Licenciatura Engenharia Informática e Multimédia

Codificação de Sinais Multimédia

Trabalho 2

Docente:

Engº José Nascimento

Grupo: 10

Turma: 41D

Miguel Távora Nº45102

João Cunha Nº45412

Arman Freitas Nº45414

Data de entrega: 19/05/2019

Índice

Índice de Figuras

Índice de Tabelas

1. Introdução e Objetivos

O presente trabalho prático tem como intuito a implementação e, melhor entendimento da norma JPEG, explicada nas aulas anteriores.

A norma JPEG é um tipo de compressão (com perdas), utilizado em imagens. Este género de compressão pode ser controlado com um parâmetro que vamos explicar mais à frente no trabalho (a qualidade).

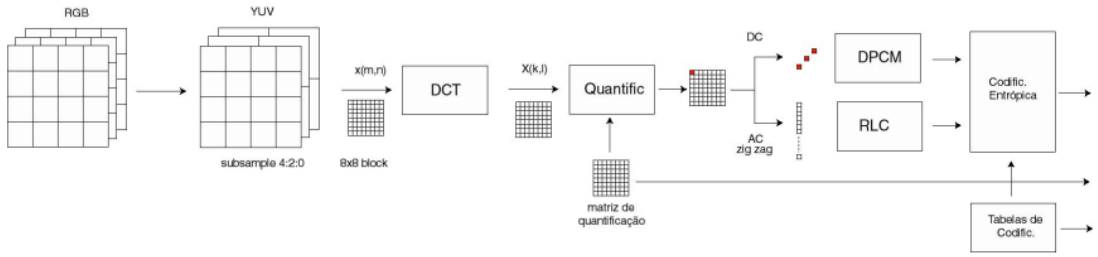

Figura 1 - Diagrama de blocos do algoritmo

Como se pode verificar acima, o algoritmo segue as seguintes etapas:

1. Primeiramente a compressão da imagem no espaço de cores RGB, para YUV.
2. Em seguida, fazer-se-há o DCT (Discrete Cosine Transform) desta mesma imagem por blocos 8x8.
3. A quantificação é o passo que se segue. Este, utiliza uma matriz de quantificação e um nível de qualidade de forma a quantificar a imagem.
4. A codificação DC.
5. A codificação AC (em zigue-zague)
6. Por fim, a imagem é codificada num ficheiro e, é feita a inversa destas mesmas

etapas. É de notar que a matriz de quantificação é enviada no “cabeçalho” do ficheiro JPEG de forma a poder quantificar inversamente a imagem.

2. Desenvolvimento

2.1. DCT – Discrete Cosine Transform

A DCT é uma extensão da transformada contínua do cosseno para um domínio discreto. A recuperação dos dados pode ser feita com a operação inversa chamada Inverse DCT (Inverse DCT). Como uma imagem é um sinal 2D, é possível aplicar o conceito de transformações 2D ortonormadas. A transformação 2D é separável em duas operações, sendo que uma aplica a todas as linhas e a outra a todas as colunas.

A transformada bidirecional resulta numa matriz onde os coeficientes mais significativos se acumulam no canto superior esquerdo (inicio da matriz) e, os restantes possuem um valor mais reduzido podendo ser mais facilmente armazenados.

Apesar de poder parecer, a transformada DCT não faz qualquer tipo de compressão quando aplicada à imagem. Isto é, após a transformada ser calculada os valores de output utilizam mais bits para guardar os dados de cada pixel e, não só os típicos 8 bits que a imagem tinha em cada pixel.

Desta forma, a transformada DCT apenas nos permite detetar os pixels menos relevantes de cada bloco 8x8 da imagem, para que mais tarde essa informação seja apagada (mais à frente na quantização).

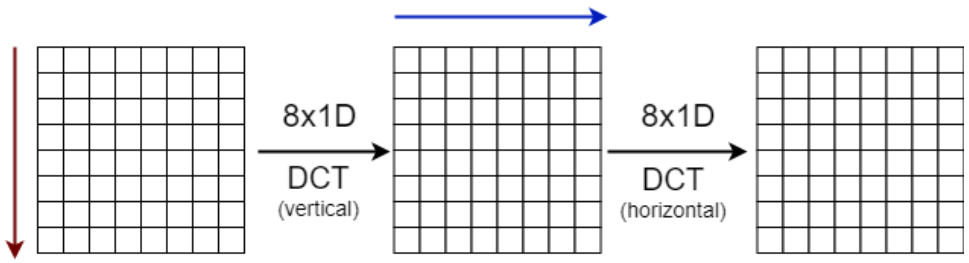

Figura 2 - Aplicação do DCT em duas dimensões

De seguida apresentam-se as equações de cálculo do DCT para cada pixel *x(m, n)*

$$y(k,l) = \frac{2}{\sqrt{MN}}\alpha(\mathrm{k})\alpha(\mathrm{l}) \sum_{m=0}^{M-1}\sum_{n=0}^{N-1}\left(\frac{2\mathrm{m}+1}{2\mathrm{M}}\pi k\right)\cos\left(\frac{2\mathrm{n}+1}{2\mathrm{N}}\pi l\right)x(m,n)$$

$$x(m,n) = \frac{2}{\sqrt{MN}}\sum_{k=0}^{M-1}\sum_{l=0}^{N-1}\alpha(\mathrm{k})\alpha(\mathrm{l})\,cos\left(\frac{2\mathrm{m}+1}{2\mathrm{M}}\pi k\right)\cos\left(\frac{2\mathrm{n}+1}{2\mathrm{N}}\pi l\right)y(k,l)$$

2.2. Quantificação

Após a transformada DCT ter sido calculada para todos os 4096 blocos 8x8 da imagem, passamos finalmente à compressão da imagem.

O método para reduzir o número de bits do DCT é a quantificação. Este método apenas reduz a "precisão" dos inteiros guardados na imagem e, adiciona então perda ao algoritmo.

No presente algoritmo, de modo a quantificar a imagem existe o conceito de matriz de quantificação. Existem diversas matrizes de quantificação, das quais nós vamos utilizar a seguinte:

$$Q = \begin{bmatrix} 16 & 11 & 10 & 16 & 24 & 40 & 51 & 61 \\ 12 & 12 & 14 & 19 & 26 & 58 & 60 & 22 \\ 14 & 13 & 16 & 24 & 40 & 57 & 69 & 56 \\ 14 & 17 & 22 & 29 & 51 & 87 & 80 & 62 \\ 18 & 22 & 37 & 56 & 68 & 109 & 103 & 77 \\ 24 & 35 & 55 & 64 & 81 & 104 & 113 & 92 \\ 49 & 64 & 78 & 87 & 103 & 121 & 120 & 101 \\ 72 & 92 & 95 & 98 & 112 & 100 & 103 & 99 \end{bmatrix}$$

Tabela 1 Matriz de Quantificação

Para quantificar em concreto utilizamos a seguinte equação:

$$Valor\ Quantificado\ (i,j) = \frac{DCT\ (i,j)}{Matriz\ de\ Quantificação\ (i,j)\ \times qualidade}$$

É de notar que esta equação se aplica a cada bloco 8x8 da imagem. Neste caso, como se pode ver, faz-se a divisão de cada bloco de 8x8 (com o DCT calculado) pela matriz de quantificação multiplicada por um fator de qualidade. Este fator de qualidade permite, aumentar ou diminuir o rácio de compressão, afetando assim a imagem final.

Após o cálculo é possível observar que, para cada bloco 8x8 da imagem, encontramos agora, no canto inferior direito vários zeros, o que traz perdas ao algoritmo como referido anteriormente.

$$Bloco\ Quantificado = \begin{bmatrix} 80 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Tabela 2 - Exemplo de bloco 8x8 quantificado

Por fim, nesta fase basta apenas acrescentar que os resultados da divisão devem todos ser arredondados ao número inteiro mais próximo.

2.3. Codificação

2.3.1. DC

O último passo do processo de compressão JPEG é a codificação, que consiste no DC e no AC.

Neste "subpasso" (DC), o valor de cada bloco 8x8 da imagem é "convertido" de um valor absoluto, num valor relativo.

Isto pois cada primeiro byte de cada bloco 8x8, passa a ser a igual à sua subtração pelo valor do primeiro byte do bloco anterior. A única exceção é o primeiro bloco que permanece o mesmo, de modo a conseguirmos obter os valores originais apartir do mesmo.

2.3.2. AC

Por fim, antes de ser escrito no ficheiro, ainda é necessário organizar os coeficientes da imagem (que não os DC) em zigue-zague, como sugere a seguinte figura:

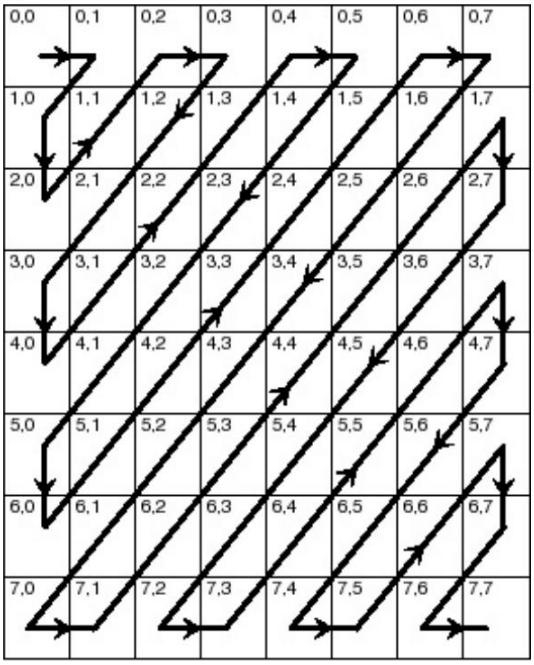

Figura 3 - Caminho ziguezague no bloco 8x8

Após organizada a imagem, é contado o número de zeros até chegar a qualquer valor que não seja um zero. De seguida são guardados os valores relevantes num tuplo com o seguinte formato "**(número de zeros, valor)**".

Quando se chega ao final do bloco é sempre colocado um tuplo "**(0, 0)**".

Uma exceção é feita, quando o número de zeros até a um valor diferente de zero é igual a 15. Este caso adicionará outro tuplo antes de chegar ao número diferente de zero, tuplo este com o seguinte aspeto : "**(15, 0)**".

3. Conclusões e Resultados

Através da implementação em Python do algoritmo de compressão JPEG obtivemos os seguintes resultados, para cada qualidade, respetivamente:

Tabela 3 - Resultados obtidos

Qualidade	Compressão (segundos)	Descompressão (segundos)	Tempo total (segundos)	SNR	Taxa de compressão
10	1.2	0.5	1.7	24.2	34.5
20	0.9	0.6	1.5	26.5	22.8
30	0.9	0.6	1.6	27.5	17.7
40	0.9	0.9	1.8	28.2	14.6
50	1	1	2	28.7	12.8
60	1	1.2	2.2	29.2	10.7
70	1	1.6	2.7	29.8	8.7
80	1.2	2.4	3.6	30.7	6.5
90	1.5	10.8	12.2	32.8	3.9

Mais uma vez, reconhecemos que a nossa implementação da norma JPEG com ajuda da linguagem de programação Python não é a mais eficiente.

No entanto, conseguimos verificar que uma compressão com perdas nem sempre se nota, ao olhar do utilizador. Ao remover blocos da imagem com frequências maiores, muitas das alterações não são visíveis ao olho humano.

De seguida encontra-se um gráfico que relaciona os SNR's obtidos com as taxas de compressão:

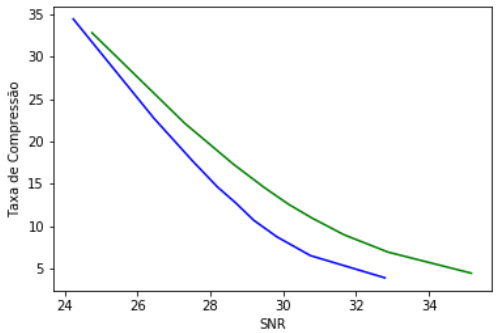

Figura 4 - Taxa de compressão em função do SNR (em azul, a nossa implementação e em verde o algoritmo da biblioteca PIL)

Outra das nossas observações relativamente à nossa mesma implementação é o facto de, mais uma vez, a utilização de ciclos *for* e *while*. Com certeza que, se abandonássemos o uso de ciclos como estes, tanto a compressão quanto a descompressão do ficheiro seriam feitas num espaço de tempo muito reduzido.

Em boa verdade, o presente trabalho prático permitiu ao grupo aprender a implementar a técnica de compressão JPEG com perdas pouco significativas (dependendo do nível de qualidade).

Primeiramente, foi feita a transformação do espaço de cores da imagem de RGB para YUV e foi tratada a imagem para sub-blocos de 8x8, aplicando a transformação DCT, tendo-se perdido as componentes de alta frequência (menos importância) posteriormente, na fase da quantificação.

A quantificação utiliza tabelas pré-definidas maximizando a taxa de compressão sem perdas.

Por fim, é realizada a codificação DC e AC, resultando assim na imagem comprimida. Desta forma é possível obter uma grande taxa de compressão com uma qualidade semelhante á original.

4. Anexos

4.1. Main

```
import cv2
import numpy as np
import matplotlib.pyplot as plt
import urllib
from os import path
from time import time
from PIL import Image

from Transformada import codificadorDCT, descodificadorDCT
from Quantificacao import codificadorQuantificacao, descodificadorQuantificacao
from DC import codificadorDC, descodificadorDC
from AC import codificadorAC, descodificadorAC
from CodificacaoFinal import codificarImagem, descodificarImagem


def downloadImage():

urllib.request.urlretrieve("https://homepages.cae.wisc.edu/~ece533/images/lena.bmp","len
a.bmp")

imagemPIL = Image.open("LenaGreyScale.bmp")
x = cv2.imread("lena.bmp")

img_yuv = cv2.cvtColor(x, cv2.COLOR_BGR2YUV)
x, y, y1 = cv2.split(img_yuv)

qualidade = [10, 20, 30, 40, 50, 60, 70, 80, 90]
SNRs = []
compressões = []
SNRsPIL = []
compressõesPIL = []


def splitImage (x):
    dct_array = np.full((int(len(x) * len(x[0]) / 64), 64), 10, dtype='float32')

    array_To_DCT = np.zeros(64, dtype='float32')
    indice_bloco = 0
    indice_array = 0

    # Separar em blocos de 8x8
    for l in range(int(len(x)/8)):
        for c in range(int(len(x)/8)):
            for xx in range(8):
                for y in range(8):
                    array_To_DCT[indice_bloco] = x[(l*8)+xx][(c*8)+y]
                    indice_bloco += 1

            dct_array[indice_array] = array_To_DCT
            indice_array += 1
            array_To_DCT = np.zeros(64, dtype='float32')
            indice_bloco = 0
    return dct_array

def CalcularSNR (pr, original):
       return (10 * np.log10
(np.sum(np.sum(np.power(pr.astype('float'),2)))/np.sum(np.sum(np.power((pr.astype('float
')-original.astype('float')),2)))))

for q in qualidade:
    print("############################## QUALIDADE",
q,"##############################")
    #print ("Qualidade " + str(q))

    time1 = time()
    arr = splitImage(x)
    dct = codificadorDCT(arr)
    array = codificadorQuantificacao(dct, q)
    array_diferencial = codificadorDC(array)
```

```
    tuplos = codificadorAC(array_diferencial)
    codificarImagem(tuplos, str(q) + ".bin")
    tempoC = time() - time1
    print ("Tempo de Compressão:", tempoC)

    time3 = time()
    i_huff = descodificarImagem(str(q) + ".bin")
    array_dif = descodificadorAC(i_huff)
    array_quant = descodificadorDC(array_dif)
    blocos = descodificadorQuantificacao(array_quant, q)

    idct = descodificadorDCT(blocos)
    tempoD = time() - time3
    print ("Tempo de Descompressão:", tempoD)
    print ("Tempo de Compressão e Descompressão:", (tempoC + tempoD))

    SNR = CalcularSNR (idct, x)
    print ("SNR:", SNR)
    original = path.getsize("LenaGreyScale.bmp")
    final = path.getsize(str(q) + ".bin")
    taxa = original/final
    print ("Taxa de compressão:", taxa)
    cv2.imwrite("Output" + str(q) + ".jpg", idct)

    imagemPIL.save(("OutputPIL" + str(q) + ".jpeg"), "JPEG", quality=q)
    finalPIL = path.getsize(("OutputPIL" + str(q) + ".jpeg"))
    taxaPIL = original/finalPIL

    pil = cv2.imread(("OutputPIL" + str(q) + ".jpeg"))
    orig = cv2.imread("LenaGreyScale.bmp")
    SNRPIL = CalcularSNR (pil, orig)

    SNRs.append(SNR)
    compressões.append(taxa)

    SNRsPIL.append(SNRPIL)
    compressõesPIL.append(taxaPIL)

# Grafico SNR / Taxa
plt.plot(SNRs, compressões, 'b')
plt.plot(SNRsPIL, compressõesPIL, 'g')

plt.xlabel('SNR')
plt.ylabel('Taxa de Compressão')
plt.savefig('Grafico.png')
plt.show()
```

4.2. Transformada DCT

```
import numpy as np
from scipy.fftpack import dct, idct


def DCT2D (arr):
    return dct(dct(arr.T, norm='ortho').T, norm='ortho')

def IDCT2D (arr):
    return idct(idct(arr.T, norm='ortho').T, norm='ortho')

def codificadorDCT(dct_array):
    dcts = []

    for bloco in dct_array:
        bloco8x8 = bloco.reshape(int(len(bloco)/8), 8)
        dcts.append(DCT2D(bloco8x8))
    return np.rint(dcts)

def descodificadorDCT(dct_values):

    idcts = []
    for bloco in dct_values:
```

```
        idcts.append(IDCT2D(bloco))

    idcts = np.rint(idcts)
    length = 64
    new_array =[]

    for i in range(length):
        for c in range(8):
            for x in range(length):
                new_array.append(idcts[x+i*length][c])

    return np.array(new_array).reshape(512, 512)
```

4.3. Quantificação

```
from tab_jpeg import Q
import numpy as np

def codificadorQuantificacao (arr, q):

    imagem = arr.copy()

    for i in range(len(imagem)):
        imagem[i] = np.divide(imagem[i], (Q * fatorQualidade(q)))

    return np.rint(imagem)


def descodificadorQuantificacao (arr, q):

    imagem = arr.copy()

    for i in range(len(imagem)):
        imagem[i] = np.multiply(imagem[i], (Q * fatorQualidade(q)))

    return imagem

def fatorQualidade (q):
    if(q <= 50):
        a = 50.0 / q
    else:
        a = 2.0 - (q * 2.0)/100.0
    return a
```

4.4. DC

```
def codificadorDC(arr):

    delta = 0
    for i in range(len(arr)):
        arr[i][0][0] -= delta
        delta = arr[i][0][0] + delta

    return arr


def descodificadorDC(arr):

    delta = 0
    for i in range(len(arr)):
        arr[i][0][0] += delta
        delta = arr[i][0][0]

    return arr
```

4.5. AC

```
import numpy as np

def codificadorAC (arr):
    # zig-zag order
    zigzag = np.zeros((8, 8))
    zigzag[0] = [ 0,  1,  5,  6, 14, 15, 27, 28]
    zigzag[1] = [ 2,  4,  7, 13, 16, 26, 29, 42]
    zigzag[2] = [ 3,  8, 12, 17, 25, 30, 41, 43]
    zigzag[3] = [ 9, 11, 18, 24, 31, 40, 44, 53]
    zigzag[4] = [10, 19, 23, 32, 39, 45, 52, 54]
    zigzag[5] = [20, 22, 33, 38, 46, 51, 55, 60]
    zigzag[6] = [21, 34, 37, 47, 50, 56, 59, 61]
    zigzag[7] = [35, 36, 48, 49, 57, 58, 62, 63]
    zigzag = zigzag.reshape(64,order='F').astype('int')

    length = len(arr)

    # Preencher o array ordenadamente
    sortedArray = np.zeros_like(arr).reshape(length ,64)
    sort = np.argsort(zigzag)

    for blocoPos in range(length):
        blocoFlat = arr[blocoPos].flatten(order='F')
        sortedArray[blocoPos] = blocoFlat[sort]

    #print(sortedArray)
    zeros = 0
    tuplos = []
    for x in range(length):
        for y in range(64):
            # Se for a primeira posicao do bloco
            if y != 0:
                if int(sortedArray[x][y]) != 0:
                    tuplos.append((int(zeros),int(sortedArray[x][y])))
                    zeros = 0
                else:
                    zeros +=1
                    if y == 63 or zeros == 15:
                        tuplos.append((int(zeros),int(sortedArray[x][y])))
                        zeros = 0
            else:
                # Guardar os DC
                tuplos.append(int(sortedArray[x][y]))
        tuplos.append((0,0))

    # Retirar os zeros a mais
    tuplos = tuplos[::-1]

    i = 0
    while i < len(tuplos) - 1:
        tuplo = tuplos[i]
        if isinstance(tuplo, tuple) and tuplo == (0, 0):
            bfrTuplo = tuplos[i + 1]
            if type(bfrTuplo) == tuple and bfrTuplo [1] == 0:
                tuplos.pop(i + 1)
                i -=1
        i += 1

    tuplos =  tuplos[::-1]

    return tuplos

def descodificadorAC(arr):

    # zig-zag order
    zigzag = np.zeros((8, 8))
    zigzag[0] = [ 0,  1,  5,  6, 14, 15, 27, 28]
    zigzag[1] = [ 2,  4,  7, 13, 16, 26, 29, 42]
    zigzag[2] = [ 3,  8, 12, 17, 25, 30, 41, 43]
    zigzag[3] = [ 9, 11, 18, 24, 31, 40, 44, 53]
    zigzag[4] = [10, 19, 23, 32, 39, 45, 52, 54]
    zigzag[5] = [20, 22, 33, 38, 46, 51, 55, 60]
```

```
    zigzag[6] = [21, 34, 37, 47, 50, 56, 59, 61]
    zigzag[7] = [35, 36, 48, 49, 57, 58, 62, 63]

    ind_O = zigzag.reshape(64,order='F').astype('int')

    zigzag = []
    bloco8x8 = []

    for valor in arr:

        if isinstance(valor, tuple):

            if valor[0] == 0 and valor[1] == 0:

                resto = 64 - len(bloco8x8)

                for i in range(resto):
                    bloco8x8.append(0)

                bloco8x8 = np.array(bloco8x8)
                zigzag.append(bloco8x8[ind_O].reshape((8,8),order='F'))
                bloco8x8 = []
            else:
                zeros = valor[0]
                for i in range(zeros):
                    bloco8x8.append(0.)

                if valor[1] != 0:
                    bloco8x8.append(valor[1])
        else:
            bloco8x8.append(valor)

    return zigzag
```

4.6. Escrever/Ler no ficheiro

```
import numpy as np
import array as arr
from tab_jpeg import K3, K5

def codificarImagem (lista, filename):

    bits = []
    for valor in lista:
        if isinstance(valor, tuple):
            if (valor[0] == 0 and valor[1] == 0):
                bits.append('1010')
            else:
                if valor[1] > 0:
                    val = valor[0]
                    binario = bin(valor[1])[2:]
                    size = len(binario)
                elif valor[1] < 0:
                    val = valor[0]
                    binario_aux = bin(valor[1])[3:]
                    binario = ''.join('1' if i == '0' else '0' for i in binario_aux)
                    size = len(binario)
                else:
                    val = valor[0]
                    size = 0
                    binario = ""

                bits.append(K5[(val, size)])
                bits.append(binario)
        else:
            if valor > 0:
                a = bin(valor)[2:]
                size = K3[len(a)]
            elif valor < 0:
                bin_aux = bin(valor)[3:]
                a = ''.join('1' if i == '0' else '0' for i in bin_aux)
                size = K3[len(a)]
            else:
                size = K3[valor]
```

```
                a = ""

            bits.append(size)
            bits.append(a)

    mensagem = ''.join(bits)
    escreverFicheiro(mensagem, filename)
    return mensagem

def descodificarImagem (filename):
    mensagem = lerFicheiro(filename)
    iK3 = {}
    for key, value in K3.items():
        iK3[value] = key

    iK5 = {}
    for key, value in K5.items():
        iK5[value] = key

    elFinal = []
    codigo = ""
    EOF = True

    while len(mensagem) != 0:
        codigo += mensagem[0]
        mensagem = mensagem[1:]

        if codigo in iK3 and EOF:
            length = iK3[codigo]
            value = mensagem[0: length]
            if len(value) != 0:
                if value[0] == '0':# Negativo
                    value = ''.join('1' if i == '0' else '0' for i in value) #
Complemento
                    value = -int(value, 2)
                else:
                    value = int(value, 2)
            else:
                value = 0

            elFinal.append(value)
            codigo = ''
            mensagem = mensagem[length:]
            EOF = False

        if codigo in iK5 and not EOF:

            tuplo = iK5[codigo]

            if tuplo == (0, 0):
                elFinal.append((0, 0))
                EOF = True
            else:
                length = tuplo[1]
                nmr = 0
                if length != 0:
                    nmr = mensagem[0:length]
                    # Negativo
                    if nmr[0] == '0':
                        nmr = ''.join('1' if i == '0' else '0' for i in nmr) #
Complemento
                        nmr = -int(nmr, 2)
                    else:
                        nmr = int(nmr, 2)
                zeros = tuplo[0]
                elFinal.append((zeros, nmr))
                mensagem = mensagem[length:]
            codigo = ''
    return elFinal

def escreverFicheiro (text, nome):
    colocar = len(text)%8
    valor = 0
    if colocar != 0:
        valor = 8 - colocar
        for i in range(valor):
```

```
            text += "0"

    binarios = np.array(list(text))
    a = int(len(binarios)/8)
    binarios = binarios.reshape(a,8)
    ficheiro = arr.array('B')
    ficheiro.append(valor)

    for numeroBits in range (len(binarios)):
        stringInt = ''.join(map(str, binarios[numeroBits]))
        inteiro = int(stringInt,2)
        ficheiro.append(inteiro)

    f = open(nome, 'wb')
    ficheiro.tofile(f)

def lerFicheiro(nomeFicheiro):
    ficheiro = np.fromfile(nomeFicheiro, dtype='B')
    remover = ficheiro[0]
    ficheiro = ficheiro[1:]
    arr_ = []

    for char in ficheiro:
        arr_.append(bin(char)[2:].zfill(8))

    binario = ''.join(arr_)
    binario = binario[:len(binario) - remover]

    return binario
```