INSTITUTO FEDERAL
Santa Catarina
Câmpus São José

Avaliação 5

Códigos de Huffman

Disciplina: COM029008 - SISTEMAS DE COMUNICAÇÃO II (2025 .2 - T01)
Professor: Roberto Wanderley da Nóbrega
Aluno: Wagner Santos

Dezembro de 2025

Sumário

1. Questão 1

Considere uma fonte discreta sem memória (DMS) com alfabeto dado por $\mathcal{X} = \{a, b, c\}$ e probabilidades respectivas dadas por $p\text{x} = \left[\frac{3}{10}, \frac{6}{10}, \frac{1}{10}\right]$.

1.1. Calcule a entropia da fonte.

Fonte:

$$\mathcal{X} = \{a, b, c\}, \quad p\text{x} = \left[\frac{3}{10}, \frac{6}{10}, \frac{1}{10}\right] \tag{1}$$

Definição:

$$H(X) = \sum_{x \in X} p(x) \log_2 \left(\frac{1}{p(x)}\right) \tag{2}$$

Substituindo:

$$H(X) = 0.3 \log_2 \left(\frac{1}{0.3}\right) + 0.6 \log_2 \left(\frac{1}{0.6}\right) + 0.1 \log_2 \left(\frac{1}{0.1}\right) \approx 1.295 \tag{3}$$

Assim:

$$H(X) \approx 1.296 \text{ bits} \tag{4}$$

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

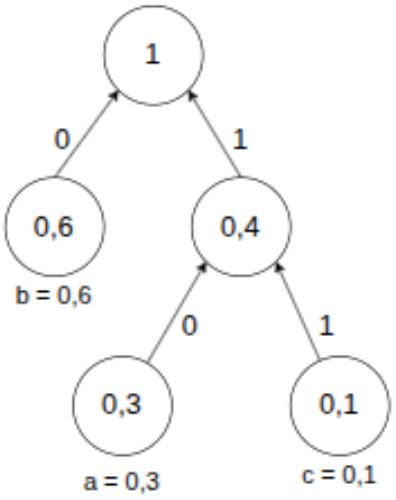

Figure 1: Elaborada pelo autor

Um código de Huffman:

$$\begin{aligned} b(0.6) &\rightarrow 0 \\ a(0.3) &\rightarrow 10 \\ c(0.1) &\rightarrow 11 \end{aligned} \tag{5}$$

Comprimento médio:

$$L(c, x) = 0.6 * 1 + 0.3 * 2 + 0.1 * 2 = 1.4 \text{ bits/letra} \tag{6}$$

1.3. Calcule a entropia da extensão de segunda ordem da fonte.

Par	Cálculo	Probabilidade
bb	$0.6 * 0.6$	0.36
ba	$0.6 * 0.3$	0.18
ab	$0.3 * 0.6$	0.18
bc	$0.6 * 0.1$	0.06
cb	$0.1 * 0.6$	0.06
aa	$0.3 * 0.3$	0.09
ac	$0.3 * 0.1$	0.03
ca	$0.1 * 0.3$	0.03
cc	$0.1 * 0.1$	0.01

Table 1: Elaborada pelo autor

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

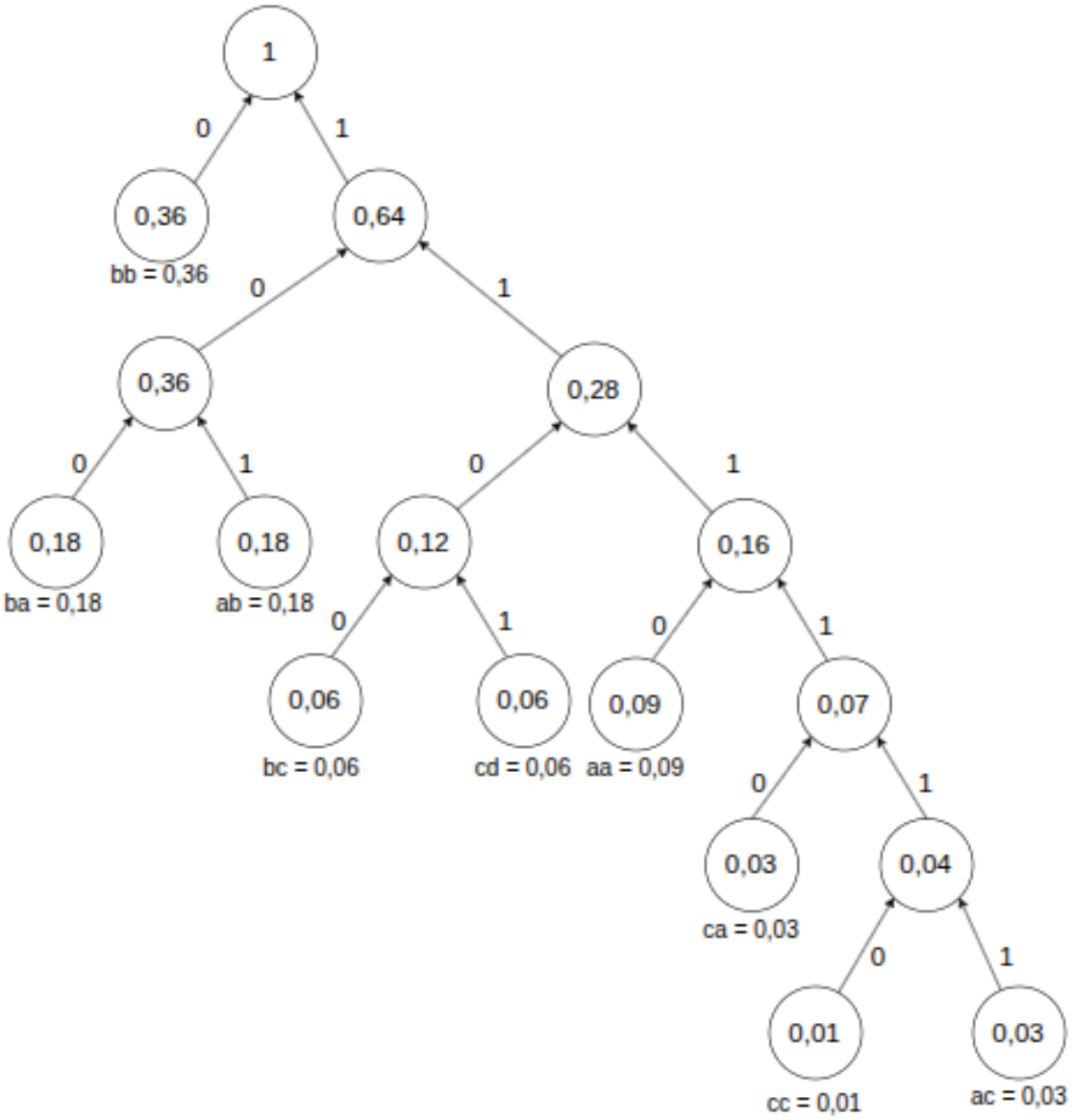

Figure 2: Elaborada pelo autor

Convenção: esquerda = 0 e direita = 1.

Códigos obtidos (caminho da raiz até a folha)

Par	Probabilidade	Código	Nº de bits
bb	0.36	0	1
ba	0.18	100	3
ab	0.18	101	3
aa	0.09	1110	4
bc	0.06	1100	4
cb	0.06	1101	4
ca	0.03	11110	5
ac	0.03	111111	6
cc	0.01	111110	6

Table 2: Elaborada pelo autor

Comprimento médio por par:

$$\begin{aligned} L(C^2, X^2) = \\ 0.36 * 1 \\ +0.18 * 3 \\ +0.18 * 3 \\ +0.09 * 4 \\ +0.06 * 4 \\ +0.06 * 4 \\ +0.03 * 5 \\ +0.03 * 6 \\ +0.01 * 6 \\ = 2.79 \text{ bits/par} \end{aligned} \tag{7}$$

Comprimento médio por símbolo original:

$$\frac{L(C^2, X^2)}{2} = \frac{2.79}{2} = 1.395 \text{ bits/simbolo} \tag{8}$$

Ao considerar a extensão de segunda ordem da fonte, observa-se que o comprimento médio por símbolo reduz-se para 1.395 bits/símbolo. Como visto nos slides, a utilização de extensões maiores permite explorar melhor as probabilidades conjuntas, aproximando o comprimento médio da entropia da fonte. Nota-se ainda que o valor obtido permanece acima da entropia, conforme previsto pelo limite de Shannon para códigos prefixo.

2. Questão 2

Escreva um programa para comprimir e descomprimir arquivos de texto usando códigos de Huffman. Seu programa deve:

2.1. Determinar a frequência de cada caractere do arquivo de entrada.

Comentários:
O arquivo é lido e cada byte é tratado como um símbolo (caractere em nível de byte). Em seguida, a frequência absoluta de cada símbolo é contada com Counter:

Trecho do código correspondente:

```
with open("Alice.txt", "rb") as f:
    data = f.read()

livro = list(data)
freq = Counter(livro)
```

2.2. Utilizar essas frequências para construir o código de Huffman.

Comentários:
As frequências são convertidas em probabilidades (PMF) e essa distribuição é usada para construir o código de Huffman:

Trecho do código correspondente:

```
pmf = np.zeros(256, dtype=float)
N = len(livro)

for i in range(256):
    pmf[i] = freq[i] / N

huff = komm.HuffmanCode(pmf=pmf)
```

2.3. Comprimir o arquivo de entrada .txt usando o código de Huffman, gerando um arquivo de saída com extensão .bin.

Comentários:
O conteúdo do arquivo é codificado em bits usando Huffman, depois os bits são empacotados em bytes e gravados em um arquivo .bin:

Trecho do código correspondente:

```
bits = huff.encode(livro)
payload, n_bits = pack_bits(bits)
```

```
with open("Alice_huff.bin", "wb") as f:
    f.write(int(N).to_bytes(8, "big"))
    f.write(int(n_bits).to_bytes(8, "big"))
    f.write(payload)
```

2.4. Descomprimir o arquivo de saída .bin , gerando um arquivo .txt idêntico ao arquivo de entrada.

Por simplicidade, assuma que o código de Huffman seja conhecido tanto na compressão quanto na descompressão — na prática, o código deve ser armazenado no arquivo de saída para que o arquivo de entrada possa ser descomprimido.

Teste seu programa com o livro Alice's Adventures in Wonderland, de Lewis Carroll, disponível em https://www.gutenberg.org/files/11/11-0.txt.

Comentários:
Na descompressão, o .bin é lido, o fluxo de bits é recuperado e decodificado usando a tabela de códigos salva (Alice_codebook.json). O arquivo de saída é gravado e comparado com o original:

Trecho do código correspondente:

```
with open("Alice_huff.bin", "rb") as f:
    N_r = int.from_bytes(f.read(8), "big")
    n_bits_r = int.from_bytes(f.read(8), "big")
    payload_r = f.read()

bits_r = unpack_bits(payload_r, n_bits_r)

decoded = []
current = ""
for b in bits_r:
    current += str(b)
    if current in decode_table:
        decoded.append(decode_table[current])
        current = ""
        if len(decoded) == N_r:
            break

with open("Alice_huff_out.txt", "wb") as f:
    f.write(bytes(decoded))

assert data == bytes(decoded)
```

O assert confirma que o arquivo descomprimido (Alice_huff_out.txt) é idêntico ao arquivo original (Alice.txt).

2.5. (a) A entropia da distribuição de frequências dos caracteres do livro.

Trecho do código correspondente:

```
# Entropia da fonte
H = 0.0
for p in pmf:
    if p > 0:
        H += p * np.log2(1 / p)

print(f"Entropia da fonte H(X)              : {H:.4f} bits/símbolo")
```

Saída:

```
Entropia da fonte H(X)              : 4.6420 bits/símbolo
```

2.6. (b) O comprimento médio do código de Huffman obtido.

Trecho do código correspondente:

```
# Comprimento médio do código de Huffman
L_med = comp_bits / len(data)
print(f"Comprimento médio do Huffman        : {L_med:.4f} bits/símbolo")
```

Saída:

```
Comprimento médio do Huffman        : 4.6840 bits/símbolo
```

2.7. (c) O tamanho (em bytes) e a taxa de compressão do arquivo comprimido.

No código, o tamanho dos arquivos em bytes é obtido diretamente a partir do sistema de arquivos:

```
print(f"Tamanho Alice.txt (bytes)           : {len(data)}")
print(f"Tamanho Alice_huff.bin (bytes)      :
{os.path.getsize('Alice_huff.bin')}")
```

O número de bits válidos gerados pela codificação de Huffman é armazenado na variável n_bits:

```
comp_bits = n_bits
orig_bits = len(data) * 8
```

Saída:

```
Tamanho Alice.txt (bytes)           : 167546
Tamanho Alice_huff.bin (bytes)      : 98114
```

A taxa de compressão é calculada conforme a definição utilizada nos slides:

$$\text{Taxa} = \left(1 - \frac{T_{\text{orig}}}{T_{\text{comp}}}\right) \times 100\% \tag{9}$$

```
print(
    f"Taxa de compressão (Huffman)        : "
    f"{(1 - comp_bits / orig_bits):.2%}"
)
```

Saída:

```
Taxa de compressão (Huffman)        : 41.45%
```

2.8. Compare com a tabela a seguir, que mostra o tamanho do arquivo original e dos arquivos comprimidos com diferentes formatos de compressão

Formato	Tamanho (bytes)	Taxa de compressão
original	154573	0.00%
zip	54176	64.95%
gz	54037	65.04%
zst	48789	68.44%
7z	48280	68.77%
xz	48232	68.80%
bz2	42779	72.32%
bz3	40362	73.89%

A tabela a seguir apresenta o tamanho do arquivo original e os tamanhos obtidos por diferentes

algoritmos de compressão amplamente utilizados. Em seguida, são comparados esses valores com o resultado obtido pela codificação de Huffman implementada neste trabalho.

Formato	**Tamanho (bytes)**	**Taxa de compressão**
original	154573	0.00%
zip	54176	64.95%
gz	54037	65.04%
zst	48789	68.44%
7z	48280	68.77%
xz	48232	68.80%
bz2	42779	72.32%
bz3	40362	73.89%
Huffman (este trabalho)	**98114**	**41.45%**

Observa-se que a taxa de compressão obtida pelo código de Huffman é inferior à dos formatos apresentados na tabela. Esse resultado é esperado, uma vez que algoritmos como ZIP, GZ e XZ combinam a codificação estatística com técnicas adicionais de compressão por dicionário e modelagem de dependências entre símbolos, enquanto o código de Huffman utiliza apenas as probabilidades individuais dos símbolos.

2.9. Cógigo python

```
from collections import Counter
import numpy as np
import komm
import os
import json

# ------------------------------------------------
# Funções auxiliares: bits <-> bytes
# ------------------------------------------------

def pack_bits(bits):
    bits = list(map(int, bits))
    n_bits = len(bits)

    pad = (-n_bits) % 8
    if pad:
        bits.extend([0] * pad)

    out = bytearray()
    for i in range(0, len(bits), 8):
        byte = 0
        for b in bits[i:i+8]:
```

```
            byte = (byte << 1) | b
        out.append(byte)

    return bytes(out), n_bits


def unpack_bits(payload, n_bits):
    bits = []
    for byte in payload:
        for k in range(7, -1, -1):
            bits.append((byte >> k) & 1)
    return bits[:n_bits]


# ------------------------------------------------
# 1) Leitura do arquivo
# ------------------------------------------------

with open("Alice.txt", "rb") as f:
    data = f.read()

livro = list(data)          # símbolos = bytes
freq = Counter(livro)

print(len(set(livro)))      # número de símbolos distintos

# ------------------------------------------------
# 2) Frequências, PMF e Entropia
# ------------------------------------------------

pmf = np.zeros(256, dtype=float)
N = len(livro)

for i in range(256):
    pmf[i] = freq[i] / N

# Entropia da fonte
H = 0.0
for p in pmf:
    if p > 0:
        H += p * np.log2(1 / p)

print(f"Entropia da fonte H(X)              : {H:.4f} bits/símbolo")

# ------------------------------------------------
# 3) Código de Huffman
# ------------------------------------------------

huff = komm.HuffmanCode(pmf=pmf)

# Tabela símbolo -> código (string de bits)
codebook = {
```

```
    i: ''.join(map(str, huff.codewords[i]))
    for i in range(256) if pmf[i] > 0
}

# ------------------------------------------------
# 4) Compressão Huffman e escrita do .bin
# ------------------------------------------------

bits = huff.encode(livro)
payload, n_bits = pack_bits(bits)

with open("Alice_huff.bin", "wb") as f:
    f.write(int(N).to_bytes(8, "big"))        # número de símbolos
    f.write(int(n_bits).to_bytes(8, "big"))   # número de bits válidos
    f.write(payload)

with open("Alice_codebook.json", "w") as f:
    json.dump(codebook, f)

print("OK: arquivo Alice_huff.bin gerado.")

# ------------------------------------------------
# 5) Descompressão Huffman
# ------------------------------------------------

with open("Alice_codebook.json", "r") as f:
    codebook = json.load(f)

decode_table = {v: int(k) for k, v in codebook.items()}

with open("Alice_huff.bin", "rb") as f:
    N_r = int.from_bytes(f.read(8), "big")
    n_bits_r = int.from_bytes(f.read(8), "big")
    payload_r = f.read()

bits_r = unpack_bits(payload_r, n_bits_r)

decoded = []
current = ""

for b in bits_r:
    current += str(b)
    if current in decode_table:
        decoded.append(decode_table[current])
        current = ""
        if len(decoded) == N_r:
            break

with open("Alice_huff_out.txt", "wb") as f:
    f.write(bytes(decoded))

# Verificação de integridade
```

```python
assert data == bytes(decoded)
print("OK: descompressão Huffman idêntica ao original.")

# ------------------------------------------------
# 6) Métricas finais
# ------------------------------------------------

orig_bits = len(data) * 8
comp_bits = n_bits

L_med = comp_bits / len(data)

print(f"Comprimento médio do Huffman        : {L_med:.4f} bits/símbolo")
print(f"Tamanho original (bits)             : {orig_bits}")
print(f"Tamanho comprimido (Huffman, bits)  : {comp_bits}")
print(f"Taxa de compressão (Huffman)        : {(1 - comp_bits /
orig_bits):.2%}")
print(f"Tamanho Alice.txt (bytes)           : {len(data)}")
print(f"Tamanho Alice_huff.bin (bytes)      :
{os.path.getsize('Alice_huff.bin')}")
print(f"Tamanho Alice_huff_out.txt
(bytes)  : {os.path.getsize('Alice_huff_out.txt')}")
```

2.10. Saída

```
(.venv) wagner-ThinkPad-Edge-E431% /home/wagner/Documentos/IFSC/2025.2/
COM2/.venv/bin/python /home/wagner/Docum
entos/IFSC/2025.2/COM2/alice.py
85
Entropia da fonte H(X)              : 4.6420 bits/símbolo
OK: arquivo Alice_huff.bin gerado.
OK: descompressão Huffman idêntica ao original.
Comprimento médio do Huffman        : 4.6840 bits/símbolo
Tamanho original (bits)             : 1340368
Tamanho comprimido (Huffman, bits)  : 784782
Taxa de compressão (Huffman)        : 41.45%
Tamanho Alice.txt (bytes)           : 167546
Tamanho Alice_huff.bin (bytes)      : 98114
Tamanho Alice_huff_out.txt (bytes)  : 167546
(.venv) wagner-ThinkPad-Edge-E431%
```

2.11. Conclusão

Conclui-se que a codificação de Huffman é capaz de reduzir significativamente a redundância do texto, produzindo um comprimento médio coerente com os limites teóricos da entropia da fonte. Embora apresente desempenho inferior a algoritmos de compressão

modernos, os resultados obtidos confirmam, na prática, os conceitos fundamentais da Teoria da Informação abordados em aula.