Data Compression

Enrico Marchionni
enrico.marchionni@studio.unibo.it

December 27, 2024

Abstract

Data compression is intended as the practice of reducing the size of binary digital data. It could be considered as a procedure that takes a bit-stream in input and returns another bit-stream as output. The output stream may be of equal length or shorter than the input.

The key to understand data compression is to discuss the distinction between data and information. It can be said that data is how information is represented[1]. In simple terms, data can be compressed because its original representation is not the shortest possible. The goal of data compression is to reduce data by maintaining the same information.

The counterpart is that in our time data is intrinsically redundant. And this redundancy is needed. So data compression isn't only a procedure that goes from a bit-stream to another one not longer, but it requires also another procedure that regenerates the original bit-stream of data, necessary for practical use, from the previously given output bit-stream of information.

This makes clear that the task of compression consists of two components, an *encoding* algorithm that takes a message and generates a "compressed" representation (hopefully with fewer bits), and a *decoding* algorithm that reconstructs the original message or some approximation of it from the compressed representation.

...

[1] ex. the number 0 can be expressed in binary as a sequence of a certain number of zeros, from 1 to ∞, and we know that calculators use at least 8 bits, let's say n (considering it as a multiple of 8), to represent an integer number. So at the end $n - 1$ bits are redundant in the 0 representation on a calculator.

Contents

1 Information Theory **2**
1.1 Quantifying Information . . . 2

2 Entropy **4**
2.1 Quantifying Entropy . . . 4

3 Randomness **6**
3.1 Kolmogorov . . . 6
3.2 Martin-Lof . . . 6
3.3 Random Sequences . . . 7

4 Data Compression **8**
4.1 Techniques . . . 8
4.1.1 Modeling . . . 8
4.1.2 Coding . . . 9
4.1.3 Implementations . . . 9
4.2 Comparison of Algorithms . . . 19
4.2.1 Compression Ratio . . . 19
4.2.2 Compression Speed . . . 19
4.2.3 Decompression Speed . . . 19
4.2.4 CPU usage . . . 19
4.2.5 Memory usage . . . 20
4.2.6 Time . . . 20

5 Compression Tools **21**
5.1 Archiving . . . 21
5.1.1 Unix . . . 21
5.1.2 Windows . . . 21
5.1.3 Mac OS X . . . 22
5.2 Compression . . . 22
5.2.1 lz4 . . . 22
5.2.2 compress . . . 22
5.2.3 gzip . . . 22
5.2.4 bzip2 . . . 22
5.2.5 bzip3 . . . 22
5.2.6 lzma . . . 22
5.2.7 xz . . . 23
5.2.8 zstd . . . 23

Chapter 1

Information Theory

In 1948, Shannon[1], while working at the Bell Telephone Laboratories, published "A Mathematical Theory of Communication" [Sha48], a seminal paper that marked the birth of information theory. In that paper, Shannon defined the concept of "information" and proposed a precise way to quantify it-in his theory, the fundamental unit of information is the bit.

Moreover, this discipline plays behind the concepts of entropy, randomness and data compression, all topics that will be discussed later on.

1.1 Quantifying Information

For what concerns data compression, information of theory has developed a usable measure of the information we get from observing the occurrence of an event having probability p. Therefore information is defined in terms of the probability.

The information measure $I(p)$ has to match the following axioms (from [Car07]):

- Information is non negative: $I(p) \geq 0$.
- If an event has probability 1, we get no information: $I(1) = 0$.
- If two independent events occur (whose probability is the product of their individual probabilities), then the information we get from observing the events is the sum of the two computed individually: $I(p_1 \cdot p_2) = I(p_1) + I(p_2)$.
- Information measure must be continuous and monotonic (slight changes in probability should result in slight changes in information).

Considering the previous properties as axioms it can be said that: $I(p^2) = I(p \cdot p) = I(p) + I(p) = 2 \cdot I(p)$. Thus: $I(p^n) = n \cdot I(p)$ (by induction). Then: $I(p) = I(p^{(\frac{1}{m})^m}) = m \cdot I(p^{\frac{1}{m}})$, so $I(p^{\frac{1}{m}}) = \frac{1}{m} \cdot I(P)$, therefore: $I(p^{\frac{n}{m}}) = \frac{n}{m} \cdot I(p)$. In general, considering r as a real number: $I(p^a) = a \cdot I(p)$.

From this analysis it was discovered that:

$$I(p) = -\log_b p \ (= \log_b \frac{1}{p}) \tag{1.1}$$

[1] Claude Elwood Shannon (1916-2001) was an American mathematician, electrical engineer, computer scientist, cryptographer and inventor known as the "father of information theory".

Where: $p = b_1^{\log_{b_1} p}$ and therefore: $\log_{b_2} p = \log_{b_2} b_1^{\log_{b_1} p} = \log_{b_2} b_1 \cdot \log_{b_1} p$. So: $\log_{b_2} b_1$ is a constant, a scaling factor. From another point of view it is a simple change in the unit of measurement.

For this reason:

$$I(p) = -\log_2 p \tag{1.2}$$

Equation 1.2 is the same expression of Equation 1.1 where the unit of measurement is called bits (look at Table 1.1). Equation 1.1 was first introduced by Hartley[2] in 1928 trying to measure uncertainty, without talking about probability, and lately reviewed by Shannon.

Unit of measurement	Base
bit (or shannon)	2
trit	3
nat (natural unit of information)	e
hartley (or dit)	10

Table 1.1: Information units of measurement

Example 1. *Let's talk about flipping a fair coin n times. It gives us:* $-\log_2 \frac{1}{2}^n = \log_2 2^n = n \cdot \log_2 2 = n$ *bits of information. In fact a sequence of heads (coded as* 1*) and tails (coded as* 0*) could be expressed as:* $010010111\ldots$*, these are the* n *bits of information.*

[2]Ralph Vinton Lyon Hartley (1888-1970) was an American electronics researcher. He invented the Hartley oscillator and the Hartley transform, and contributed to the foundations of information theory.

Chapter 2

Entropy

Entropy is a concept that was explained in many fields. Previously defined by Clausius[1] and Boltzmann[2] was later used by Shannon. It is believed that these three definitions are indeed equivalent although no formal proof of this is available (as discussed in [Ben19]). Informally it is advantageous to remember that in statistical physics entropy represents the randomness or disorder of a system.

2.1 Quantifying Entropy

Here is how Shannon introduced the measure of Information:

> Suppose we have a set of possible events whose probabilities of occurrence are $p_1, p_2, \ldots, p_n$. These probabilities are known but that is all we know concerning which event will occur. Can we find a measure of how much "choice" is involved in the selection of the event or how uncertain we are of the outcome?

If there is such a measure, say, $H(p_1, p_2, \ldots, p_n)$[3], it is reasonable to require of it the following properties:

- H should be continuous in the p_i.
- If all the p_i are equal, $p_i = \frac{1}{n}$ then H should be a monotonic increasing function of n. With equally likely events there is more choice, or uncertainty, when there are more possible events.
- If a choice be broken down into two successive choices, the original H should be the weighted sum of the individual values of H (The entropy of an entire stream is simply the sum of the entropy of all individual symbols).

Then Shannon proved that the only H satisfying the three assumptions above has the form:

$$H = -K \sum_{i=1}^{n} p_i \ln p_i \tag{2.1}$$

[1] Rudolf Julius Emanuel Clausius (1822-1888) was a German physicist and mathematician and is considered one of the central founding fathers of the science of thermodynamics.

[2] Ludwig Eduard Boltzmann (1844-1906) was an Austrian physicist and philosopher. His greatest achievements were the development of statistical mechanics and the statistical explanation of the second law of thermodynamics.

[3] Where H refers to Hartley.

Equation 2.1 includes a constant K, in the Shannon article it is any constant. In application to thermodynamics K turns into Boltzmann Constant. It is simply a scaling factor. Note that if K is $\frac{1}{\ln b}$ or equivalently $\log_b e$, the formula, considering only K and the logarithm, becomes $\log_b e \cdot \ln p$ that is the same of $\log_b e^{\ln p}$ that can be simply written as $\log_b p$. So:

$$H(P) = -\sum_{i=1}^{n} p_i \log_2 p_i \tag{2.2}$$

Where $P = p_1, p_2, \ldots, p_n$ is the distribution of probability considered. remind that in Equation 2.2 base 2 could be a general base b and it can be simply view as a simple change in the unit of measurement (as it was seen in Table 1.1).

An intuitive way to explain the origin of this formula is now discussed. We want to obtain the average amount of information from each symbol we see in a stream. Let's suppose we start from n symbols $a_1, a_1, \ldots, a_n$. A stream of these symbols is provided with probabilities $p_1, p_1, \ldots, p_n$ respectively. As it was seen in Equation 1.2 for a symbol a_i we get $-\log_2 p_i$ information. In a long run, say N observations, we will see (approximately) $N \cdot p_i$ occurrences of the symbol a_i. Thus in the N independent observations, we will get total information of:

$$I = -\sum_{i=1}^{n} (N \cdot p_i) \log_2 p_i \tag{2.3}$$

So then, from Equation 2.3 the average information is:

$$\frac{I}{N} = -\frac{1}{N} \sum_{i=1}^{n} (N \cdot p_i) \log_2 p_i = -\sum_{i=1}^{n} p_i \log_2 p_i \tag{2.4}$$

This makes clear that entropy is a probability weighted average of the self information of each message. At this point we get Equation 2.4 that is the same as Equation 2.2. Furthermore, it is shown in Equation 2.5 that $H(P)$ is bounded (for further information see [Car07]):

$$0 \leq H(P) \leq \log_2 n \tag{2.5}$$

Larger entropies represent larger average information, and perhaps counter-intuitively, the more random a set of messages (the more even the probabilities) the more information they contain on average.

Example 2. *Returning to the example of the coin:*

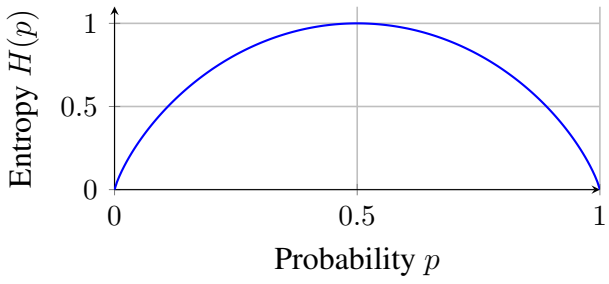

Figure 2.1: Graph of entropy $H(p) = -p \log_2(p) - (1-p) \log_2(1-p)$ for a fair coin toss.

Figure 2.1 shows an example of the entropy in function of the probability of heads or tails when flipping a fair coin.

Chapter 3

Randomness

Compression, logically, can be interpreted as the removal of redundancy. The compressed data therefore has no structure and cannot be distinguished from random data; in fact, it is random ([Sal07]).

3.1 Kolmogorov

Definition 3.1.1. ***Kolmogorov complexity*** *of a binary sequence is the length of the shortest binary program that generates that sequence on a universal Turing machine ([Kol68]).*

The concept essentially asserts that a binary sequence is considered random if there is no algorithm shorter than the sequence itself that can generate it.

That is related to the concept of incompressibility in algorithmic randomness: a random sequence cannot be compressed into a shorter representation than its original size.

So, loosely speaking, the randomness (or Kolmogorov complexity) of a finite sequence is equal to its shortest description.

It is known that the Kolmogorov complexity is not computable.

3.2 Martin-Lof

Martin-Lof in Algorithmic Randomness and Complexity ([Mar66]) shows that the random elements as defined by Kolmogorov possess all conceivable statistical properties of randomness.

He also extended the definition for random elements with three approaches to the definition of algorithmic randomness for infinite sequences:

Definition 3.2.1. *The* ***computational paradigm****: Random sequences are those whose initial segments are all hard to describe, or, equivalently, hard to compress.*

Definition 3.2.2. *The* ***measure-theoretic paradigm****: Random sequences are those with no "effectively rare" properties. If the class of sequences satisfying a given property is an effectively null set, then a random sequence should not have this property. This approach is the same as the stochastic paradigm: a random sequence should pass all effective statistical tests.*

Definition 3.2.3. *The* ***unpredictability paradigm****: This approach stems from what is probably the most intuitive conception of randomness, namely that one should not be able to predict the next bit of a random sequence, even if one knows all preceding bits, in the same way that a coin toss is unpredictable even given the results of previous coin tosses.*

Taken from [DH10].

The previous citations aim to observe that the idea of Kolmogorov that random generators didn't exist was lately reviewed and while remaining true, extended to infinite sequences.

3.3 Random Sequences

A random sequence should satisfy three conditions:

- The sequence follows a uniform distribution.
- Each element of the sequence is independent of each other.
- The rest of the sequence can not be predicted from any sequence.

Random numbers can be divided into two categories: true random numbers and pseudo-random numbers. True random number generators (RNGs) are composed of two parts: entropy source and algorithm post-processing. Pseudo random number generators (PRNGs) take a seed as input and generate an output sequence by function.

Chapter 4

Data Compression

When discussing compression algorithms it is important to make a distinction between two components: the **model** and the **coder**. The model component somehow captures the probability distribution of the symbols (or messages) by knowing or discovering something about the structure of the input. The coder component then takes advantage of the probability knowledge acquired in the model to generate codes to represent the symbols (or messages). It does this by effectively lengthening low probability messages and shortening high-probability messages. It should be pointed out that the line between model and coder components of algorithms is not always well define, on the other hand modeling and coding are two distinctly different things.

4.1 Techniques

The compression techniques can be: *lossless* and *lossy*. The first category is the most similar to the theory and the most practical intuition of it. Lossless compression ratios are generally in the range of 2:1 to 8:1. Lossy compression, in contrast, works on the assumption that the data doesn't have to be stored perfectly. Most information can be simply thrown away; the data will still be of acceptable quality. This technique is frequently used in image and video compression, where a group of pixels can be approximated into a single value. In this case the ratio can be of orders greater. In conclusion lossless compression has a lower ratio it preserves all information that can be reload back to the original data, on the other hand, lossy compression has a bigger ratio but it discards some information and doesn't generate the same data when it is reloaded.

4.1.1 Modeling

It can be statistical or dictionary-based. Statistical modeling reads in and encodes a single symbol at a time using the probability of that character's appearance. Dictionary-based modeling uses a single code to replace strings of symbols. This techniques can be static or adaptive. In the second case the problem of knowing the generated structure, the table of probabilities or the dictionary, both when encoding and when decoding, is avoided.

Statistical Modeling

It uses a static table of probabilities. At the beginning the model was created only one time and reused for different data. The next enhancement was to build a statistics table for every unique input stream. The number of symbols (usually characters) is called order of the model.

In particular order-N indicates that are considered N characters before the current one. The order is important because it determines the size of the table. The higher the order, the larger the table. However, as the order increases, the algorithm's efficiency ratio also improves (up to the point where the overhead of reading these tables becomes significant). An alternative approach is to use adaptive models, that generates statistics while reading data.

Dictionary Modeling

It follows a different approach. It reads in input data and looks for groups of symbols that appear in a dictionary. It uses pointers to refer to groups of symbols at some point in the dictionary. The longer the match, the better the compression ratio. In this case the modeling is the core of the compression algorithm. Also in this case the dictionary can be static or adaptive.

4.1.2 Coding

It generate codes to represent symbols or groups of symbols.

4.1.3 Implementations

For details on these implementations and their examples, see [NG95].

Morse Code (1837)

It was based on the fact that shorter code should be assigned to the most frequent symbols used in English communication.

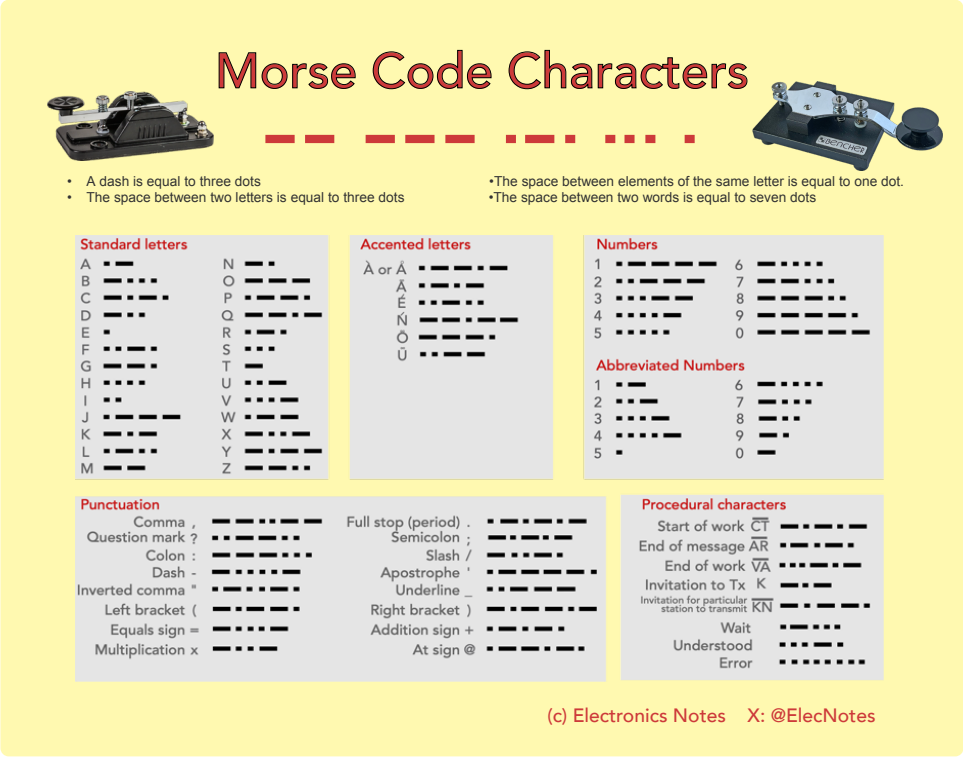

Figure 4.1: Table of the various Morse code characters

In Figure 4.1 it can be observed that the letter "e" which is the most commonly used letter in the English language is assigned the symbol of a single dot. The difference in the length of encoding different symbols can be considered as a need that explains why data compression is important. It takes the name from Morse[1], a co-developer, that in 1837 helped to develop the commercial use of telegraphy. For further information see `https://www.electronics-notes.com/articles/summary-infographics/morse-code/morse-code-characters-infographic.php`.

Shannon-Fano coding (1950)

Shannon at Bell Labs and Fano[2] at MIT developed this method nearly simultaneously[3]. It is necessary to know the probability of each symbol's appearance in a message. A table of codes could be constructed that has several important properties:

- Different codes have different numbers of bits.
- Codes for symbols with low probabilities have more bits, and codes for symbols with high probabilities have fewer bits.
- Though the codes are of different bit lengths, they can be uniquely decoded.

Arranging the codes as a binary tree solves the problem of decoding these variable-length codes.

The actual algorithm is:

1. For a given list of symbols, develop a corresponding list of probabilities or frequency counts so that each symbol's relative frequency of occurrence is known.
2. Sort the lists of symbols according to frequency, with the most frequently occurring symbols at the top and the least common at the bottom.
3. Divide the list into two parts, with the total frequency counts of the upper half being as close to the total of the bottom half as possible.
4. The upper half of the list is assigned the binary digit 0, and the lower half is assigned the digit 1. This means that the codes for the symbols in the first half will all start with 0, and the codes in the second half will all start with 1.
5. Recursively apply the steps 3 and 4 to each of the two halves, subdividing groups and adding bits to the codes until each symbol has become a corresponding code leaf on the tree.

[1]Samuel Finley Breese Morse (1791-1872) was an American inventor and painter. After establishing his reputation as a portrait painter, Morse, in his middle age, contributed to the invention of a single-wire telegraph system based on European telegraphs. He was a co-developer of Morse code in 1837 and helped to develop the commercial use of telegraphy.

[2]Roberto Mario "Robert" Fano (1917-2016) was an Italian-American computer scientist and professor of electrical engineering and computer science at the Massachusetts Institute of Technology.

[3]This method was proposed in Shannon's "A Mathematical Theory of Communication" (1948) and in a later technical report by Fano (1949).

Example 3. *In the following tables the execution steps of the algorithm are explained in a specific example (taken from [NG95]).*
The input data are:

Symbol	Count
A	15
B	7
C	6
D	6
E	5

Table 4.1: Input data.

Table 4.1 shows data of a widespread example about minimum redundancy coding[4].
The result of the first step is:

Symbol	Count	
A	15	0
B	7	0
C	6	1
D	6	1
E	5	1

Table 4.2: Shannon-Fano output of the first step.

Table 4.2 shows with a red line the split of the first step.
The result of the other steps is:

Symbol	Count			
A	15	0	0	
B	7	0	1	
C	6	1	0	
D	6	1	1	0
E	5	1	1	1

Table 4.3: Shannon-Fano final output.

Table 4.3 shows with horizontal red lines the Shannon-Fano steps by showing the splits.
Another view of the structure is:

[4] It is a code constructed in such a way that the average number of coding digits per message is minimized.

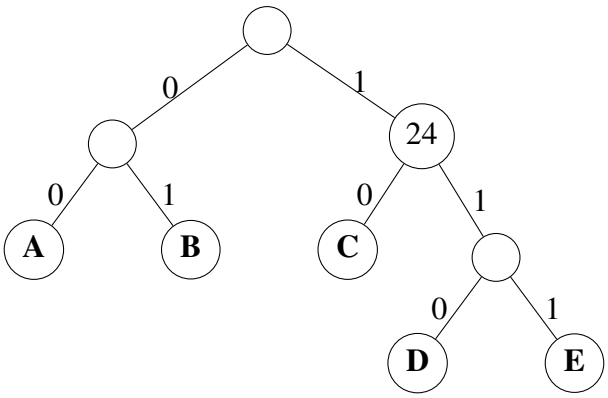

Figure 4.2: Shannon-Fano tree.

Figure 4.2 shows the Shannon-Fano binary tree.

The symbols with higher probability of occurrence (higher count) have fewer bits in their code.

The given results can be compared with some computations about the information theory. In the following table .

Symbol	Count	Information	Number of Bits	SF Size	SF Bits
A	15	1.38	20.68	2	30
B	7	2.48	17.35	2	14
C	6	2.70	16.20	2	12
D	6	2.70	16.20	3	18
E	5	2.96	14.82	3	15

Table 4.4: Information for each symbol.

Table 4.4 shows the amount of ***Information*** *that is computed with Equation 1.2, while the* ***Number of Bits*** *is computed with* $count \cdot Equation\ 1.2$

Huffman coding (1952)

It is similar to the Shannon-Fano coding. It was introduced two years later then the Shannon-Fano coding. It was first published in 1952 by Huffman[5] with a paper: "A Method for the Construction of Minimum Redundancy Codes". It is more efficient than the other one. The Shannon-Fano tree is built top-down, starting by assigning the most significant bits to each code and working down the tree until finished. Instead, Huffman codes are constructed bottom-up, beginning with the tree's leaves and progressing toward its root.

The tree is then built with the following steps:

- The two free nodes with the lowest weights are located.
- A parent node for these two nodes is created. It is assigned a weight equal to the sum of the two child nodes.

[5]David Albert Huffman (1925-1999) was an American pioneer in computer science, known for his Huffman coding.

- The parent node is added to the list of free nodes, and the two child nodes are removed from the list.
- One of the child nodes is designated as the path taken from the parent node when decoding a 0 bit. The other is arbitrarily set to the 1 bit.
- The previous steps are repeated until only one free node is left. This free node is designated the root of the tree.

Example 4. *Reconsidering the Table 4.1 input let's apply the Huffman coding algorithm. The result is shown:*

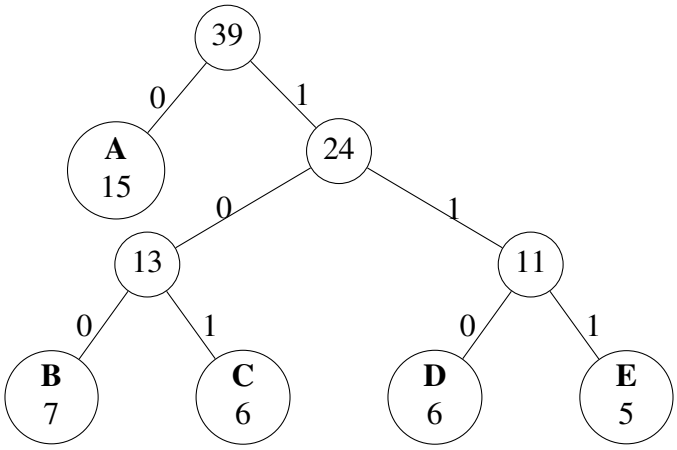

Figure 4.3: Huffman tree.

Figure 4.3 shows the Huffman binary tree, this can be compared with Figure 4.2.
The codes generated are:

Symbol	Code
A	0
B	100
C	101
D	110
E	111

Table 4.5: The Huffman codes table.

Table 4.5 shows them.
A comparison between this example for Shannon-Fano and Huffman codes follows:

Symbol	Count	Shannon-Fano Code Length	Shannon-Fano Bits	Huffman Code Length	Huffman Bits
A	15	2	30	1	15
B	7	2	14	3	21
C	6	2	12	3	18
D	6	3	18	3	18
E	5	3	15	3	15

Table 4.6: Information for each symbol.

Table 4.6 shows that Huffman is better than Shannon-Fano because it uses less bits. As a matter of fact, for a message with an information content of 85.25 bits, Shannon-Fano coding requires 89 bits, but Huffman coding requires only 87.

RLE (1960s)

Run-length encoding algorithm stores consecutive occurrences of the same data value as a single occurrence of that data value and a counter of the number of consecutive occurrences. In some certain circumstances it can be very useful and efficient specially if combined with other techniques. In 1983 RLE was patented by Hitachi.

Arithmetic coding (1976)

Basic algorithms for arithmetic coding were developed independently by Jorma J. Rissanen[6], at IBM Research, and by Richard C. Pasco, a Ph.D. student at Stanford University; both were published in May 1976.

Huffman coding is a fixed-length coding method available and it is the best available. Nevertheless Huffman codes have to be an integral number of bits long, and this can sometimes be a problem. If the probability of a character is 1/3, for example, the optimum number of bits to code that character is around 1.6 bits. Huffman coding has to assign either one or two bits to the code, and either choice leads to a longer compressed message than is theoretically possible. In fact Huffman is a non optimal coding technique.

Arithmetic coding bypasses the idea of replacing an input symbol with a specific code. It replaces a stream of input symbols with a single floating-point output number.

Example 5. *The message "BILL GATES", for example, would have the following probability distribution:*

[6]Jorma Johannes Rissanen (1932-2020) was an information theorist, known for originating the minimum description length (MDL) principle and practical approaches to arithmetic coding for lossless data compression.

Symbol	Probability	Range
‘ ’	1/10	$0 < r < 0.1$
A	1/10	$0.1 < r < 0.2$
B	1/10	$0.2 < r < 0.3$
E	1/10	$0.3 < r < 0.4$
G	1/10	$0.4 < r < 0.5$
I	1/10	$0.5 < r < 0.6$
L	2/10	$0.6 < r < 0.8$
S	1/10	$0.8 < r < 0.9$
T	1/10	$0.9 < r < 1$

Table 4.7: Input data.

Table 4.7 shows input data, how the are sorted and their range of probability. The Arithmetic coding encoding algorithm is:

```
low = 0.0;
high = 1.0;
while ((c = getc(input)) != EOF) {
  range = high - low;
  high = low + range * high_range(c);
  low = low + range * low_range(c);
}
output(low);
```

Listing 4.1: Arithmetic coding encoding algorithm

Following the encoding process shown in Listing 4.1 brings to the following result:

Symbol	Low Value	High Value
	0.0	1.0
B	0.2	0.3
I	0.25	0.26
L	0.256	0.258
L	0.2572	0.2576
‘ ’	0.2572	0.25724
G	0.257216	0.25722
A	0.2572164	0.2572168
T	0.25721676	0.2572168
E	0.257216772	0.257216776
S	0.2572167752	0.2572167756

Table 4.8: Arithmetic encoding.

So the final value 0.2572167752 (as shown in Table 4.8), will uniquely encode "BILL GATES".

The Arithmetic coding decoding algorithm is:

```
number = input_code();
for (;;) {
  symbol = find_symbol_straddling_this_range(number);
  putc(symbol);
  range = high_range(symbol) - low_range(symbol);
  number = number - low_range(symbol);
  number = number / range;
}
```

Listing 4.2: Arithmetic coding encoding algorithm

Following the decoding process shown in Listing 4.2 brings to the following result:

Encoded Number	Output Symbol	Low Value	High Value	Range
0.2572167752	**B**	0.2	0.3	0.1
0.572167752	**I**	0.5	0.6	0.1
0.72167752	**L**	0.6	0.8	0.2
0.6083876	**L**	0.6	0.8	0.2
0.041938	**‘ ’**	0.0	0.1	0.1
0.41938	**G**	0.4	0.5	0.1
0.1938	**A**	0.2	0.3	0.1
0.938	**T**	0.9	1	0.1
0.38	**E**	0.3	0.4	0.1
0.8	**S**	0.8	0.9	0.1
0				

Table 4.9: Arithmetic decoding.

In summary, the encoding process is simply one of narrowing the range of possible numbers with every new symbol. The new range is proportional to the predefined probability attached to that symbol. Decoding is the inverse procedure, in which the range is expanded in proportion to the probability of each symbol as it is extracted (see Table 4.9).

Practical Matters Encoding and decoding a stream of symbols using arithmetic coding at first glance seems completely impractical. Particularly because floating point numbers have many limitations on digital computers. As it turns out, arithmetic coding is best accomplished using integer math. Floating point math is not required.

LZ77 (1977)

This algorithm was introduced by Lempel and Ziv in 1977 with the paper: "A Universal Algorithm for Sequential Data Compression" as part of *IEEE Transactions on Information Theory*

journal.

LZ77 is revolutionary because that compression uses previously seen text as a dictionary. It achieves compression by replacing variable-length strings in the input text with fixed-size pointers into the dictionary.

The main data structure in LZ77 is a text window, divided into two parts. The first consists of a large block of recently decoded text. The second, normally much smaller, is a look-ahead buffer. The look-ahead buffer has characters read in from the input stream but not yet encoded.

The normal size of the text window is several thousand characters. The look-ahead buffer is generally much smaller, maybe ten to one hundred characters. The algorithm tries to match the contents of the look-ahead buffer to a string in the dictionary.

LZ77 encodes in sequences of tokens. Each token consists of three items:

1. An offset to a phrase in the text window.
2. The length of the phrase.
3. The first symbol in the look-ahead buffer that follows the phrase.

A token is a tuple: `(offset, length, next symbol)`.

The decompression algorithm is very similar to the compression one. It reads in a token, outputs the indicated phrase, outputs the following character, shifts, and repeats. It maintains the window, but it does not work with string comparisons.

The dictionary is the text window, so it is adaptive.

LZ78 (1978)

This algorithm was introduced by Lempel and Ziv in 1978 with the paper: "Compression of Individual Sequences via Variable-Rate Coding" as part of *IEEE Transactions on Information Theory* journal.

LZ78 abandons the concept of a text window. Under LZ78, the dictionary is a potentially unlimited list of previously seen phrases. LZ78 also outputs a series of tokens with essentially the same meanings of LZ77. Unlike LZ77, the phrase length is not passed since the decoder knows it.

This time token is a tuple: `(index, next symbol)`. Here "index" takes the place of "offset" because of the different structure of the dictionary.

LZ78 adds that phrase to the dictionary. After the phrase is added, it will be available to the encoder at any time in the future, not just for the next few thousand characters.

Also in this case the decompression algorithm is very similar to the compression one.

This algorithm is "greedy", so it takes locally optimal decision optimal choice at each stage. The problem is that in many cases, like in this one, a greedy strategy does not produce an optimal solution.

Example 6. *Consider the following input text: "DAD DADA DADDY DADO...".*

Output Phrase	Output Character	Encoded String
0	D	D
0	A	A
1	‘ ’	‘D ’
1	A	DA
4	‘ ’	‘DA ’
4	D	DAD
1	Y	DY
0	‘ ’	‘ ’
6	O	DADO

Table 4.10: LZ78 encoding.

In Table 4.10 each new phrase that was not seen before is also added to the dictionary that initially was empty. Step by step the dictionary helps to reduce the volume of data.

The previous example makes clear that the decompression is done without giving the dictionary to the decompressor, because the compression algorithm always outputs the phrase and character components of a code before it uses them. This is very important because in this way the compressed data is not burdened with carrying a large dictionary.

LZW (1984)

Lempel-Ziv-Welch algorithm was published by Welch in 1984 as an improved implementation of the LZ78 algorithm published by Lempel and Ziv in 1978.

Here the token is simply a value: `index`.

LZW is able to simplify the token because of the way it stores items on the dictionary. A new entry to the dictionary is the concatenation of the matched pattern and the next character from the input stream. The "next symbol" is always part of the new entry being added to the dictionary.

The decoder, upon receiving the index, retrieves the matched pattern and updates its dictionary concatenating the matched pattern and the next inferred character. This means that the dictionary is maintained adaptive.

BWT (1994)

Burrows-Wheeler transform rearranges a character string into runs of similar characters. It was invented by Michael Burrows[7] and David Wheeler[8] in 1994.

If the original string had several substrings that occurred often, then the transformed string will have several places where a single character is repeated multiple times in a row.

[7]Michael Burrows (born 1963) is a British computer scientist and the creator of the Burrows-Wheeler transform.

[8]David John Wheeler (1927-2004) was a computer scientist and professor of computer science at the University of Cambridge.

ANS (2014)

Asymmetric numeral systems is a family of entropy encoding methods introduced by Jarek Duda[9] and used in data compression since 2014 due to improved performance compared to previous methods. ANS combines the compression ratio of arithmetic coding with a processing cost similar to that of Huffman coding. It's tabled variant, tANS, is achieved by constructing a finite-state machine.

4.2 Comparison of Algorithms

They are useful when done on algorithms of the same compression technique: lossless or lossy. The following factors refer mainly to lossless data compression, in the lossy case it is more complicated (for example it should be considered how good is the approximation). Measurements have to be considered as averages in practice, because they are different for every single sample so their value is an approximation.

4.2.1 Compression Ratio

An important factor when considering data compression is the "compression ratio". It is simply the sample size reduction factor (look at Equation 4.1).

$$compressionratio = \frac{uncompressedfilesize}{compressedfilesize} \tag{4.1}$$

4.2.2 Compression Speed

It is important is specific use cases, usually when the network is involved (look at Equation 4.3).

$$compressionspeed(SIZE/s) = \frac{uncompressedfilesize(SIZE)}{compressiontime(s)} \tag{4.2}$$

4.2.3 Decompression Speed

As the compression speed, it is important is specific use cases, usually when the network is involved (look at Equation 4.3).

$$decompressionspeed(SIZE/s) = \frac{uncompressedfilesize(SIZE)}{uncompressiontime(s)} \tag{4.3}$$

4.2.4 CPU usage

The use can differ from 100% (a single core) to $N \times 100\%$ (N cores), where N is the number of logical cores of the CPU. It has to be considered than even if a multi-threaded algorithm is generally faster than it's equivalent single-threaded algorithm, the higher CPU usage is something

[9] Jarosław Duda (born 1963), also known as Jarek Duda, is a Polish computer scientist and an assistant professor at the Institute of Computer Science and Computational Mathematics of the Jagiellonian University in Kraków. He is known as the inventor of asymmetric numeral systems (ANS), a family of entropy encoding methods widely used in data compression.

that could be considered in the comparison of algorithms, depending on the needs. In practical tests the percentage could be different from a multiple of 100%, for example an average.

In most cases every algorithm is implemented to work on the CPU only, because GPUs are expensive and the hardware is needed. If it could work on the GPU it should be specified and tests should consider that. In fact their results should not be compared with other algorithms implementations that work on the CPU.

4.2.5 Memory usage

How much RAM the implementation of the algorithm needs.

4.2.6 Time

It could be simply considered the total elapsed time, but a more accurate analysis can be done.

User Time

The amount of CPU time spent executing user-level code. It does not include time spent in kernel mode (e.g., for system calls or I/O operations).

System Time

The amount of CPU time spent executing system-level code on behalf of the program (in kernel mode). It includes time spent handling system calls (reading a file, memory allocation, networking...) and overhead for interacting with hardware or operating system resources.

Wall Time

The actual elapsed real-world time taken for the program to run from start to finish. It is the sum of the `user time` and the `system time`.

Chapter 5

Compression Tools

It is important to consider that compression tools are logically separated:

- A file archiver combines several files into one file.
- A compression tool compresses and decompresses a single file.

These tools are often used in sequence to create compressed archives. In some programs the two phases are hidden by the program itself.

5.1 Archiving

It combines several files into a single file with a size equivalent to the sum of each single file dimension.

5.1.1 Unix

According to the Unix philosophy the archiving process is separated from the compression process. Unix known programs that aim to archiving only are `ar`, `cpio`, `tar` and `libarchive`.

`ar` was the original archiver, but has been largely replaced by `tar`. `tar` derived from "tape archive" , as it was originally developed to write data to sequential I/O devices with no file system of their own, such as devices that use magnetic tape.

In 2001, IEEE defined a new pax format which is basically tar with additional extended attributes. The name "pax" is an acronym for portable archive exchange, but is also an allusion to the Latin word for "peace"; intended as a peaceful unification of both `tar` and `cpio`.

`libarchive` is a free and open-source library, it's development was started in 2003. It works on most Unix-like systems and Windows.

5.1.2 Windows

By default treats ZIP archive format that supports lossless data compression. Native support was added as of the year 2000 in Windows ME.

One of the most common programs used in Windows is `7z` (7-Zip) that has its own archive format called 7z, but can read and write several others, so it is widespread for compatibility.

5.1.3 Mac OS X

Apple has included built-in ZIP support in macOS 10.3 (via BOMArchiveHelper, now Archive Utility) and later.

5.2 Compression

A comparison between some of the following compression tools can be found online but it is important to stay updated. About compression in Linux many articles can be found. At `https://www.redpill-linpro.com/techblog/2024/12/18/compression-tool-test.html` different algorithms are tested with a single large file. Moreover at `https://www.baeldung.com/linux/compression-methods-comparison` a more accurate comparison that uses the Silesia Corpus GitHub repository as a collection of files used for compression benchmarks.

The following compression programs are lossless.

5.2.1 lz4

It is focused on compression and decompression speed. It belongs to the LZ77 family. It is multi-threaded by default.

5.2.2 compress

It is focused on compression and decompression speed. It uses LZW. It is not multi-threaded.

5.2.3 gzip

GNU zip is based on the DEFLATE algorithm. DEFLATE is a combination of LZ77 and Huffman coding. It is not multi-threaded. `pigz` is the parallel version.

5.2.4 bzip2

It is the successor of the `bzip` obsolete version. It uses several layers of compression techniques, such as run-length encoding (RLE), Burrows-Wheeler transform (BWT), move-to-front transform (MTF), and Huffman coding. It is not multi-threaded. `pbzip2` is the parallel version.

5.2.5 bzip3

It is a modern implementation of bzip2 with enhancements. It is not multi-threaded.

5.2.6 lzma

For compression/decompression the Lempel-Ziv-Markov chain algorithm (LZMA) is used. This algorithm was introduced in 1998 and is based on LZ77. It was developed by Igor Pavlov the author of 7-Zip. It is not multi-threaded.

5.2.7 xz

For compression/decompression LZMA2 is used, it is an improved version of the Lempel-Ziv-Markov chain algorithm (LZMA). It is multi-threaded by default.

5.2.8 zstd

Zstandard is a lossless data compression algorithm developed by Yann Collet at Facebook. Zstandard uses a dictionary windows (like LZ77) combined with fast entropy-coding. It uses Huffman coding and finite-state entropy (FSE) (a version of ANS, tANS). Zstd is the corresponding reference implementation in C, released as open-source software[1] in 2016. It is multi-threaded by default.

[1] the official implementation is available at `https://github.com/facebook/zstd`

Appendix

Ultimately, the selection of a compression method depends on the specific requirements of the task at hand.

For tasks prioritizing speed, tools with a lower compression ratio like `compress` and `lz4` are excellent choices. Small files might be more efficiently handled by fast compressing tools like `gzip` (or better `pigz`). Large files benefit from high compression ratio tools like `bzip2` (or better `pbzip2`) and `bzip3`.

The file type also matters, with textual data typically compressing well with the tools previously seen. On the other hand binary files might see better results with `zstd` which has a high compression ratio and it is very fast. Furthermore `lzma` and `xz` have impressive compression levels but require a lot of time.

Bibliography

[Ben19] Arieh Ben-Naim. "Entropy and Information Theory: Uses and Misuses". In: *Entropy* 21.12 (2019). ISSN: 1099-4300. DOI: 10.3390/e21121170. URL: https://www.mdpi.com/1099-4300/21/12/1170.

[DH10] Rodney G. Downey and Denis R. Hirschfeldt. *Algorithmic Randomness and Complexity*. Springer, 2010. URL: https://books.google.it/books?id=FwIKhn4RYzYC&lpg=PR3&ots=_nbPVe6IUi&dq=randomness%20in%20computer%20science&lr&pg=PA226#v=onepage&q&f=false.

[Car07] Tom Carter. "An introduction to information theory and entropy". In: *Complex systems summer school, Santa Fe* (2007).

[Sal07] D. Salomon. *A Concise Introduction to Data Compression*. Undergraduate Topics in Computer Science. Springer London, 2007. ISBN: 9781848000728. URL: https://books.google.it/books?id=mnpeizY0btYC.

[NG95] Mark Nelson and Jean-Loup Gailly. "The data compression book 2nd edition". In: *M & T Books, New York, NY* (1995).

[Kol68] A. N. Kolmogorov. "Three approaches to the quantitative definition of information". In: *International Journal of Computer Mathematics* 2.1-4 (1968), pp. 157–168. DOI: 10.1080/00207166808803030. eprint: https://doi.org/10.1080/00207166808803030. URL: https://doi.org/10.1080/00207166808803030.

[Mar66] Per Martin-Löf. "The definition of random sequences". In: *Information and Control* 9.6 (1966), pp. 602–619. ISSN: 0019-9958. DOI: https://doi.org/10.1016/S0019-9958(66)80018-9. URL: https://www.sciencedirect.com/science/article/pii/S0019995866800189.

[Sha48] C. E. Shannon. "A mathematical theory of communication". In: *The Bell System Technical Journal* 27.3 (1948), pp. 379–423. DOI: 10.1002/j.1538-7305.1948.tb01338.x.