

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

In our increasingly digitalized existence, the constant influx of data has become a defining characteristic of the contemporary era. The intricate dance between interconnected devices, the marvels of artificial intelligence, and the insights drawn from machine learning applications collectively contribute to an unprecedented surge in data generation. As we stand on the precipice of a digital revolution, the management, transmission, and storage of this colossal volume of data emerge as paramount challenges.

At the forefront of addressing these challenges is the unsung hero – data compression. This technological wizardry enables us to navigate the complexities of our data-driven world more efficiently. In this exploration, we seek to unravel the multifaceted significance of data compression, shedding light on why it holds a pivotal role in the landscape of modern technology.

Imagine a world where the internet seamlessly connects your devices, where artificial intelligence not only aids decision-making but also refines its understanding over time, and where machine learning algorithms continuously glean insights from data patterns. These scenarios paint a vivid picture of the vast data landscape we find ourselves in. Without the magic of data compression, this landscape could quickly become an overwhelming and unwieldy terrain, fraught with inefficiencies in data transmission and soaring storage costs.

Our journey into the realm of data compression goes beyond its role in the Internet of Things (IoT). It extends to the core of our digital existence, touching every aspect where data plays a pivotal role. From the efficiency of data transmission to the optimization of storage space, data compression emerges as a silent force that keeps our digital world running smoothly.

In this narrative, we'll navigate the challenges presented by the surge in data volumes, driven not only by IoT but also by the pervasive influence of artificial intelligence and the ever-evolving capabilities of machine learning applications. Amidst the myriad of compression techniques, one luminary stands out – Huffman coding. Born out of a student's ingenious breakthrough in 1951, this coding technique has become an integral part of the data compression landscape. As we delve deeper, we'll uncover the elegance of Huffman coding and its unique contribution to efficiently managing the deluge of data in our digital age.

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]

Cost of the 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 “BCCABBDDECCBBAEDDCC”, 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... = 001 010 010 000 001...

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

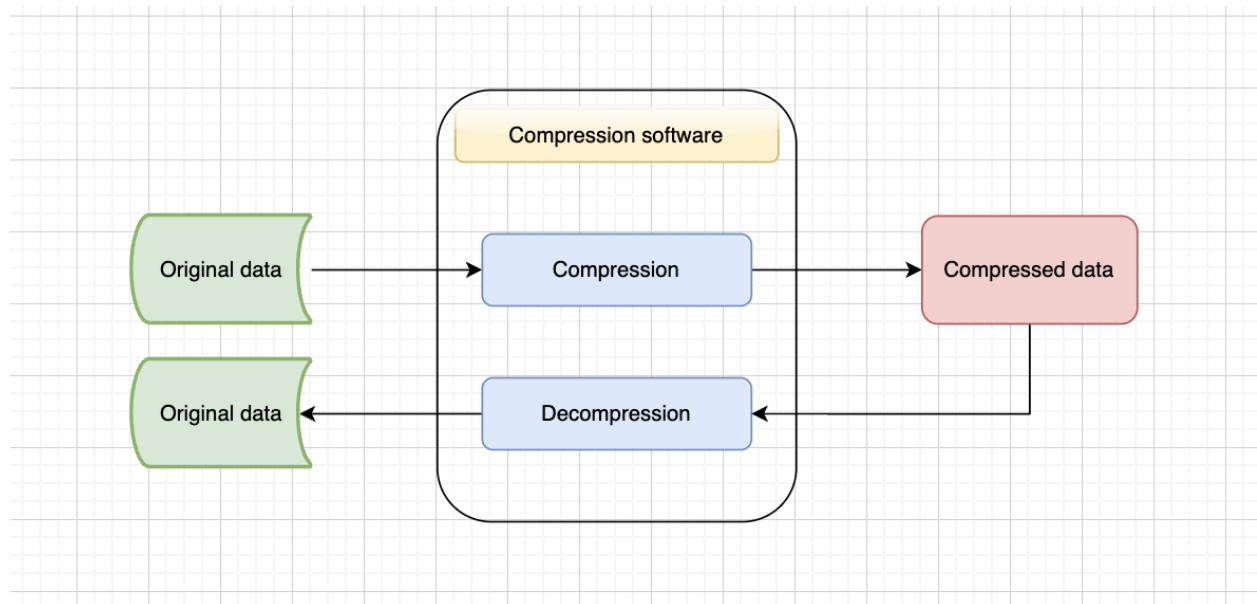
1. **Simplicity:** Fixed-length encoding is straightforward and simple to implement.
2. **Random Access:** It enables random access to elements within the encoded data structure, as each element has a fixed position.

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.

Huffman coding

In Figure 1, the simplified, general 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.



(Figure 1)

One of the primary reasons why Huffman encoding surpasses fixed-length encoding is its ability to achieve variable-length codes. In fixed-length encoding, each symbol is allocated the same number of bits, regardless of its occurrence frequency. This leads to a waste of bits on more common symbols and insufficient bits for less frequent ones. Huffman coding dynamically adjusts the code lengths, ensuring a more balanced and efficient representation of the dataset. Consider a simple example with an alphabet consisting of four symbols: A, B, C, and D. In fixed-length encoding, if each symbol is assigned 3 bits, the codes could be A: 000, B: 001, C: 010, D: 011. However, if the frequency of occurrence is A: 8, B: 6, C: 4, D: 2, the fixed-length approach results in inefficiency. Huffman encoding, on the contrary, would assign shorter codes to A and B and longer codes to C and D, reflecting their respective frequencies. Efficiency in terms of storage and transmission becomes particularly crucial in large datasets or when dealing with bandwidth constraints. Huffman encoding excels in these scenarios by minimizing the average code length, leading to a more compact representation of the data. The adaptability of Huffman coding ensures that symbols occurring more frequently benefit from shorter codes, reducing the overall size of the encoded data. Moreover, the adaptability of Huffman encoding makes it resilient to changes in the source data. In applications where the dataset dynamically evolves, fixed-length encoding might struggle to maintain efficiency. Symbols that were initially rare may become more frequent, or vice versa, disrupting the balance established by

fixed-length codes. Huffman encoding, with its ability to dynamically adjust to changing frequencies, remains effective across a broader spectrum of scenarios. Another aspect where Huffman encoding outshines fixed-length encoding is in scenarios where data follows a skewed distribution. If certain symbols occur significantly more often than others, fixed-length encoding can lead to substantial inefficiencies. Huffman coding, being inherently variable, can accommodate such skewed distributions more effectively, ensuring optimal compression ratios.[6]

In the following example, we will take the same message ("BCCABBDDAECCBBAEDDCC") from previous examples, and analyze it. As we know, the Huffman method does not require to use of fixed-size codes for alphabets. First, we create a table of the alphabet with the count of how many times each character is appearing.

Character	Count	Code
A	3	
B	5	
C	6	
D	4	
E	2	

This table is just showcasing the total amount of how many times each character appears. It does not have to be sorted in any particular order. However, the binary code is missing in this table.

We need to generate our own code. First, we need to create a leaf node for each unique character. In order to achieve so, first we need to create a table in ascending order. This means that the letter E will appear first, followed by the letter A, until we reach the letter C.

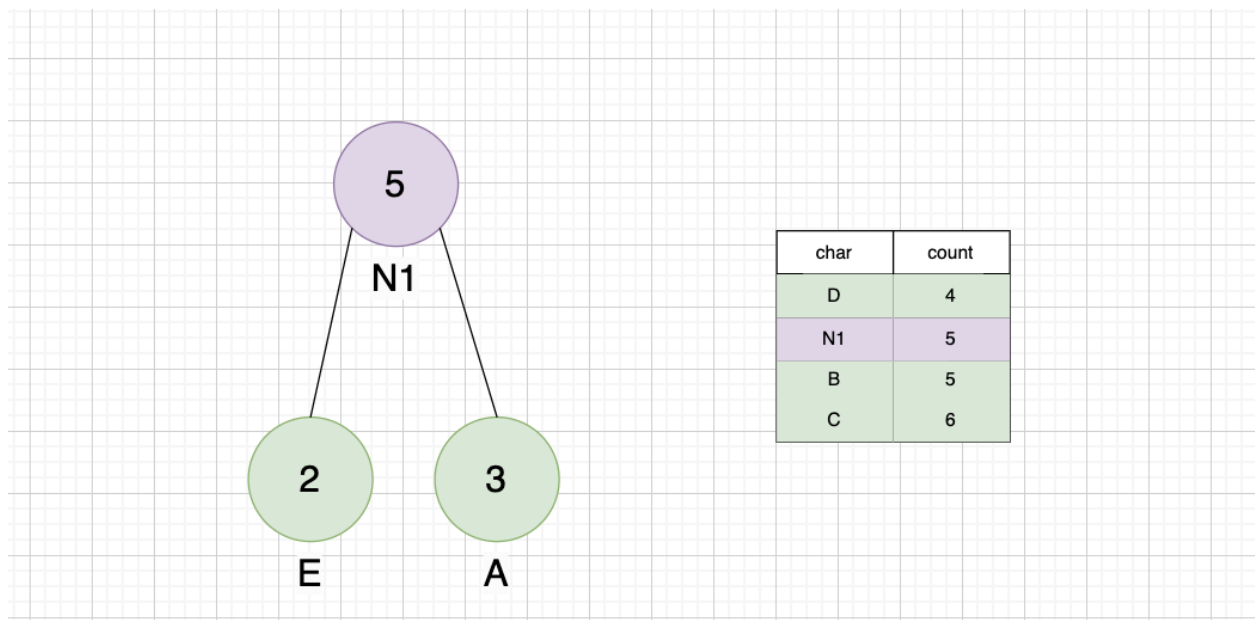
Character	Count
E	2
A	3
D	4
B	5
C	6

Building Huffman Tree

Step 1: To initiate the construction of a Huffman tree, individual leaf nodes are generated for each unique character present in the dataset. These leaf nodes serve as the elemental components of the upcoming tree structure. Simultaneously, a min heap is established, organizing the leaf nodes based on their frequencies. This initial step lays the groundwork for subsequent stages in the Huffman tree-building process, ensuring an organized approach to handling character frequencies.

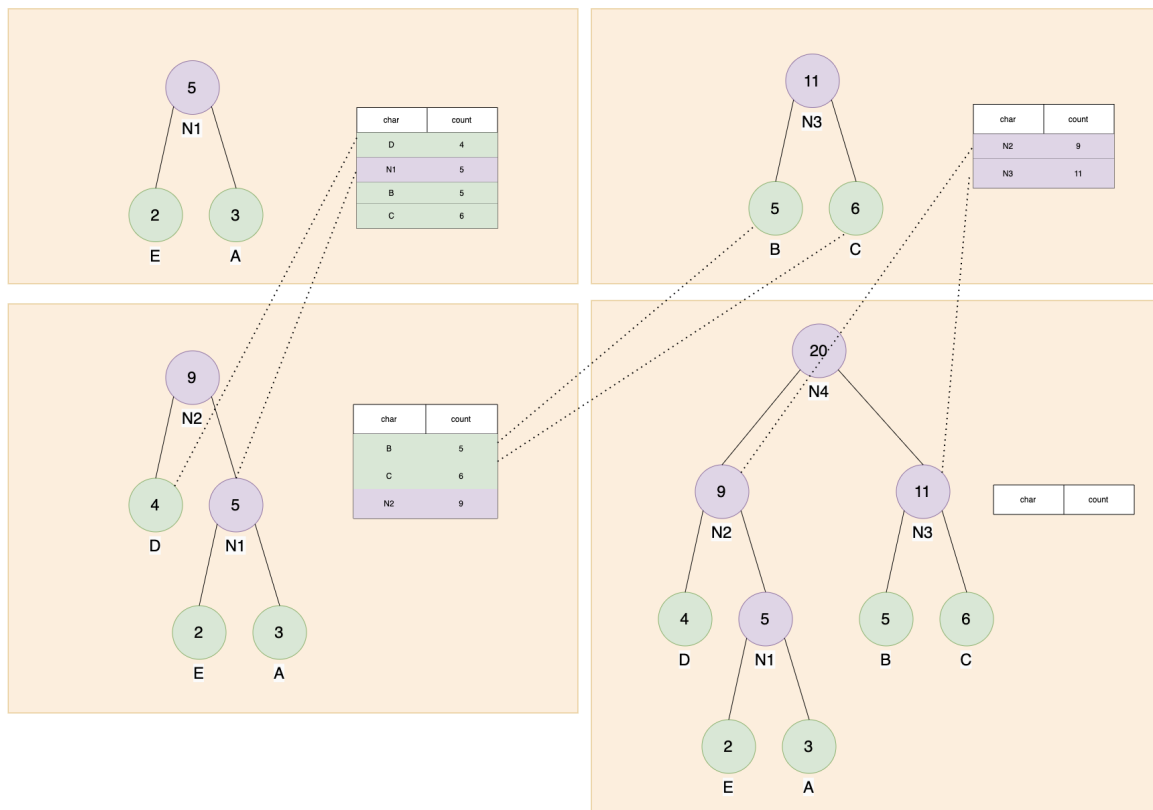
Step 2: A critical phase in Huffman tree construction involves the extraction of two nodes with the minimum frequency from the min heap. This extraction is pivotal in identifying the two least frequent characters within the dataset. Following this extraction, a new internal node is created, with a frequency equal to the sum of the frequencies of the two extracted nodes. This internal node takes on the role of a parent node, with the first extracted node becoming its left child and the second its right child. This hierarchical arrangement signifies the relationships between characters and their frequencies. From the table above, we should extract node E which has a frequency of 2, and node A with a frequency of 3.

Step 3: After the formation of the internal node, it is reintegrated into the min heap, ensuring that the heap remains updated with the latest frequencies and relationships. This iterative process of extracting nodes with minimum frequency and creating internal nodes with cumulative frequencies contributes to the gradual expansion of the Huffman tree. This recursive approach continues until the min heap contains only one node, and this remaining node ascends to become the root node of the Huffman tree, concluding the tree-building process. In Figure 2, we have built a new Node that we named N1, which has a value of 5. Then we have created a new table



(Figure 2)

The systematic execution of steps 2 and 3 results in the incremental growth of the Huffman tree, assimilating characters and their frequencies into the hierarchical structure. The recursive nature of this approach allows the tree to capture the relative importance and frequency of each character, providing a compact representation for subsequent data compression. The essence of Huffman coding lies in its ability to assign shorter codes to more frequent characters, thereby optimizing the overall encoding scheme. In Figure 3, all steps are repeated for each character until the table is empty.



(Figure 3)

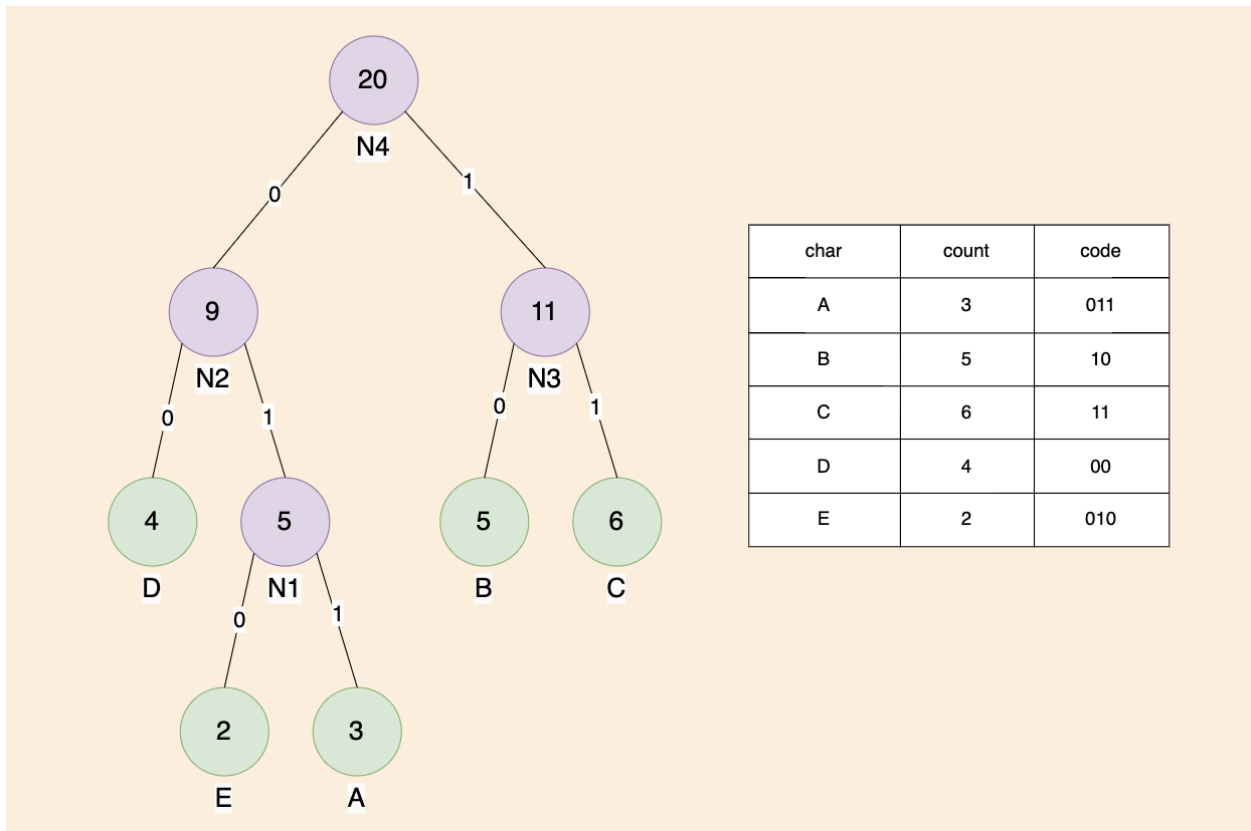
Huffman coding aligns with the concept of entropy encoding, where the length of the code is proportional to the probability of occurrence. In the context of Huffman tree construction, this translates to shorter codes for characters with higher frequencies, ultimately reducing the average code length and the overall size of the encoded data. The iterative extraction and merging of nodes contribute to the adaptability of Huffman coding, making it particularly well-suited for datasets with diverse character frequencies.

Throughout the tree-building process, the min heap functions as a dynamic repository, reflecting the evolving relationships between characters in the dataset. Its organized structure ensures that nodes with the minimum frequencies are readily available for extraction, facilitating an efficient construction of the hierarchical tree. This streamlined approach to building the tree enhances the effectiveness of Huffman coding, especially when dealing with large datasets exhibiting varied character frequencies.

In essence, the process of building a Huffman tree encompasses the creation of leaf nodes, the extraction of nodes with minimum frequency, and the iterative formation of internal nodes until a single root node remains. This hierarchical tree structure, rooted in the principles of entropy encoding, enables the compact representation of data and forms the basis for Huffman coding. The adaptability and efficiency inherent in Huffman coding make it a potent tool for data compression, particularly in scenarios where character frequency distribution plays a crucial role.

Creating Binary Codes from the Huffman Tree

In the intricate dance of Huffman coding, the process of constructing binary codes is a fascinating journey through the hierarchical structure of the Huffman tree. Imagine this tree as a branching narrative, where each bifurcation represents a decision point. As we embark on the traversal from the root to the leaf nodes, the delicate assignment of '0' and '1' unfolds with precision and purpose. A left traversal signifies the addition of '0,' while a right traversal appends '1.' This strategic ballet of binary digits is the foundation of creating distinct binary codes for each symbol residing in the leaf nodes. In Figure 4, on the left side, we assigned 0s and 1s to our tree.



(Figure 4)

The beauty of the Huffman tree lies in its ability to mirror the frequency distribution of symbols in the encoded data. Symbols that frequent the digital stage find themselves closer to the root, acquiring concise binary codes. Conversely, less frequent actors in this digital theatre take a more extended journey to the leaf nodes, resulting in longer binary codes. This dynamic allocation of binary digits optimizes the compression process, with shorter codes for commonly occurring symbols and longer codes for those with more infrequent entrances.

As we dig deeper into the construction of binary codes, a crucial aspect emerges – the generation of a comprehensive table mapping characters to their unique binary representations. This table (Figure 4), born from a depth-first traversal of the Huffman tree, becomes the Rosetta Stone of encoding and decoding. During this traversal, the binary code for each character is meticulously crafted, emerging as a concatenation of '0s' and '1s' along the journey to the leaf node. This table serves as a quick reference guide, enabling the seamless translation between the original symbols and their binary counterparts.

The brilliance of this process lies not only in its efficiency but also in its adaptability. The Huffman tree dynamically adjusts its structure based on the frequency characteristics of the symbols. In the grand symphony of data representation, the binary codes orchestrated within the tree offer an elegant solution to the complexities of compression. The result is not merely a collection of bits but a carefully choreographed ballet where each binary digit plays a vital role in conveying the essence of the original data. This harmonious convergence of structure and purpose showcases the artistry inherent in the Huffman coding dance, unraveling the intricate layers of data compression with every step.

In the end, our message “BCCABBDDAECCBBAEDDCC” will be encoded to
“1011110010101000111001101001111000111000101101001101001010000110100110011001101101100010011100111001110010110101100110011011001101101”

Decoding a message

Decoding in the realm of Huffman coding unfolds as a meticulous process, leveraging the structured hierarchy of the Huffman tree. The encoded message, represented by a sequence of bits, serves as the starting point. The essence of decoding lies in traversing the Huffman tree, distinguishing between 0s and 1s to reconstruct the original message. The tree's distinctive feature is its binary nature, where each internal node bifurcates into left and right branches, designated as 0 and 1, respectively. As the decoding process commences, each bit read directs the traversal along the corresponding branch, progressing through the tree until a leaf node is reached. The leaf nodes encapsulate the original characters, effectively decoding the bits into their original form. This binary-tree-based approach imbues Huffman coding with efficiency, as shorter codes for more frequent characters expedite the decoding process. The elegance of decoding in Huffman coding lies in its ability to precisely reconstruct the message while maintaining optimal compression ratios. In practical terms, the decoding algorithm starts at the tree's root, traversing down its branches based on the bits of the encoded message until a leaf node is encountered. The character associated with the leaf node is then appended to the

decoded message, and the process iterates until the entire encoded sequence is deciphered. This binary-tree-guided decoding exemplifies the synergy between efficient data compression and accurate retrieval of information, showcasing why Huffman coding remains a cornerstone in various applications where space optimization and rapid data transmission are paramount. The suffixes within the binary codes introduce an additional layer of nuance to the decoding process. Each binary code, representing a specific character in the encoded message, is uniquely structured in a way that avoids ambiguity during decoding. The absence of any common prefix among the binary codes ensures that at each step of the traversal, a distinct decision point arises, allowing for unambiguous identification of characters. This characteristic is crucial for the efficiency of Huffman decoding, contributing to its ability to rapidly reconstruct the original message. The structure of Huffman codes, with their carefully crafted suffixes, embodies a balance between compact representation and unambiguous interpretation, further highlighting the elegance and effectiveness of the Huffman coding algorithm in the landscape of data compression. Later, in the section where we build Huffman coding compression on Golang, we will explain how decoding works in great detail. For now, when we receive binary code “101111001010100011100110100111100011100010110100110100101000011010011001100110101100010011100111001110010110101100110011011001101101”, we are sure that it decodes to “BCCABBDDAECCBBAEDDCC”

Calculations

The input text has $N = 20$ symbols of which 5 are unique. After computing the probability of each and building the tree, we can use it to fill the alphabet table with every encoded symbol. Then, we just have to swap each symbol for its binary result, and we get our encoded string. With the obtained table we can compute the average length of the code words,

Average Length, $L = 2.250$ bits

Which is very close to the minimum, defined by the Shannon Entropy,

Entropy, $H = 2.228$ bits/symbol.

Other conclusions we can obtain are:

Code Efficiency: $\eta = H/L = 0.990$

Residual Efficiency: $\tau = H - L = -0.022$

Code Redundancy: $r = 1 - \eta = 0.010$

If we would have encoded the text naively, it would have weighed 160 bits. On the other hand, with Huffman is 45 bits. Therefore, we can calculate its compression ratio:

Compression Ratio: $45/160$ bits = 28.125%

Building Huffman coding compression application using Go programming language


Golang, or Go, stands out as a programming language developed by Google with a focus on simplicity, efficiency, and concurrency. Its inception in 2009 introduced a language that embodies fast compilation, static typing for improved reliability, and automatic garbage collection for simplified memory management. With a comprehensive standard library covering networking, file I/O, encryption, and more, Go provides a solid foundation for various applications. [8 - REFERENCE TODO] The language has found notable applications in web development, cloud-native systems, system programming, and network programming. Its concurrent programming model, through lightweight goroutines and channels, enables efficient multicore utilization. Go's fast compilation speed and execution efficiency make it particularly suitable for performance-critical tasks. The language's simplicity and clean syntax contribute to readability and ease of maintenance. In the realm of data compression, Go's strengths shine, offering a compelling choice for implementing algorithms like Huffman coding. Its built-in support for concurrency can be leveraged for parallel processing, enhancing compression and decompression times. With cross-platform compatibility and excellent standard library support, Go emerges as a versatile and efficient tool for developing a wide range of applications, including robust and high-performance data compression tools like Huffman coding applications. Hence, Go will be our preferred choice for building this application. The entire project can be found on link: <https://github.com/Ammce/data-compression-application>



```
1 func main() {
2     input := "BCCABBDDECCBBAEDDCC"
3     compressed, codes := Compress(input)
4
5     fmt.Println("Original:", input)
6     fmt.Println("Compressed:", compressed)
7     fmt.Println("Huffman Codes:")
8     for char, code := range codes {
9         fmt.Printf("%c: %s\n", char, code)
10    }
11
12    /*
13    THE OUTPUT
14    Original: BCCABBDDECCBBAEDDCC
15    Compressed: 101111011101000000110101111101001101000001111
16    Huffman Codes:
17    C: 11
18    D: 00
19    E: 010
20    A: 011
21    B: 10
22    */
23 }
```

(Figure 5)

In Figure 5, we can see initialization of go application. First, we definite input to be the message we used through many examples in this seminar paper. Next, we call *Compress* function that will return the actual compressed message stored in variable *compressed*. In variable *codes*, which is a type of *map[byte]string*, we store the map where byte stores the actual character, and string actually stores the binary code. After that, we log the data we got back. In the readme.md file, you can learn more how to start the project or run tests.



```
1 func Compress(input string) (string, map[byte]string) {
2     frequencies := make(map[byte]int)
3     for _, char := range input {
4         frequencies[byte(char)]++
5     }
6
7     root := BuildHuffmanTree(frequencies)
8
9     codes := make(map[byte]string)
10    GenerateHuffmanCodes(root, "", codes)
11
12    compressed := ""
13    for _, char := range input {
14        compressed += codes[byte(char)]
15    }
16
17    return compressed, codes
18 }
```

(Figure 6)

In Figure 6, The *Compress* function serves as the core component for Huffman coding within the Golang application. It begins by analyzing the input string to determine the frequency of each character, encapsulating this information in a map named *frequencies*. Subsequently, the function proceeds to construct the Huffman tree by invoking the *BuildHuffmanTree* function. Following the tree's creation, the next step involves generating Huffman codes for every character present in the tree. These codes are stored in the *codes* map. The actual compression of the input message is then executed by iteratively appending the Huffman codes

corresponding to each character. The function culminates by returning both the compressed message and the comprehensive map of Huffman codes. This systematic approach encapsulates the intricacies of Huffman coding, showcasing a well-structured and modular design that enhances the efficiency and readability of the Golang application.

```
func BuildHuffmanTree(frequencies map[byte]int) *HuffmanNode {
    priorityQueue := make(HuffmanPriorityQueue, len(frequencies))
    i := 0
    for char, frequency := range frequencies {
        priorityQueue[i] = &HuffmanNode{Value: char, Frequency: frequency}
        i++
    }
    heap.Init(&priorityQueue)
    for len(priorityQueue) > 1 {
        left := heap.Pop(&priorityQueue).(*HuffmanNode)
        right := heap.Pop(&priorityQueue).(*HuffmanNode)

        internalNode := &HuffmanNode{
            Frequency: left.Frequency + right.Frequency,
            Left:      left,
            Right:     right,
        }
        heap.Push(&priorityQueue, internalNode)
    }
    return priorityQueue[0]
}

func GenerateHuffmanCodes(root *HuffmanNode, code string, codes map[byte]string) {
    if root == nil {
        return
    }
    if root.Left == nil && root.Right == nil {
        codes[root.Value] = code
        return
    }
    GenerateHuffmanCodes(root.Left, code+"0", codes)
    GenerateHuffmanCodes(root.Right, code+"1", codes)
}
```

(Figure 7)

In Figure 7, The *BuildHuffmanTree* function is a pivotal component in constructing the Huffman tree for the Golang application. It initializes a priority queue, named *priorityQueue*, using a custom type *HuffmanPriorityQueue*. This queue is populated with leaf nodes, each representing a unique character from the input string along with its corresponding frequency. The priority queue is then transformed into a min heap using the *heap.Init* function. The function proceeds to iterate until the heap contains a single node, popping the two nodes with the minimum frequencies, creating a new internal node, and pushing it back into the heap. This process continues until the heap is reduced to a single node, representing the root of the Huffman tree. The *GenerateHuffmanCodes* function complements the tree-building process by recursively traversing the tree. It assigns binary codes to each leaf node character based on the path taken to reach that node. The codes are stored in the *codes* map, where the character serves as the key, and the corresponding Huffman code as the value. The function effectively creates a comprehensive mapping of characters to their respective Huffman codes.

Together, these functions embody the core logic of Huffman coding. The *BuildHuffmanTree* function constructs the tree itself, employing a priority queue to ensure efficient node selection, while *GenerateHuffmanCodes* traverses the tree to generate the binary codes associated with each character. This modular and systematic approach enhances the readability and maintainability of the Golang application, providing a clear implementation of the Huffman coding algorithm.

```
type HuffmanNode struct {
    Value      byte
    Frequency  int
    Left       *HuffmanNode
    Right      *HuffmanNode
}

type HuffmanPriorityQueue []*HuffmanNode

func (pq HuffmanPriorityQueue) Len() int {
    return len(pq)
}

func (pq HuffmanPriorityQueue) Less(i, j int) bool {
    return pq[i].Frequency < pq[j].Frequency
}

func (pq HuffmanPriorityQueue) Swap(i, j int) {
    pq[i], pq[j] = pq[j], pq[i]
}

func (pq *HuffmanPriorityQueue) Push(x interface{}) {
    *pq = append(*pq, x.(*HuffmanNode))
}
```



```

func (pq *HuffmanPriorityQueue) Pop() interface{} {
    old := *pq
    n := len(old)
    node := old[n-1]
    *pq = old[0 : n-1]
    return node
}

```

(Figure 8)

In Figure 8, the code defines the structures and methods for implementing a priority queue specific to the *HuffmanNode* type, tailored for building a Huffman tree in the Golang application.

HuffmanNode is a struct representing a node in the Huffman tree. It contains fields for the character value (*Value*), the frequency of the character in the input data (*Frequency*), and pointers to the left and right child nodes (*Left* and *Right*).

HuffmanPriorityQueue is a custom type defined as a slice of pointers to *HuffmanNode*. This type will be used to implement the priority queue.

The *Len*, *Less*, and *Swap* methods are part of the *heap.Interface* implementation. They define the basic operations required for managing the priority queue.

Len() returns the length of the priority queue.

Less(i, j int) bool compares the frequencies of two nodes at positions *i* and *j* in the priority queue, indicating whether the node at index *i* has a lower frequency.

Swap(i, j int) swaps the nodes at positions *i* and *j* in the priority queue.

The *Push* and *Pop* methods are also part of the *heap.Interface* implementation, allowing the *HuffmanPriorityQueue* type to be used as a min heap.

Push(x interface{}) adds a new element (*x*) to the priority queue. In this case, it appends a *HuffmanNode* to the slice.

Pop() interface{} removes and returns the minimum element from the priority queue. It extracts the last element from the slice, updates the slice to exclude the last element, and returns the extracted element.

These methods enable the *HuffmanPriorityQueue* to be utilized with the *heap* package in Golang, facilitating the construction and manipulation of the priority queue essential for building the Huffman tree efficiently.

Usage of Huffman Coding

Huffman coding has become a silent hero in the digital realm, quietly assisting in various applications across our daily lives. Its widespread use has significantly contributed to the efficiency and optimization of digital processes. From reducing file sizes for quicker sharing and smoother streaming experiences to enhancing the overall performance of electronic devices, Huffman coding has proven to be an invaluable tool. Its adaptability and effectiveness have led to its integration into a myriad of applications, making it an unsung champion behind the scenes of our modern, interconnected world. Let's explore the expansive reach of Huffman coding, uncovering the multitude of areas where it has made a profound impact.

In the realm of data compression, ZIP and GZIP file formats stand out as stalwarts. Whether you're archiving a folder or sending a compressed document via email, Huffman coding is the unsung hero behind the scenes. It meticulously analyzes the frequency of characters in a file and assigns shorter codes to more common characters, resulting in reduced file sizes without compromising data integrity. This not only saves storage space but also expedites file transfers, especially in bandwidth-constrained scenarios.

When it comes to multimedia, Huffman coding has become an integral part of image and audio compression. In the PNG (Portable Network Graphics) image format, Huffman coding contributes to visually lossless compression, ensuring that images remain crisp and vibrant while occupying minimal disk space. Similarly, in the MP3 audio compression standard, Huffman coding aids in compressing audio files without perceptible loss in quality. This enables users to enjoy their favorite music seamlessly, whether it's stored on a device or streamed online.

Communication protocols heavily rely on Huffman coding to enhance efficiency in data exchange. In the HTTP (Hypertext Transfer Protocol) used for web communication, Huffman coding optimizes the transmission of textual data, reducing latency and improving overall browsing experiences. Additionally, in the MQTT (Message Queuing Telemetry Transport) protocol, commonly used in the Internet of Things (IoT), Huffman coding streamlines the communication between connected devices, making it resource-efficient and responsive.

In HTTP/2, headers are compressed using the HPACK compression format, which includes Huffman coding. Huffman coding is employed to efficiently represent commonly used strings, reducing the size of headers and improving overall data transfer efficiency. This is particularly important for optimizing the performance of gRPC communication over HTTP/2, where minimizing latency and bandwidth usage is crucial. In 2023, gRPC played a significant role in micro-services communication. In the years to come, this technology tends to be dominant in a world of microservices.

Network communication, a cornerstone of the internet, benefits significantly from Huffman coding. It plays a crucial role in optimizing various network protocols, such as TCP/IP (Transmission Control Protocol/Internet Protocol), contributing to faster data transmission and reduced latency. This acceleration is particularly noticeable in online activities like video

streaming, where large amounts of data need to be transferred swiftly to ensure smooth playback.

The landscape of data storage owes much to Huffman coding, especially in hard drives and solid-state drives (SSDs). By intelligently compressing data before storage, these devices maximize their capacity, allowing users to store more files without the need for additional physical space. This is particularly advantageous in environments where efficient use of storage resources is paramount, such as data centers and cloud computing infrastructures.

Smartphone applications harness the power of Huffman coding to balance the trade-off between functionality and storage space. Whether you're downloading a new app or updating existing ones, Huffman coding ensures that the process is swift and data-efficient. This is particularly crucial in the mobile ecosystem, where users prioritize both application performance and conserving their limited mobile data plans.

In the realm of digital television broadcasting, Huffman coding contributes to efficient transmission of audio and video data. This ensures that viewers receive high-quality broadcasts with minimal bandwidth requirements, enhancing the overall viewing experience. Additionally, it aids in the implementation of advanced features such as electronic program guides (EPGs) and interactive services.

Distributed DNN Training with Huffman-Based Encoders

In the dynamic landscape of deep neural network (DNN) training, the prominence of distributed stochastic algorithms equipped with gradient compression techniques is escalating. Specifically, codebook quantization has emerged as a state-of-the-art method for training large-scale DNN models. However, the efficient communication of quantized gradients among the nodes in a network poses a critical challenge, necessitating sophisticated encoding techniques.

Traditionally, Elias encoding has been the go-to choice, but this discussion explores a paradigm shift. Leveraging Huffman coding, we propose a family of lossless encoding techniques tailored to exploit diverse characteristics of quantized gradients during distributed DNN training. Huffman coding, renowned for its entropy encoding capabilities, diverges from conventional methods by adapting code lengths based on symbol frequency. The flexibility inherent in Huffman coding allows for a more nuanced approach to encoding, proving particularly advantageous in the context of quantized gradient communication. Our proposed Huffman-based encoders—namely RLH, SH, and SHS—introduce innovative strategies to optimize the communication of quantized gradients in distributed DNN training. RLH, the first encoder in our paradigm, refines the encoding process through recursive adaptation to the local structure of quantized gradients. This adaptability enables RLH to capture nuanced patterns dynamically, improving compression ratios, especially in scenarios with changing gradient distributions during training. SH, the second encoder, introduces selective Huffmanization, strategically applying Huffman coding to specific subsets of quantized gradient data. This targeted approach optimizes the encoding process for varying data characteristics, outperforming traditional encoders, especially in handling heterogeneous gradient distributions. SHS, the final member of our proposed family, incorporates sequential Huffmanization into the encoding process. This approach encodes

quantized gradients sequentially, leveraging context to enhance compression efficiency. SHS showcases significant improvements in data volume reduction while maintaining a balance between computational complexity and encoding performance. Experimental validation spans a range of DNN models and involves training on diverse datasets. The results underscore the efficacy of RLH, SH, and SHS in substantially reducing encoded data volume compared to traditional Elias-based encoders. These Huffman-based techniques demonstrate superior compression ratios, promising to revolutionize the communication aspect of distributed DNN training. [7]

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:

The future of Huffman coding and data compression presents a myriad of intriguing possibilities that extend beyond the current state of the art. As computational capabilities continue to evolve, one promising avenue for exploration involves the integration of Huffman coding with emerging technologies such as quantum computing. The potential application of quantum principles to data compression algorithms opens up new frontiers in terms of speed and efficiency. Quantum superposition and entanglement may offer novel approaches to encoding and decoding processes, transforming the landscape of compression. However, the challenges associated with harnessing the power of quantum computing for practical applications, including error correction and scalability, must be carefully addressed to unlock the full potential of this futuristic direction. [9 REFERENCE TODO]

Moreover, the advent of machine learning introduces an exciting dimension to Huffman coding. Research initiatives could investigate how machine learning algorithms can be leveraged to dynamically adapt Huffman coding strategies based on the evolving characteristics of data streams. This self-optimizing capability could lead to more intelligent compression algorithms that adapt to changing patterns in real-time, making them particularly suitable for dynamic environments such as internet of things (IoT) applications and streaming services. The intersection of machine learning and Huffman coding not only promises enhanced compression performance but also opens avenues for exploring automated decision-making processes within the compression framework. [10 REFERENCE TODO]

Additionally, the field of data compression is increasingly relevant in the context of security and privacy concerns. Future research directions may explore the integration of Huffman coding with advanced encryption techniques to ensure secure communication and storage of compressed data. This fusion of compression and encryption could offer a holistic approach to safeguarding sensitive information, addressing potential vulnerabilities associated with traditional compression methods.

Furthermore, as data volumes continue to skyrocket, there is a growing need for scalable and distributed compression solutions. Investigating how Huffman coding algorithms can be parallelized and optimized for distributed computing architectures represents a critical future direction. This exploration could lead to advancements in cloud-based compression services, facilitating efficient and rapid compression of massive datasets across distributed networks.

In the realm of human-computer interaction, future directions may focus on enhancing the user experience in applications utilizing Huffman coding. Designing more intuitive interfaces, exploring real-time compression for interactive environments, and considering the psychological aspects of user perception during compression and decompression activities are potential areas for investigation.

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
6. Suiliang Ma, Rutgers, the State University of New Jersey Math 436, Final Paper Huffman Coding
7. Huffman Coding Based Encoding Techniques for Fast Distributed Deep Learning
Rishikesh R. Gajjala¹ , Shashwat Banchhor¹ , Ahmed M. Abdelmoniem^{1*} Aritra Dutta, Marco Canini, Panos Kalnis KAUST
8. Boyd, W. (2023, June 8). What is Go? An intro to Google's Go programming language. Pluralsight Blog.

<https://www.pluralsight.com/resources/blog/cloud/what-is-go-an-intro-to-googles-go-programming-language-aka-golang>

9. Tolba, A. S., Rashad, M. Z., & El-Dosuky, M. A. (July 2007). "Quantum-inspired Huffman Coding." Dept. of Computer Science, Faculty of Computers and Information Sciences, Mansoura University, Mansoura, Egypt
10. Frąckiewicz, Marcin. (May 10, 2023). "The Future of AI: Exploring the Potential of PCA for Intelligent Data Compression." Artificial Intelligence, TS2 Space.