

ARQUIVOS COMPACTADOS COM ALGORITMO DE HUFFMAN

Rodrigo Berino Pereira

Resumo: Criado pelo cientista da computação David Huffman, o algoritmo ou codificação de Huffman é utilizado para fazer a compactação de dados e usa um método de compressão baseado nas probabilidades de ocorrência de símbolos e caracteres no conjunto de dados a ser comprimido.

Palavras-chave: Algoritmo, compressão, dados, redução, árvore binária, desempenho.

Abstract: Made by the computer scientist David Huffman, that is, Huffman's algorithm or encoding is used to compress data and uses a compression method based on the probability of occurrence of symbols and characters not data sets to be compressed.

Palavras-chave: Algorithm, compression, data, reduction, binary tree, performance.

INTRODUÇÃO

Na computação, sabemos que um dos maiores problemas enfrentados pelos profissionais desta área é a otimização do uso de espaço do armazenamento de um arquivo. Para isso, estudaremos neste artigo um método eficaz desenvolvido por Huffman, capaz de fazer com que os dados sejam armazenados de maneira mais econômica possível.

A ideia inicial de Huffman foi criar um algoritmo para encontrar a codificação binária mais eficiente até aquele momento (1951, no MIT). Logo, conseguiu identificar a possibilidade de usar uma árvore binária utilizando frequências relativas para fazer a demonstração que precisava. A conceitualização é criar um formato binário onde são atribuídos menos bits para símbolos e caracteres mais frequentes e mais bits para símbolos e caracteres menos frequentes em um arquivo, seja ele de texto (preferencial), imagem ou áudio.

IMPLEMENTAÇÃO E EXPLANAÇÃO DO ALGORITMO

Inicialmente, para compreendermos o algoritmo de Huffman, dividiremos o seu passo a passo em 5 etapas com exemplos e demonstrações do comportamento da memória, na qual serão:

- Contagem de frequência de símbolos e caracteres.
- Montagem de uma árvore binária, agrupando símbolos e caracteres por sua frequência.
- Leitura da árvore para montar o dicionário com o novo código de cada símbolo.
- Mostrar os dados usando o dicionário criado.
- Decodificação

1.1 Contagem de frequência de símbolos e caracteres:

Para a contagem de frequência, um exemplo prático e comum utilizando a frase “Hello world!” pode ser desenvolvido usando a tabela abaixo:

S/CARACTERE	FREQUENCIA
H	1
e	1
l	3
o	2
w	1
r	1
d	1
!	1
espaço	1

Tabela de frequência.

Agora que a frequência foi montada, o processo ocorrerá de maneira bem simples e consistirá em ordenar essa tabela do símbolo menos frequente até o mais frequente.

S/CARACTERE	FREQUENCIA
H	1
e	1
w	1
r	1
d	1
!	1
espaço	1
o	2
l	3

Tabela de frequência ordenada.

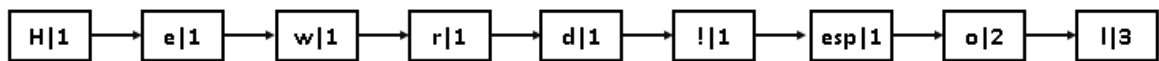
Usando a linguagem c, para realizarmos esse processo usaremos a seguinte função:

```
//funcao que conta frequencia de um simbolo
void contaFreq(FILE *ent, unsigned int* listaTam){
    byte aux;
    while (fread(&aux, 1, 1, ent) >= 1){
        listaTam[(byte)aux]++;
    }
    rewind(ent);
}
```

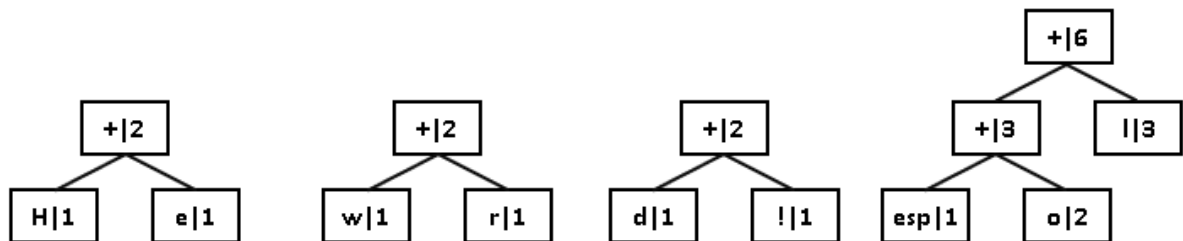
1. 2 Montagem de uma árvore binária, agrupando símbolos e caracteres por sua frequência:

Agora, inicia-se o processo de montagem da árvore binária. Nesse processo, é preciso retirar os dois nós de menor frequência da lista e agrupá-los em um nó pai, na qual o nó pai será a soma da frequência de seus nós filhos.

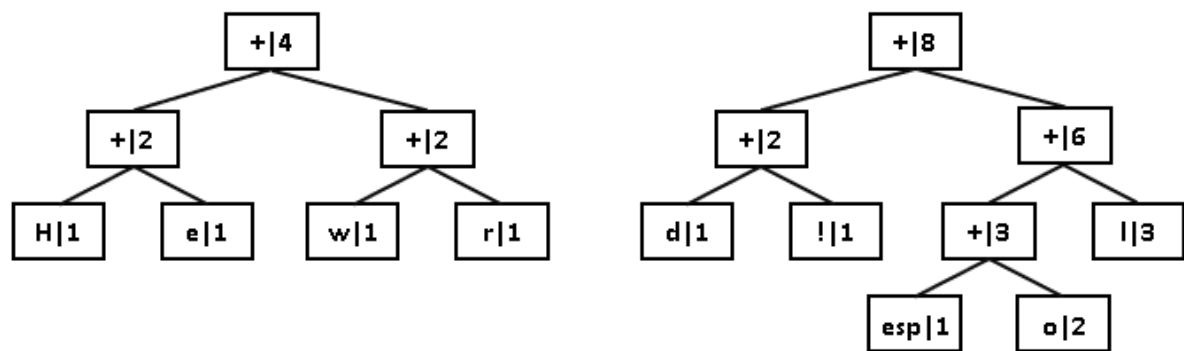
Lista:



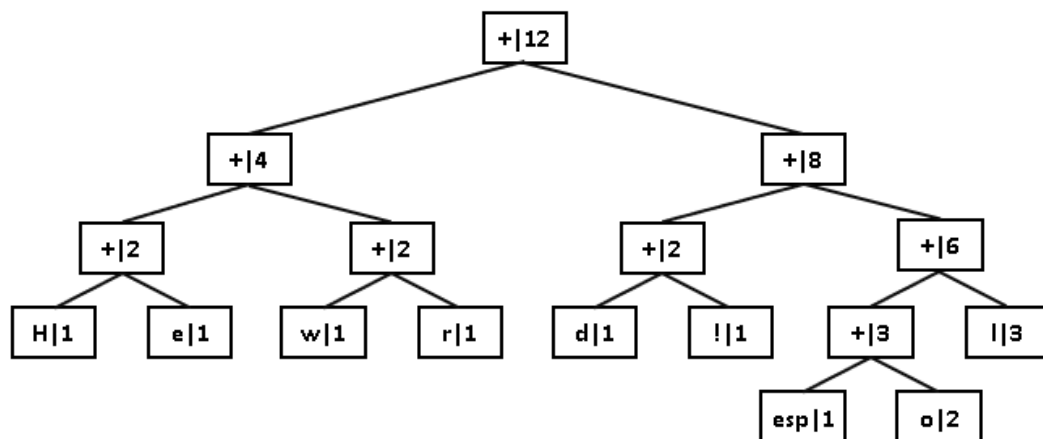
Assim que esse processo for executado, a lista terá a seguinte aparência:



Na próxima execução, a lista terá a seguinte aparência:



E por fim, teremos a nossa árvore binária pronta para execução:



Como é um processo recursivo, ele só acabará quando não houver mais nós na lista de frequência.

Usando a linguagem c, para realizarmos esse processo usaremos a seguinte função:

```

//funcao de insercao na cabeca se a lista de frequencia estiver vazia ou nao
void insereL(NO* n, lista* el){
    if (!el->cab){
        el->cab=n;
    }
    else if (n->n->freq < el->cab->n->freq){
        n->prox = el->cab;
        el->cab = n;
    } else{

```

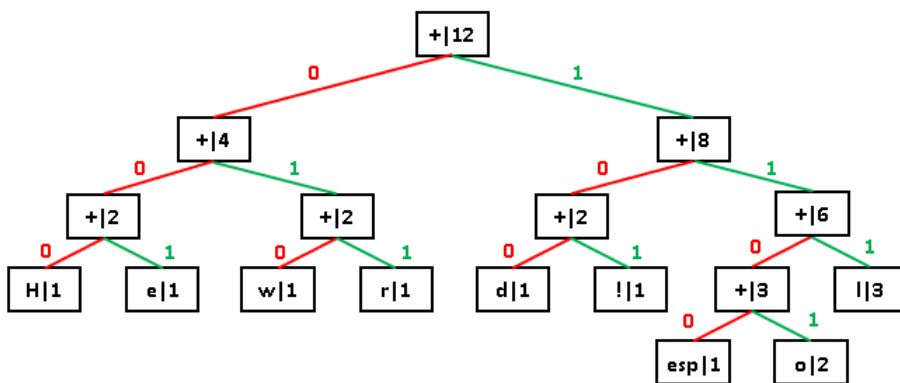
```

    NO *temp = el->cab->prox;
    NO *aux = el->cab;
    while (temp && temp->n->freq <= n->n->freq){
        aux = temp;
        temp = aux->prox;
    }
    aux->prox = n;
    n->prox = temp;
}
el->valor++;
}

```

1. 3 Leitura da árvore para montar o dicionário com o novo código de cada símbolo:

Agora, que a árvore binária foi criada, faz-se necessário montar a tabela que relaciona o símbolo ou caractere a nova sequência binária. Portanto, percorre-se a árvore colocando 0 nos nós a esquerda e 1 nos nós a direita.



S/CARACTERE	FREQUENCIA
H	0 0 0
e	0 0 1
w	0 1 0
r	1 1 0
d	1 0 0
!	1 0 1
espaço	1 1 0 0
o	1 1 0 1
l	1 1 1

Nesse processo, ao encontrar uma folha, ou seja, um nó que não possui descendentes, adiciona-se a tabela.

Usando a linguagem c, para realizarmos esse processo usaremos a seguinte função:

```

//funcao que cria a arvore de huffman
ARVORE* criaHuffmanTree(unsigned* listaB){
    lista nlista = {NULL, 0};
    for (int i = 0; i < 256; i++){
        if (listaB [i]){
            insereL(novoNoNaLista(novoNoNaArv(i, listaB[i], NULL, NULL)), &nlista);
        }
    }
}

```

```

while (nlista.valor > 1){
    ARVORE *noesq = desenfMenor(&nlista);
    ARVORE *nodir = desenfMenor(&nlista);
    ARVORE *soma = novoNoNaArv('/', noesq->freq + nodir->freq, noesq, nodir);
    insereL(novoNoNaLista(soma), &nlista);
}
return desenfMenor(&nlista);
}

```

1. 4 Mostrar os dados usando o dicionário criado:

Nessa fase, podemos percorrer o texto substituindo cada símbolo ou caractere pelo seu correspondente em binário e reescrever a frase “Hello world!” da seguinte forma:

S/CARACTERE	FREQUENCIA
H	0 0 0
e	0 0 1
w	0 1 0
r	1 1 0
d	1 0 0
!	1 0 1
espaço	1 1 0 0
o	1 1 0 1
l	1 1 1

“000 001 111 111 1101 1100 010 1101 110 111 100 101”

Logo, nota-se que não foi preciso repetir o caractere “l” que se repete 3 vezes na frase, fazendo com que somente o seu código em binário seja utilizado. Isso representa uma redução de custo de memória expressiva, tornando esse algoritmo muito poderoso em seu uso.

Usando a linguagem c, para realizarmos esse processo usaremos a seguinte função:

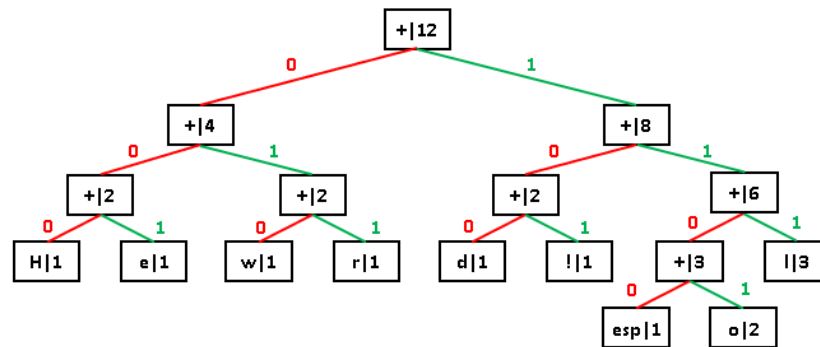
```

//funcao que verifica um no e retorna um bit
int retornaBit(FILE* ent, int p, byte* n){
    (p % 8 == 0) ? fread(n, 1, 1, ent) : NULL == NULL;
    return !!( (*n) & (1 << (p % 8)) );
}

```

1. 5 Decodificação:

Esse é o último processo do algoritmo de Huffman e requer extremo cuidado, pois a execução será feita iniciando na raiz e pulando para cada nó. Primeiro, verifica-se o bit do nó atual é 0 e se for verdade, percorre o nó para esquerda, caso contrário e o bit seja 1.



“000 001 111 111 1101 1100 010 1101 110 111 100 101”




“H e l l o esp w o r l d !”

A leitura será para a direita e ao encontrar o último nó da árvore, ou seja, uma folha específica, imprime-se o símbolo ou caractere que está naquele nó, formando a frase/texto desejado.




ANÁLISE DOS TESTES

O primeiro teste realizado foi com arquivos JPEG, na qual obtive bons resultados de compressão, superando a compressão de arquivo ZIP. Porém, o arquivo teste criado ficou com problemas e não pode ser operado.

```
C:\Users\Rodrigo\Desktop\Huffman coding\Codigos>gcc -o teste.jpg mainHuffman.c -c passaro.jpg
gcc: warning: passaro.jpg: linker input file unused because linking not done
```

 passaro.zip	23/06/2021 22:07	Arquivo ZIP do Wi...	144 KB
 passaro.jpg	21/06/2021 23:08	Arquivo JPG	144 KB
 teste.jpg	21/06/2021 23:38	Arquivo JPG	2 KB

O segundo teste realizado foi com arquivo de texto denominado Hello, que foi o alvo da pesquisa até aqui. O resultado, neste arquivo já foi favorável para compressão e ocorreu naturalmente como foi desenvolvido, superando o arquivo ZIP:

 Hello.txt	23/06/2021 21:17	Documento de Te...	10 KB
 Hello.zip	23/06/2021 22:09	Arquivo ZIP do Wi...	3 KB
 helloTeste.txt	23/06/2021 21:19	Documento de Te...	2 KB

RESULTADOS E CONCLUSÃO

É notável a grande possibilidade de utilizar o algoritmo de Huffman atualmente, tendo em vista que a quantidade de dados que nós consumimos diariamente é extremamente larga e difícil de ser mensurada. Várias ferramentas foram desenvolvidas após o advento desse mecanismo, possibilitando compartilhar e receber arquivos de forma mais rápida e mais segura, pois conseguimos manusear esses dados de maneira mais eficaz. O uso desse algoritmo apresenta um ótimo desempenho, pois ela garante que nunca um símbolo seja idêntico ao outro, gerando ambiguidade, isto é, nenhum código binário pode ser igual ao outro na construção da árvore binária. Porém, apresenta um desempenho ruim quando as probabilidades dos símbolos e caracteres não estão presentes na tabela ASCII, o que é muito difícil, visto que todas as tecnologias no mundo moderno levam em consideração essa regra internacional.

REFERÊNCIAS

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

GOODRICH, Michael T. **TAMASSIA**, Roberto. *Estruturas de dados & algoritmos em Java*. Porto Alegre: Grupo A, 2013.

GEEKSFORGEEKS. *Huffman Coding*, 2021. Disponível em:
<https://www.geeksforgeeks.org/huffman-coding-greedy-algo-3/>.
Acesso em: 10 de junho de 2021.

IME. *Algoritmo de Huffman para compressão de dados*, 2021. Disponível em:
<https://www.ime.usp.br/~pf/estruturas-de-dados/aulas/huffman.html>.

Acesso em: 10 de junho de 2021.