



**INSTITUTO  
FEDERAL**

Santa Catarina

---

Câmpus  
São José

# **Compressão e Descompressão com Algoritmo de Huffman**

Sistemas de Comunicação II

Arthur Cadore Matuella Barcella

03 de Fevereiro de 2024

Engenharia de Telecomunicações - IFSC-SJ

# Sumário

<b>1. Introdução .....</b>	<b>3</b>
<b>2. Desenvolvimento .....</b>	<b>3</b>
2.1. Análise Teórica .....	3
2.1.1. Fonte Discreta sem Memória .....	3
2.1.1.1. Item A .....	3
2.1.1.2. Item B .....	3
2.1.1.3. Item C .....	4
2.1.1.4. Item D .....	5
2.2. Compressão Huffman Em Arquivo .....	5
2.2.1. Leitura do Arquivo .....	5
2.2.2. Análise dos caracteres contidos .....	6
2.2.3. Calculo do percentual individual .....	7
2.2.4. Calculando a PMF (Probability Mass Function) .....	8
2.2.5. Indexação dos caracteres .....	9
2.2.6. Aplicando codificação sobre o texto .....	9
2.2.7. Imprime o arquivo codificado .....	9
<b>3. Analisando o arquivo codificado .....</b>	<b>10</b>
3.1. Decodificando o arquivo .....	10
3.1.1. Leitura do arquivo codificado .....	10
3.1.2. Decodificação do arquivo por huffman: .....	10
3.1.3. Aplicando a indexação inversa: .....	11
3.1.4. Exportando decodificado em um arquivo: .....	11
<b>4. Conclusão .....</b>	<b>12</b>

# 1. Introdução

Neste relatório, será apresentado um estudo sobre o algoritmo de Huffman, incluindo a leitura de um arquivo de texto, a contagem de ocorrências de cada caractere, a análise dos caracteres contidos no arquivo, o cálculo do percentual de cada caractere, a criação de uma Função de Massa de Probabilidade (PMF), a aplicação do algoritmo de Huffman sobre o texto, a codificação do texto e a análise do arquivo codificado.

O algoritmo de Huffman é um método de compressão de dados que utiliza a codificação de caracteres para reduzir o tamanho de um arquivo. O algoritmo foi desenvolvido por David A. Huffman em 1952, e é amplamente utilizado em sistemas de comunicação e armazenamento de dados.

## 2. Desenvolvimento

### 2.1. Análise Teórica

#### 2.1.1. Fonte Discreta sem Memória

Considere uma fonte discreta sem memória (DMS) com alfabeto dado por  $X = \{a, b, c\}$  e probabilidades respectivas dadas por  $p_x = [\frac{3}{10}, \frac{3}{10}, \frac{1}{10}]$

##### 2.1.1.1. Item A

- Calcule a entropia da fonte.

```
1 pmf = [3/10, 6/10, 1/10]
2 BFR = math.ceil(math.log2(len(pmf)))
3
4 print(f"Numero de simbolos: {BFR}")
5 entropy = -sum(p * math.log2(p) for p in pmf)
6
7 print(f"Entropia da fonte: {entropy:.4f} bits/símbolo")
```

- Numero de simbolos: 2
- Entropia da fonte: 1.2955 bits/símbolo

##### 2.1.1.2. Item B

- Determine um código de Huffman para a fonte. Qual o comprimento médio do código obtido?

```
1 code = kmm.FixedToVariableCode.from_codewords(3, [(0,), (0,1), (1,1)])
2
3 print("Unicamente decodificavel: ", code.is_uniquely_decodable())
4 print("Prefixo Livre:", code.is_prefix_free())
5 print("Huffman Rate: ", code.rate(pmf))
6 print("Huffman Codewords: ", code.codewords)
7
8 # Calculate the compress ratio
```

```
9 print("Compress Ratio: ", BFR - code.rate(pmf))
```

- Unicamente decodificavel: True
- Prefixo Livre: False
- Huffman Rate: 1.7
- Huffman Codewords: [(0,), (0, 1), (1, 1)]
- Compress Ratio: 0.30000000000000004

### 2.1.1.3. Item C

- Calcule a extensão e segunda ordem da fonte.

```
1 # Generate the PMF of the second order
2 pmf_2nd = [p1 * p2
3             for p1, p2 in itertools.product(pmf, repeat=2)]
4
5 # Print the PMF of the second order
6 print("Tabela de PMF de segunda ordem:")
7 for i, p in enumerate(pmf_2nd):
8     print(f"p({i//3}, {i%3}) = {p:.4f}")
9 print("\n")
10
11 # Calculate the entropy of the second order
12 entropy_2nd = -sum(p * math.log2(p) for p in pmf_2nd if p > 0)
13 print(f"Entropia de segunda ordem: {entropy_2nd:.4f} bits por par de
14       símbolos")
15
16 # Calculate the average entropy per symbol
17 entropy_per_symbol = entropy_2nd / 2
18 print(f"Entropia média por símbolo (segunda ordem):
19       {entropy_per_symbol:.4f} bits/símbolo")
```

- Tabela de PMF de segunda ordem:
- $p(0, 0) = 0.0900$
- $p(0, 1) = 0.1800$
- $p(0, 2) = 0.0300$
- $p(1, 0) = 0.1800$
- $p(1, 1) = 0.3600$
- $p(1, 2) = 0.0600$
- $p(2, 0) = 0.0300$
- $p(2, 1) = 0.0600$
- $p(2, 2) = 0.0100$

Alem disso, temos que:

- Entropia de segunda ordem: 2.5909 bits por par de símbolos
- Entropia média por símbolo (segunda ordem): 1.2955 bits/símbolo

#### 2.1.1.4. Item D

- Determine um código de Huffman para a extensão de segunda ordem da fonte. Qual o comprimento médio do código obtido?

```
1 huff = komm.HuffmanCode(pmf_2nd)
2
3 print("Huffman Ratio: ", huff.rate(pmf_2nd))
4 print("Huffman Code: ", huff.codewords)
5
6 # Calculate the compress ratio
7 print("Compress Ratio: ", BFR_2nd - huff.rate(pmf_2nd))
```

- Huffman Ratio: 2.6699999999999995
- Huffman Code: [(0, 1, 0, 0), (1, 1), (0, 1, 0, 1, 0, 0), (1, 0), (0, 0), (0, 1, 1, 1), (0, 1, 0, 1, 1), (0, 1, 1, 0), (0, 1, 0, 1, 0, 1)]
- Compress Ratio: 1.3300000000000005

## 2.2. Compressão Huffman Em Arquivo

### 2.2.1. Leitura do Arquivo

Inicialmente, deve-se coletar todos os caracteres contidos no arquivo de texto. Para isso, é necessário realizar a leitura do arquivo e contar a quantidade de ocorrências de cada caractere. Isso é executado através do código abaixo:

```
1 # Read the file and count the number of occurrences of each character
2 with open("alice.txt", "r", encoding="utf-8") as file:
3     # Ler o arquivo
4     text = file.read()
5
6     # Create a variable to store the characters and their occurrences
7     characters = {}
8
9     for char in text:
10        # Check if the character is on the variable
11        if char in characters:
12            # Increment the character
13            characters[char] += 1
14        else:
15            # Add the character to the variable and set it to 1
16            characters[char] = 1
17
18    # Order the characters
19    sorted_letters = sorted(characters.items())
20
21    # Get the letters and occurrences
22    letters = [letter for letter, occurrences in sorted_letters]
23    occurrences = [occurrences for letter, occurrences in sorted_letters]
24
25    # Print the letters count:
26    # Print the letters count:
27    print("Letters count:", len(letters))
28
```

```
29 BFR = math.ceil(math.log2(len(letters)))
30 print("Bits for representation:", BFR)
```

Desta forma, o vetor `letters` contém todos os caracteres contidos no arquivo de texto, enquanto o vetor `occurrences` contém a quantidade de ocorrências de cada caractere.

Também podemos notar que com base na impressão de “letters count”, o arquivo contém 91 caracteres distintos, necessitando de 91 símbolos diferentes para representar cada caractere, portanto, a codificação mais simples precisará de 7 bits ( $2^7 = 128$ ) para representar todos os caracteres.

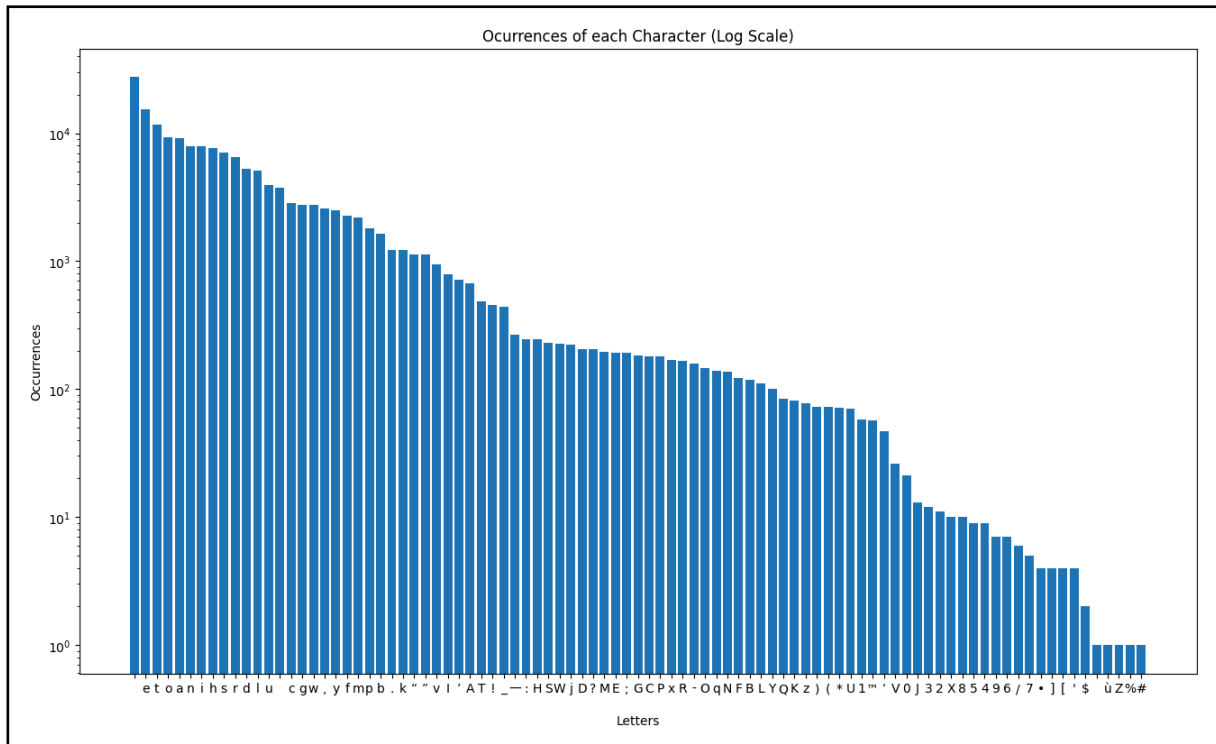
### 2.2.2. Análise dos caracteres contidos

Na sequência, é possível analisar a quantidade de ocorrências de cada caractere contido no arquivo de texto. Para isso, foi aplicado um plot de barras, onde o eixo x representa os caracteres e o eixo y representa a quantidade de ocorrências de cada caractere. O código abaixo realiza essa operação:

```
1 sorted_letters = [letter for _, letter in sorted(zip(occurrences, letters),
2 reverse=True)]
3 sorted_occurrences = [occurrence for occurrence, _ in
4 sorted(zip(occurrences, letters), reverse=True)]
5 plt.figure(figsize=(16,9))
6 plt.bar(sorted_letters, sorted_occurrences)
7 plt.yscale("log")
8 plt.xlabel("Letters")
9 plt.ylabel("Occurrences")
10 plt.title("Occurrences of each Character (Log Scale)")
11 plt.show()
```

Com base neste código, é gerado a seguinte figura:

Figura 1: Elaborada pelo Autor



Note que o primeiro caractere mais frequente é o espaço, seguido pelas letras “e”, “t”, “a” e “o”.

Nota: Após o caracter “u”, aparentemente outro caracter espaço está sendo representado, porém, esse representa o caracter de quebra de linha (“\n”).

### 2.2.3. Calculo do percentual individual

Com base na quantidade de ocorrências de cada caractere, é possível calcular o percentual de cada caractere em relação ao total de caracteres contidos no arquivo de texto.

Para isso, calculamos a quantidade total de caracteres lidos, e determinamos a razão entre a quantidade de ocorrências de cada caractere e o total de caracteres. O código abaixo realiza essa operação:

Total de caracteres: 163919

```
1 # sum all the occurrences of the letters and create a graph with the
  # percentage of each letter
2
3 # Get the total number of letters
4 total_letters = sum(occurrences)
5 print("Total letters:", total_letters)
6
7 # Calculate the percentage of each letter
8 def percentage(occurrences, total):
9     return [occurrence / total for occurrence in occurrences]
10
11 percentages = percentage(occurrences, total_letters)
12
13 # sort the letters by occurrences to make the graph more readable
```

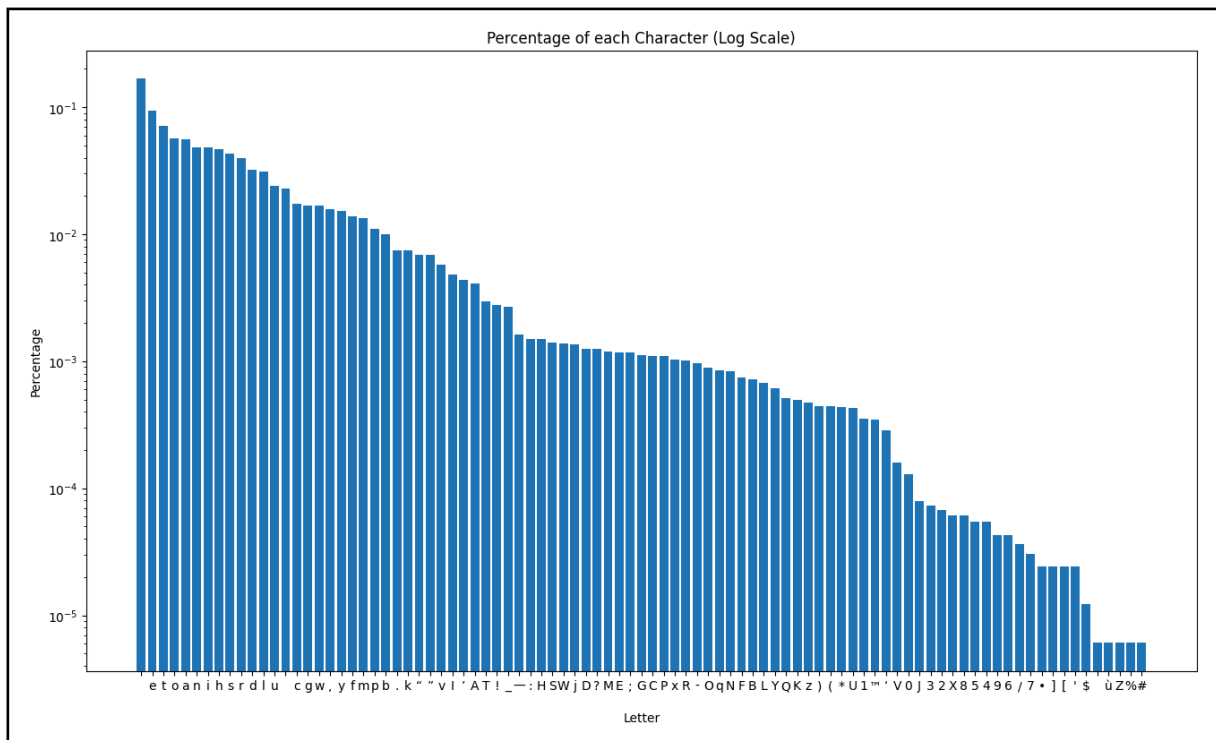
```

14 sorted_letters = [letter for _, letter in sorted(zip(occurrences, letters),
15 sorted_percentages = [percentage for _, percentage in
sorted(zip(occurrences, percentages), reverse=True)]
16
17 # Create the gprint("Letters:", letters) graph
18 plt.figure(figsize=(16,9))
19 plt.bar(sorted_letters, sorted_percentages)
20 plt.xlabel("Letter")
21 plt.ylabel("Percentage")
22 plt.title("Percentage of each Character (Log Scale)")
23 plt.yscale("log")
24 plt.show()

```

Dessa forma é possível visualizar a porcentagem de cada caractere em relação ao total de caracteres contidos no arquivo de texto, conforme apresentado abaixo:

Figura 2: Elaborada pelo Autor



#### 2.2.4. Calculando a PMF (Probability Mass Function)

Com base na lista de caracteres e seus respectivos percentuais, é possível calcular a Função de Massa de Probabilidade (PMF) de cada caractere. Para isso, é necessário criar um dicionário contendo o caractere e seu respectivo percentual. O código abaixo realiza essa operação:

```

1 # create a pmf vector with the percentage of each letter
2
3 pmf = {letter: percentage for letter, percentage in zip(letters,
4 percentages)}
5 print("PMF:", list(pmf.values()))

```



```

6 # Calculate the huff code
7 huff = kmm.HuffmanCode(list(pmf.values()))
8
9 # Print the huff code values and the huff ratio
10 print("Huffman code:", huff.codewords)
11 print("Huff Ratio:", huff.rate(list(pmf.values())))
12 print("Compress Ratio:", BFR - huff.rate(list(pmf.values())))

```

Com base nisso, podemos determinar o código de huffman para cada caractere com base em sua participação do percentual total. Além disso, podemos determinar parâmetros resultantes do código de huffman, como o huff ratio e a taxa de compressão:

Huff Ratio: 4.643275 Compress Ratio: 2.356724

Dessa forma, podemos ver que o código aplicado foi capaz de reduzir a quantidade de bits necessária para representar os caracteres do arquivo em 2.35 bits (média).

### 2.2.5. Indexação dos caracteres

Em seguida, é necessário criar uma indexação do caractere e sua respectiva letra, de forma que cada caractere seja representado por um índice. O código abaixo realiza essa operação:

```

1 index = {i: letter for i, letter in enumerate(letters)}
2 range(len(letters)):
3     print(letters[i], occurrences[i])
4 print ("Index:", index)

```

### 2.2.6. Aplicando codificação sobre o texto

Com base nesta indexação, é possível codificar o texto original, de forma que cada caractere seja representado por um índice. O código abaixo realiza essa operação. De forma que inicialmente o texto é “codificado” pelo index, transformando cada caractere em uma letra, e posteriormente é aplicado o código de huffman sobre o texto codificado.

```

1 # encode the text using the index to create a integer index of the letters
  in the text
2
3 encoded_text = [list(index.keys())[list(index.values()).index(letter)] for
  letter in text]
4
5 print("Encoded text (by index):", encoded_text)
6
7 # create a huffman code for the encoded text
8 huff_encoded = huff.encode(encoded_text)
9
10 print("Huffman encoded text:", huff_encoded)

```

### 2.2.7. Imprime o arquivo codificado

Por fim, é possível imprimir o arquivo codificado, de forma que cada caractere seja representado por um índice. O código abaixo realiza essa operação:

```

1 # export the huffman code to a file.
2 with open("huff_encoded.com2", "w") as file:
3     file.write("".join(map(str, huff_encoded)))

```

### 3. Analisando o arquivo codificado

Por fim, é possível analisar o arquivo codificado, de forma que cada caractere seja representado por um índice. O código abaixo realiza essa operação:

```

1 0000000010110111100010111110001110000000100101011101010001000001011101101010
2 10011110110001010111000110101011110000000111010001000011110110000111110101
3 1101011011100010100111000010101110101100000000101110110100101010001111100
4 0011100101100000011011000101101100111000110110110011011011000000001011011
5 1000000111100010101000100001111011100000010111110000001101001101111101010
6 000111110000111111001101101110110001011111100110110111010110110111100011
7 11111001011010011011110000000010001011110001001110110000000011000100100101
8 00111000011001101101100101001100000001010010000100110111100011000100010010
9 00110100000011000011111010011011110101101000000110110101100100101001011100
10 00101001000000010100100110110110001011011000100110110010110110100110000000
11 10100101110000001111100001010000011110001010001001100100010110000101011011
12 01100101000000110001110000001011100011000100101111000101011010011001011101
13 0101001000000100001110111000100010101110000000010111100010010010101101001
14 01110110000001100011111101010000111111000100101011011001110001101001101
15 1110100111000111010010000110000111110100110111100010111100011100000010010
16 10111010100010000010111011010101001111011000101011100011010101000000011111
17 [...]

```

#### 3.1. Decodificando o arquivo

##### 3.1.1. Leitura do arquivo codificado

Para decodificar o arquivo, é necessário realizar a leitura do arquivo codificado e aplicar o algoritmo de Huffman para decodificar o texto. O código abaixo realiza essa operação:

```

1 # read the huffman code from the file
2
3 with open("huff_encoded.com2", "r") as file:
4     huff_encoded = file.read()
5
6 # decode the huffman code
7 huffDecode = kmm.HuffmanCode(list(pmf.values()))
8 print("Huffman code:", huffDecode.codewords)

```

Com base no código acima, deve-se obter as palavras código utilizadas originalmente para codificar o texto.

##### 3.1.2. Decodificação do arquivo por huffman:

Tendo as palavras código e o texto codificado, é possível decodificar o texto original. Esse processo é realizado através do código abaixo:

```

1 # Decode the encoded text
2
3 # Convert the string back to a list of integers
4 huff_encoded_list = list(map(int, huff_encoded))
5
6 decoded_text = huffDecode.decode(huff_encoded_list)
7
8 # print more values of the decoded text
9 print("Decoded text:", decoded_text[:100])

```

O texto decodificado é apresentado abaixo, note que ainda não é possível ler pois o texto está estruturado com base em um índice de cada letra correspondente.

```

1 90 46 63 60 1 42 73 70 65 60 58 75 1 33 76 75 60 69 57 60 73 62 1 60
2 28 70 70 66 1 70 61 1 27 67 64 58 60 6 74 1 27 59 77 60 69 75 76 73
3 60 74 1 64 69 1 49 70 69 59 60 73 67 56 69 59 0 1 1 1 1 0 46 63
4 64 74 1 60 57 70 70 66 1 64 74 1 61 70 73 1 75 63 60 1 76 74 60 1
5 70 61 1 56

```

### 3.1.3. Aplicando a indexação inversa:

Dessa forma, para resolver o problema de leitura do texto decodificado, é necessário aplicar a indexação inversa, de forma que cada índice seja representado por um caractere. Abaixo está o índice utilizado no processo de codificação (gerado automaticamente durante a codificação).

```

1 0: '\n', 1: ' ', 2: '!', 3: '#', 4: '$', 5: '%', 6: '"', 7: '(', 8: ')'
2 9: '*', 10: ',', 11: '-', 12: '.', 13: '/', 14: '0', 15: '1', 16: '2',
3 17: '3', 18: '4', 19: '5', 20: '6', 21: '7', 22: '8', 23: '9', 24: ':',
4 25: ';', 26: '?', 27: 'A', 28: 'B', 29: 'C', 30: 'D', 31: 'E', 32: 'F',
5 33: 'G', 34: 'H', 35: 'I', 36: 'J', 37: 'K', 38: 'L', 39: 'M', 40: 'N',
6 41: 'O', 42: 'P', 43: 'Q', 44: 'R', 45: 'S', 46: 'T', 47: 'U', 48: 'V',
7 49: 'W', 50: 'X', 51: 'Y', 52: 'Z', 53: '[', 54: ']', 55: '_', 56: 'a',
8 57: 'b', 58: 'c', 59: 'd', 60: 'e', 61: 'f', 62: 'g', 63: 'h', 64: 'i',
9 65: 'j', 66: 'k', 67: 'l', 68: 'm', 69: 'n', 70: 'o', 71: 'p', 72: 'q',
10 73: 'r', 74: 's', 75: 't', 76: 'u', 77: 'v', 78: 'w', 79: 'x', 80: 'y',
11 81: 'z', 82: 'ù', 83: '-', 84: "'", 85: '"', 86: '"', 87: '"', 88: '•',
12 89: '™', 90: '\uffeff'

```

Esse índice é aplicado no código abaixo para realizar a indexação inversa do texto, retornando-o em palavras.

```

1 # use the index to convert the integers back to letters, the index need to
  check all the values of the index to find the letter
2
3 decoded_text = [index[i] for i in decoded_text]
4
5 print("Decoded text:", decoded_text[:100])

```

### 3.1.4. Exportando decodificado em um arquivo:

Por fim, é possível exportar o texto decodificado para um arquivo de texto. O código abaixo realiza essa operação:

```
1 # print the text into a file "huff_decoded.txt"
2
3 with open("huff_decoded.txt", "w") as file:
4     file.write("".join(decoded_text))
```

## 4. Conclusão

Neste relatório, foi apresentado um estudo sobre o algoritmo de Huffman, incluindo a leitura de um arquivo de texto, a contagem de ocorrências de cada caractere, a análise dos caracteres contidos no arquivo, o cálculo do percentual de cada caractere, a criação de uma Função de Massa de Probabilidade (PMF), a aplicação do algoritmo de Huffman sobre o texto, a codificação do texto e a análise do arquivo codificado.

O algoritmo de Huffman é um método de compressão de dados que utiliza a codificação de caracteres para reduzir o tamanho de um arquivo. O algoritmo foi desenvolvido por David A. Huffman em 1952, e é amplamente utilizado em sistemas de comunicação e armazenamento de dados.