Corrección de Errores.

Adrian Fernando Loya Sabido, Luis Angel Tun Reyes,

Alan Daniel Villanueva Paredes.

February 8, 2021

Abstract

En este documento se realizó un programa capaz de comprimir un texto codificando su contenido con el Algoritmo de Huffman y también capaz de corregir errores con el código de Hamming basado en matrices. La finalidad de este documento es la manipulación de información.

1 Introducción

La finalidad principal de este reporte será documentar los resultados obtenidos de realizar la codificación y decodifiación de un texto. Este texto puede contener cualquier tipo de símbolo, lo que se busca es realizar un programa capaz de leer cualquier cadena de caracteres, codificarla, corregir errores y decodificarla a su estado original.

Para que se realice todo este proceso se siguió con el algoritmo de Huffman [1] el cual es un algoritmo de compresión de datos. Se utiliza comúnmente para codificar un determinado símbolo. Para asignar este código se estima la probabilidad de aparición de cada símbolo dentro del texto. Este código se llama código prefijo y hace referencia a una cadena de bits que representa a un símbolo y que no puede pertenecer a otro símbolo. Fue desarrollado por David Huffman y para su uso normalmente se realiza un árbol binario en donde se representan la frecuencia de cada uno de los símbolos en el texto. Realizando el recorrido de este árbol se obtiene el código prefijo para dicho símbolo en específico.

Este algoritmo se utiliza meramente para codificar cualquier símbolo dentro de un texto y volverlo un código binario. Todo esto con la finalidad de obtener un archivo que contenga una cantidad menor de información, con esto ahorrando espacio de memoria.

Para el siguiente punto dentro del proceso de codificación y decodificación, se utilizó el algoritmo de Hamming que generalmente se usa para corrección de errores, sin embargo, para el uso de este algoritmo se deben cumplir ciertas condiciones. No entraremos en detalles, sin embargo, dentro de [1] se encuentra la información más a detalle de lo que se hace. El algoritmo de Hamming es un código detector y corrector de fallos basado en la paridad.

Una vez completado la codificación en Huffman y la corrección de errores en Hamming se realiza la decodificación la cual es simplemente el proceso inverso de lo que se realiza en la codificación. Con esto en primera instancia se debe obtener la misma información, es decir, el texto original.

2 Desarrollo

En esta sección se mostrará a detalle como se realizó el programa que es capaz de codificar y decodificar un texto.

El texto que se usará el libro llamado "Llano en Llamas" del escritor Juan Rulfo. Como primera instancia se realizaron las siguientes lineas de código que corresponden a la codificación de Huffman.

```
    def __init__(self,dato,symbol):
        self.freq=dato
        self.symbol=symbol
        self.left=None
        self.right=None
        self.encode=None
        self.visit=False

def mini(Q):
    minimo=Nodo(999999,'None')
    eliminar=0
    for i in range(0,len(Q)):
        if Q[i].freq < minimo.freq:
            minimo= Q[i]
            eliminar=i #esta variable es innecesaria
    Q.pop(eliminar)
    return minimo
def huffman(directorio):#aquí se crea el arbol pero solo returnea el nodo raiz principal
    n=len(directorio)
    Q=[]
    for i in directorio:
        Q.append(Nodo(directorio.get(i),i)) #Metemos a Q el nodo que se forma con el simbolo y su frecuencia
    for i in range(0,n-1):
        #simb=[]
        #for j in range (0,len(Q)): #estas tres lineas son namás pa ver lo que contiene Q como simbolos
            #simb+=[Q[j].symbol]
        #print(simb)
        #simb=[] #reinicio el conjunto de elementos de Q
        x=mini(Q) #escogemos al mas chiquito de todos
        y=mini(Q) #escogemos al segundo mas chiquito de todos
        z=Nodo(x.freq+y.freq,x.symbol+y.symbol)
        z.left=x
        z.right=y
        z.left.encode=0
        z.right.encode=1
        z.freq #pa que
        x.freq #pa que
        y.freq #pa que
        Q.append(z)
        #for j in range (0,len(Q)): #estas tres lineas son namás pa ver lo que contiene Q como simbolos
            #simb+=[Q[j].symbol]
        #print(simb)
    return Q[0] #inicialmente returnea mini(Q) pero cuál es la diferencia con returnear solo al único elemento de Q? Q[0]
        #
```

Figure 1: Funciones para crear el árbol de Huffman.

Lo que sigue en el código son las funciones que permitieron realizar el recorrido del árbol de Huffman.

En la figura 3, se observan las funciones que hacen la decodificación de Huffman. En la figura 4 se observan las funciones dividir4bits , creatingcodeword y creatingcode que son funciones que se utilizan para la creación del código que utiliza Hamming para corregir los errores. En la figura 5 se observan las funciones que definen el diccionario para la codificación Hamming, y las funciones que corrigen el error en Hamming. En la figura 6 se muestra una función de verificación para revisar el resultado de la multiplicación de las matrices de Hamming. Por último la función corregir bit.

Como siguiente en las figuras 8, 9, 10 y 11 se muestra el código principal que es la implementación de todas las funciones anteriores en sus diferentes secciones. En la sección de Hamming se uso un programa externo que inyectaba errores al código de Hamming obtenido por la codificación de Huffman, todo esto con la finalidad de demostrar que el Algoritmo de Hamming funciona.

```
def codificacionH(raiz,i,codificacion=""): #Le metemos el nodo desde donde empezamos, símbolo i de nuestro dicci de caracteres
    #print("codificacion en proceso:",codificacion)
    #print("símbolo del nodo actual:",repr(raiz.symbol), "y su longitud: ", len(raiz.symbol))
    if len(raiz.symbol)>1:                                  #y cadena vacía codificacion que será la codificacion
        #print("el símbolo en nodo izq es ",repr(raiz.left.symbol), "y en el derecho", repr(raiz.right.symbol))
        if i in raiz.left.symbol:#checamos si nuestro símbolo está en el nodo izquierdo
            codificacion+=str(raiz.left.encode) #meto el símbolo adecuado a la codificación
            return codificacionH(raiz.left,i,codificacion) #si sí está vamos a ese nodo y repetimos

        else:
            codificacion+=str(raiz.right.encode) #meto el símbolo adecuado a la codificación
            return codificacionH(raiz.right,i,codificacion) #si no estuvo en el izq. está en el derecho y repetimos
    #print("codificación final: ", codificacion)
    return codificacion

def v(raiz,codeH,i): #Le metodomos un nodo y la codificación de huffman de un simbolo. Nos permite pasar al siguiente nodo
    #print(raiz.left.symbol)                                                    tomando en cuenta la codificacion
    #print("i;",i)
    #print("codeH;", codeH)
    #print("esto es en v. i:",i,"Longitud codeH", len(codeH))
    if raiz.left.encode == int(codeH[i]):
        raiz=raiz.left
        i=i+1 #este es el contador para saber en qué posición de la cadena estoy
    else:
        #print(raiz.right.freq)
        raiz=raiz.right
        i=i+1
    #print(raiz.freq)
    return raiz,i

def v2(raiz_aux,codeH,i): #en conjunto con v1, hace el recorrido del arbol hasta su respectiva hoja
    c=""
    while len(raiz_aux.symbol) > 1: #siempre y cuando no estemos en un nodo hoja
        #print("raiz_aux:", raiz_aux.symbol)
        (raiz_aux,i)=v(raiz_aux,codeH,i)
    c+=raiz_aux.symbol #el simbolo de la hora a la que llegamos. ¿Es necesario declarle una variable?
    #print("c:", c)
    return c,raiz_aux,i #regresa el símbolo de la hoja en donde acabamos, el nodo hoja y en qué elemento de la codificación
                                                                                #nos quedamos
```

Figure 2: Recorrido del árbol de Huffman para crear la codificación.

```
def decodificacion(values,keys,text):
    aux=""
    res=""
    while text:
        for key in range(0,len(keys)):
            valor_key=keys[key]
            if text.startswith(valor_key):
                res += str(values[key])
                text = text[len(valor_key):]
    return res

def huffman_Inv(raiz,codeH): #pide la codificación de una palabra y la raiz principal del arbol con el que codificó la palabra
    c=""
    i=1 #contador para la posición en la cadena. La posición cero se pone fuera del ciclo
    raiz_aux=raiz    #guardamos la raiz en una variable auxiliar
    if raiz_aux.left.encode==int(codeH[0]): #si yendo hacia la izquierda está nuestro primer símbolo de la codificación...
        raiz_aux=raiz_aux.left #nos movemos hacia ese nodo izquierdo

        while(i < len(codeH)): #continuamos el proceso hasta que acabamoscon la cadena
            #print("i en el while explicitamente acotado:",i,"Longitud codeH", len(codeH))
            (c_aux,raiz_aux,i)=v2(raiz_aux,codeH,i) #recorremos toda el arbol hasta un nodo hoja de acuerdo a la codificacion
            raiz_aux=raiz #¿Para qué pedimos raiz_aux en la función anterior si la vamos a reiniciar de todas formas?
            c+=c_aux #sumamos los símbolos de las hojas a las que llegamos para formar la cadena
        if len(codeH)==1:
            (c_aux,raiz_aux,i)=v2(raiz_aux,codeH,i)
            raiz_aux=raiz
            c+=c_aux
    else:
        raiz_aux=raiz_aux.right
        while(i < len(codeH)):
            #print("i en el while explicitamente acotado:",i,"Longitud codeH", len(codeH))
            (c_aux,raiz_aux,i)=v2(raiz_aux,codeH,i)
            raiz_aux=raiz
            c+=c_aux
        if len(codeH)==1:
            (c_aux,raiz_aux,i)=v2(raiz_aux,codeH,i)
            raiz_aux=raiz
            c+=c_aux
    #print("decodificación con huffman_inv:",c)
    return c
```

Figure 3: Decodificación de Huffman.

```python
def dividir4bits(valor_binario):
    power_set=[]
    valor=""
    for i in range(0,len(valor_binario)):
        valor=valor+valor_binario[i]

        if len(valor) == 4:
            power_set.append(valor)
            valor=""
    return power_set

def creating_code_word(H, bit_string,dirHam):
    #*print("transformamos este byte con hamminh: ", bit_string)
    codeword = ''
    r = len(H)
    for row in H:
        counter = 0
        #print(len(row))
        for i in range(r, len(row)):
            if bit_string[i - r] == '1':
                counter += int(row[i])
            else:
                continue
        codeword += str((counter % 2))
        #print("la contrucción de Hammin va en: ",codeword)
    codeword += bit_string
    #print("la codeword de Hamming de", bit_string, "es ",codeword)
    value=fill_DirHam(codeword)
    dirHam[bit_string]=value
    return codeword,dirHam

def creating_code(H,power_set,dirHam):
    r = len(H)
    code = []
    #power_set = ['0'*(r- len(bin(a)[2:])) + bin(a)[2:] for a in range(2**r)]
    #print(len(power_set))
    for word in power_set:
        #print("vamos a ver qué es code y cómo evoluciona cuando se le añaden los chunks de 4bits: ", code)
        (codeword,dirHam)=creating_code_word(H, word,dirHam)
        code.append(codeword)
        #code.append(creating_code_word(H, word,dirHam))
    return code #code es el conjunto de 8bits después de hamming
```

Figure 4: Creación del código Hamming.

```python
def fill_DirHam(codeword_aux):
    codeword=[]
    dirHam_aux={}
    codeword[:0]=codeword_aux
    #codeword.pop(-1) por qué el pop?
    codeword = "".join(codeword)
    return codeword

def columnaerror (Vector): #lista de listas despues de aplicar la multiplicacion de H1\n". Nota: en realidad parece que solo
    ilist=[]                                                    #la posición del bit erróneo de un vector
    for j in range (0, len(Vector)):
        for i in range (0, len(H0[0])):
            columna= []
            columna = [fila[i] for fila in H0]
            #print("comparamos el vector y la columna respectivamente: ", Vector[j], columna)
            if Vector[j]==columna:
                #ilist+= [i]
                return i
                #break
            elif Vector[j]== [0]*len(Vector[j]): #no creo que sea necesario declarar un caso sin error xq solo se entra
                ilist+= ["no hay error"]          #a esta función cuando se ha detectado un error
                break
    #print("hola")
    return ilist

def corregirerror(codeword,ilist): #codeword es lista, ilist es la lista de la columna que tiene el error de codeword\n",
    for i in range (0, len(ilist)):
        if ilist[i]!= "no hay error":
            for j in range (0,ilist[i]):
                codeword[ilist[i]]
```

Figure 5: Funciones para corrección de errores.

```python
def verificador(code,H):
    suma_aux=0
    suma=""
    resul=""
    res_mult=[]
    valido=1
    for row in H:
        for i in range(0,len(code)):
            val1=int(code[i])
            val2=int(row[i])
            suma_aux=val1*val2
            if suma_aux%2 == 0:
                resul='0'
            else:
                resul='1'
            suma+=resul
            #print(suma)
            #print(row[i])
        res_mult.append(suma)
        suma=""

    #print(res_mult)
    aux=0
    vector_resultante=[]
    for i in res_mult:
        #print(i)
        for j in i:
            aux=aux+int(j)
        if aux%2 ==0:
            vector_resultante.append(0)
        else:
            vector_resultante.append(1)
        aux=0

    if max(vector_resultante)==1:
        valido=0

    return valido,vector_resultante
```

Figure 6: Función verificador.

```python
def corregir_bit(bit_malo,codeword):
    codewordcorregido=""
    for i in range(0,len(codeword)):
        if i == bit_malo:
            if codeword[i]==0:
                codeword[i]=1
                break
            else:
                codeword[i]=0
                break
    codeword_=""
    for i in codeword: #transformamos en string
        codeword_=codeword_+str(i)
    return codeword_
```

Figure 7: Corregir bit.

```
##########################################################################################
#########################################Codificación con huffman#################
```

```
ruta1=('C:/Users/loya_/Desktop/cosas de maestria/prueba.txt')
archivo1=codecs.open(ruta1,encoding='utf-8')
linea1 = archivo1.read() #dejo o quito el .replace("\n", " ") ?
#print("Archivo abierto: ", linea1)
directorio = {}
#print("input: ",repr(linea1)) #repr para ver los \n, \r etc


#Generar el directorio de las frecuencias del archivo que se abra en ruta1
for i in linea1:
    if i in directorio:
        directorio[i]+=1
    else:
        directorio[i]=1
print("directorio: ",directorio)
raiz=huffman(directorio) #la raiz principal del arbol que se crea con huffman
caracteres=[]
for i in directorio:
    caracteres.append(i) #metemos a todos los símbolos que tenemos

dirFinal={}
for i in caracteres:
    #print("símbolo:",repr(i))
    dirFinal[i]=codificacionH(raiz,i)

print("dirFinal: ", dirFinal)
```

Figure 8: Codificación Huffman

```
#########################################################################################################################
#####################################Codificación con Hamming#############################################################


code_Huffman=""
cont=0
for i in linea1: #sustituimos los caracteres por su codigo huffman
    code=dirFinal[i]
    code_Huffman=code_Huffman+code

H0 = [[1, 0, 0, 0, 0, 0, 0,1],
      [0, 1, 0, 0, 1, 1, 1,0],
      [0, 0, 1, 0, 0, 1, 1,1],
      [0, 0, 0, 1, 1, 0, 1,0]]

power_set=dividir4bits(code_Huffman)
dirHam={} #directorio para las palabra-codigo (hamming)

codeword=creating_code(H0,power_set,dirHam)

code_Hamming=""
for i in power_set:
    code=dirHam[i]
    code_Hamming=code_Hamming+code
print("code_Hamming: ", code_Hamming)

ruta3=('C:/Users/loya_/Desktop/Hamming.txt') #escribimos el output de codde_Hamming para que sea el input del interferenciador
archivo3=open(ruta3,"w")
archivo3.writelines([str(len(code_Hamming)),"\n",code_Hamming]) #dejo o quito el .replace("\n", " ") ?
archivo3.close()

with open('C:/Users/loya_/Desktop/Hamming.txt', 'r') as txtfile:
    Hamming = txtfile.read()

change text into a binary array
binarray = ' '.join(format(ch, 'b') for ch in bytearray(Hamming,"utf-8"))
print(type(binarray.encode()))
with open('C:/Users/loya_/Desktop/Entrada.bin', 'wb') as binfile:
    binfile.write(binarray.encode())
binfile.close()

with open('C:/Users/loya_/Desktop/Salida.bin', "rb") as file:
     data = file.read(8)
print(data)
```

Figure 9: Codificación Hamming.

```
with open('C:/Users/loya_/Desktop/out.txt', "w") as doc:
    doc.write(" ".join(map(str,data)))
    doc.write("\n")
#########################################################################################################################
#####################################decode Hamming######################################################################
code=""
code_word=[]
ruta2=('C:/Users/loya_/Desktop/Hamming.txt')
archivo2=codecs.open(ruta2,encoding='utf-8')
code_interferido = archivo2.read() #dejo o quito el .replace("\n", " ") ?
for i in code_interferido: #podemos insertar aquí el mensaje interferido. #cambiar a code_word para cero errores
    code+=i
    if len(code)==8:
        code_word.append(code)
        #print(code)
        code=""
#print(code_word[0], code_word[1],code_word[2])
Vector=[]
Lista=[]
resultado=""
#print("otra vez code_word: ",code_word)
#codewordbad solo funciona para la frase los gatos en la casa
for i in code_Hamming: #poner code_word en vez de codewordbad si queremos que todo sea correcto aquí se inserta el mensaje in
    #i='11000001'
    (valido,vector_res)=verificador(i,H0)
    #print("vector resultante ",vector_res," de aplicar H a ",i)
    #print("valido",valido)
    if valido==0:
        bit_incorrecto=columnaerror([vector_res])
        #print("bit_incorrecto: ", bit_incorrecto)
        code_word_aux=[int(x) for x in i] #Convertir la cadena a una lista de enteros
        code_coregido=corregir_bit(bit_incorrecto,code_word_aux)
        #print("Bit malo:"+str(bit_incorrecto))
        #print("code word malo: "+i)
        #print("code word bien: "+code_coregido)
        resultado+=code_coregido
    else:
        resultado+=i #La cadena de hamming completa con correcciones hechas
#print("code",resultado, "longitud", len(resultado)) #resultado es la cadena ya corregida
#print("interferido:", code_interferido, "longitud", len(code_interferido))
```

Figure 10: Decodificación Hamming.

```
code_Huffman=""

for i in linea1:
    code=dirFinal[i]
    code_Huffman=code_Huffman+code

codeH=code_Huffman
#print("(codeH) Codificación de",linea1, "a Huffman", codeH)


#print(list(ca)) #huffman inverso
#print(list(dirFinal.keys()))  #huffman original
codigo_huff_inv=huffman_Inv(raiz,codeH)
print(codigo_huff_inv)
#print("decodificación: ",codigo_huff_inv)
```

Figure 11: Decodificación Huffman.

3 Resultados.

Como resultados se obtuvieron las siguientes salidas que se muestran en las figuras 12, 13 y 14.

```
directorio:  {'J': 7, 'u': 964, 'a': 3135, 'n': 1433, ' ': 5088, 'R': 2, 'l': 1335, 'f': 82, 'o': 2575, '\r': 137, '\n': 137,
'(': 4, 'M': 8, 'é': 91, 'x': 2, 'i': 882, 'c': 815, ',': 277, '1': 5, '9': 4, '8': 2, '-': 2, '6': 2, ')': 3, 'E': 49, 'L':
50, 'e': 2557, 'm': 674, 's': 1909, 'O': 6, 'r': 1656, 'g': 277, 't': 828, 'p': 453, 'b': 422, 'd': 1116, 'v': 247, 'A': 37,
'N': 16, 'º': 1, '4': 3, '5': 2, '0': 3, '3': 3, 'Y': 44, 'q': 279, '.': 309, 'C': 28, '“': 15, '¡': 14, 'V': 7, 'P': 75,
'F': 11, '!': 15, '”': 17, 'y': 202, 'ó': 99, 'h': 298, 'á': 125, 'z': 91, 'j': 140, 'í': 247, ':': 19, 'D': 31, 'S': 41,
'B': 6, ';': 29, 'ñ': 29, '–': 27, '¿': 1, 'Q': 4, '?': 1, 'Z': 29, '&': 4, 'ú': 16, 'I': 3, 'U': 3, 'T': 16, 'H': 10, 'G':
3, 'ü': 3, '\xad': 9, 'É': 2}
dirFinal:  {'J': '100101000001', 'u': '10011', 'a': '010', 'n': '0011', ' ': '111', 'R': '10010100000011', 'l': '11001', 'f':
'110001011', 'o': '000', '\r': '10110011', '\n': '10110100', '(': '1011010101000', 'M': '100101011101', 'é': '00100101', 'x':
'10110101100000', 'i': '10001', 'c': '01100', ',': '1011011', '1': '001001001001', '9': '1011010101001', '8': '1011010110000
1', '-': '10110101100010', '6': '10110101100011', ')': '0010010011101', 'E': '001001111', 'L': '011010010', 'e': '1101', 'm':
'00101', 's': '1010', 'O': '001001001111', 'r': '0111', 'g': '1100000', 't': '10000', 'p': '100100', 'b': '011011', 'd': '101
11', 'v': '1001011', 'A': '1011010111', 'N': '10010101111', 'º': '00100100111000', '4': '0110100110010', '5': '001001001000
0', '0': '0110100110011', '3': '0110100110100', 'Y': '001001110', 'q': '1100001', '.': '001000', 'C': '1001010001', '“': '100
10101010', '¡': '10010100001', 'V': '100101011100', 'P': '110001010', 'F': '00100100110', '!': '10010101011', '”': '101101010
11', 'y': '0110101', 'ó': '01101000', 'h': '1100011', 'á': '10110010', 'z': '00100110', 'j': '11000100', 'í': '1011000', ':':
'10110101101', 'D': '1001010110', 'S': '001001000', 'B': '011010011000', ';': '1001010010', 'ñ': '1001010011', '–': '01101001
11', '¿': '00100100111001', 'Q': '1011010101010', '?': '10010100000010', 'Z': '1001010100', '&': '1011010101011', 'ú': '10110
101000', 'I': '0110100110101', 'U': '0110100110110', 'T': '10110101001', 'H': '00100100101', 'G': '0110100110111', 'ü': '1001
010000000', '\xad': '101101011001', 'É': '0010010010001'}
```

Figure 12: Directorio de frecuencias y codificación de Huffman.

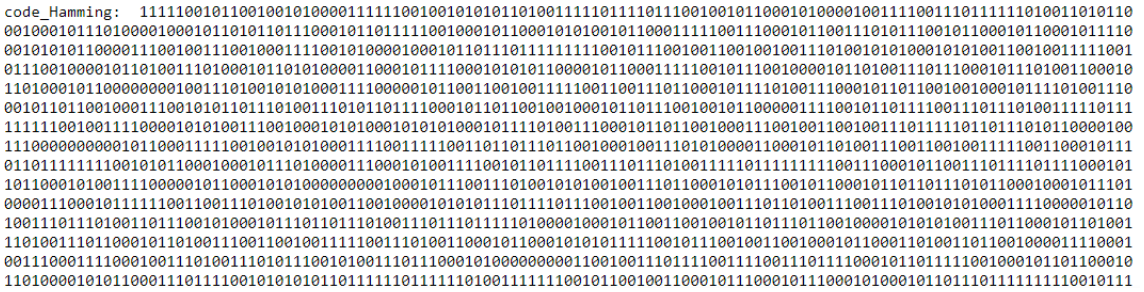

Figure 13: Código Hamming.

```
Juan Rulfo
(México, 1918-1986)

El Llano en llamas
Originalmente publicado en la revista América
Nº 64, diciembre, 1950
(El Llano en llamas, 1953)

Ya mataron a la perra,
pero quedan los perritos...
(Corrido popular)

“¡Viva Petronilo Flores!”
El grito se vino rebotando por los paredones de la barranca y subió hasta donde estábamos nosotros. Luego se deshizo.
Por un rato, el viento que soplaba desde abajo nos trajo un tumulto de voces amontonadas, haciendo un ruido igual al que hace
el agua crecida cuando rueda sobre pedregales.
En seguida, saliendo de allá mismo, otro grito torció por el recodo de la barranca, volvió a rebotar en los paredones y llegó
todavía con fuerza junto a nosotros:
“¡ Viva mi general Petronilo Flores!”
Nosotros nos miramos.
```

Figure 14: Texto decodificado.

4 Conclusiones

El código de Huffman y el código Hamming se usan en informática para la compresión y corrección de errores. En este documento se demostró que se pueden utilizar para cualquier cadena de caracteres, lo importante es entender el funcionamiento y aplicación de cada uno de estos algoritmos. Por lo que representan una herramienta importante para compresión de datos y corrección de errores.

References

[1] N.L. Biggs and P.M.L.S.E.N.L. Biggs. *Discrete Mathematics*. Oxford science publications. OUP Oxford, 2002.