Building data compression application using Huffman coding

Abstract

As civilization evolves, the data is in exponential growth. The ubiquity of Internet-based applications, coupled with emerging technologies like the Internet of Things (IoT), artificial intelligence, cloud computing, and machine learning, has led to an unprecedented surge in the volume of data transmission and storage. Managing this influx of data has become a paramount challenge. Scientists and engineers are working closely to develop new and improve existing technologies used to compress the data to increase the speed of transmission, reduce the cost of storage, and ultimately reduce the total cost of the maintenance of the data. Although there are numerous ways of compressing the data, this research paper focuses on the pivotal role of Huffman coding in addressing the challenges posed by rapidly increasing data.

Introduction:

Establishing the significance of data compression in the contemporary digital landscape. Highlighting the surge in data volumes propelled by IoT, artificial intelligence, and machine learning applications.

Background:

Even though many believe that data compression is a new branch in the field of technology, the reality is that the first signs of data compression are coming from 1838 when the Morse code was invented. As the earliest example of data compression, morse code used shorter codewords for letters such as “e” and “t” which are very common in the English language. More than 100 years later, the modern era of data compression began with a significant development of the information theory. In 1977, Abraham Lempel and Jacob Ziv proposed the basic concept of pointer-based encoding. By the mid-1980s, after work by Terry Welch, the LZW algorithm quickly became the preferred method for most general-purpose compression systems. It found use in various programs like PKZIP and hardware devices like modems. As digital images became more widespread in the late 1980s, standards for compressing them started emerging. In the early 1990s, lossy compression methods also gained widespread use. Current image compression standards include FAX CCITT 3 (using run-length encoding with codewords

determined by Huffman coding based on a specific distribution of run lengths), GIF (LZW), JPEG (lossy discrete cosine transform followed by Huffman or arithmetic coding), BMP (using run-length encoding, etc.), and TIFF (FAX, JPEG, GIF, etc.). Typical compression ratios achieved for text are approximately 3:1, for line diagrams and text images around 3:1, and for photographic images around 2:1 for lossless compression and 20:1 for lossy compression. [3 REFERENCE TODO]

Over the years, two main methods of data compression emerged - lossy and lossless.
With lossless compression, every bit of a file is restored after uncompression. This is very useful in scenarios where the quality of the data should be maintained. For example, text files will not lose a single letter after they are decompressed. Another example is compressing images. After decompression, images will maintain their original quality. With this method, the file sizes are reduced, but not as much as they are with Lossy compression.
In Lossy compression, the unnecessary data is permanently deleted. After the decompression, the data is not restored to its original form. Because the data can not be returned to its original form, this compression is known as irreversible compression, whilst lossless compression is known as reversible compression. Lossy compression is mainly used in scenarios where some amount of loss of information is not important, such as images, videos, and sound.

This paper focuses on Huffman Coding which originally started in 1951. That year, David A. Huffman and his classmates at MIT studying information theory had to choose between writing a term paper or taking a final exam. Robert M. Fano was the professor who assigned them a term paper about finding the best and most efficient way to create a binary code. Huffman was stuck for quite some time, unable to prove that any of the given codes were the best choice. He almost gave up and considered studying for the final exam instead. Suddenly, he had a breakthrough idea. Instead of using the traditional method, Huffman thought of organizing the binary code based on how often each symbol appeared (this will be discussed more in Fundamentals of Huffman Coding and Entropy). He quickly proved that this approach, using a frequency-sorted binary tree, was the most efficient way. In achieving this, Huffman surpassed Fano, who, along with Claude Shannon, had developed a similar code. Huffman's bottom-up method of building the tree guaranteed optimality, a better approach than the top-down method used in Shannon–Fano coding. [4 REFERENCE TODO]

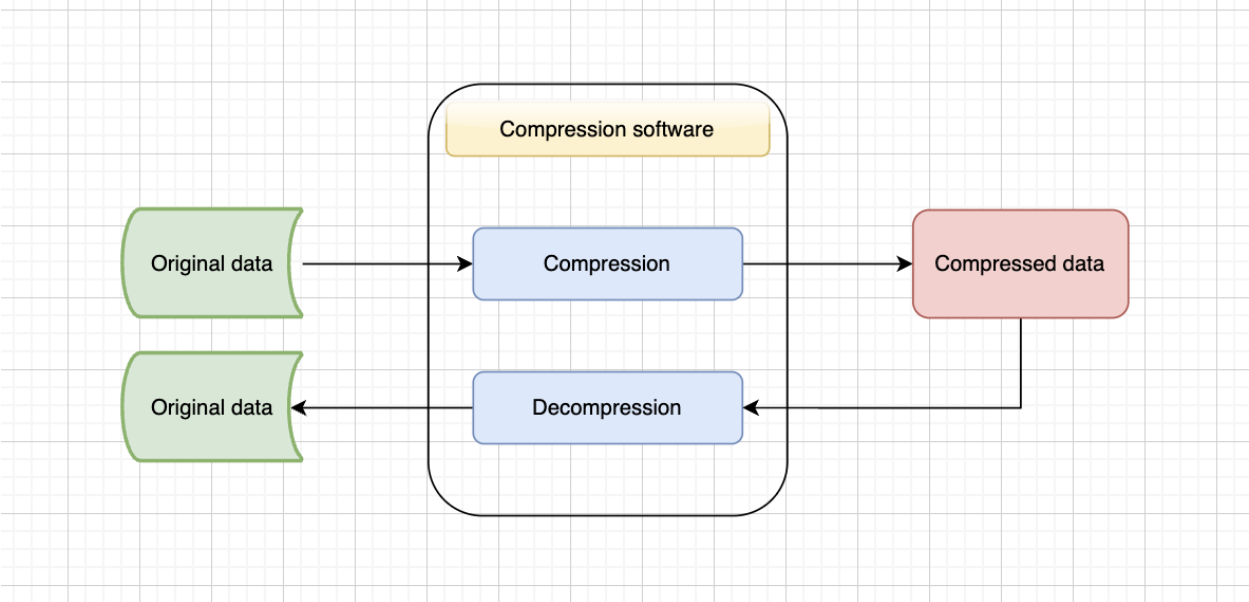
Fundamentals of Huffman Coding

In computer science and information theory, there's something called Huffman coding, and it is a special kind of code that's good for compressing data without losing (lossless data compression) any of it. So, what does Huffman coding do? Imagine you want to send a message, and you want to make it as short as possible. Huffman's idea is to give shorter codes to the things that show up more often in your message. For example, if the letter "e" appears a lot, it gets a shorter code. This way, the whole message becomes shorter. Huffman's way of doing this is smart and doesn't take too long. It makes a kind of table that says how to turn each part of the

message into a special code. This table is based on how often each part of the message shows up. The more common things get shorter codes. Even though there are other ways to make messages shorter, like arithmetic coding, Huffman coding is still really good at what it does, especially when you want to give different codes to different things in the message. [5 REFERENCE TODO]

Building Huffman codes

In the figure below, the core flow of Huffman coding is presented. On the left side, we see the original data that we want to store. Instead of storing it directly, we need compression software (algorithm) that will transform the original data into a compressed file by reducing its bit size. As we mentioned above, Huffman coding is a lossless, reversible type of data compression. This means we should have a way to return the compressed data in the original form without losing any single bits of it.



Cost of normal message

To understand the best Huffman coding, we will start first with understanding the cost of a normal message. Imagine we have a message “BCCABBDDAECCBBAEDDCC” that we want to encode and send over the network. The first thing that we can determine is the length of the message, which is in this case 20 letters. How this message can be sent? It has to be sent using ASCII code. Computers and electronic devices use ASCII code for characters or in this case English letters. Each ASCII code can be represented within 8 bits. In this message, we used only the letters A to E, to simplify the examples and understand easier how compression is done. The single letter A is represented as 65 in the ASCII code, which in the binary system can be represented as 01000001. Letter B in the ASCII code can be presented as 66, or in the binary system as 01000010. In the table below, we can see the representation of each one of the letters we used in the message above.

English letter	ASCII code	8-bit Binary
A	65	01000001
B	66	01000010
C	67	01000011
D	68	01000100
E	69	01000101

From the table above, we can see that 8 bits exist for each letter. Considering that in the original message, we have 20 letters, we can easily calculate the total number amount of bits needed for the message. The calculation formula is as follows:
$Total\ number\ of\ bits\ =\ 8\ *\ total\ number\ of\ letters\ in\ message$.
In the formula above, the number 8 represents the number of bits required to present ASCII code. In our example, the total number of bits = 8 * 20 = 160 bits.
The greatest advantage of this approach is that it is standardized and easy to decode. However, the main disadvantage is the length of the uncompressed messages.

Fixed-length encoding

Fixed-length encoding refers to a type of data encoding where each element of the data is represented using the same number of bits or bytes, regardless of the actual value. In fixed-length encoding, each data element is allocated a fixed amount of storage space, and any unused space is typically filled with padding.
In our example, we are not using all possible English letters or all possible characters we can find on the keyboard, or in general, we are not using all possible characters from the character set. In our message “BCCABBDDAECCBBAEDDCC”, we are only using the first 5 capital letters of the English alphabet. Based on that, we see that we do not need 8-bit codes to represent just 5 letters. In essence, we only need a few bits to represent all of them. In Fixed-length encoding, we are allowed to use our own fixed-size codes. If we had only 1 bit, we could write either 0 or 1, meaning with 1 bit we can represent only 2 characters. If we had 2 bits, then we could have pairs of 00, 01, 10, 11. In total, we could represent 4 letters. Because we have 5 letters, we will need 3 bits to represent all of them.

Character	Count	Frequency	Code
A	3	3/20	000
B	5	5/20	001
C	6	6/20	010

D	4	4/20	011
E	2	2/20	100

With the table above, we present our 3-bit codes for characters. We will now use these codes to encode the message from above in order to reduce its size.
$BCCAB\ldots = 001\ 010\ 010\ 000\ 001\ldots$
As we know, the message is 20 characters long. Each character is represented as a 3-bit code. The total size of the message is: $length\ of\ the\ message\ *\ number\ of\ bits$
In our example, the size of the message is 20 * 3, which is equal to 60 bits.
However, this is not the total cost of the message. We need to consider the receiver of the message. If we just send the message like this, the receiver will not be able to decode and understand the message using plain English letters. To prevent this, alongside the message, we should send the table of codes used for encoding. This way, the receiver will be able to decode its content easily. In this case, the total size of the message will not be 60 bits. We need to send each letter from the table. We know from the previous example that each letter can be represented in ASCII code which is 8 bits. Since we have 5 letters, the total length of the letters is 5 * 8 (bits) which equals 40 bits. We also need to send the codes used for each letter. We have 5 letters, which means that we will have 5 codes. Each code is 3-bit in size. The total size of all codes is 15 bits. Now, we should recap everything we have so far, and calculate the total length.
$Encoded\ Message\ size = 60\ bits$
$Table\ of\ Characters\ size = 40\ bits$
$Table\ of\ Codes\ size = 15\ bits$
In total, the message is 115 bits in size. Comparing it to the normal message example, where the exact same message was 160 bits in size, we can already see significant improvement.

Advantages and disadvantages of fixed-length encoding

Advantages

1. Simplicity: Fixed-length encoding is straightforward and simple to implement.
2. Random Access: It enables random access to elements within the encoded data structure, as each element has a fixed position.

Disadvantages:

1. Wasted Space: Padding can result in wasted space, especially if many values don't use the full allocated space.
2. Not Suitable for Variable-Length Data: It is not suitable for encoding data where the length of individual elements varies widely.
3. Inefficiency: It can be inefficient in terms of storage space for data with a wide range of values, as some values may require fewer bits.

How it relates to Entropy

Detailing the core principles and mechanisms behind Huffman coding.
Establishing the connection between Huffman coding and information entropy.
Exploring how Huffman coding achieves entropy-based compression by assigning shorter codes to more frequent symbols.

Applications of Huffman Coding:

Analyzing real-world scenarios where Huffman coding finds practical utility.
Showcasing its effectiveness in diverse domains such as image and text compression.

Performance Evaluation:

1. COMPARE How it works with compression and without compression on 100k requests.
2. Presenting empirical studies and performance comparisons with other compression algorithms.
3. Discussing the trade-offs and limitations associated with Huffman coding.

Future Directions:

MAYBE MENTION GRPC?

Identifying potential advancements and avenues for further research in Huffman coding.
Considering its adaptability to emerging technologies and evolving data landscapes.

Conclusion:

Summarizing the key findings and contributions of the research.
Emphasizing the ongoing relevance and future prospects of Huffman coding in the dynamic field of data compression.

Sources

1. General about data compression
2. Data compression use cases (Here also mentioned Huffman coding, nice for transition)
3. Stephen Wolfram, A New Kind of Science (Wolfram Media, 2002), page 1069.
4. Huffman, Ken (1991). "Profile: David A. Huffman: Encoding the "Neatness" of Ones and Zeroes". Scientific American: 54–58.
5. https://en.wikipedia.org/wiki/Huffman_coding#Basic_technique