

INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA

Licenciatura Engenharia informática e multimédia

Trabalho 2

**Codificação Sinais Multimédia**

**Docentes:**

José Nascimento

**Trabalho realizado por:**

Miguel Távora Nº45102

João Cunha Nº45412

Arman Freitas Nº45414

# Sumário

[Desenvolvimento 5](#_Toc6146252)

[Função gera\_huffman 5](#_Toc6146253)

[Função Codifica 7](#_Toc6146254)

[Função Descodifica 9](#_Toc6146255)

# Índice de Ilustrações

[Figura 1 Descodificação da palavra 10](#_Toc6146184)

# Introdução

O presente trabalho foi nos proposto com o objetivo de aperfeiçoar as habilidades do grupo no que toca á construção de algoritmos. Insidindo este trabalho na compressão e descompressão de diversos tipos de ficheiros (.txt, .mp3, etc.).

Neste trabalho utilizamos a codificação de Huffman, que não possuí quaisquer perdas e, tem uma baixa taxa de redundância. O algoritmo tira por base as probabilidades de cada símbolo na mensagem, construindo uma árvore.

Esta agrega sempre os últimos dois caracteres com menor número de ocorrências. Recolocando o novo caracter obtido lista por ordem de probabilidade sucessivamente até so ter dois caracteres com diversos simbolos agregados. Atribuindo-se então para cada aresta da árvore um digito binário (0 ou 1) sendo o código obtido quando é percorrida a árvore e anotando nas arestas os digitos binários desde a raiz até as folhas que corresponde ao simbolo desejado.

É também nos pedido a descompressão através da árvore de Huffman. Ou seja, através de uma trama de números em binário, descodificar os mesmos para que consigamos ler a mensagem. Isto é feito por parte do receptor quando recebe a trama de bits. Este processo será demonstrado no desenvolvimento do trabalho.

# Desenvolvimento

## Função gera\_huffman

Para a realização do primeiro exercício, foi necessário realizar a função que gera o código de huffman. De modo a construir a função *gera\_huffman*, primeiramente foi criada uma outra função, *geraDicionario*, esta tem como parâmetro a *string* a que se pretende aplicar o algoritmo de Huffman.

A partir deste texto, é criada uma lista com o número de vezes que o caractere se repete (frequência do caractere) e o carácter correspondente. Após a construção desta lista, era indispensável criar outra função cujo seu objetivo é ordenar de forma crescente os caracteres por número de vezes que o mesmo se repete. Assim, criámos a função *ordemCrescente*.

A função *gera\_huffman* propriamente dita, gera a árvore que nos ajuda na codificação e, descodificação da mensagem. Esta foi implementada utilizando recursividade, utilizando a tabela antes construída (retorno de *geraDicionario*) como argumento, onde vai acrescentando a cada caracter um novo numero obtendo desta forma uma tabela.

De seguida podemos verificar um exemplo de uma árvore de Huffman gerada, e, como o algoritmo em si funciona (explicado na introdução do trabalho):

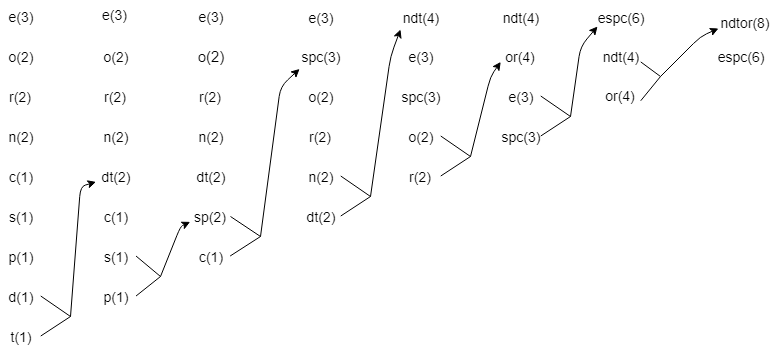


Figura 1 – Algoritmo de Huffman para a mensagem “correspondente”

.

## Função Codifica

Para codificar a mensagem, é recebido como parâmetro esta (sequência de caracteres) e, a árvore de Huffman gerada anteriormente.

Percorrendo esta árvore podemos ir “juntando” os bits encontrados de forma a criar cada caractere presente na mensagem.

Para simplificar o processo criámos outra função, *code()*, que apenas codifica um caractere e, no *codifica* apenas percorremos todos os caracteres da *string*.

De seguida podemos verificar como funciona a pesquisa na árvore por um determinado caractere. Apenas procuramos pelo mesmo e, vamos juntando os bits até chegar eventualmente ao caractere pretendido.

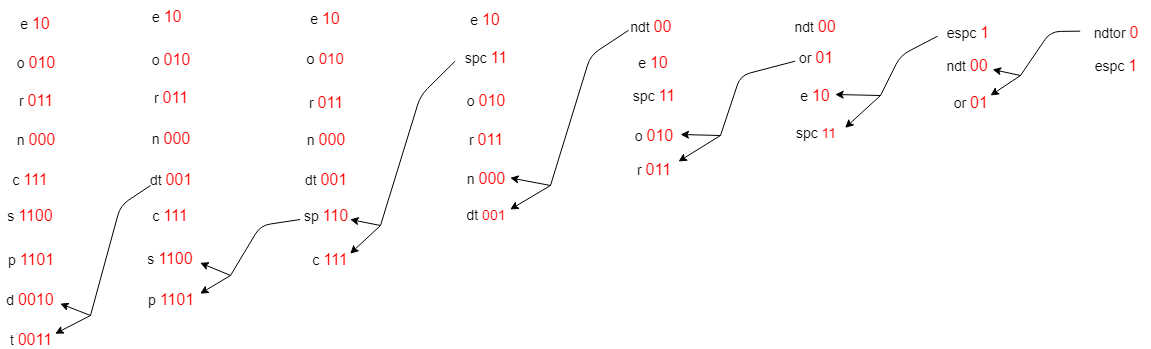


Figura 1 – Atribuição do código para os caracteres

Por fim, criamos um método *tabelaCodificacao()* para nos ajudar mais a frente a descodificar a mensagem mais facilmente.

Esta função apenas cria uma tabela com todos os caracteres presentes na árvore de Huffman (não repetidos) junto com as suas respetivas codificações:

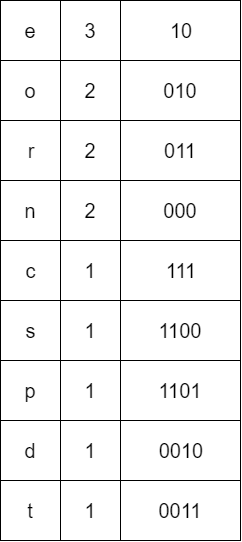


Figura 3 – Tabela obtida apartir da atribuição dos códigos

## Função Descodifica

De modo a podermos descodificar uma certa trama de bits é necessário a tabela de codificação obtido anteriormente pela função de *tabelaCodificacao*.

A partir disso, a função *descodifica* lê os códigos binários e, verifica pela tabela a existência de certas codificações se não encontrar nenhuma, apenas acrescenta o próximo bit e procura outra vez na tabela até encontrar uma codificação que exista na tabela.

Quando encontrado é apenas feito o mesmo processo até ao final da trama de bits.

Obtemos no final, desta forma, a mensagem codificada. O processo descrito pode ser encontrado na seguinte imagem com o total de 43 bits:



Figura 3 - Descodificação da palavra

## Função Escrever

Tal como o nome diz a função *escreve*, escreve uma mensagem a partir do código binário cedido como argumento.

De forma a poder a guardar a trama de bits num array mais pequeno, utilizamos o método *packbits* do Numpy e, guardamos o array num ficheiro utilizando o método *tofile()*.

É de notar que essa mensagem é guardada em um ficheiro do tipo “*.txt”* .

## Função Ler

A partir de um ficheiro encontrado no sistema operativo, a função *ler* vai retornar uma *string* com o conteúdo do ficheiro.

Esta função recebe como parâmetro, não só o nome do ficheiro (com a sua extensão ex: “.txt”), mas também uma variável booleana, que vem inicialmente com o seu valor falso. No entanto se for passado o valor de True, a função irá ler o conteúdo do ficheiro como bytes.

## Conclusões

Em suma com a realização deste trabalho o grupo aprendeu a construir a tabela de Huffman.

Permitindo fazer a codificação e descodificação de mensagens. Conseguindo desta forma otimizar bastante todo o tipo de ficheiro. Conseguimos ver diversos objetivos para a utilização deste tipo de compressão sem perdas, como por exemplo o envio de mensagens via internet que requer que nenhum caractere desapareça na receção da mensagem.

A característica que o torna este algoritmo distinto dos outros é o facto de ser não redundante, ou seja, uma sequência de números binários nunca pode dar 2 caracteres diferentes.

No entanto nem tudo é perfeito e, como se pode verificar na seguinte tabela, os nossos resultados foram deveras ineficientes no que diz respeito ao tempo de processamento da codificação e descodificação.

Achamos que estes números possam ser diminuídos, mas não nos conseguimos “escapar” à utilização de ciclos *for*, que utilizam a maior parte do tempo de CPU devido a estarmos a codificar ficheiros com muitos caracteres.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Árvore de Huffman | *Codificação* | *Descodificação* | Entropia | Número médio de bits | Eficiência |
| *ecg.txt* | 0,31941724 | 0,67318273 | 8,35173678 | 3,26516166 | 9,82352941 | 0,33238173 |
| *ubuntu.txt* | 2,02020431 | 4,6073401 | 82,2575569 | 4,6779341 | 9,63366337 | 0,48558206 |
| *mp3* | 5,94306684 | 26,2587106 | 169,2293 | 4,28631471 | 8,85263158 | 0,48418537 |
| *midi* | 0,19938469 | 0,93242192 | 10,5513976 | 3,73814176 | 9,37894737 | 0,3985673 |
| *imagem* |  |  |  |  |  |  |

## Anexos