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

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 Teorica

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 $p_x = \left[\frac{3}{10}, \frac{3}{10}, \frac{1}{10}\right]$

2.1.1.1. Item A

- Calcule a entropia da fonte.

```
pmf = [3/10, 6/10, 1/10]
BFR = math.ceil(math.log2(len(pmf)))

print(f"Numero de simbolos: {BFR}")
entropy = -sum(p * math.log2(p) for p in pmf)

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

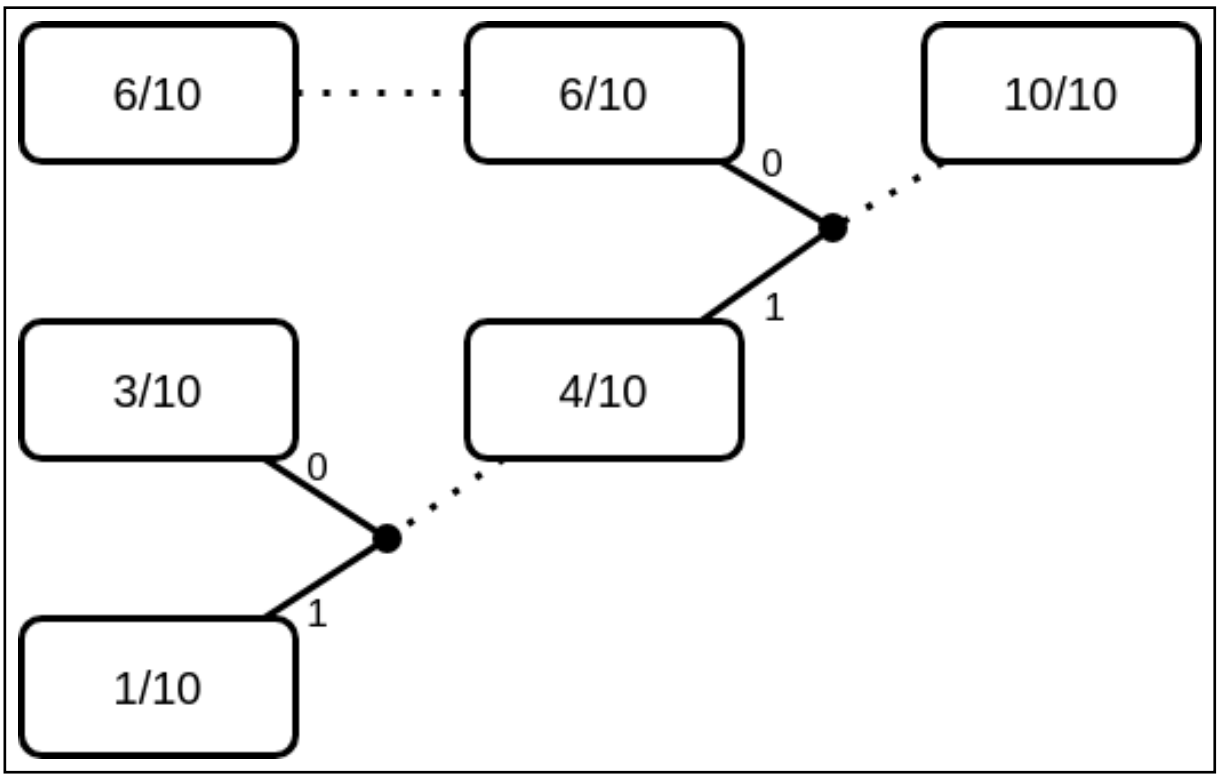
- Determine um código de Huffman para a fonte. QUal o comprimento médio do código obtido?

```
code = komm.FixedToVariableCode.from_codewords(3, [(1,0), (0,), (1,1)])

print("Unicamente decodificavel: ", code.is_uniquely_decodable())
print("Prefixo Livre:", code.is_prefix_free())
print("Huffman Rate: ", code.rate(pmf))
print("Huffman Codewords: ", code.codewords)

# Calculate the compress ratio
```

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

- Unicamente decodificavel: True
- Prefixo Livre: True
- Huffman Rate: 1.4
- Huffman Codewords: [(1, 0), (0,), (1, 1)]
- Compress Ratio: 0.6000000000000001

Figura 1: Elaborada pelo Autor



2.1.1.3. Item C

- Calcule a extensão e segunda ordem da fonte.

```
# Generate the PMF of the second order
pmf_2nd = [p1 * p2
           for p1, p2 in itertools.product(pmf, repeat=2)]

# Print the PMF of the second order
print("Tabela de PMF de segunda ordem:")
for i, p in enumerate(pmf_2nd):
    print(f"p({i//3}, {i%3}) = {p:.4f}")
print("\n")

# Calculate the entropy of the second order
entropy_2nd = -sum(p * math.log2(p) for p in pmf_2nd if p > 0)
print(f"Entropia de segunda ordem: {entropy_2nd:.4f} bits por par de
símbolos")

# Calculate the average entropy per symbol
```

```
entropy_per_symbol = entropy_2nd / 2
print(f"Entropia média por símbolo (segunda ordem): {entropy_per_symbol:.4f} bits/símbolo")
```

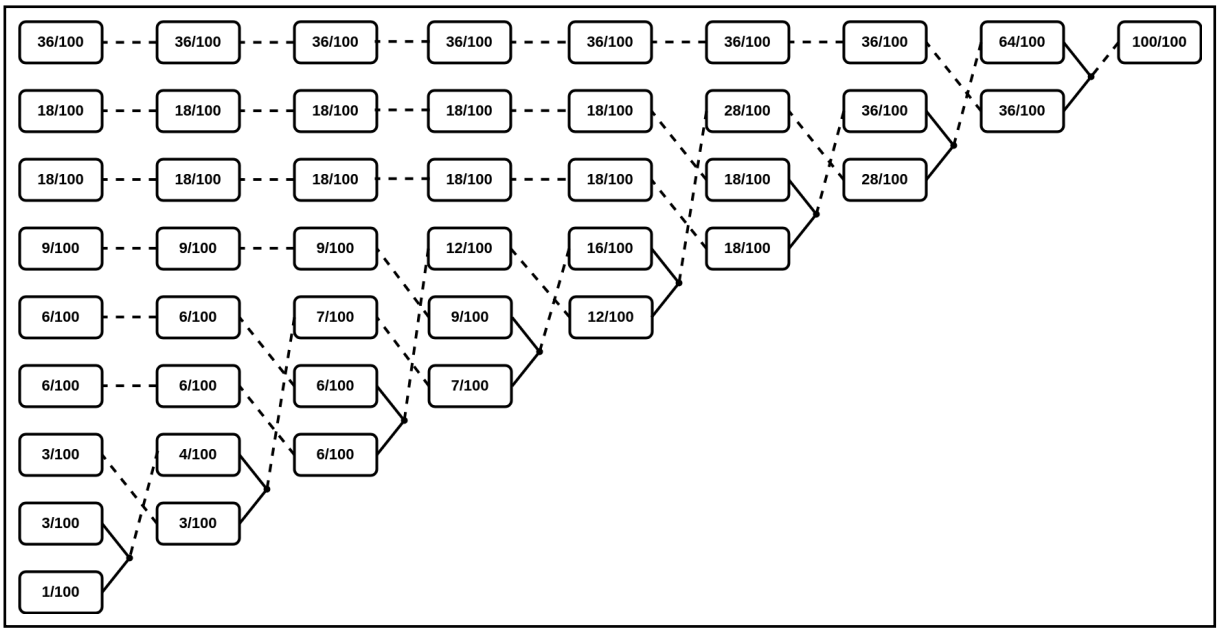
- Tabela de PMF de segunda ordem:
- p(0, 0) = 0.0900
- p(0, 1) = 0.1800
- p(0, 2) = 0.0300
- p(1, 0) = 0.1800
- p(1, 1) = 0.3600
- p(1, 2) = 0.0600
- p(2, 0) = 0.0300
- p(2, 1) = 0.0600
- p(2, 2) = 0.0100

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

Dessa forma, resulta-se no seguinte diagrama

Figura 2: Elaborada pelo Autor



Dessa forma, podemos verificar da seguinte maneira:

```
#print the huufman codewords in a more readable way (one per line)
print("Huffman Codewords:")
for i, c in enumerate(huff.codewords):
    print(f"p({i//3}, {i%3}) = {c}")
```

- Huffman Codewords:
- p(0, 0) = (0, 1, 0, 0)

- p(0, 1) = (1, 1)
- p(0, 2) = (0, 1, 0, 1, 0, 0)
- p(1, 0) = (1, 0)
- p(1, 1) = (0, 0)
- p(1, 2) = (0, 1, 1, 1)
- p(2, 0) = (0, 1, 0, 1, 1)
- p(2, 1) = (0, 1, 1, 0)
- p(2, 2) = (0, 1, 0, 1, 0, 1)

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?

```
huff = komm.HuffmanCode(pmf_2nd)

print("Huffman Ratio: ", huff.rate(pmf_2nd))
print("Huffman Code: ", huff.codewords)

# Calculate the compress ratio
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:

```
# Reed the file and count the number of occurrences of each character
with open("alice.txt", "r", encoding="utf-8") as file:
    # Ler o arquivo
    text = file.read()

    # Create a variable to store the characters and their occurrences
    characters = {}

    for char in text:
        # Check if the character is on the variable
        if char in characters:
            # Increment the character
            characters[char] += 1
        else:
            # Add the character to the variable and set it to 1
            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 contem 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.

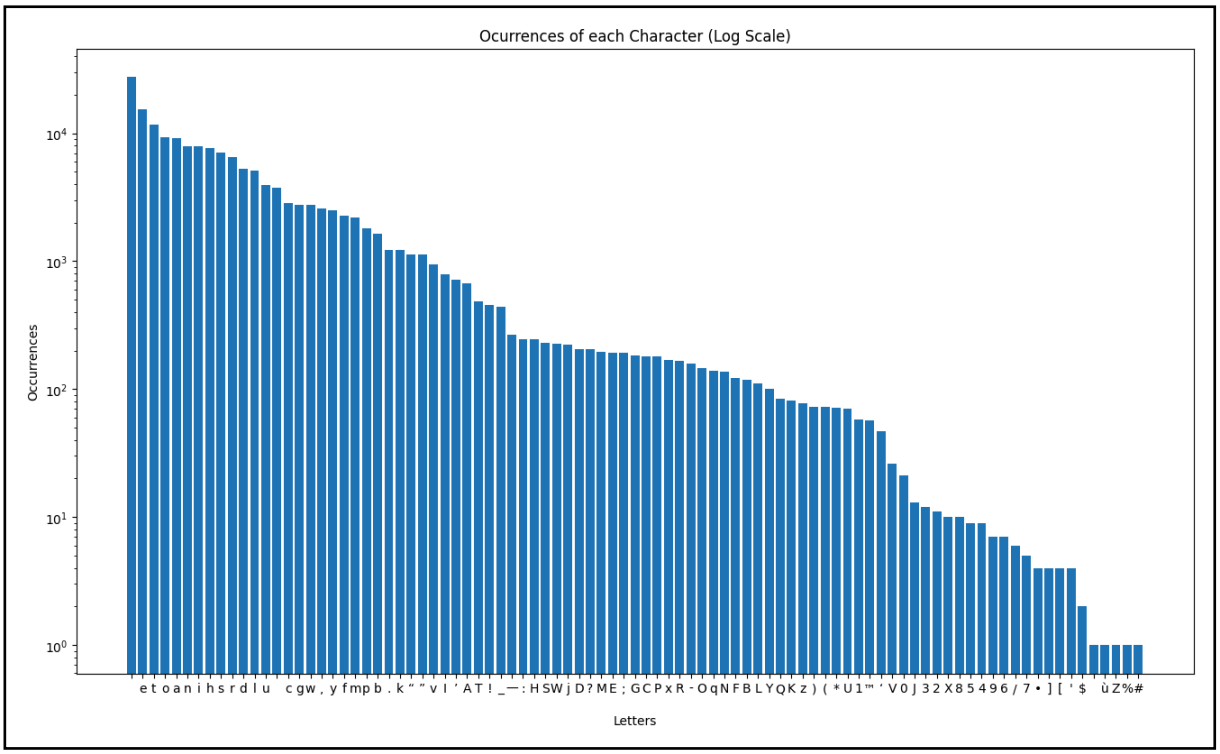
2.2.2. Análise dos caracteres contidos

Na sequencia, 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),
   reverse=True)]
2
3  sorted_occurrences = [occurrence for occurrence, _ in
   sorted(zip(occurrences, letters), reverse=True)]
4
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("Ocurrences of each Character (Log Scale)")
11 plt.show( )
```

Com base neste código, é gerado a seguinte figura:

Figura 3: 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:

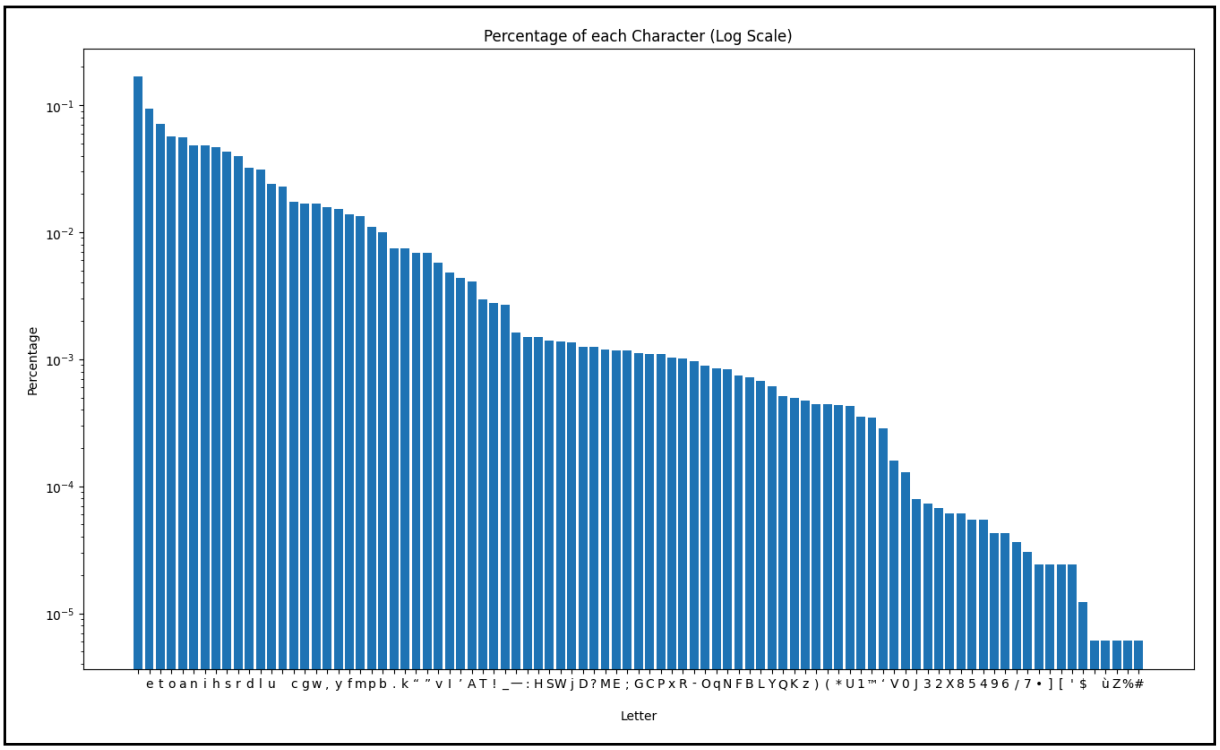
Total de caracteres: 163919

```
# sum all the occurrences of the letters and create a graph with the percentage of each letter

# Get the total number of letters
total_letters = sum(occurrences)
print("Total letters:", total_letters)

# Calculate the percentage of each letter
def percentage(occurrences, total):
    return [occurrence / total for occurrence in occurrences]

percentages = percentage(occurrences, total_letters)

# sort the letters by occurrences to make the graph more readable
```

```
sorted_letters = [letter for _, letter in sorted(zip(occurrences, letters), reverse=True)]
sorted_percentages = [percentage for _, percentage in sorted(zip(occurrences, percentages), reverse=True)]

# Create the gprint("Letters:", letters) graph
plt.figure(figsize=(16,9))
plt.bar(sorted_letters, sorted_percentages)
plt.xlabel("Letter")
plt.ylabel("Percentage")
plt.title("Percentage of each Character (Log Scale)")
plt.yscale("log")
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 4: 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:

```
# create a pmf vector with the percentage of each letter

pmf = {letter: percentage for letter, percentage in zip(letters, percentages)}
print("PMF:", list(pmf.values()))

```

```
# Calculate the huff code
huff = komm.HuffmanCode(list(pmf.values()))

# Print the huff code values and the huff ratio
print("Huffman code:", huff.codewords)
print("Huff Ratio:", huff.rate(list(pmf.values())))
print("Compress Ratio:", BFR - huff.rate(list(pmf.values())))
```

Com base nisso, podemos determinar o codigo de huffman para cada caractere com base em sua participação do percentual total. Além disso, podemos determinar parâmetros resultantes do codigo 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 quandidate 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:

```
index = {i: letter for i, letter in enumerate(letters)}for i in range(len(letters)):
    print(letters[i], occurrences[i])

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.

```
# encode the text using the index to create a integer index of the letters in the text

encoded_text = [list(index.keys())[list(index.values()).index(letter)] for letter in text]

print("Encoded text (by index):", encoded_text)

# create a huffman code for the encoded text
huff_encoded = huff.encode(encoded_text)

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:

```
# export the huffman code to a file.
with open("huff_encoded.com2", "w") as file:
    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:

```
0000000101101111000101111100011100000010010101110101000100000101110110101
1001111011000101011100011010101111000000011101000100001111011000011111010
1101011011100010100111000010101110101100000000010111101101001010100011111100
0011100101100000011011000101101100111000110110110011011011000000000101101 1
1000000111100010101000100001111011100000010111111000000011010011011111010 10
0001111100001111110011011011101100010111111001101101110101011011011110001 1
1111100101101001101111000000001000101111000100111011000000000110001001001 01
0011100001100110110110010100110000000010100100001001101111000110001000100 10
0011010000001100001111101001101111010110100000011011010110010010100101110 0
0010100100000000101001001101101100010110110001001101100101101101001100000 00
1010010111000000111110000101000001111000101000100110010001011000010101101 1
0110010100000011000111000000101110001100010010111100010101101001100101110 1
0101001000000100001110111000100010101110000000010111110001001001010110100 1
0111011000000110001111110101000011111100010010101101101100111000110100110 1
1110100111000111010010000110000111110100110111100010111110001110000001001 0
1011101010001000001011101101010100111101100010101110001101010100000001111 1
[...]
```

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:

```
# read the huffman code from the file

with open("huff_encoded.com2", "r") as file:
    huff_encoded = file.read()

# decode the huffman code
huffDecode = komm.HuffmanCode(list(pmf.values()))
print("Huffman code:", huffDecode.codewords)
```

Com base no codigo 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 possivel decodificar o texto original. Esse processo e realizado através do código abaixo:

```
# Decode the encoded text

# Convert the string back to a list of integers
huff_encoded_list = list(map(int, huff_encoded))

decoded_text = huffDecode.decode(huff_encoded_list)

# print more values of the decoded text
print("Decoded text:", decoded_text[:100])
```

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

```
90 46 63 60  1 42 73 70 65 60 58 75  1 33 76 75 60 69 57 60 73 62  1 60
28 70 70 66  1 70 61  1 27 67 64 58 60  6 74  1 27 59 77 60 69 75 76 73
60 74  1 64 69  1 49 70 69 59 60 73 67 56 69 59  0  1  1  1  1  0 46 63
64 74  1 60 57 70 70 66  1 64 74  1 61 70 73  1 75 63 60  1 76 74 60  1
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 index utilizado no processo de codificação (gerado automaticamente durante a codificação).

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

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

```
# 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

decoded_text = [index[i] for i in decoded_text]

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:

```
# print the text into a file "huff_decoded.txt"

with open("huff_decoded.txt", "w") as file:
    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.