# INFORMATION THEORY AND HUFFMAN ENCODING

A THESIS SUBMITTED TO THE FACULTY OF ARCHITECTURE AND ENGINEERING
OF EPOKA UNIVERSITY

BY

KEVIN TERVOLI
DANIEL AVDIU
BLERIM BRAHJA
DANIEL FIGU

MONTH (JANUARY), YEAR (2023)

**We hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. We also declare that, as required by these rules and conduct, We have fully cited and referenced all material and results that are not original to this work.**

Name, Surname:

- Daniel Avdiu
- Kevin Tervoli
- Blerim Brahja
- Daniel Figu

Signatures:

# ABSTRACT


## INFORMATION THEORY AND HUFFMAN ENCODING

The topic for this research paper is about information theory and huffman encoding. First we are going to have a brief introduction to information theory and some of its features. The following sections introduce the different data compression techniques, and code implementations of Huffman Encoding. This topic is of quite importance due to the reason being that data compression is essential in today's technology. Without compression it would mean we would have a lot more packets flowing in a network, whereas with compression the number is reduced significantly.

Here we are going to have a look at the basics of compression and when compression (Huffman Encoding) was first introduced. The aim of this research paper is to present the usefulness of Huffman Encoding. The algorithm and pseudocode of this technique will be provided and then both the advantages and disadvantages are discussed in detail with the respective arguments. At the end, we are going to be able to tell whether certain techniques work or not in some situations.

The approach that we have followed is mainly theoretical, by explaining the terms and formulas for information theory. Then the various compression techniques are discussed, with the next section being code implementations of what was theoretically explained beforehand. After having considered data compression from various perspectives, we have proposed a model for a network where Huffman Encoding can be integrated in a server responsible for only its corresponding subnet.


**Keywords:** Huffman Encoding Algorithm, Huffman Encoding, Data Compression

# Table of Contents

# LIST OF FIGURES



LIST OF TABLES

# 1. INTRODUCTION

## 1.1 Information theory

Information theory has come a long way, from a single theoretical paper by Claude Shannon in 1948, to a broad field that has redefined many processes. The field has evolved in such a great amplitude that it can be used in the studies of social, political and technological interactions.

Even though Shannon was the first one to put substantial emphasis on the field of information theory with his work, there were also predecessors who did contribute in the development of information theory, and this statement can be supported by the fact that Shannon himself acknowledged work done before him. However, he was the one who revolutionized communication, which went on to be the base of creation for information theory.

By definition, information theory is the mathematical treatment of the concepts, parameters and rules governing the transmission of messages through communication systems[1]. The methods used in information theory are of probabilistic nature and that is the reason for information theory to be considered as an outcome of probability theory. Despite the fact that the topic seems a bit far-fetched, and information theory may seem unheard of, the scientific discipline itself has a wide application in many other areas. One of them is Data Compression. Through information theory we can determine the probability to compress data, without any loss. It becomes quite interesting when encoding and decoding is involved in this process.

Shannon was one of the first advocates of this scientific field, however there was little practical implementation. The idea of compression is to find a way of representation for data, in a more efficient way than just a byte for a character. Compression comes in handy in the cases of storage and transmission. The primary concern in computer networks is the transmission of data, which has to be divided into chunks of data, and then can be sent. If there was a way, which there is, to compress data, it would make the whole process considerably easier.

In this research paper the primary focus will be put on getting insight of applied information theory techniques for encoding. Huffman Encoding is one of the main topics that will be discussed in this in the following sections. The algorithm and code implementation are going to be provided in order to get a full understanding of this encoding technique. In addition, prospective interesting features of information theory are not going to be left out.

One of the main problems that we face in encoding and decoding is variability. Different data objects may need different implementations to be encoded and decoded, which raises a serious

problem in the real life scenarios. One optimal solution to this problem is Huffman Encoding and Decoding.

## 1.2 Entropy

The amount of information we receive at the occurrence of an event is inversely proportional related to the probability of p of the event. To get other natural properties of information, it can be shown that the appropriate definition is:

$$information\ gained\ upon\ learning\ event\ of\ probability\ p\ occurred\ =\ \ log_2\frac{1}{p} bits$$

The measure denoted as entropy is quite significant due to the fact that it represents useful information, and that information is crucial in the process of coding and decoding the information coming from the source. If someone is sending us a stream of symbols from a source, there is generally some uncertainty about the sequence of the symbols. Otherwise, if there was no uncertainty, there would be no point in sending and receiving the data. A commonly used way to model this uncertainty is to assume that the data is coming randomly according to some probability distribution. The least complex model is one where the symbols $s_1, s_2, \ldots, s_n$ have associated probabilities of occurrences in the stream of data. Supposing that the receiving end gets the symbols from $s_1$ to $s_n$ and the sum of their probability distributions is 1, how much information does the source provide? Assuming that for symbol (formula) $s_i$ the probability of occurrence is $p_i$, then we get $log_2\frac{1}{p_i}\ bits$ of information. Therefore the average number of bits of information we get by observing one symbol is:

$$p_1 log_2\frac{1}{p_1}\ +\ldots\ + p_M log_2\frac{1}{p_M} = \sum_{i=1}^{M} p_i log_2\frac{1}{p_i}$$

The above formula represents the Entropy of the source and it is denoted by $H$. $H$ represents the the average number of bits of information per symbol from the source, meaning that we might expect that we need to use at least $H$ bits per symbol to encode each symbol in order to represent the source with a uniquely decodable code[1]. If we make encode longer and longer strings of symbols, it would mean that with reaching infinity, the performance of the codework would get close to $H$ (average number of bits per symbol). Analyzing it from the mathematical aspect, it would mean that no program can do better than $H$ (average number of bits per symbol). In cases where the string we are operating on is sufficiently long, the performance tends to go towards $H$.

The uncertainty that is mentioned above is somehow ambiguous. Information theory itself is concerned with means of transmitting/conveying information among two or more entities (being them people or machines). The transmitter therefore sends a series of partial messages that provide clues for the original message. The information content of one of these partial messages is a measure of how much uncertainty this resolves for the receiver. A partial message may provide useful information, or information that does not serve the receiver on the other hand any good[2]. A partial message that cuts the number of possibilities

[1] Computer Networking Information Theory
[2] Formal Definition of Entropy

in half transmits only one bit of information. If the source is trying to send a randomly chosen digit to the receiver, the partial message would be "The number is even", therefore reducing the possibilities. However, a partial message conveying no information at all, a typical one being "The number is less than 10", which does not reduce the probability in measurable terms. In this context, the "Entropy" can be perceived as how much useful information a message is expected to contain. In addition it provides a lower bound for the size of an encoding scheme [3].

# 2. HUFFMAN ENCODING

## 2.1 General Overview

Huffman Encoding & Decoding seems to be an optimal solution, out of other possible ones, due to the reason that it is based on the fact that it can adapt to variable length codes given the probabilities of the symbols. The resulting code, from Huffman algorithm, is utterly decodable and instantaneous. This compression technique is widely used in image compression, video compression and codes used in fax machines[2].

The result after applying the Huffman Encoding technique to given data (can be text or any other type) is a tree-like structure which is created by using the frequencies of characters in the data. The feature of being decodable comes from the fact that the codeworks of a Huffman code are the leaves of the binary tree. The paths to specific leaves in a binary tree are unique, there is no ambiguity in the process, which is why the algorithm and code are versatile. The final result is a string of bits, which is constructed alongside the Huffman Tree. After the string is completed, it can be decoded using the binary tree. A Huffman tree makes the best use of storage as it is not wasteful, in the sense that all leaf nodes correspond to symbols. If there are empty lead nodes, those would be removed and shorten certain codeworks and still have the tree code. We should not leave out the fact that the average length of the Huffman code may not equal the entropy. Whatever the case, Huffman code does satisfy the following:

$$H \leq average\ length\ of\ Huffman\ code \leq H\ +\ 1 \quad [3]$$

In order to approach the entropy, we can use the block coding technique. The average length of the corresponding Huffman code approaches the entropy as the block size gets larger. For instance, using a block size of k, the following holds true:

$$H \leq average\ length\ per\ original\ symbol \leq H\ +\ 1/k$$

[3] Entropy Information Theory

For practical reasons, the block size cannot be taken to be too large. In the case when the source has M symbols, it will follow that for the block of length k there are M^k strings of symbols to be considered. [4] . The result would be an exponential growth in the block of size k. The encoder performs the codework and constructs the Huffman Tree. To counterbalance the effectiveness of Huffman Encoding, in furtherance of the previous statement, the decoder also needs the codework. Either the Huffman Tree has to be attached to the sending block as overhead, or the decoder has to be provided with the frequencies and probabilities of the data. In other words, Huffman Encoding gives us the opportunity to not send the data itself for communication purposes, rather the tree structure and the end result string can be used to decode the message or the data at the destination. The overhead, on the other hand, seems to become cumbersome as it suggests that some cases of communication tend to not become efficient if encoding is involved.

Although it is beneficial using Huffman Encoding, it is not always the best solution, as there are numerous disadvantages as well. This type of coding algorithm can be less effective on data where there are very few unique symbols, or where the symbols are already highly compressed, and there is no more room for compression [4]. The Huffman Tree, in cases of communication, tends to become overhead in the sending of messages. This drawback can be avoided when the communicating units share the Huffman Tree among them and do not have to store and send it to the others, with the exception that it should be accessible only by the communicating units inside the network, otherwise there would be leakage of information.

David Huffman was an MIT student who took the course of Information Theory taught by Professor Fano (a colleague of Shannon). The class was given the option that instead of a final exam, they could do research on a preferred topic instead. David Huffman had been working for months on his thesis and was about to give up until he had a realization. Consider a string (text) that you want to encode or compress. It is logical that more bits are used by the character that appears most frequently in the string. He figured that one solution would be to use the shortest code for the most frequent character/symbol. By using the Huffman Tree, the shortest path is to the node with the character that was most repeated.

## 2.3 Algorithm

Huffman encoding follows the same algorithm all the time, however, the tree that is constructed is not always the same. As it has already been mentioned above, Huffman makes use of the frequency of the characters in a data stream (which is why we mentioned the entropy measure). Encoding basically means converting a string into another format of data which is beneficial for numerous reasons, one of them being space-usage. It is also referred to as

[4] Information Theory & The digital revolution

## 2.4 Huffman Encoding in DNA

Computations involving DNA sequencing have shown to be expensive in terms of storage due to the length of numerous genomes ranging from megabases to gigabases. A simple implementation of the Huffman encoding would simply not be sufficient to get satisfactory results. Skewed Huffman Tree[5], on the other hand, provides a significant improvement in DNA compression for the genomes ranging from 5 to 50 Mbp[6].

DNA sequences are composed of nucleotides, namely A, C, G, and T. The number of these characters used in gene sequencing is enormous, This presents an inconvenience for Huffman Encoding. The greater the number of characters used in a data stream, the less the chance that we can represent characters with shorter codes than they already have. In brief, Huffman Encoding takes advantage of the most frequent elements as they are represented by the shortest code. However, the probability distribution and hence the frequencies of the characters increase when the data source provides such a stream that it is almost impossible to use the normal approach of Huffman Encoding. There is another subsequent approach that can be followed. Instead of keeping track of the most frequent characters, we can try and find the most repeated DNA strings, and then encoding those, and not the individual characters. However, it is even harder to check the string of DNA over and over again to identify the substrings that are repeated by taking into consideration all the possible combinations. The time and space complexities suggest that this modified approach does not perform well. In order to store and check the chunks of strings one by one, we would have to buffer them in order to find the frequencies of some specific. Since the length of the substring is not defined, it becomes an even greater problem, not that it won't converge at some point. The issue itself is not the search for substrings, but how it is implemented with the least time and space complexity for the algorithm, because the solution comes at a high cost.

The bit count of the DNA sequence "ACTGAACGATCAGTACAGAAG," for instance, is 21x8 = 168 bits if each base is encoded with an 8-bit ASCII code, with A = 01000001, C = 01000011, G = 01000111, and T = 01010100. This is based on the assumption that the DNA sequence comprises 21 bases. The Huffman compression algorithm functions as follows. The algorithm initially counts each base's frequency, as shown in Table 1. Each base should be viewed as a tree node together with the frequency that it corresponds to. Identify the first four nodes as unprocessed. The following stages are repeated to create a binary tree. The two unprocessed nodes with the lowest frequencies should be searched first.

Create a parent node next, whose frequency is equal to the total of the frequencies of the two child nodes. Return the parent node to the list of nodes that have not been processed. When each

[5] See Appendix A for detailed information on skewed Huffman Trees
[6] Article Reference

node has been processed, the tree construction process is complete. Fig. 1 depicts the Huffman tree's creation for the provided situation. Keep in mind that each internal node's left link is labeled 0, while the right link is labeled 1.

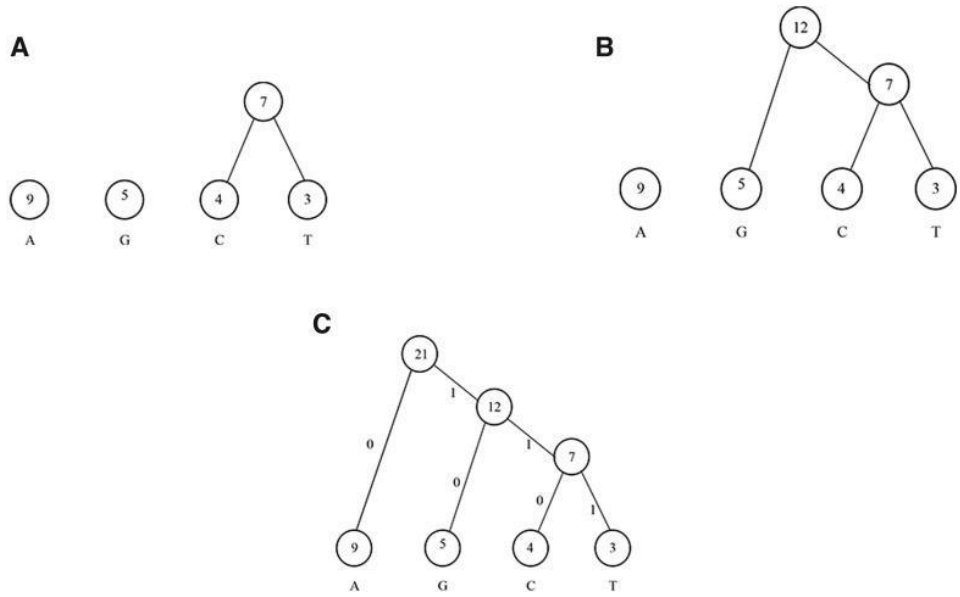


*Figure 1. Huffman Tree From DNA sequence*

| Base | Frequency | The new representation | The old representation |
|---|---|---|---|
| A | 9 | 0 | 01000001 |
| G | 5 | 10 | 01000111 |
| C | 4 | 110 | 01000011 |
| T | 3 | 111 | 01010100 |

*Table 1. DNA sequences in Huffman Encoding*

Each symbol has a new bit representation after the Huffman tree has been constructed, which may be obtained by traversing the edges and recording the bits connected to the required symbol from the tree's root (which must be located at the leaf level). For instance, the bit representation for the path from the root to base A is 0, while the bit representation for the path from the root to base T is 111.

### 2.5.1 Unbalanced Huffman Tree

For the improvement in DNA encoding it is recommended to create the unbalanced huffman tree instead. The only guarantee we get is that 3 of the 4 DNA bases can be represented using only 2 bits and the fourth one has to be represented with 4 bits[7]. We can force the Huffman Tree to be unbalanced in the following way:

1. Based on their frequencies, order all k-mers[8] that have been gathered, where k 2.
2. Choose the k-mer that has the highest frequency, let's say f.
3. Until a k-mer with a frequency lower than equation $\frac{f}{2}$is discovered, scan all k-mers with frequencies lower than f in descending order. Choose this k-mer, and then set f to match the frequency of the chosen k-mer. Keep in mind that no chosen k-mer is appropriately contained in another. This makes it possible to decrease overlap between the chosen k-mers, increasing the amount of bases that can be compressed.
4. Repeat steps 2 and 3 until no more k-mers can be selected.
5. Finally, feed the four bases and the k-mers (chosen above) into the Huffman encoding technique.

# 3. SHANNON THEOREM

Shannon published his paper “A Mathematical Theory of Communication” in the Bell Systems Technical Journal, by showing how information could be quantified with absolute precision, and demonstrated the essential unity of all information media. Shannon showed that essentially every mode of communication, telephone signals, text, radio waves, and pictures, could be encoded in bits. His journal was used as a blueprint for the digital era, as the practical implementations took place afterwards.

There are four major concepts in Shannon’s paper.

- **Channel Capacity & The Noisy Channel Coding Theorem**

[7] Al-Okaily, A., Almarri B., Al Yami S., Huang Ch., “Towards a Better Compression for DNA Sequences Using Huffman Encoding”, National Library of Medicine, National Center for Biotechnology Information.

[8] K-mers refers to the substrings of length up to k bases

Shannon proposed the connect that every communication channel had a speed limit, measured in binary digits per second, which is exemplified by the familiar formula of a White Gaussian Noise Channel:

$$C_t = Wlog_2 \frac{P+N}{N}$$

Unfortunately, it is mathematically impossible to get an error free communication medium with efficiency above the limit. No matter how much data is compressed, you cannot make the channel go faster without any information loss. The noisy channel coding theorem is what gave rise to the entire field of error-correcting codes and channel coding theory. We may have experienced the phenomenon of error correction code in our everyday lives. One pure example is the CD. Even if you scratch it, it would still perform as if there was no change, this is the result of the noisy channel theorem. In order to get a better understanding of the topic, below we have included an illustrative picture of the general architecture of communication systems.

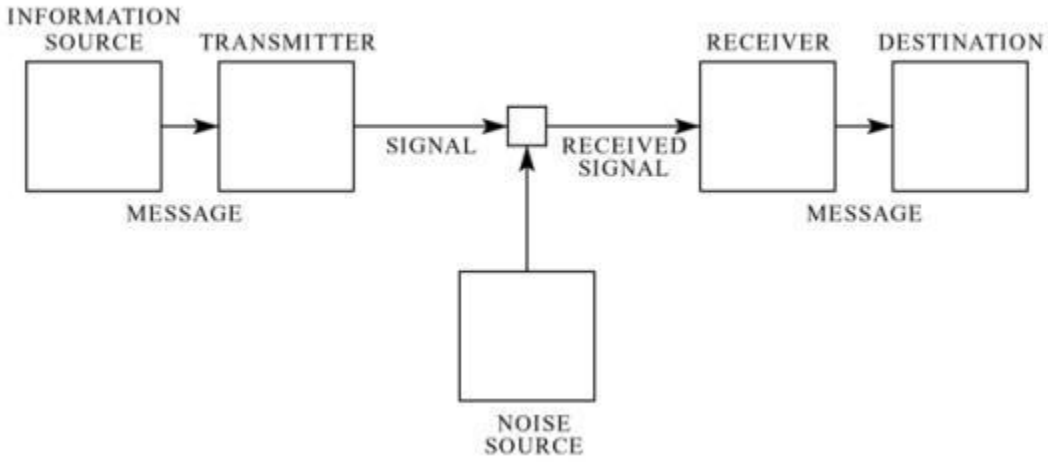


*Figure 2. Communication System's Structure*

The vast majority of the new era's communication systems are based on this model shown above, supporting the suggestion that it serves as a blueprint for the digital era.

Shannon realized and pointed out that the content of the message was irrelevant to its transmission. Any kind of data that he would consider, would be turned into bits and then transmission would follow. He introduced a new way of encoding data (which we now refer to as data compression), the Shannon-Fano code. It was later shown that his method was not always optimal in reducing the expected length of the message. A couple of years later, David Huffman, an MIT student came up with the Huffman Code (which is mentioned in the sections above). Almost everyone uses zip files, because we find the need to compress data in order to save space.

Huffman encoding is the technique that is used in the creation of zip files (data compression). It is also widely used in image and audio compression. [14]

## 3.1 Shannon’s Noisy Channel Theorem

A source that produces source letters, but not always randomly and independently, is covered by Shannon's Noisy Channel Theorem. It is remarkable that while being widely considered as the foundation of information theory, the Noisy Channel Theorem's assertion and its proof are essentially useless in real-world applications. Furthermore, the theory cannot be stated to have indirectly influenced the development of effective "error-correcting" codes or their theory, both of which might and frequently are presented without mentioning the word "entropy."

The main claim of the theorem is that if the rate of information flow from the source is lower than the channel capacity, you have enough room to afford the luxury of error correction. If you're willing to put in the effort, you can reduce the maximum error probability as low as you like while still allowing the information flow from the source to proceed without any accumulated delays or backlog. Doesn't that make the channel capacity sound like the highest rate at which information might possibly flow? The average error probability cannot be decreased below a specific positive amount, a function of the source frequencies and time rate as well as the channel's transition probabilities, according to Shannon's original formulation of the theorem, which makes a second, similarly telling claim.

- **Channel Capacity Analogy**

Consider a flash flood that is approaching a culvert. The culvert pipe has a maximum amount of water it can move in a given amount of time. In theory, water can be routed through the pipe without a drop splashing over the road above the culvert or a pond of unconveyed water amassing on the flood side of the culvert if the rate at which the flood is arriving at the pipe entrance is less than that maximum rate. In practice, the directing of the flood waters into the pipe, a civil engineering problem, will not be perfect – some water will be lost by sloshing. But as long as the flood flow is below the theoretical maximum that the pipe can handle (the pipe’s capacity), steps can be taken to reduce the sloshing loss (error) below any required positive threshold, while maintaining flow and avoiding backup. If the flood rate exceeds the pipe capacity, then no engineering genius will be able to avoid some combination of water loss and backup; the flood volume per unit time in excess of the pipe capacity has to wind up somewhere other than the pipe.

# 4. Encoding

## 4.1 Block to variable encoding

Block coding is a technique for encoding data in units of constant size, whereas variable-length coding uses code words of varying sizes. Because it allows for the efficient encoding of frequently occurring symbols using shorter code words and infrequently occurring symbols are efficiently recorded using longer code words, variable-length coding is more effective than block coding. As a result, the bit space that is open is used more effectively, resulting in a smaller overall code size and less need for data transmission and storage.

Huffman coding is an illustration of a variable-length coding system that uses shorter code words for symbols that appear more frequently in the data and longer code words for symbols that do not as often. As a result, the bit space that is available is used more effectively since the symbols that are used more frequently occupy less space than they would in a fixed-size block coding scheme. Arithmetic coding is another type of variable-length encoding that represents symbols as intervals on a number line as opposed to fixed-length code words. As a result, the bit space can be used even more effectively because the intervals can be set to be as brief as is required to adequately represent the data.

Variable-length coding also has the benefit of enabling data compression because the encoded data often takes up less space than the original data. This is especially helpful for applications like digital communication, digital video, and picture compression when data storage or transmission is an issue. Block coding, in contrast, is primarily utilized for error detection and correction and typically doesn't offer significant data reduction. When data integrity is a problem, such as in disk storage or digital communication systems, it is frequently utilized.

Overall, block coding and variable-length coding are two distinct methods, each with its own advantages and uses. Block coding is better suitable in circumstances where data integrity is a concern, even though variable-length coding is more effective in terms of bit usage and data reduction.

# 5. DATA COMPRESSION

## 5.1 Image Compression

Modern computers employ graphics extensively. The progress that we have witnessed throughout the years has been incredible. Since modern hardware can display many colors, it is common to have a pixel represented internally as a 24-bit number, where the percentages of red, green, and blue occupy 8 bits each. As it was mentioned in the prior sections, Shannon pointed out that any type of data in the communication medium can be represented as bits, the same is valid for images. A 24-bit number can specify 16.78 million colors. An image at the resolution of 512x512 would occupy 786,432 bytes.[9] An image of resolution 1024x1024 would be four times as large in size. The numbers illustrate why image compression is important.

In general terms, information can be compressed if it is redundant. The same principle is used in Huffman Encoding. The more frequent a character is, the more opportunity there is for data compression. Another term used for this technique is lossless data compression. With lossy compression, however, we have a new concept, namely compressing information by removing irrelevant data [10]. An image with no redundancy is not always random. In an image where each color appears with the same frequency  does not necessarily mean that it is random.

Digitizing an image involves two steps: sampling and quantization. Sampling is what we refer to as pixels. Quantization is the process when we assign an integer value to each pixel. The compression techniques that are most commonly used are RLE, scalar quantization, statistical methods, and dictionary-based methods. However, none of these techniques shows satisfactory results when used for images or even for gray scale images. *Facsimile compression* [11] uses RLE combined with Huffman coding and gets good results, but only for bi-level images.

Let's consider scalar quantization. It can be used to compress images, but its performance is not as good as we think. If we have an image with 8-bit pixels, it can be compressed with scalar quantization by cutting off the four least-significant bits of each pixel. The compression ratio will only be 0.5 in this case (4 out of 8 bits are removed), and this ratio is not satisfactory. Considering that 4 out of 8 bits of every pixel are removed, it means that in one way or another we may have data loss, and this can be seen by the number of colors available. After scalar quantization, we get only 16 colors out of 256 available.

Let's consider a simple case of 12 pixels. After performing the scalar quantization method, we would be left with 12 4 bit codes  4 bits for one pixel. After this compression we would still have

[9] Salomon, *Data Compression The Complete Reference,* 2004
[10] Salomon, *Data Compression The Complete Reference,* 2004
[11] Facsimile compression

the need to reconstruct or decompress the compressed pixels. This is where the problem arises. The supposed values of the 12 pixels are shown below:

11010111 11010110 11010101 11010011 11010010 11010001 11001111 11001110 11001101 11001100 11001011 11001010

By ‘four least significant bits’ we refer to the second half of the 8 bit pixel, which means we will simply cut off 4 bits from the pixel, resulting in the following bit codes for the pixels:

1101 1101 1101 1101 1101 1101 1100 1100 1100 1100 1100 1100

Even though we accomplished the process known as scalar quantization, we need to ‘retrieve’ information, we need to find a way to get back those 4 bits that we simply cut off. One possible solution is to add 4 ‘0’ bits at the end of each 4 bit code. At the end we would get 12 8-bit codes as follows:

11010000 11010000 11010000 11010000 11010000 11010000 11000000 11000000 11000000 11000000 11000000 11000000

By analyzing these codes carefully, one observation is that the first 6 codes are the same and the other remaining 6 codes are the same as well. This similarity among groups creates a band, easily visible to the human eye. It is usually preferable to avoid this kind of result. There is actually a way to avoid creating bands in an image because of compression. Instead of just adding 0’s at the end of the compressed pixels, the additional bits are chosen randomly, in order to avoid the creation of pixel bands in the decompressed image. Even though it offers a solution it is not necessarily the best one.

Until this point we have considered an input stream where a number of elements are frequent, which leads to compression. One hidden condition is that these items have to be different from the other ones in order to make a distinction and use Huffman Encoding to group the less frequent characters together and define short codes for the more frequent characters. However this is the case when the elements of the data stream are easily differentiated between them, due to the characters having different shapes, different pronunciation etc,. What happens when the data show similarity but are not completely identical? This is the instance when the problem arises. The principle of Huffman Encoding is to have a data stream and determine the probability distribution of the elements to construct the Huffman Tree. Characters will necessarily be different, and we do not have to worry about that.

Image compression tends to work in a different manner. If we select a pixel in an image, there is a high probability that the neighboring pixels are similar to the one we just chose, and represent

similar color bands.This phenomenon is also known as correlated pixels. In order to make sense of this similarity we need another way of analyzing the pixels in an image.

The following stream of numbers has the property that adjacent numbers have a small difference, making it very difficult to encode the numbers. Only two of all the numbers have a great difference.

12, 17, 14, 19, 21, 26, 23, 29, 41, 38, 31, 44, 46, 57, 53, 50, 60, 58, 55, 54, 52, 51, 56, 60[12]

As it is illustrated by the numbers themselves, there is not much room for compressing the data into codes. One approach is to take the difference between pairs.

12, 5, −3, 5, 2, 4, −3, 6, 11, −3, −7, 13, 4, 11, −4, −3, 10, −2, −3, 1, −2, −1, 5, 4

To make the result more understandable there is a graph shown below.

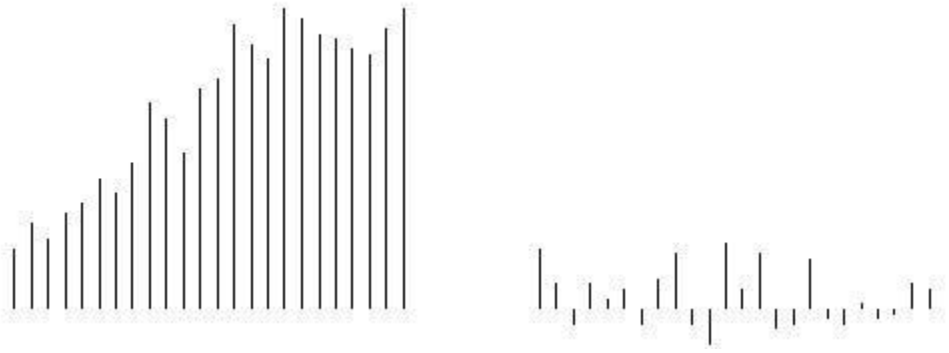

*Figure 3. Difference of Pixel Pairs*

This illustration shows that there is actually an opportunity to work with the data. The numbers towards the center (after taking pair differences) make up the best elements for encoding. After this step we have gotten rid of some similarity of the pixels (pixels in this example are represented by single numbers rather than a set of numbers for the purpose of simplicity). If the same step is repeated again a greater difference between the elements. The two images below show what is the result of taking the difference of the pairs two times.

[12] 24 adjacent pixels represented as single numbers for simplicity

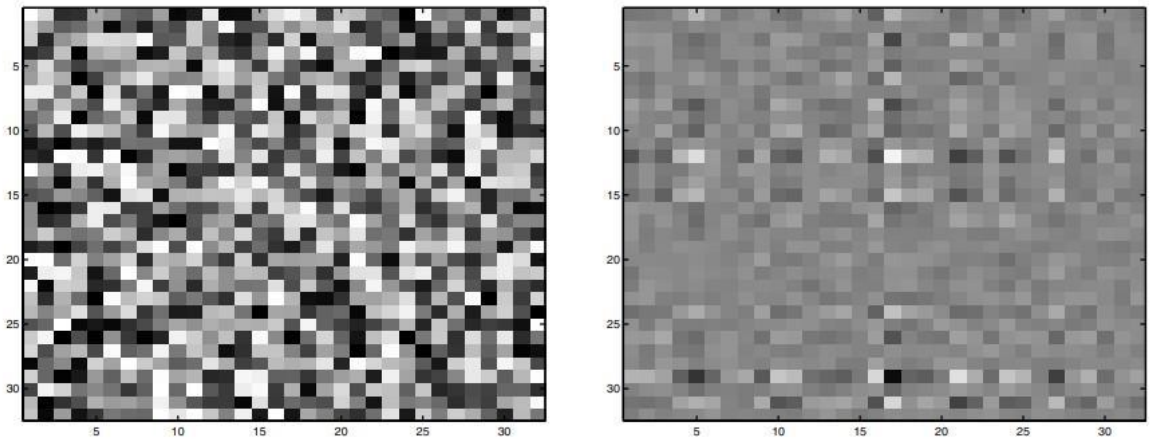


*Figure 4. Difference of Pixel Pairs*

## 5.2 Audio Compression

Compared to images, audio takes up less space. Depending on the number of colors used in an image, a single pixel occupies between one bit and three bytes, which can comprise a considerable amount of space keeping in mind that an image may have hundreds of thousands of pixels. Sound is quite a familiar phenomenon, since we hear it all the time. When we try to define sound, we find that we can approach this concept from two different points of view. Based on intuition we can define audio to be a sensation detected by our ears and interpreted by our brain in a certain way[13]. Scientifically, on the other hand, sound is a physical disturbance as it propagates through media such as water or air.

As mentioned above, sound propagates through different media, in the form of waves. The most important attributes of waves are speed, amplitude, and period. We are sensitive to only a certain range of frequencies, from 20 Hz to 22,000 Hz, depending on many physical factors. There are a lot of features we can mention in this paper, however the main focus of this section is audio compression. The scale by which we measure the loudness of sound is denoted by dB (pronounced decibel). It is somehow ambiguous to realistically understand what those numbers mean in the physical world, so the Figure below illustrates the level of loudness of real life objects.

[13] Salomon, *Data Compression The Complete Reference,* 2004

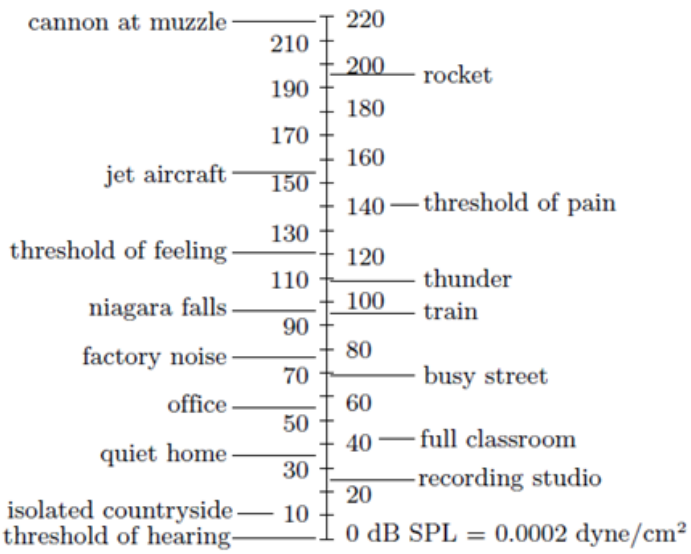


*Figure 5. Decibel levels of sounds*

Much as an image can be digitized and broken up into pixels, where each pixel is a number, sound can also be digitized. When sound is played into a microphone, it is converted into a voltage that varies continuously with time. The voltage in this case is the analog representation of the sound. Converting a sound into digital material is done by measuring the voltage at several points in time, and then translating each measurement into a number, and those numbers are then stored in files. It is quite practical to store sound in the form of numbers, since they are easier to operate from the mathematical point of view. On the contrary, the goal is not to just store the sound by using numbers just for the sake of studying it. The overall aim is to reuse the files with those numbers in order to recreate the sound, by converting those exact numbers into voltages. The higher the accuracy of keeping record of the sound (sampling) the better it will be reproduced.

The process of keeping record of sound by measuring the voltages and converting them into numbers is known with the term *sampling*. A high sampling rate would result in better sound reproduction, but also in many more samples and therefore bigger files. The main issue is that in terms of storage we cannot afford a high sampling rate, meaning that we cannot go over the sound many times. It is scientifically proven that the sampling rate should be at least twice the measure of the bandwidth of the wave of the sound. Such a sampling rate guarantees a true reproduction of the sound. Ideally, each sample will have a size, which normally becomes either 8 bits or 16 bits. Assuming that the highest voltage in a sound wave is 1 volt, an 8-bit sample can distinguish voltages as low as 0.004 volt, or 4 millivolts. A quiet sound, generating a wave lower than 2 millivolt, would be sampled as zero and played back as silence. In that case it would be more appropriate to use 16-bit sampling where the lowest distinguishable voltage would be 15 microvolts.

We can think of the sample size as a quantization of the original audio data. Eight-bit samples are more coarsely quantized than 16-bit ones. As a result, they produce better compression but poorer reconstruction (the reconstructed sound has only 256 levels). We are quite familiar with FM[14] and AM[15] when it comes to the radio. They refer to the method of sampling. The general term used for audio sampling is PCM[16]. PCM refers to the technique for converting a continuous wave to a stream of binary numbers.

### 5.2.1 Audio Compression Methods

RLE, statistical, and dictionary-based methods can be used to losslessly compress sound files, but the result depends on the sound used. RLE may work well when the sound contains long runs of identical samples. With 8-bit samples this may be common. Statistical methods assign variable-size codes to the samples according to their frequency occurrence (which was also used in Huffman Encoding). With 8-bit samples, there are only 256 different samples, so in a large sound file, the samples may sometimes have a flat distribution. Such a file will therefore not respond well to Huffman Encoding[17]. Consequently, with 16-bit samples there are more than 65,000 possible samples, so they may sometimes feature skewed probabilities. Dictionary-based methods expect to find the same phrases again and again in the data, in order to increment the value of the data encountered. This would work well with text, where certain phrases or haracters may repeat themselves and the dictionary is relatively easy to construct. However, it is not the case for analog signals (sound). For instance, in 8-bit samples, waves of 8mv may be represented by a size of 2, whereas the waves with a voltage value similar to the first one's may be represented by a different size, even though the difference may be quite small among the waves. Sounds in audio that shall appear almost the same to the human ear, would consequently be classified as different ones by the dictionary-based method and thus reproducing a different sound. Therefore, our mini-conclusion is that dictionary-based methods are not suited for audio compression.c

#### 5.2.2 Compression of data with loss

There is an approach from which we can get satisfactory results, and that is compressing the sound with a small loss. How does that work exactly? Scientifically it is not the right way, considering that data loss is the last thing we want. In this case we compromise in order to get a result with which we can work. As in image compression where we eliminate some of the pixels,

[14] *Frequency modulation*
[15] *Amplitude modulation*
[16] *Pulse Code modulation*
[17] *Recall from the **Huffman Encoding** section that it is important for the stream of data to have a different probability distribution for its elements (or said differently the frequency of occurrence). The more uniform the distribution is, the more complex it becomes to distinguish the elements/characters from each other with a unique code.*

we again allow some data loss by discarding the sound frequencies that our ears are not sensitive to. It is of importance to mention that there are two other sub-approaches in this case, silence compression and compandig[18]. The main idea of silence compression is to treat small samples as if they were actually silence. We have to keep in mind that this method takes advantage of the different sensibility levels of hearing sounds of different frequencies, as such tolerating the loss of the data that was not received to begin with. Companding on the other hand relies on the fact that more precise sounds are required at low frequencies and less precise sounds are allowed while compressing data/sound of higher frequencies. In other words compading[19] gets the meaning of compressing and expanding (depending on the usage). In this research paper we are not going to dive into technicalities of audio compression as it gets cumbersome the deeper we go.

## 5.3 Video Compression

Two guiding concepts govern video compression. The first is that every frame has some spatial redundancy. The second is that a video frame's near neighbors tend to resemble it quite a little. Temporal redundancy refers to this. So, a typical video compression approach should begin by encoding the first frame using a still image compression technique. Then, for each succeeding frame, it should determine and encode the changes between that frame and its predecessor.

It is best to code a frame separately from every other frame if it differs significantly from its predecessor (as the initial frame of a shot does). Inter frame (or just inter) and intra frame (or independent frame) are terms used in the video compression literature to describe how a frame is coded (or just intra). Most often, lossy compression is used for video. A frame's distortions are introduced when it is encoded in terms of its predecessor, frame Fi1. The distortion is thereby increased by encoding frame Fi+1 in terms of frame Fi. A frame might lose some bits even though video is compressed without loss. This may occur either during transmission or following a protracted shelf life.

A frame should be coded separately from any previous frames if it differs significantly from its predecessor (as is the case with the first frame of a shot). A frame that is coded using its predecessor is referred to as an inter frame (or just inter) in the video compression literature, while a frame that is coded independently is referred to as an intra frame (or just intra). Lossy video compression is the norm. There are some distortions introduced when encoding a frame Fi in terms of its predecessor Fi1. As a result, the distortion is increased when frame Fi+1 is encoded in terms of frame Fi. A frame may lose some bits even when compressed using lossless video. This can take place during transmission or following an extended shelf life.

[18] Approach for audio compression
[19] Another expression for Compressing and Expanding

If a frame Fi loses a bit, all frames up to the next intra frame are decoded incorrectly, potentially resulting in compounded mistakes. Intra frames should therefore be used throughout a sequence, not simply at the start. I stands for an intra frame and P for an inter frame (for predictive). The concept of an inter frame can be broadly applied once this concept is understood. One of its ancestors as well as one of its successors can be used to code such a frame. We are aware that an encoder should not use any data that the decoder cannot access, yet video compression is unique due to the enormous

The encoder often doesn't bother us, but the decoder must be quick. An example of this is playing back video that has been recorded on a DVD or hard drive. The data can be encoded by the encoder in a matter of minutes or hours. However, the decoder must be quick in order to replay it at the proper frame rate (a certain number of frames per second). Because of this, a standard video decoder operates in parallel. It features numerous decoding circuits that can process several frames at once. Now that we are aware of this, we can envision a scenario in which the encoder encrypts frame 2 based on both frames 1 and 3 and writes the frames on the compressed stream in the sequence 1, 3, 2. They are read in this order by the decoder, which simultaneously decodes frames 1 and 3 before outputting frame 1 and decoding frame 2 depending on frames 1 and 3. Of course, the frames need to be clearly marked (or time stamped). B is the name of a frame that has been encoded using both previous and subsequent frames (for bidirectional). When an object in the image gradually reveals the background, it makes sense to predict the next frame based on the previous one.

*Figure 6. Representations of frames*

There are a few intuitive video compression methods:

**Subsampling**: The encoder picks out every other frame and adds it to the stream that has been compressed. A compression factor of 2 is the result. A frame is input by the decoder, and it is duplicated to produce two frames.

**Differentiating**: A frame is contrasted with its forerunner. The encoder writes three integers for each individual pixel—its picture coordinates, the difference between the pixel's values in the two frames, and any other relevant information—on the compressed stream if the difference between them is minimal (a few pixels or less). If there is a significant gap between the frames, the current frame is output in raw format. Lossy differencing examines the magnitude of change in a pixel. A pixel is not regarded as different if the disparity between its intensities in the current frame and the frame before it is smaller than a predetermined threshold.

**Block Differencing**: It is a development over differencing. The pixels in the image are separated into blocks, and each block B in the current frame is compared to its matching block P in the frame before. B is compressed by recording its image coordinates, followed by the values of all its pixels (represented as differences), on the compressed stream if the blocks differ by more than a predetermined threshold. The benefit is that the block coordinates are short (smaller than a pixel's coordinates) and only need to be typed once for the entire block. The output must include the values of each and every pixel in the block, even those that are unchanged. But given that they

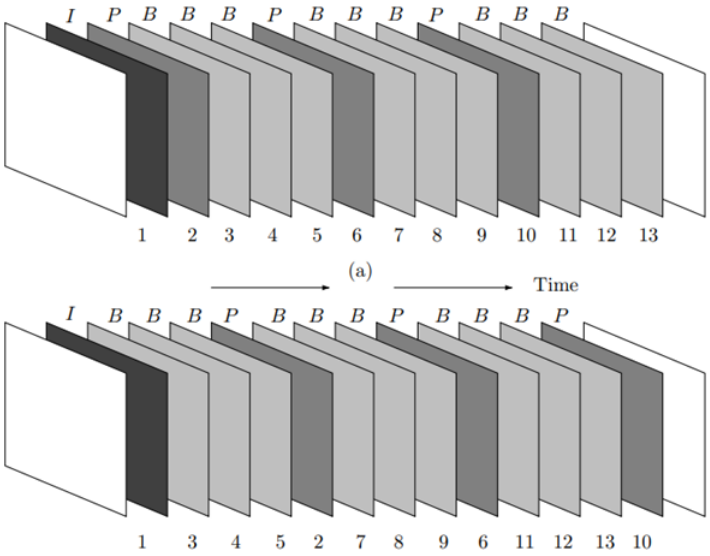


*Figure 7. Organization of Frames*

**Motion Compensation:** Anyone who has seen a movie before is aware that there isn't much of a change between two frames because the scene, the camera, or both have moved in between them. Therefore, it is possible to use this function to improve compression. The encoder can compress part P by writing its former position, present location, and information defining the bounds of P to the compressed stream if it finds that part P of the previous frame has been rigorously shifted to a different point in the current frame.

Such a component is theoretically open to any shape. In reality, we can only use equal-sized blocks (normally square but can also be rectangular). Block by block, the encoder reads the current frame. If lossless compression is desired, it looks for an identical block C (for each block B) or one that is similar in the preceding frame (if it can be lossy). When the encoder locates such a block, it outputs the difference between its past and present locations. This difference is known as a motion vector since it has the form (Cx Bx, Cy By) = (x, y). In a straightforward example, as the youngster moves, the sun and trees in Figure 6.10a,b are rigidly pushed to the right due to camera movement.

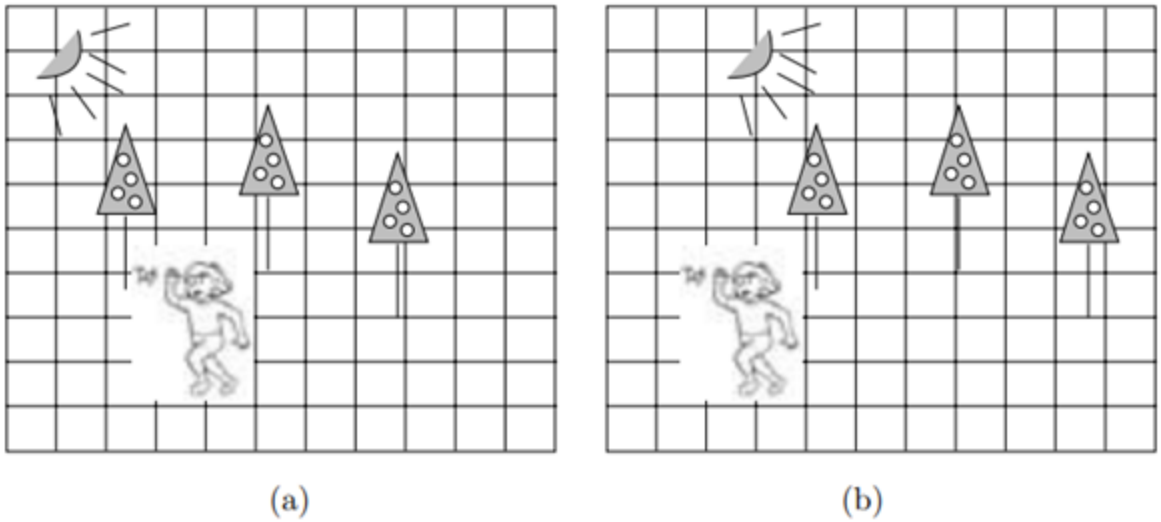


*Figure 8. Change in consecutive frames*

**Frame Segmentation**: There are equal-sized, non-overlapping blocks within the current frame. The blocks could be rectangular or squares. In the second option, it is assumed that most motion in video is horizontal, hence horizontal blocks fewer motion vectors without lowering compression ratio. The block size is crucial since both large and small blocks produce several motion vectors, which lowers the likelihood of finding a match. In reality, block sizes that are integer powers of 2, such 8 or 16, are employed because they make the software more user-friendly.

**Search Threshold**: There are equal-sized, non-overlapping blocks within the current frame. The blocks could be rectangular or squares. In the second option, it is assumed that most motion in video is horizontal, hence horizontal blocks fewer motion vectors without lowering compression ratio. The block size is crucial since both large and small blocks produce several motion vectors, which lowers the likelihood of finding a match. In reality, block sizes that are integer powers of 2, such 8 or 16, are employed because they make the software more user-friendly.

**Block Search**: This requires a lengthy process, therefore it must be well planned. If B is the current block in the current frame, then a block that is similar to or extremely close to B must be found in the previous frame. Normally, the search is limited to a narrow region (referred to as the search area) centered on B, as determined by the maximum displacement parameters dx and dy. The maximum horizontal and vertical distances, in pixels, between B and any matching block in the preceding frame are specified by these parameters. The search region contains (b + 2dx)(b + 2dy) pixels and is made up of (2dx + 1)(2dy + 1) separate, overlapping bb squares if B is a square with side b. There are consequently a total of candidate blocks in this area.

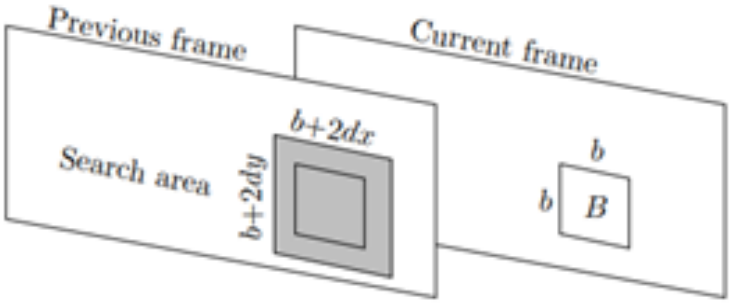


*Figure 9. Block search in frames*

**Distortion Measure**: The encoder's most delicate component is this one. The distortion measure chooses block B's ideal match. It must be quick, easy, and trustworthy at the same time. Below are a few options discussed. The average of the absolute differences between a pixel Bij in block B and its corresponding pixel Cij in candidate block C is calculated by the mean absolute difference (or mean absolute error):

$$\frac{1}{b^2}\sum_{i=1}^{b}\sum_{j=1}^{b}\left|B_{ij}-C_{ij}\right|$$

This requires one division, b2 additions, b2 subtractions, and absolute value operations. The least distortion, say for block Ck, is looked at after this measure is calculated for each of the (2dx + 1)(2dy + 1) unique, overlapping bb candidate blocks. Ck is chosen as the match for B if it is less than the search threshold. If not, B cannot be matched and must be encoded without motion correction. At this point, a reasonable query is, "How can such a thing happen?" How is it possible for a block in the current frame to match nothing in the frame before? Imagine a camera sweeping from left to right to find the answer. Constantly, new objects will come into view from the right. Thus, a block on the right side of the frame can have items that weren't present in the prior frame. A comparable measurement is the mean square difference, which calculates the square rather than the absolute magnitude of a pixel difference:

$$\frac{1}{b^2}\sum_{i=1}^{b}\sum_{j=1}^{b}\left(B_{ij}-C_{ij}\right)^2$$

### 5.3.1 Video Compression Methods

The main advantage of video compression standards is their capacity to provide interoperability, or communication, between encoders and decoders produced by various individuals or businesses. Standards reduce risk in this way for both consumers and manufacturers, which can speed up adoption and wide use. These standards are also made for a wide range of applications, which results in economies of scale that lower costs and increase usage. The popular families of video compression standards are displayed in Table 1 (Current and Emerging Video Compression Standards), which was carried out by the International Telecommunications Union-Telecommunications (ITU-T, formerly the International Telegraph and Telephone Consultative Committee, CCITT), the International Organization for Standardization (ISO), and the Moving Pictures Expert Group (MPEG), which was founded by the ISO in 1988 to develop a standard.

The H.261 standard was the first to be widely adopted for video compression. Video over ISDN can be sent using H.261 and H.263. They are utilized to deliver video over meager bandwidths (Marcel et al., 1997). For applications that need larger bit rates, the MPEG standards offer a variety of compression formats. Standard VHS grade video compression is offered by the MPEG-1 standard. The MPEG-2 is capable of handling applications requiring data rates of up to 100 Mbps and is easily adaptable to applications for digital television transmission.

| Video coding standards | Year developed | Publisher | Primary Intended Applications | Bit rate |
|---|---|---|---|---|
| H.261 | 1990 | ITU | Video telephony and teleconferencing over ISDN | $p \times 64$ kb $sec^{-1}$ |
| MPEG-1 | 1991 | ISO/IEC | Video on digital storage media (CDROM) | 1.5 Mb $sec^{-1}$ |
| MPEG-2 | 1994 | ISO/IEC | Digital television | 2-20 Mb $sec^{-1}$ |
| H.263 | 1996 | ITU | Video telephony over PSTN | 33.6 kb $sec^{-1}$ and up |
| MPEG-4 | 1998 | ISO/IEC | Object-based coding, synthetic content, interactivity, video streaming | Variable |
| MPEG-7 | 2001 | ISO/IEC | Real-time and non-real time applications, to tag the contents and events of video streams for more intelligent processing in video management software or video analytics applications | Variable |
| H.264/AVC | 2003 | ITU-T/ ISO/IEC | Improved video compression | 10's to 100's of kb $sec^{-1}$ |

*Table 2. Video Compression Methods*

# 6. HUFFMAN CODE IMPLEMENTATION

### *General Pseudocode*

Huffman Encoding/Decoding pseudocode for the implementation

1. Get the input data from the GUI application or read it from a file
2. Pass the inputted string as arguments for the Encode(parameter)

***Encoding***

1. Calculate the frequency of each character in the string
2. Sort the characters in increasing order of the frequency (Priority Queue)
3. Make each unique character as a leaf node
4. Create an empty node and assign the minimum frequency to the left and the second minimum to the right
5. Remove them from the priority queue and insert the node we created into the tree and repeat , for each non leaf assign 0 to the left and 1 to the right edge

***Decoding***

1. Traverse through the tree to get all the 0's and 1's from the root to get the decoded values

***Time complexity for encoding*** : O(log n)

## 6.1 C++ implementation

To support the theoretical aspect of this research paper we saw it fit to have some code implementation for Huffman Encoding. In total we did 3 different implementations, each serving a different purpose in this project. Initially, we assumed that we have a stream of characters, or a string that is going to be compressed using Huffman Encoding. The algorithm for the compression technique is explained in the preceding sections, so we are going to create a tree-like structure using C++.

Initially we ask the user to enter a string as an input and then call the function buildHuffmanTree(text), which has the string as a parameter. We are going to explain the implementation in simple terms. Basically we have to construct a tree-like structure in order to get the compression correct. There is a struct we have used, which represents the node of the tree having the as variables a character, the frequency and pointers to the left and right nodes (if there are any). The structures we have used are a map and a priority queue. At the beginning, when we get the string, we count the frequencies of each character and store them in a map, which is unordered. Then we create a priority queue for each character assuming that each character is a leaf node initially without pointing anywhere.

Since we used a priority queue, the top node or element is always going to be either the greatest one in the set or the smallest, by default it is the greatest. We pop the top two elements in the priority queue, find their sum (the sum of the frequencies of their characters) and then push the

newly created node in the priority queue. Another function is then called to encode the characters with their respective codes and then we traverse the priority queue to print the encoded characters. What the function encode does is that it traverses the tree recursively, adding a 0 when it goes to a left node and then adding a 1 when going to a right node. At the end we have the encoded string for the initial string we provided as input. The decode function works quite similarly, it reads the encoded string, which will lead us through the huffman tree and the initial text will be printed. This is the first implementation where we demonstrate Huffman Encoding.

## 6.2 Python Implementation

### 6.2.1 Method 1

In this first implementation in python I have used tkinter for the user interface, which even though is simplistic serves the purpose. The steps are quite similar to the first implementation using C++, however we have a slight difference here. The encoded string is stored in binary files and then read from the binary file to be presented to the user, in order to not have any changes in the process of encoding and decoding.

I have created a class called AnotherHuffman, it contains a path for the file, heap, codes as a dictionary, reverse mapping (to make the whole process easier), and the different forms of the string. When we get the initial string as input (the string that will be encoded). The heapq is used for the same reason as in the previous implementation, to make it more practical in use. The only difference is that when we store the bit string into a binary file, we have to add some padding, if the length of the encoded string is not a multiple of 8, we have to add some 0's to make it a multiple of 8 and then write it as an array of bits into a binary file. The figure below shows what happens when we run the program. It is self explanatory.

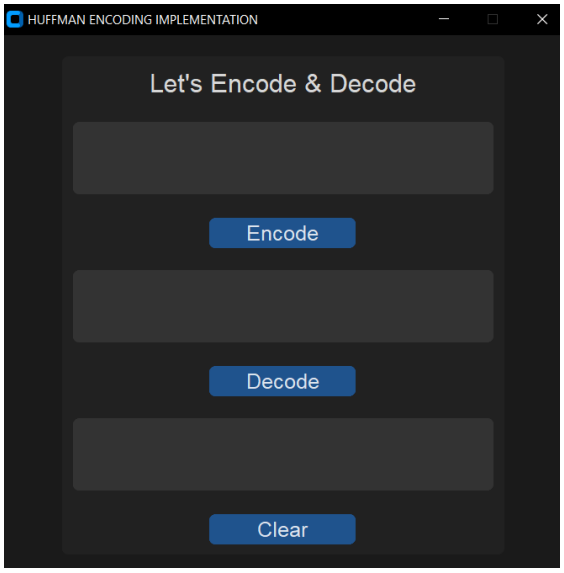


*Figure 10. Huffman Implementation in Python*

### 6.2.2 Method 2

From the previous implementations we have seen how Huffman Encoding works, but we have not really measured the performance when we provide a string as input. In this implementation we have not used files and have not saved the encoded string in binary files, we just use the input text field in the text area as a source, and then use Huffman Encoding on the provided string. The UI will print the representation of the codes in this encoding technique. There is another added section that represents the length in bits of the text prior to being encoded and after being encoded using Huffman Encoding. This approach is slightly modified from the first one. It is different in structure but follows the same logic as the first one. Below is the figure that represents when the program is run.

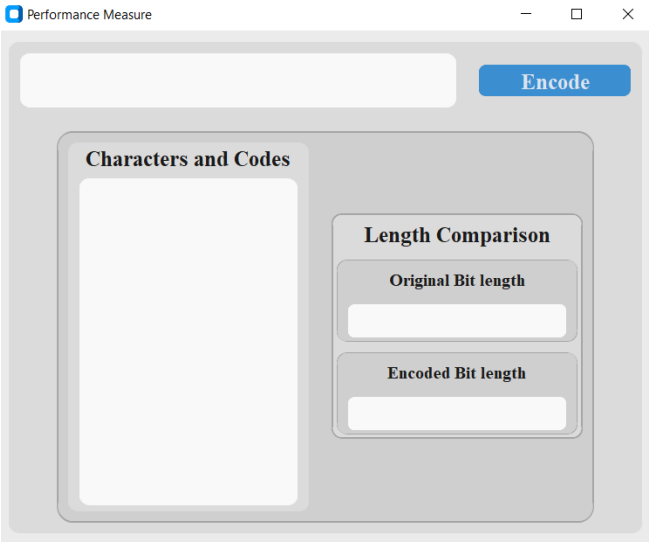


*Figure 11. Performance Measure with Python*

# 7. MEASUREMENTS

It is not quite that simple to provide satisfactory results with the measures we can make by using the shallow knowledge and the simple implementations we have created. The second implementation of Huffman Encoding in Python can be used to analyze the behavior of this encoding technique. Let's say we take several string inputs to encode and see the results. The two strings that we are going to use are: (a) Hi there (b) Computer Networks.

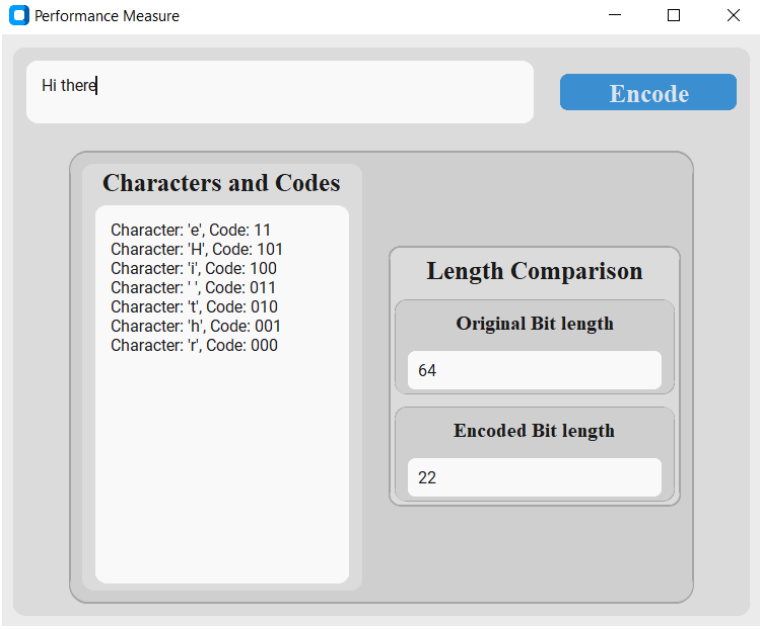


*Figure 12. Huffman Encoding example*

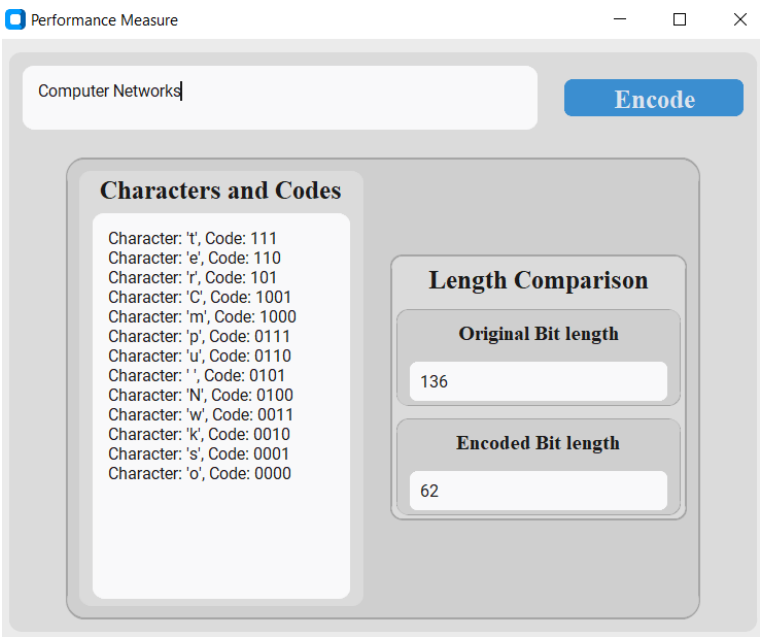


*Figure 13. Implemented example in Python*

From the results that we got it is easy to jump to conclusions and say that Huffman Encoding is highly efficient, but is it really the case? As it is mentioned in the preceding sections, when the probability distribution of the characters in the string is flat, it means that the encoded result will not meet the requirements to be considered a successful technique. We took a paragraph from the book "Computer Networks" and tried to encode it using our code implementation in python. Below are the results.

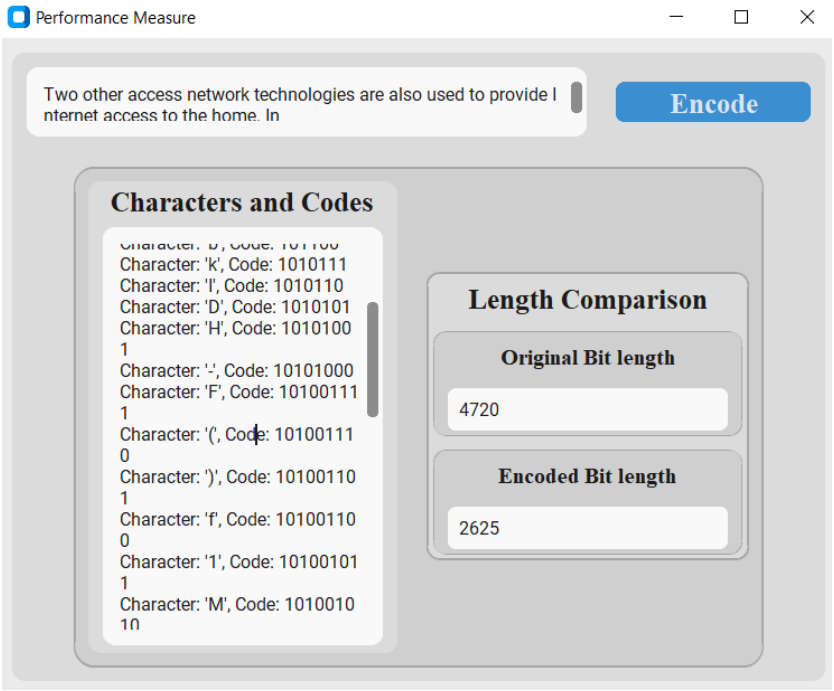

*Figure 14. Encoded Paragraph*

There are two perspectives in this result we got. In terms of compression, our code implementation achieved good results, but in terms of character representation it has failed in some cases. For the character 'F' it has used 9 bits for the encoding (a character is usually represented in the 8-bit format). This is one of the drawbacks of Huffman Encoding, it depends on the input we provide, it may use significantly less bits for one character, but at the same time, more bits for another character.

# 8. CONCLUSIONS

After considering all the topics covered in this research paper, it is of importance to point out some of the conclusions. Huffman Encoding can be used in many applications, but there is no guarantee that the performance will be of satisfactory results, the reason being that the outcome depends solely on the input. Probability distribution of the text input can vary greatly, due to the reason that we have to consider all the possible cases for the characters, as there may be certain unconsidered conditions.

**Conclusion (a):** Huffman Encoding, in terms of data compression, can provide a basis for communication improvement. Characters are typically represented by 8 bits. In Huffman Encoding this rule is not obeyed, for the better, achieving compression.

**Conclusion (b):** Different compression techniques do not perform well under certain conditions. There need to be certain modifications in order to guarantee the least worst outcome, such as in the case of the unbalanced Huffman Tree in DNA (which was just a theoretical implication).

# 9. PROPOSAL

Compression is essential in today's communication systems, especially when heavy traffic in the network becomes problematic. If we have to transmit some data from one host to another host, the data is separated into chunks and then assembled at the receiving end of the network. The greater the information we have to transmit, the greater the number of data chunks. If the data was compressed, the number of data chunks is consequently less in number. The Huffman Tree has to be part of the data chunks, it can be added as overhead, or it can be part of the first packets to be sent, in order for the receiving end to start processing the data the packets are sending immediately.

We haven't taken into consideration the work a host has to perform in order to decode the encoded string/data. Several factors should be considered to determine whether this approach is

beneficial or not, and if it should be implemented for further research. Our proposal is to use a server for each subnet in the network to deal with the decoding, so that each host does not have to deal with computation obstacles just because of encoding. In this way the big network itself performs faster as less packets circulate in between routers, hosts and servers. The question is till if this modification is beneficial or not. We can let the server decide, if it is text of a small size, it can send it directly to the host and it can perform the operations of decoding itself. On the other hand, if the string is of great size, it can be decoded by the server. If a large number of hosts are waiting for their data to be decoded, the server can implement parallel programming techniques, such as distributed memory programming, which makes it a lot easier for the server to deal with several decoding operations at the same time. The figure below shows the theoretical implementation of the proposed model, where each subnet has its own server to deal with encoding and decoding data if the computers do not have the proper requirements.

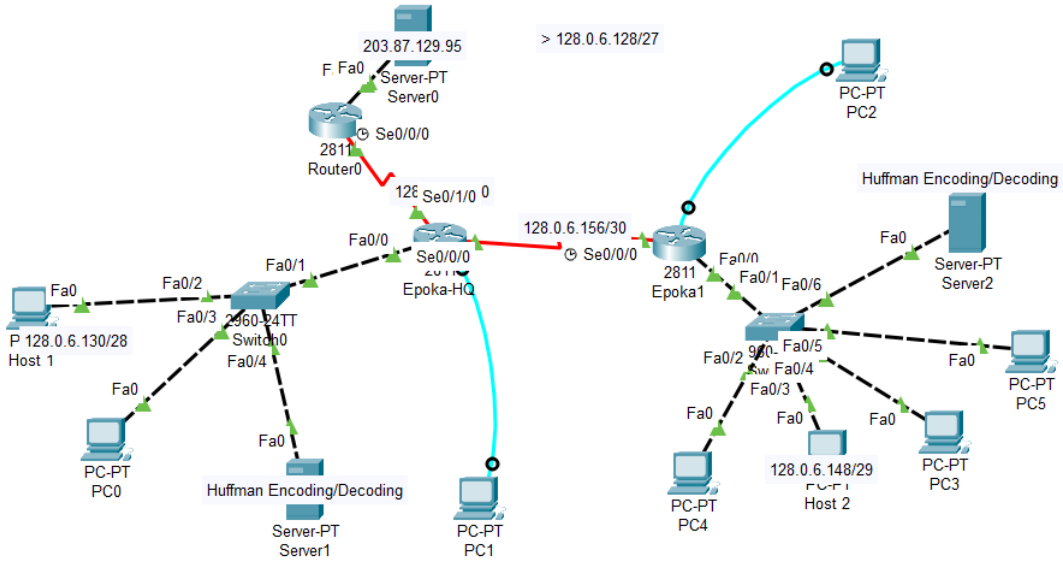


*Figure 15. Proposed Network Model*

This is the proposed solution. The performance of the network has to be determined if it is implemented. The messages have to pass through the server first and a choice has to be made. The server is the one that has to deal with the encoding and decoding. It is a lot easier for the server to encode than decode messages, for the reason that for every decode operation it has to follow instructions provided by the sender, and they tend to be different based on the data that they are sending. In short terms, the hosts may be unaware of the encoding and decoding part, whereas the server is responsible to coordinate the communication of the subnet with the entire network.

# Appendix A

# References

Salomon, David. “Data Compression.” *The Complete Reference*, 2006. *Bowker*

Yeung, Raymond W. *Information Theory and Network Coding*. 2008.

Harris, Greg A., et al. *Introduction to Information Theory and Data Compression*. 2003. *Bowker*

Al-Okaily, Anas, et al. “Toward a Better Compression for DNA Sequences Using Huffman Encoding.” *PubMed Central (PMC)*, 1 Apr. 2017

“Huffman Coding | Greedy Algo-3 - GeeksforGeeks.” *GeeksforGeeks*, 3 Nov. 2012

*Huffman Coding*. (n.d.). Huffman Coding. Retrieved January 30, 2023,

MIT News Office, L. H. (2010, January 19). *Explained: The Shannon limit*. MIT News | Massachusetts Institute of Technology.

Manubolu, S. (2021, December 14). *Huffman Coding on Image*. Medium.

*David A. Huffman Collection - MIT Museum*. (n.d.). MIT Museum.

Gray, R. M. (2013, March 14). *Entropy and Information Theory*.

*Discovery of Huffman Codes | Mathematical Association of America*. (n.d.). Discovery of Huffman Codes | Mathematical Association of America.