Huffman y Hamming

Gillermo Reyes Martinez, Pierre Armando Solis Rodriguez

Diciembre 2022

1 Introduction

La informacion en la actualidad necesita ser almacenada y transmitida de manera mas eficiente año con año, ya que la poblacion mundial consume, crea y transmite todo tipo de informacion todos los dias en tasas que van en aumento, esta informacion necesita llegar a todas partes del mundo lo mas rapido posible, y lo mas importante, en buen estado pues la gente que recibe y utiliza estos datos tiene que ser capaz de leer, escuchar e intepretar el contenido, la informacion comprimida es mas liviana en cuanto necesita menos ancho de banda y la informacion codificada puede arreglar posibles errores en la transmision debido a interferencias en el camino, pensando en que cualquier cosa que tambien transmita datos podria causar estragos en nuestro mensaje, si la informacion no esta codificada no existiria una referencia de como debe ser, o mejor dicho como era originalmente, no es de esperar que hoy en dia exista toda una infraestructura dedicada al control de la transmision de los datos desde nivel local hasta nivel internacional.
En esta ocasion hablaremos de un algoritmo de compresion desarrollado por David A Huffman, que utiliza un alfabeto de n simbolos para crear un codigo a partir de sus frecuencias en el contenido y un codigo de deteccion y correccion de errores desarrollado por Richard Hamming, los codigos de Hamming pueden detectar errores en uno o dos bits y corregir errores en un bit.

2 Codigos de Huffman

El codigo de Huffman se crea a partir de tomar un alfabeto de n simbolos y analisar la frecuencia con la que aparecen en un texto, por ejemplo[1].
El algoritmo consiste en la creacion de un arbol binario que tiene cada uno de los sibolos por hoja, y se construye de tal forma que al recorrerlo desde la raiz a cada una de sus hojas se puede obtener un codigo asociado (codigo de Huffman):

- 1 Se crean varios arboles, uno por cada simbolo del alfabeto, cada uno seria un nodo sin hijos, cada uno con una etiqueta que lleva el simbolo asociado y la frecuencia con la que aparecen.
- 2 se toman los dos arboles de menor frecuencia y se unen creando un nuevo arbol, la etiqueta sera la suma de las etiquetas de los arboles anteriores y cada nodo se convierte en un hijo del nuevo nodo raiz, se etiquetan con un 0 al de la izquierda y un 1 al de la derecha.
- 3 se repite el paso 2 hasta que solo quede un arbol. este arbol permitira conocer el codigo asociado a un simbolo y obtener el simbolo asociado a dicho codigo.

Si se quiere conocer el codigo asociado al simbolo entonces:

- 1 Debe comenzar el recorrido del arbol en la raiz.
- 2 Extraer el codigo del primer simbolo a decodificar.

- 3 Descender por la rama etiquetada con ese simbolo.
- 4 volver al paso dos hata que llegue a la hoja que corresponde con el simbolo que se esta buscando.

Si por el contrario se quiere conocer el simbolo asociado al codigo entonces:

- 1 Comenzar con un código vacío.
- 2 Iniciar el recorrido del árbol en la hoja asociada al símbolo.
- 3 Comenzar un recorrido del árbol hacia arriba.
- 4 Cada vez que se suba un nivel, añadir al código la etiqueta de la rama que se ha recorrido.
- 5 Tras llegar a la raíz, invertir el código.
- 6 El resultado es el código Huffman deseado.

3 Codigos de Hamming

Los codigos de Hamming son una familia de codigos lineales de correccion de errores, pueden detectar errores de hasta dos bits o corregir errores de un bit sin deteccion de errores no corregidos. Se dice que los codigos de Hamming son perfectos ya que alcanzan la tasa maxima posible con su longitud de bloque y distancia minima de tres. Richard W Hamming inveto estos codigos en 1950 como una forma de corregir automaticamente los errores introducidos por los lectores de tarjetas perforadas. En su articulo original, Hamming elaboro su idea general, pero se centro especificamente en el codigo Hamming(7,4) que agrega tres bits de paridad a cuatro bits de datos[2].
El numero de bits de paridad requeridos para n bits de datos esta determinado por la desigualdad: $2^p >= n + p + 1$ donde p es el numero de bits de paridad y n es el numero de bits de datos. Se debe tener en cuenta que en la palabra de codigo que se quiere generar, las posiciones que son potencia de dos se utilizan como bits de paridad, mientras que el resto se utiliza como bits de datos. La posicion de cada bit de paridad que se genera en la etapa de codificacion, determina la secuencia de los bits de datos que verifica. El valor de cada bit de paridad se obtiene sumando la cantida de unos que hubo en las posiciones comprobadas, si esta suma es impar, el valor del bit de paridad es uno, de lo contrario es cero. Si se deseara enviar la secuencia 1010, en la palabra codigo se asignarian los bits de datos en forma de: _ _ 0 _ 1 0 1, los espacios en blanco corresponden a los bits de paridad de modo que la palabra codigo quedaria 0 1 0 0 1 0 1[3].
Para decodificar una palabra codigo existen dos etapas, la deteccion y la correccion:

- Deteccion: Se detecta cuando existe un bit erroneo, se genera una nueva palabra codigo recalculando sobre las posiciones de los bits de paridad, y se compara con la secuencia recibida, si son iguales no hubo errores, de lo contrario se determina que hubo un error y se procede a la correccion.

- Correccion: Se compara la secuencia recibida con la palabra codigo calculada en la etapa anterior, se suman las posiciones de los bits de paridad que difieren, el resultado determina la posicion del bit erroneo y se sustituye por su valor opuesto.

4 Codigo

El siguiente es un programa de python que comprime un texto por el algoritmo de huffman, lo codifica por medio de hamming, despues se descomprime por medio de Huffman con los errores corregido por Hamming.

Primero es necesario crear el arbol de Huffman a partir del cual se realizara la codificacion, en las Fig 1. y Fig 2. se puede observar como se ve en el codigo de python:

```
# -*- coding: utf-8 -*-
import sys
import codecs
import os
import csv
import operator

#creacion del arbol de Huffman
class Nodo:
    def __init__(self,dato,symbol):
        self.freq=dato
        self.symbol=symbol
        self.left=None
        self.right=None
        self.encode=None
        self.visit=False

def insertar(raiz,nodo):
    if raiz is None:
        raiz=nodo
    else:
        if raiz.freq < nodo.freq:
            if raiz.right is None:
                raiz.right=nodo
            else:
                insertar(raiz.right,nodo)
        else:
            if raiz.left is None:
                raiz.left=nodo
            else:
                insertar(raiz.left,nodo)

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
    Q.pop(eliminar)
    return minimo
```

Fig 1. Primera parte de la creacion del arbol de Huffman

```
    #creacion del nodo raiz principal
def huffman(directorio):
    n=len(directorio)
    Q=[]
    for i in directorio:
        #Se añaden los nuevos nodos que se forman con la etiqueta de simbolo y frecuencia
        Q.append(Nodo(directorio.get(i),i))
    for i in range(0,n-1):
        #Se toma al nodo mas pequeño
        x=mini(Q)
        #Tambien se selecciona al segudo nodo mas pequeño
        y=mini(Q)
        z=Nodo(x.freq+y.freq,x.symbol+y.symbol)
        z.left=x
        z.right=y
        z.left.encode=0
        z.right.encode=1
        Q.append(z)
    return mini(Q)

#la funcion codificacionH() recibe por parametro el nodo inicial, el simbolo i del diccionario de caracteres y la codificacion.
def codificacionH(raiz,i,codificacion=""):
    if len(raiz.symbol)>1:
        #verificamos si el simbolo se encuentra en el nodo izquierdo
        if i in raiz.left.symbol:
            #Introducimos el simbolo correcto a la codificacion
            codificacion+=str(raiz.left.encode)
            #si el simbolo se encuentra en el nodo izquierdo recorremos hasta llegar y repetimos el proceso
            return codificacionH(raiz.left,i,codificacion)

        else:
            #Introducimos el simbolo correcto a la codificacion
            codificacion+=str(raiz.right.encode)
            #En caso de no encontrarlo en el nodo izquierdo podemos concluir que se encuentra en el nodo derecho, luego repetimos el proceso
            return codificacionH(raiz.right,i,codificacion)
    return codificacion
```

Fig 2. Segunda mitad de la creacion del arbol de Huffman

A continuacion en la Fig 3. se puede observar la parte del codigo que realiza el recorrido del arbol de Huffman

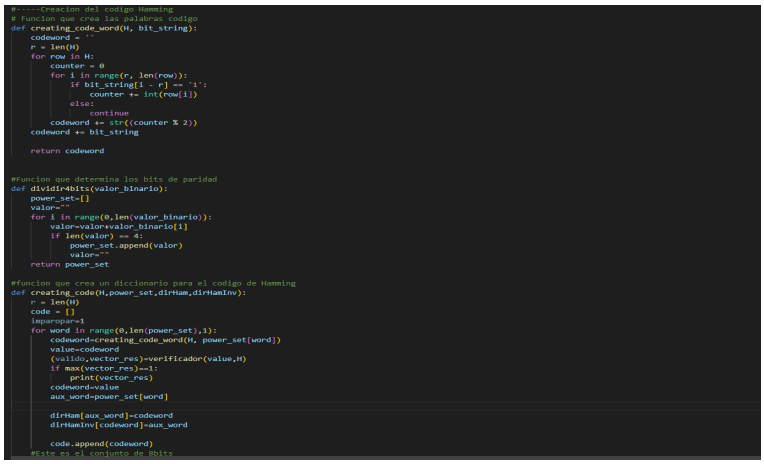
```
#En esta funcion se recibe por parametro un nodo y la codificacion de Huffman de un simbolo, esto nos permite dirigirnos al siguiente nodo basandonos en la codificacion
def v(raiz,codeH,i):

    if raiz.left.encode == int(codeH[i]):
        raiz=raiz.left
        #i es un contador que nos indica en que posision nos encontramos
        i=i+1
    else:
        raiz=raiz.right
        i=i+1
    return raiz,i

#Esta funcion trabaja en conjunto con la funcion v, de modo que se recorre el arbol hasta su respectiva hoja
def v2(raiz_aux,codeH,i):
    c=""
    #El ciclo seguira repitiendose mientras no estemos en un nodo hoja
    while len(raiz_aux.symbol) > 1:
        (raiz_aux,i)=v(raiz_aux,codeH,i)
    #Aqui se obtiene el simbolo de la hoja a la que se llego
    c+=raiz_aux.symbol
    #Retornamos el simbolo de la hoja final, el nodo al que corresponde, y en que elemento de la codificacion nos encontramos
    return c,raiz_aux,i
```

Fig 3. Recorrido del arbol de Huffman

En la fig 4. se pueden observar las funciones relacionadas a la codificaicion de Hamming.



```
#-----Creacion del codigo Hamming
# Funcion que crea las palabras codigo
def creating_code_word(H, bit_string):
    codeword = ''
    r = len(H)
    for row in H:
        counter = 0
        for i in range(r, len(row)):
            if bit_string[i - r] == '1':
                counter += int(row[i])
            else:
                continue
        codeword += str((counter % 2))
    codeword += bit_string

    return codeword


#Funcion que determina los bits de paridad
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

#funcion que crea un diccionario para el codigo de Hamming
def creating_code(H,power_set,dirHam,dirHamInv):
    r = len(H)
    code = []
    imparopar=1
    for word in range(0,len(power_set),1):
        codeword=creating_code_word(H, power_set[word])
        value=codeword
        (valido,vector_res)=verificador(value,H)
        if max(vector_res)==1:
            print(vector_res)
        codeword=value
        aux_word=power_set[word]

        dirHam[aux_word]=codeword
        dirHamInv[codeword]=aux_word

        code.append(codeword)
    #Este es el conjunto de 8bits
```

Fig 4. Codificacion de Hamming.

En las fig 5 y 6 se observan la lineas de codigo relacionadas a la correccion de errores de Hamming.

```
#Lista de listas generada despues de multiplicar H1
def columnaerror (Vector,H):
    ilist=[]
    for j in range (0, len(Vector)):
        for i in range (0, len(H[0])):
            columna= []
            columna = [fila[i] for fila in H]
            if Vector[j]==columna:
                return i
            elif Vector[j]== [0]*len(Vector[j]):
                ilist+= ["no hay error"]
                break
    return ilist

def recuperar_Archivo(ruta1,H):
    archivo1=codecs.open(ruta1,encoding='utf-8')
    cont_archivo_Codi = archivo1.read().replace("\n", " ")
    binario_recuperado=cont_archivo_Codi
    archivo_corregido=corregir_Archivo(binario_recuperado,H)

    return archivo_corregido
```

Fig 5. Primera mitad de la correccion de errores de Hamming.

```python
def corregir_Archivo(binario_recuperado,H):
    code_word=[]
    code=""
    for i in binario_recuperado:
        code+=i
        if len(code)==8:
            code_word.append(code)
            code=""

    resultado=""
    for i in code_word:
        (valido,vector_res)=verificador(i,H)
        if valido==0:
            bit_incorrecto=columnaerror([vector_res],H)

            #Se convierte la cadena en una lista de enteros
            code_word_aux=[int(x) for x in i]
            code_coregido=corregir_bit(bit_incorrecto,code_word_aux)
            resultado+=code_coregido
        else:
            resultado+=i

    return resultado


def dividir8bits(valor_binario):
    tam_8=[]
    valor=""
    for i in range(0,len(valor_binario)):
        valor=valor+valor_binario[i]
        if len(valor) == 8:
            tam_8.append(valor)
            valor=""
```

Fig 6. Segunda mitad de la correccione de errores de Hamming.

En la Fig 7. se muestra un algoritmo que verifica el resultado de la multiplicacion de la matrices de Hamming.

```python
def verificador(code,H):
    suma_aux=0
    suma=""
    resul=""
    res_mult=[]
    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
        res_mult.append(suma)
        suma=""

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

    return valido,vector_resultante
```

Fig 7. Verificador de matrices de Hamming.

Las Fig 8. y Fig 9. muestran el codigo en el que se aplica la decodificacion de Huffman

```
def decodificar(archivo_corregido,restoHam,dirHamInv,raiz,ruta_Archivo_Decodificado):
    tam_8=dividir8bits(archivo_corregido)
    huffman_recuperado=""

    for i in tam_8:
        code=dirHamInv[i]
        huffman_recuperado=huffman_recuperado+code

    huffman_recuperado=huffman_recuperado+restoHam
    codigo_huff_inv=huffman_Inv(raiz,huffman_recuperado)

    file = open(ruta_Archivo_Decodificado, "w")
    file.write(codigo_huff_inv)
    file.close()
```

Fig 8. Primera mitad de la decodificacion de Huffman.

```
#Se pasa por parametro la codificacion de una palabra y la raiz principal del arbol con la que se codifico
def huffman_Inv(raiz,codeH):
    c=""
    #Contador que indica la posicion en la cadena; La posicion cero no es parte del ciclo
    i=1
    #La raiz se almacena en una variable temporal
    raiz_aux=raiz
    #Si nuestro primer simbolo de la codificacion se encuentra del lado izquierdo, recorremos el arbol hacia el nodo izquierdo
    if raiz_aux.left.encode==int(codeH[0]):
        raiz_aux=raiz_aux.left

        #El proceso continua hasta llegar al final de la cadena
        while(i < len(codeH)):
            #Se recorre todo el arbol hasta un nodo hoja que coincide con la codificacion codeH, los parametros c_aux, raiz_aux e i corresponden a: el simbolo de la hoja, el nodo de la hoja y l
            (c_aux,raiz_aux,i)=v2(raiz_aux,codeH,i)
            raiz_aux=raiz
            #Se suman los simbolos de las hojas visitadas para formar la cadena
            c+=c_aux
        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)):
            (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
    return c
```

Fig 9. Segunda mitad de la decodificacion de Huffman.

Para finalizar se muestra la parte del codigo en la que se llaman a las funciones y se declara la matriz de Hamming. (Fig 10)

```
#-------------------------------------
#Matriz de 4x8
H =  [[1, 0, 0, 0, 1, 1, 1, 0],
      [0, 1, 0, 0, 0, 1, 1, 1],
      [0, 0, 1, 0, 1, 1, 1, 0],
      [0, 0, 0, 1, 0, 0, 1, 1]]

#Ruta del archivo original
ruta='C:/Users/user/Desktop/textos/archivo.txt'

ruta_Archivo_Original=ruta
#Ruta en donde se escribira el archivo de codificacion
ruta_Archivo_Codificado='C:/Users/user/Desktop/codificacion.txt'
ruta_Archivo_a_corregir=ruta_Archivo_Codificado
#Ruta donde se escribira el archivo decodificado
ruta_Archivo_Decodificado='C:/Users/user/Desktop/decodificacion.txt'


#Aqui se llama a la funcion que creara el archivo codificado
(dirHamInv,raiz,restoHam)=GenerarArchivoCodificado(ruta_Archivo_Original,ruta_Archivo_Codificado,H)

#Aqui se llama a la funcion para la correccion de errores
archivo_corregido=recuperar_Archivo(ruta_Archivo_a_corregir,H)

#Aqui se llama a la funcion para crear el archivo corregido
decodificar(archivo_corregido,restoHam,dirHamInv,raiz,ruta_Archivo_Decodificado)
```

Fig 10. Llamado de funciones.

Para que el programa funciones hay que cambaiar los directorios que se muestran en la imagen con unos propios de quien vaya a ejercutarlo.

5 Referencias

1. https://es-academic.com/dic.nsf/eswiki/65575#cite_note-0
2. https://hmong.es/wiki/Hamming_code
3. http://servicio.bc.uc.edu.ve/facyt/v2n2/2-2-4.pdf