_fano"
	"github.com/spf13/cobra"
	"io"
	"os"
)

var unpackCmd = &cobra.Command{
	Use:   "unpack",
	Short: "Decompress target file",
	Run:   unpack,
}

func init() {
	rootCmd.AddCommand(unpackCmd)

	unpackCmd.Flags().StringP(method, methodShort, "",
		fmt.Sprintf("decompression method available values: %q, %q", actionMethodShort, actionMethodFull))

	unpackCmd.Flags().StringP(extension, extensionShort, "",
		fmt.Sprintf("desired extension of the decomressed file. %q by default", ".txt"))

	if err := unpackCmd.MarkFlagRequired(method); err != nil {
		fmt.Printf("Error: Flag --%q (or -%q) is required\n", method, methodShort)
		panic(err)
	}
}

// unpack is intended to decompress compressed .sf file
// available flags and opts:
// "--method", "-m":
// values: "sf", "shannon-fano"
// "--extension", "-e":
// values: any "txt", "rtf" - "txt" by default
func unpack(cmd *cobra.Command, args []string) {
	fmt.Println("Start decompressing your file...")

	var decoder compression.Decoder

	if len(args) == 0 || args[0] == "" {
		panic(errEmptyPath)
	}

	method := cmd.Flag(method).Value.String()
```

```
49      switch method {
50      case actionMethodShort, actionMethodFull:
51          decoder = compression.New(shannon_fano.NewGenerator())
52      }
53
54      fileExtension := cmd.Flag(extension).Value.String()
55      if fileExtension != "" {
56          unpackedExtension = fileExtension
57      } else {
58          unpackedExtension = defaultUnpackedExtension
59      }
60
61      filePath := args[0]
62
63      file, err := os.Open(filePath)
64      if err != nil {
65          fmt.Println("Error while opening target file:", file.Name())
66          panic(err)
67      }
68      defer file.Close()
69
70      data, err := io.ReadAll(file)
71      if err != nil {
72          fmt.Println("Error while reading target file:", file.Name())
73          panic(err)
74      }
75
76      packed := decoder.Decode(data)
77
78      err = os.WriteFile(unpackedFileName(filePath), []byte(packed), 0644)
79      if err != nil {
80          fmt.Println("Error while creating a result file")
81          panic(err)
82      }
83
84  }
85
```

```go
package cmd

import (
	"errors"
	"fmt"
	"path/filepath"
	"strings"
)

var errEmptyPath = errors.New("ERROR: target file path is not specified")

var unpackedExtension string

const (
	actionMethodShort        = "sf"
	actionMethodFull         = "shannon-fano"
	packedExtension          = "sf"
	defaultUnpackedExtension = "txt"
	method                   = "method"
	methodShort              = "m"
	extension                = "extension"
	extensionShort           = "e"
)

func packedFileName(path string) string {
	return filename(path, packedExtension)
}

func unpackedFileName(path string) string {
	return filename(path, unpackedExtension)
}

func filename(path, ext string) string {
	fileName := filepath.Base(path)
	fileExt := filepath.Ext(fileName)
	baseName := strings.TrimSuffix(fileName, fileExt)

	return fmt.Sprintf("%v.%v", baseName, ext)
}

```

Raw Data:

		Algorithm	
Sample	**Original File Size (KB)**	Huffman	Shannon-Fano
Sample Text	699	385	398
Bee Movie Script	49	29	30
"Crazy? I was crazy once" copypasta	52	27	28
Shrek Script	38	22	24

Processed Data:

Sample	**Compression ratio (Huffman)**	**Compression ratio (Shannon-Fano)**
Sample Text	699:385	699:398
Bee Movie Script	49:29	49:30
"Crazy? I was crazy once" copypasta	52:27	52:28
Shrek Script	38:22	38:24
Average	196177:109620 Kilobytes ≈ 196:110 Megabytes = 98:55 Megabytes	570859:334320 Kilobytes ≈ 571:334 Megabytes