Licenciatura em Engenharia informática e Multimédia

Codificação de Sinais Multimédia

Trabalho prático 2

Docentes

Eng.º José Nascimento

Eng.º André Lourenço

Turma 41D

Miguel Távora Nº45102

João Cunha Nº45412

Arman Freitas Nº45414

Sumário

SUMÁRIO **2**
ÍNDICE DE ILUSTRAÇÕES **3**
ÍNDICE DE TABELAS **3**
INTRODUÇÃO **4**
DESENVOLVIMENTO **5**
FUNÇÃO GERA_HUFFMAN **5**
FUNÇÃO CODIFICA **7**
FUNÇÃO DESCODIFICA **9**
FUNÇÃO ESCREVER **10**
FUNÇÃO LER **10**
CONCLUSÕES **11**
ANEXOS **13**

Índice de Ilustrações

Figura 1 Algoritmo de Huffman para a mensagem “correspondente” 6

Figura 2 Atribuição do código para os caracteres 7

Figura 3 Tabela obtida apartir da atribuição dos códigos..... 8

Figura 4 Descodificação da palavra..... 9

Índice de Tabelas

Tabela 1 Testes ao programa..... 12

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):

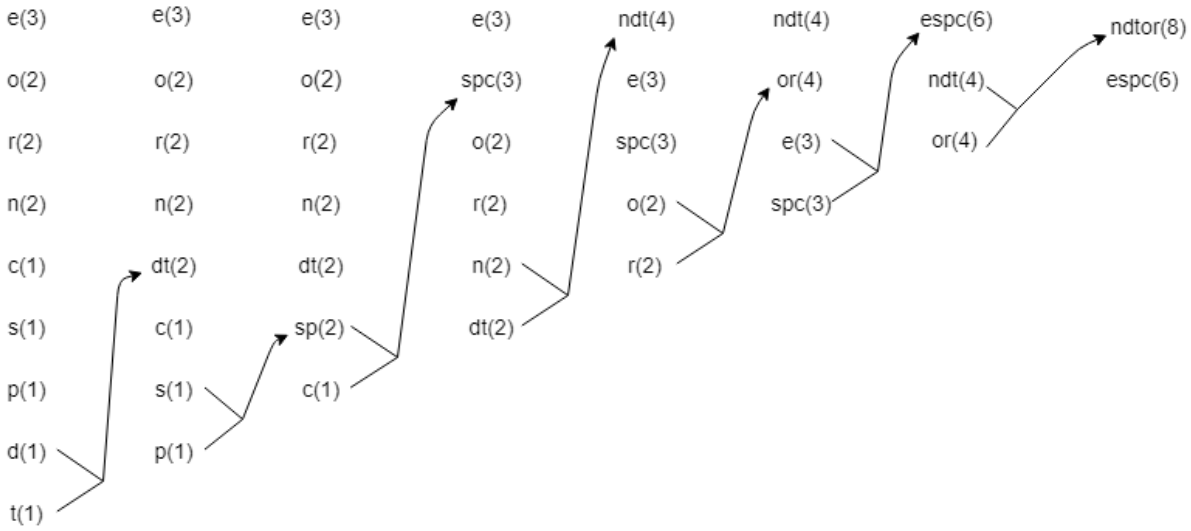

Figura 1 Algoritmo de Huffman para a mensagem “correspondente”

.

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.

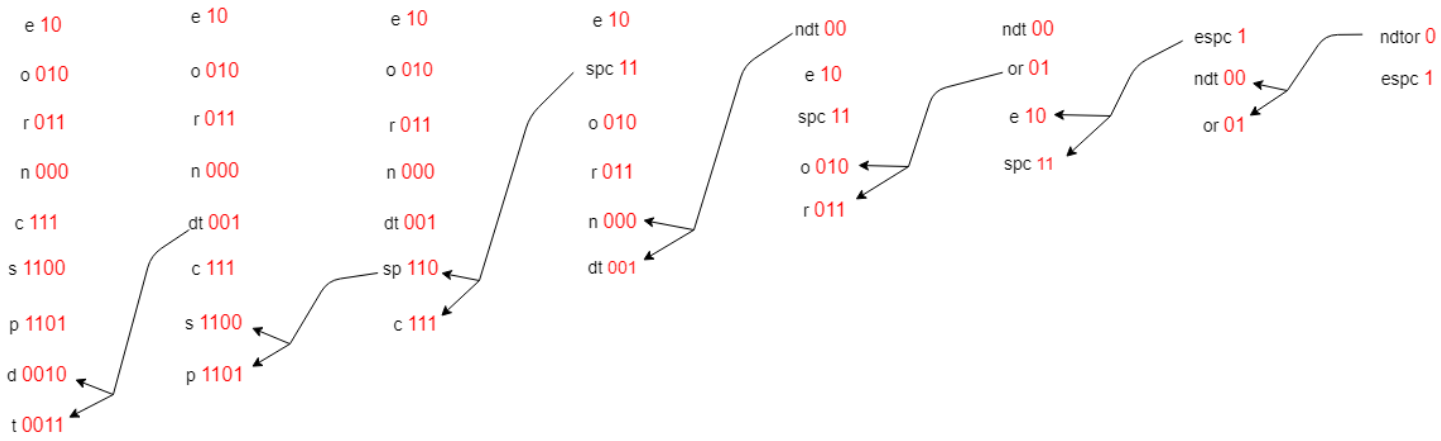

Figura 2 Atribuição do código para os caracteres

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:

e	3	10
o	2	010
r	2	011
n	2	000
c	1	111
s	1	1100
p	1	1101
d	1	0010
t	1	0011

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:

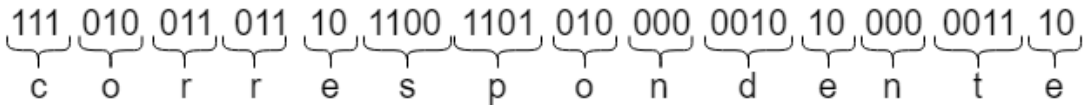

Figura 4 Descodificação da palavra

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.

Por fim, infelizmente não conseguimos testar a imagem que por alguma razão nos gera uma tabela de codificação com um número médio de bits muito grande (31622.9).

Tabela 1 Testes ao programa

	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

Anexos

```
import cv2
import numpy as np
import matplotlib.pyplot as plt
import os
from time import time

def gerarDicionario (text):
    text = list (text)
    dicionario = []
    for i in set(text):
        dicionario.append([text.count(i)/len(text), i])
    return dicionario

#tabela = gerarDicionario("OLA MUNDO!!!")

def ordemCrescente (table):
     for i in range (len (table)):
           for j in range (i+1, len (table)):
                if (table[i][0] > table [j][0]):
                     temp = table[i]
                     table[i] = table [j]
                     table[j] = temp
     return table
#####################################################################
#################


def escrever (message, fileName):
    #print("-------------------------------------ESCREVER---------
----------------------------\n")
     #file = open(str(fileName), 'w')
     np.packbits(list(map(int, message))).tofile("f.txt")
     print("Wrote: \"" + str(message) + "\" to \'" + str(fileName)
+ "\'\n")
     #file.close()

def ler (fileName, a=False):
    #print("-------------------------------------LER--------------
-----------------------\n")
    if(not a):
        file = open(str(fileName), 'r')
    else:
        file = open(str(fileName), 'rb')

    r = file.read()

    file.close()

    print("Read: \"" + str(r) + "\" from: \'" + str(fileName) +
"\'\n")
    return str(r)
#####################################################################
#################
#print("IMAGEM----------------------")
```

```
# Lê a imagem em níveis de cinzento
#x = cv2.imread("lenac.tif",cv2.IMREAD_GRAYSCALE)
# Converte a imagem (matriz) numa sequência de números (array)
#xi = x.ravel()
#aa = ""
#for i in range(len(xi)):
#    aa+=str(xi[i])
#print(aa)
# Calcula o histogram
#h, bins, patches = plt.hist(xi,256,[0,256])
# Gera o código de Huffman
#t0 = time()

#print(np.concatenate(np.arange(0, 256), h))
#img = []
#for i in range(256):
#    img.append([h[i], str(i)])

#tabela = img
#print(tabela)
#tabela = np.arange(0,256),h


########## IMAGEM

#text = ler("HenryMancini-PinkPanther30s.mp3", True)
#text = ler("HenryMancini-PinkPanther.mid", True)
text = ler("ecg.txt")
#text = ler("ubuntu_server_guide.txt")
#t0 = time()
tabela = gerarDicionario(text)
huffman_tree = []
huffman_tree.append(tabela)

def gera_huffman (nodes):
      newnode = []
      #print("LEN")
      #print(len(nodes))
      if len(nodes)>1:
            #print("nodes:", str(nodes))
            nodes.sort()
            nodes[0].append("0")
            nodes[1].append("1")
            combined_node1 = (nodes[0][0] + nodes[1][0])
            combined_node2 = (nodes[0][1] + nodes[1][1])
            newnode.append(combined_node1)
            newnode.append(combined_node2)
            newnodes = []
            newnodes.append(newnode)
            newnodes = newnodes + nodes[2:]
            nodes = newnodes
            huffman_tree.append(nodes)
            gera_huffman (nodes)
      return huffman_tree
########## IMAGEM
```

```
def printLevels(huffmanTable):
    #huffmanTable = huffmanTable[::-1]
    print("-------------------------------------PRINT-LEVELS------
-------------------------------\n")
    counter = 0
    for level in huffmanTable:
      print("Level", counter, ":", level)
      counter += 1
huffmanTable = gera_huffman(tabela)
huffmanTable = huffmanTable[::-1]
#t1 = time()
#print ("time:", t1-t0)
#print("-------------------------------------CODIFICA-----------
------------------------\n")
def code (letter, table):
      ajuda = ""
      nodeBefore = []
      for level in table:
            for node in level:
                  if letter in node[1] and len(node)>2 and node !=
nodeBefore:
                        nodeBefore = node
                        ajuda += node[2]
      return ajuda

def codifica (text, table):
      letters = list(text)
      elFinal = ""
      for letter in letters: elFinal += code(letter, table)
      return elFinal

def codificaverdadeiro(text, dicionario):
    s = ""
    for char in text:
        for values in dicionario:
            if char == values[0]:
                s += values[1]
    return s

def tabelaCodificacao(table, dicionario):
      print("TABELA DE CODIFICACAO")
      cod = []
      a = []
      for nome in dicionario:
            c = codifica(nome[1], table)
            print("Letter (" + str(nome[1]) + "): " + str(c))
            cod.append([nome[1], c])
            a.append(c)
      return cod, a
t0 = time()
codificacao, codigos = tabelaCodificacao(huffmanTable, tabela)
codificado = codificaverdadeiro(text, codificacao)
t1 = time()
print("TIME", str(t1-t0))
escrever(codificado, "ecgCODIFICADO.txt")
escrever(codificado, "ubuntu_server_guideCODIFICADO.txt")
```

```
#print("-------------------------------------DESCODIFICAR---------
----------------------------\n")
def descodifica (message, dicionario):
    strfinal = ""
    s = ""

    for char in message:
        s += char
        for values in dicionario:
            if s == str(values[1]):
                strfinal += str(values[0])
                s = ""
                break
    return strfinal
t0 = time()
descodificado = descodifica(codificado, codificacao)
t1 = time()
print(descodificado)
print("TIME", str(t1-t0))

def entropia(p):
    elFinal = 0
    for i in range(len(p)):
        elFinal += (p[i][0]*np.log2(p[i][0]))

    return -1*elFinal
entrop = entropia(tabela)
print("ENTROPIA",entrop)
def numeroMedio(a):
    nmr = 0
    count = 0
    for i in range(len(a)):
        nmr += len(list(a[i]))
        count += 1
    return nmr/count
nm = numeroMedio(codigos)
print("NUMERO MEDIO",nm)
def eficiencia(e, n):
    return e/n
print("EFICIENCIA",eficiencia(entrop, nm))
```