

**INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA DO
AMAZONAS
CAMPUS MANAUS ZONA LESTE
CURSO BACHARELADO EM ENGENHARIA DE SOFTWARE**

Pedro Lucas Ferreira Carvalho

ALGORITMO DE COMPRESSÃO DE ARQUIVOS DE HUFFMAN:

Implementação em linguagem C usando tipos abstratos de dados

**Manaus – Am
2021**

Pedro Lucas Ferreira Carvalho

ALGORITMO DE COMPRESSÃO DE ARQUIVOS DE HUFFMAN:

Implementação em linguagem C usando tipos abstratos de dados

Trabalho de Pesquisa
solicitado na disciplina
ESW013 - Algoritmos e
Estruturas de Dados 2, para
substituição de avaliação de
Segunda Chamada da
disciplina que compõe a grade
do III Período do referido Curso

Professor: Me. Carlos Augusto
de Araújo Mar

Manaus – Am

2021

INTRODUÇÃO

Em computação, um problema recorrente diz respeito à otimização do uso de armazenamento de memória e processamento de arquivos. A compressão de arquivos surge como uma possibilidade para mitigar esse problema. O algoritmo de compressão de Huffman, desenvolvido em 1952 pelo então doutorando do MIT David A. Huffman se mostrou bastante eficiente para a tarefa, consistindo em um algoritmo que usa as probabilidades de ocorrência dos símbolos no conjunto de dados a ser comprimido para determinar códigos de tamanho variável para cada símbolo. O algoritmo de Huffman recebe um fluxo de bits e devolve um fluxo de bits comprimido que representa o fluxo original. Em geral, o fluxo comprimido é mais curto que o original.

Assim sendo, este algoritmo basicamente cria uma árvore binária recursiva, criando um formato de arquivo binário onde menos bits são atribuídos a caracteres e símbolos mais recorrentes e por outro lado mais bits para caracteres e símbolos menos recorrentes, com a compressão sendo baseado no fluxo de frequência desses caracteres e símbolos. O algoritmo pode ser utilizado em arquivos de texto, cuja compressão pela implementação do algoritmo desses arquivos é o objetivo deste trabalho, assim como em imagens, entre outros.

Este trabalho irá abordar as possibilidades da implementação da codificação de Huffman em linguagem C, abordando toda a construção das funções, algoritmos e estruturas de dados, buscando explicar a finalidade de cada parte integrante do código, que por fim será implementado e testado com diferentes tipos de arquivos, culminando na análise e explanação dos resultados desses testes.

IMPLEMENTAÇÃO DO ALGORITMO

Para a codificação e implementação do algoritmo, segue-se 5 passos de construção que possibilita o fácil entendimento do seu funcionamento, assim, fica-se dividido em 5 fases de implementação, são elas: **contagem de frequência de ocorrência de símbolos e caracteres, montagem da árvore binária baseada na frequência de ocorrência, leitura da árvore e criação de um “dicionário” com o**

novo código, organização e exibição desse novo código e por último a decodificação desse novo código comprimido.

- **Contagem de frequência de ocorrência de símbolos e caracteres**

Tomando como exemplo um arquivo de texto com a mensagem “pedro carvalho”: na prática, é montada uma tabela que registra a frequência de ocorrência de cada caractere/símbolo, dessa forma o registro da frequência vai balizar a construção da árvore binária.

Na tabela abaixo temos o registro da frequência:

CARACTERE	FREQUENCIA
p	1
e	1
d	1
r	2
o	2
c	1
a	2
v	1
l	1
h	1
espaço	1

A partir dessa tabela, a frequência é ordenada do menor registro para o maior, assim:

CARACTERE	FREQUENCIA
p	1
e	1
c	1
v	1
l	1
h	1
d	1
espaço	1
r	2
o	2
a	2

Em linhas de código, usando a linguagem C, a função *countFreq* é responsável por todo esse processamento da entrada recebida até o ponto exemplificado:

```
//contar a frequência de ocorrência de caractere ou símbolo
void countFreq(FILE *entrada, unsigned int* listaTamanho){
    byte aux;
    while (fread(&aux, 1, 1, entrada) >= 1){
        listaTamanho[(byte)aux]++;
    }
    rewind(entrada);
}
```

Figura 1 - Trecho de código em linguagem C da função *countFreq*

- **Montagem da árvore binária baseada na frequência de ocorrência**

Para a montagem da árvore binária, usa-se uma lista baseada na tabela de frequência de recorrência ordenada, onde os nós da lista que tem a menor frequência são agrupados de dois em dois em um nó pai, cujo valor é a soma da frequência dos nós filhos.



Figura 2 - Lista encadeada da frequência de caracteres

A partir da execução da função que vai implementar a árvore binária, o resultado seria assim:

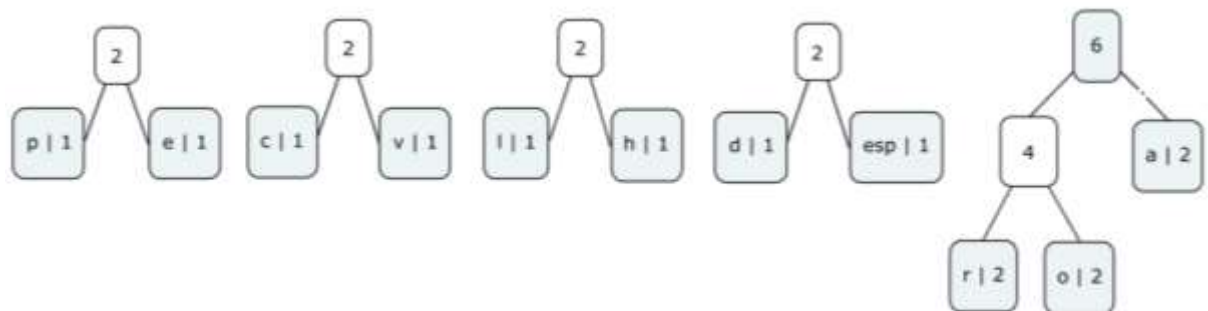


Figura 3 – Criação dos primeiros nós-pai da árvore

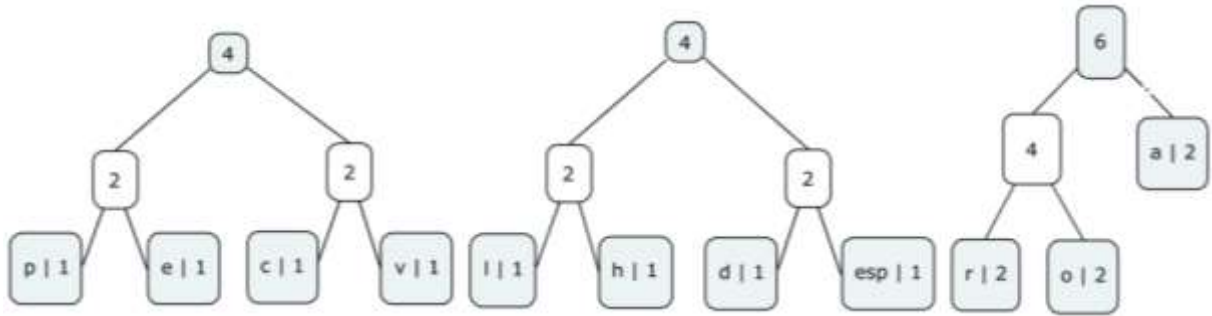


Figura 4 - Após outra execução da função

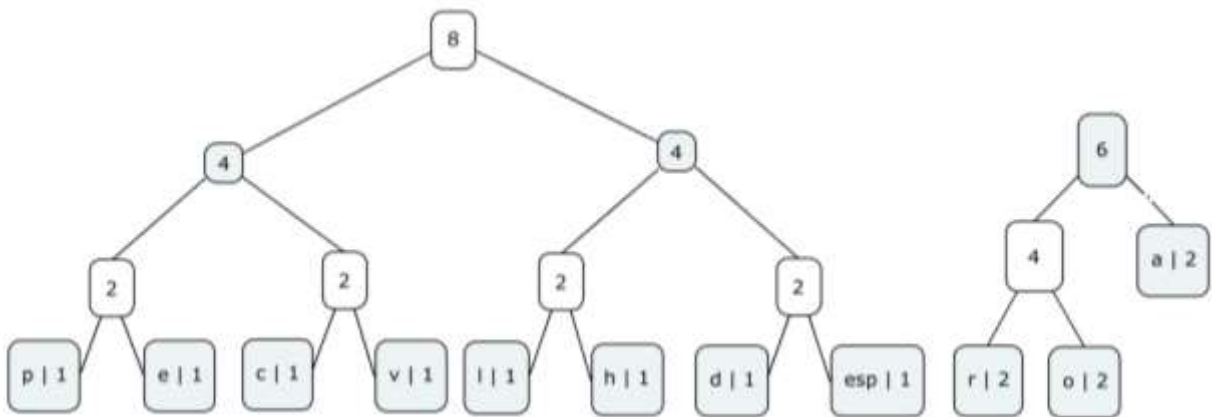


Figura 5 - Após outra execução da função

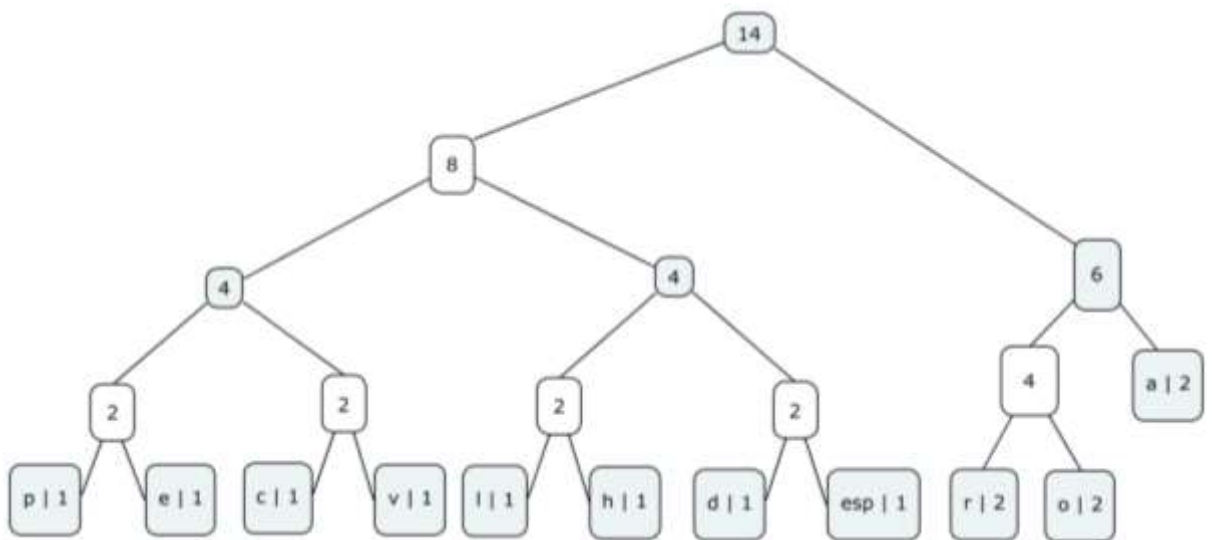


Figura 6 - Árvore finalizada

Por ser um algoritmo recursivo, a execução vai parar quando não existirem mais nós na lista de frequência de ocorrência, resultando assim na árvore binária como mostrada na imagem, onde cada par de filhos tem um nó pai. Em linguagem C, o trecho de código abaixo é responsável por esse processo:

```
//inserir na cabeca se a lista de frequencia estiver vazia ou nao
void insereCabList(NODO* n, lista* newElemento){
    if (!newElemento->cabeca){
        newElemento->cabeca=n;
    }
    else if (n->no->frequencia < newElemento->cabeca->no->frequencia){
        n->prox = newElemento->cabeca;
        newElemento->cabeca =n;
    } else{
        NODO *temp = newElemento->cabeca->prox;
        NODO *aux = newElemento->cabeca;
        while (temp && temp->no->frequencia <= n->no->frequencia){
            aux = temp;
            temp = aux->prox;
        }
        aux->prox = n;
        n->prox= temp;
    }
    newElemento->valor++;
}
```

Figura 7 – Função em C que insere na lista

```
//tirar o menor no da lista de frequencia
ARVORE* desenfileiraMenor(lista* elemento){
    NODO* aux = elemento->cabeca;
    ARVORE* aux2 = aux->no;
    elemento->cabeca = aux->prox;
    free(aux);
    aux = NULL;
    elemento->valor--;
    return aux2;
}

//contar a frequencia de ocorrencia de caractere ou símbolo
void countFreq(FILE *entrada, unsigned int* listaTamanho){
    byte aux;
    while (fread(&aux, 1, 1, entrada) >= 1){
        listaTamanho[(byte)aux]++;
    }
    rewind(entrada);
}
```

Figura 8 - Funções em C que fazem a junção dos nós filhos

- **Leitura da árvore e criação de um novo “dicionário”**

A partir da árvore binária, percorre-se a mesma relacionando o símbolo/caractere com a nova frequência de ocorrência, adicionando 0 aos nós à esquerda e 1 aos nós à direita, assim:

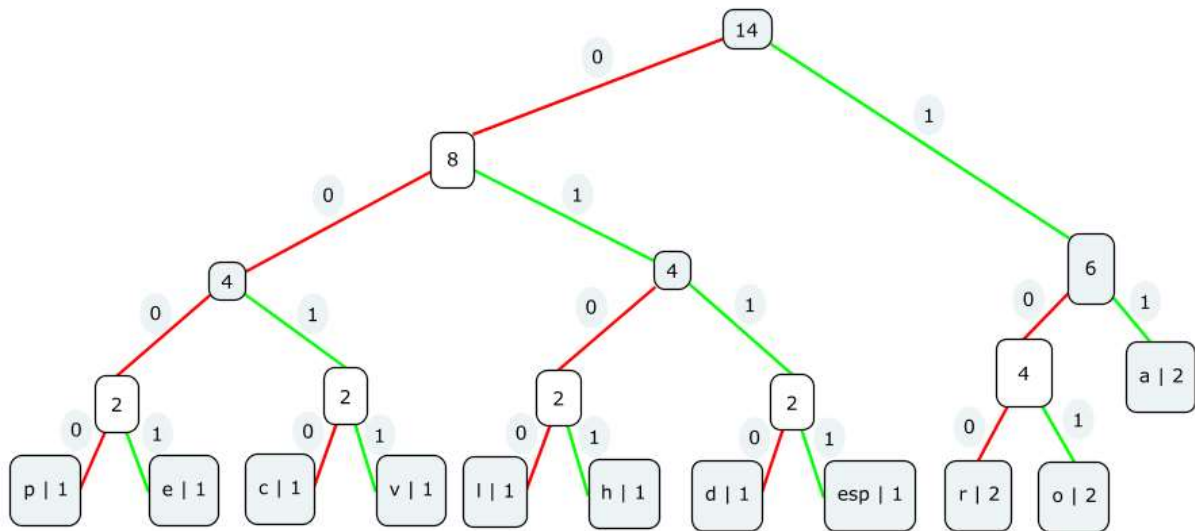


Figura 9 - Árvore com a nova frequência adicionada

A partir de então, monta-se a tabela de relacionamentos, que exibe a nova frequência obtida a partir da nova frequência.

CARACTERE	FREQUENCIA
p	0 0 0 0
e	0 0 0 1
c	0 0 1 0
v	0 0 1 1
l	0 1 0 0
h	0 1 0 1
d	0 1 1 0
espaço	0 1 1 1
r	1 0 0
o	1 0 1
a	1 1

A leitura da árvore é feita da esquerda para a direita, percorrendo os nós, assim que um nó que não possui descendente é encontrado, a frequência é adicionada à tabela. Em linguagem C, a função que cria a árvore de Huffman está representada na figura abaixo:

```
//criar a arvore de huffman
ARVORE* arvoreHuffman(unsigned* listaB){
    lista nLista = {NULL, 0};
    for (int i = 0; i < 256; i++){
        if (listaB [i]){
            insereCabList(novoNodoList(novoNodoArvore(i, listaB[i], NULL, NULL)), &nLista);
        }
    }
    while (nLista.valor > 1){
        ARVORE *noEsq = desenfileiraMenor(&nLista);
        ARVORE *noDir = desenfileiraMenor(&nLista);
        ARVORE *soma = novoNodoArvore('/', noEsq->frequencia + noDir->frequencia, noEsq, noDir);
        insereCabList(novoNodoList(soma), &nLista);
    }
    return desenfileiraMenor(&nLista);
}
```

Figura 10 - Função em C que cria a árvore binária

- **Exibição do novo código**

A partir do dicionário criado, é possível reescrever o conteúdo abordando somente a frequência que está na tabela, organizando a exibição da frequência binária conforme a frase é originalmente:

CARACTERE	FREQUENCIA
p	0 0 0 0
e	0 0 0 1
c	0 0 1 0
v	0 0 1 1
l	0 1 0 0
h	0 1 0 1
d	0 1 1 0
espaço	0 1 1 1
r	1 0 0
o	1 0 1
a	1 1

“0000 0001 0110 100 101 0111 0010 11 100 0011 11 0100 0101 101”

A troca do conteúdo pela representação em código binário representa uma economia de memória de armazenamento, o que indica que o algoritmo cumpre a sua função de compressão sem falhas ou perdas de conteúdo. Em linguagem C:

```
//verificar um nodo e retornar um bit
int returnBit(FILE* ent, int p, byte* n){
    (p % 8 == 0) ? fread(n, 1, 1, ent) : NULL == NULL;
    return !((*n) & (1 << (p % 8)));
}
```

Figura 11 - Função que verifica os nós e retorna o bit encontrado

• Decodificação

Nessa fase, o código binário é decodificado, sendo lido da esquerda para a direita e o original é reescrito conforme a frequência binária indica. A execução começa na raiz e pula de nó em nó, verificando os valores binários, 0 indica a esquerda e 1 indica a direita, ao encontrar o último nó o caractere que está nele é imprimido.

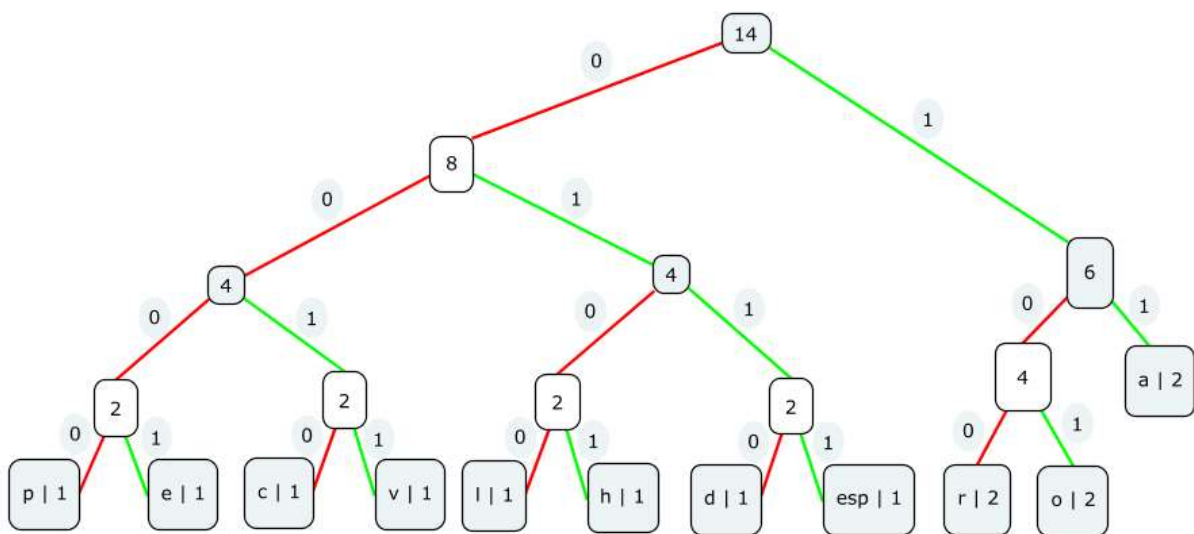


Figura 12 - Árvore que será decodificada no processo

Em linguagem C:

```
//buscar um determinado byte
bool buscaByte(ARVORE* n, byte c, char* busca, int tam){
    if (!(n->esquerda || n->direita) && n->ch==c){
        busca[tam] = '\0';
        return true;
    } else {
        bool alvo = false;
        if (n->esquerda){
            busca[tam] = '\0';
            alvo = buscaByte(n->esquerda, c, busca, tam+1);
        }
        if (!alvo && n->direita){
            busca[tam] = '1';
            alvo = buscaByte(n->direita, c, busca, tam+1);
        }
        if (!alvo){
            busca[tam] = '\0';
        }
        return alvo;
    }
}
```

Figura 13 - Função em C que busca e retorna os bytes

ANÁLISE DOS TESTES

O algoritmo é operado por linha de comando e a pouca familiaridade com os comandos para a operação dificultou o processo, mas mesmo assim foram realizados os testes com arquivos de imagem e de texto.

O teste com o arquivo de texto deu resultado, porém nada muito expressivo se comparado a outras ferramentas de compressão:

```
C:\Users\Pedro Lucas\Desktop\HUFFMAN TP>cd CODIGOS
C:\Users\Pedro Lucas\Desktop\HUFFMAN TP\CODIGOS>gcc -o huffmanEXE.c -c pedro.txt
gcc: warning: pedro.txt: linker input file unused because linking not done
C:\Users\Pedro Lucas\Desktop\HUFFMAN TP\CODIGOS>gcc -o huffmanEXE.c -c testePedro.txt
gcc: warning: testePedro.txt: linker input file unused because linking not done
C:\Users\Pedro Lucas\Desktop\HUFFMAN TP\CODIGOS>_
```

Figura 14 - Teste em linha de comando com arquivo de texto





 testePedro	Documento de Te...	1 KB
 pedro	Documento de Te...	1 KB
 pedro	Arquivo do WinRAR	1 KB
 testePedro	Arquivo do WinRAR	1 KB

Figura 15 - Comparação dos arquivos .txt e .rar

O segundo teste foi executado com um arquivo de imagem .png chamado carro e o resultado de compressão não foi diferente:

```
C:\Users\Pedro Lucas\Desktop\HUFFMAN TP\CODIGOS>gcc -o huffmanEXE.c -c carro.png
gcc: warning: carro.png: linker input file unused because linking not done
C:\Users\Pedro Lucas\Desktop\HUFFMAN TP\CODIGOS>_
```

Figura 16 - Teste com arquivo .png

 carro	Arquivo PNG	7 KB
 carro	Arquivo do WinRAR	7 KB

Figura 17 - Comparação dos arquivos .png e rar

Os testes evidenciaram que o poder de compressão do algoritmo é semelhante ao que é visto em ferramentas que já estão consolidados no mercado, mas ainda assim, o algoritmo não retorna valores satisfatórios com alguns tipos específicos de arquivos.

CONCLUSÃO

Após a implementação do Algoritmo de Huffman, ficou evidente as múltiplas possibilidades de uso para a manipulação de arquivos, em vista da massiva quantidade de dados que precisam ser armazenados. A execução do algoritmo deixa a desejar em virtude do tempo de processamento, quanto maior mais demorada a compressão e vice-versa, além do consumo de memória de processamento ser relativamente alta devido a complexidade do algoritmo, o que o caracteriza como um “algoritmo guloso”.

Codificar usando TADs facilita o entendimento de todas as partes integrantes do código, melhorando a autonomia de cada módulo de código e incitando o reuso de código, assim a criação da árvore binária foi facilitada. O algoritmo em geral se comportou bem, o que já era esperado visto que já é usado como base para várias outras ferramentas de compressão conhecidas. A estrutura binária garante que não ocorra ambiguidades e também evita perdas durante a compressão de arquivos e mensagens.

REFERÊNCIAS

BRUNO, Davidson. **Compressão de dados pelo algoritmo de Huffman**: Relato de experiência. Disponível em:

<<https://medium.com/@davidsonbrsilva/compress%C3%A3o-de-dados-pelo-algoritmo-de-huffman-5e04bc437d77>>. Acesso em: 15 de julho de 2021.

GEEKSFORGEEKS. Codificação de Huffman. Disponível em:

<<https://www.geeksforgeeks.org/huffman-coding-greedy-algo-3/?ref=rp>>. Acesso em: 18 de Julho de 2021.

GEEKSFORGEEKS. Decodificação de Huffman . Disponível em:

<<https://www.geeksforgeeks.org/huffman-decoding/?ref=rp>>. Acesso em: 18 de julho de 2021.

GEEKSFORGEEKS. Compressão de imagem usando codificação Huffman.

Disponível em: <<https://www.geeksforgeeks.org/image-compression-using-huffman-coding/?ref=rp>>. Acesso em 20 de julho de 2021.

HUFFMAN, David. A Method for the Construction of Minimum-Redundancy Codes. Massachusetts: Proceedings of the I.R.E., 1951.

IME. Algoritmo de Huffman para compressão de dados. Disponível em:

<<https://www.ime.usp.br/~pf/estruturas-de-dados/aulas/huffman.html>>. Acesso em: 17 de Julho de 2021.