Practica 1: Código Huffman y Primer Teorema de Shannon

Pablo Jiménez Poyatos

8 de febrero de 2024

1. Introducción

El objetivo de esta práctica es obtener los códigos Huffman binarios para los alfabetos español e inglés (S_{Esp} y S_{Eng}, respectivamente) a partir de muestras de texto en cada idioma. Una vez obtenidos, se aplicarán conceptos previamente estudiados tales como el cálculo de la entropía y la longitud media, así como el Primer Teorema de Shannon. Además, se llevará a cabo una comparación de la eficiencia de estos códigos en relación con la codificación binaria estándar (ASCII). La implementación práctica de este trabajo incluirá, ademas de la creación de ambos códigos binarios, el desarrollo de un algoritmo en Python capaz de codificar "Lorentz" usando ambos códigos. Posteriormente, se implementará una función para decodificar cualquier cadena, lo que permitirá comprobar la precisión de los resultados obtenidos.

2. Material usado

Para realizar esta práctica, se han utilizado los ficheros del campus virtual **"GCOM2024_ pract1_ auxiliar_ esp.txt"** y **"GCOM2024_ pract1_ auxiliar_ eng.txt"** como las muestras de texto que vamos a utilizar para crear nuestro código Huffman. Además, he importado en mi script la **librería os** para verificar cuando un string hace referencia a un archivo o a una cadena de caracteres y las funciones **log2, e** y **sqrt** del **módulo math** para el cálculo de la entropía y de su error. A continuación, se detalla el funcionamiento de las principales funciones y clases.

La clase **Nodo** representa un nodo en un árbol binario de búsqueda. Cada uno tiene asociado una clave y un valor, así como referencias a sus hijos izquierdo y derecho. Por otro lado, la clase **ArbolDePares** implementa un árbol binario de búsqueda para pares clave-valor. Proporciona métodos para **insertar, eliminar, encontrar el mínimo, contar nodos** y **generar un diccionario con el código binario**.

Para calcular el código de Huffman, primero se cuenta la frecuencia de cada carácter utilizando la función **"contar_frecuencias"**. Después, se construye un ArbolDePares con nodos (carácter, (frecuencia, primera aparición)) utilizando la función **"arbol_frecuencias"**. Este árbol se ordena según la frecuencia de cada carácter, resolviendo los empates mediante la primera aparición del carácter. Lo utilizamos para obtener los 2 nodos con frecuencias más bajas, los cuales se eliminan del árbol. Sus caracteres se concatenan formando una nueva cadena. Esta cadena junto con la suma de sus frecuencias y el mínimo de las primeras apariciones, se inserta en el árbol. Paralelamente, se crea un nuevo árbol que almacena esta unión, y se añade a un diccionario cuya clave es la unión de los caracteres y el valor, el árbol. Continuamos este proceso hasta que el árbol inicial contiene únicamente un nodo. En este momento, el valor del diccionario con clave la cadena de caracteres de este último nodo, es el árbol de Huffman que buscamos. Finalmente, se genera el código de Huffman utilizando la función **"dic_arbol"**.

Para codificar una palabra, utilizo **"codificar_palabra"**. Esta función, toma una palabra como entrada y un diccionario que asigna códigos a cada carácter. Luego, itera sobre cada carácter y busca en el diccionario su código correspondiente. Los resultados, se concatenan para formar la codificación de la palabra completa. Para comprobarlo, utilizo la función **"decodificacion"**, que toma una cadena binaria y un diccionario de traducción y te devuelve la palabra correspondiente.

Por último, para el estudio de la entropía, longitud media, error y eficiencia respecto del código binario usual, hemos utilizado las funciones **"longitudMedia_entropia"** y **"longitudCBU"**. La primera te calcula la longitud media, la entropía y el error. La segunda, calcula la longitud de una cadena después de codificarla de la forma estándar usando ASCII.

3. Resultados

La **codificación de Huffman** asigna códigos de longitud variable a cada carácter de manera que los caracteres más frecuentes tengan códigos más cortos. El código de cada idioma es:

- S_{Es} = {“a”: 000, “.”: 0010000, “é”: 00100010, “E”: 001000110, “j”: 001000111, “R”: 001001000, “í”: 001001001, “k”: 00100101, “-”: 00100110, “4”: 001001110, “C”: 001001111, “p”: 00101, “s”: 0011, “e”: 010, “r”: 0110, “m”: 01110, “u”: 01111, “n”: 1000, “o”: 1001, “ ”: 101, “l”: 11000, “b”: 1100100, “z”: 1100101, “q”: 11001100, “L”: 11001101, “ó”: 1100111, “i”: 1101, “d”: 11100, “t”: 11101, “c”: 11110, “f”: 1111100, “v”: 1111101, “M”: 111111000, “x”: 1111110010, “P”: 1111110011, “w”: 111111010, “g”: 111111011, “,”: 11111110, “y”: 111111110, “h”: 111111111}
- S_{En} = {“g”: 0000000, “,”: 0000001, “k”: 0000010, “L”: 00000110, “M”: 00000111, “c”: 00001, “f”: 00010, “l”: 00011, “e”: 001, “u”: 010000,“ x”: 010001000, “S”: 010001001, “E”: 010001010, “R”: 010001011, “.”: 0100011, “h”: 01001, “r”: 0101, “d”: 01100, “m”: 01101, “o”: 0111, “a”: 1000, “s”: 1001, “t”: 1010, “n”: 1011, “,”: 110, “4”: 111000000, “-”: 111000001, “A”: 111000010, “T”: 111000011, “w”: 1110001, “b”: 1110010, “z”: 11100110, “q”: 11100111, “y”: 1110100, “v”: 1110101, “p”: 111011, “i”: 1111}

En el anexo he añadido una tabla 1 con la codificación de cada carácter en ambos idiomas.

Una vez hemos obtenido los códigos de cada carácter en ambas lenguas, vamos a comprobar que se verifica el **Primer Teorema de Shannon**, es decir, que *el algoritmo de Huffman acota la longitud media del código simbólico según:* $H(C) \leq L(C) < H(C) + 1$. Para ello, calcularemos la **longitud media** y la **entropía** de cada sistema utilizando las siguientes fórmulas:

$$L(C) := \frac{1}{W} \sum_{i=1}^{N} w_i |c_i| \quad ; \quad H(S) := -\sum_{j=1}^{N} P_j \log_2(P_j) = -\sum_{j=1}^{N} \frac{F_j}{W} \cdot \log_2(\frac{F_j}{W}) \tag{1}$$

donde W es el número total de caracteres de la muestra, N el número de caracteres diferentes, $|c_i|$ la longitud de la codificación del carácter i, P_j las probabilidades de cada carácter y F_j son las frecuencias de cada carácter en la muestra.

Aplicando la **longitud media** 1.1 en S_{Esp} y en S_{Eng} , obtenemos que ambas son iguales y valen $L(S_{\text{Esp}}) = L(S_{\text{Eng}}) = 4{,}30$. Por otro lado, al calcular la entropía 1.2 en la funcion **“`longitudMedia_entropia(texto, S)`”**, obtenemos un **error** del 0.02 (utilizando la propagación cuadrática del error), por lo que redondeamos ambas **entropías** a 2 cifras significativas. En particular, obtenemos el mismo valor $H(S_{\text{Esp}}) = H(S_{\text{Eng}}) = 4{,}28 \pm 0{,}02$. Entonces, podemos verificar que $H(C) \leq L(C) < H(C) + 1$.

$$H(S_{\text{Esp}}) = 4{,}28 \pm 0{,}02 \leq L(S_{\text{Esp}}) = 4{,}30 < H(S_{\text{Esp}}) + 1 = 5{,}28 \pm 0{,}02$$

$$H(S_{\text{Eng}}) = 4{,}28 \pm 0{,}02 \leq L(S_{\text{Eng}}) = 4{,}30 < H(S_{\text{Eng}}) + 1 = 5{,}28 \pm 0{,}02$$

En la segunda parte de la práctica se pide codificar la palabra “Lorentz” usando los códigos anteriores. Usando S_{Esp}, la **codificación** es 1100110110010110010100011011100101 y usando S_{Eng}, 0000011001110-10100110111010111001 10, ambas con una longitud de 35 bits. Por otro lado, usando la **codificación binaria usual (ASCII)**, obtenemos 1001100110111111100101100101110111011101001111010, con una longitud de 49 bits, un 140 % mas larga. Esto prueba que es **más ineficiente** que usando Huffman.

Por último, para verificar nuestros resultados, **decodificamos** estas cadenas y comprobamos que obtenemos la palabra “Lorentz” en ambos idiomas.

4. Conclusión

Esta práctica ha confirmado la efectividad de los códigos de Huffman en la compresión de datos, al reducir la longitud de un texto mediante la asignación de códigos de longitud variable según la frecuencia de cada carácter. La aplicación del Primer Teorema de Shannon a la palabra “Lorentz” verificó que estos códigos son óptimos en términos de tamaño promedio, demostrando así su eficiencia en la representación de texto.

5. Anexo

5.1. Código binario Huffman

Char	C. Huffman esp	C. Huffman ing
a	000	1000
.	0010000	0100011
é	00100010	-
E	001000110	010001010
j	001000111	-
R	001001000	010001011
í	001001001	-
k	00100101	0000010
- (guión)	00100110	111000001
4	001001110	111000000
C	001001111	-
p	00101	111011
s	0011	1001
e	010	001
r	0110	0101
m	01110	01101
u	01111	010000
n	1000	1011
o	1001	0111
(espacio)	101	110
l	11000	00011
b	1100100	1110010
z	1100101	11100110
q	11001100	11100111
L	11001101	00000110
ó	1100111	-
i	1101	1111
d	11100	01100
t	11101	1010
c	11110	00001
f	1111100	00010
v	1111101	1110101
M	111111000	00000111
x	1111110010	010001000
P	1111110011	-
w	111111010	1110001
g	111111011	0000000
,	11111110	0000001
y	111111110	1110100
h	111111111	01001

Cuadro 1: Codificación Huffman para caracteres en español e inglés.

5.2. Código implementado

```
"""

    Asignatura: Geometría computacional
    Subgrupo: 1
    Curso: 2023-2024
    Alumno: Jiménez Poyatos, Pablo
    Curso: 4 CC
    Carrera: Grado en matemáicas.
    Práctica 1. Código de Huffman y Teorema de Shannon

"""

# Importamos el módulo os y la funcion log_2

from os import path
from math import log2,e,sqrt


# Creamos la clase Nodo y árbol de pares.

class Nodo:

    """
    Clase que representa un nodo en un árbol binario de pares.

    Atributos:
    - clave: Identificador único del nodo.
    - valor: Valor asociado al nodo.
    - iz: Referencia al hijo izquierdo del nodo.
    - dr: Referencia al hijo derecho del nodo.
    """

    def __init__(self, clave, valor):
        self.clave = clave
        self.valor = valor
        self.iz = None
        self.dr = None


class ArbolDePares:

    """
    Clase que implementa un árbol binario de búsqueda para pares clave-valor.

    Atributos:
    - raiz: Nodo raíz del árbol.

    Métodos:
    - insertar(clave, valor): Inserta un nodo con clave y valor en el árbol.
    - eliminar(clave, valor): Elimina el nodo con clave y valor del árbol.
    - encontrar_minimo(): Encuentra el nodo con el valor mínimo en el árbol.
    - nodos(): Devuelve la cantidad total de nodos en el árbol.
    - gen_dic_cod(): Genera un diccionario con el código binario de cada hoja.
    - inorden(): Imprime los nodos del árbol en orden inorden.
    """

    def __init__(self):
        self.raiz = None



    def insertar(self, clave, valor):
```

```
        """
        Inserta un nuevo nodo en el árbol.

        Parámetros:
        - clave (cualquier tipo comparable): La clave del nuevo nodo.
        - valor (cualquier tipo): El valor asociado al nuevo nodo.

        Descripción:
        Esta función inserta un nuevo nodo en el árbol binario de búsqueda.
        Si el árbol está vacío, el nuevo nodo se convierte en la raíz.
        Si el árbol no está vacío, la función busca la ubicación correcta para
        el nuevo nodo basándose en la comparación de claves y valores.
        """

        self.raiz = self._insertar(self.raiz, clave, valor)


    def _insertar(self, nodo, clave, valor):
        """ Funcion auxiliar de insertar"""

        if nodo is None:
            return Nodo(clave, valor)

        if valor[0] < nodo.valor[0]:
            nodo.iz = self._insertar(nodo.iz, clave, valor)
        elif valor[0] > nodo.valor[0]:
            nodo.dr = self._insertar(nodo.dr, clave, valor)
        else:
            # En caso de igualdad, compara el segundo elemento de la tupla
            if valor[1] < nodo.valor[1]:
                nodo.iz = self._insertar(nodo.iz, clave, valor)
            elif valor[1] > nodo.valor[1]:
                nodo.dr = self._insertar(nodo.dr, clave, valor)
            else:
                pass

        return nodo



    def eliminar(self, clave, valor):
        """
        Elimina un nodo con la clave y valor dados del árbol.

        Parámetros:
        - clave (cualquier tipo comparable): La clave del nodo a eliminar.
        - valor (cualquier tipo): El valor asociado al nodo a eliminar.

        Descripción:
        Esta función busca y elimina un nodo con la clave y valor dados del
        árbol binario de búsqueda. Si el nodo tiene dos hijos, se reemplaza
        por el nodo mínimo del subárbol derecho.
        """

        self.raiz = self._eliminar(self.raiz, clave, valor)


    def _eliminar(self, nodo, clave, valor):
        """ Funcion auxiliar de insertar"""
        if nodo is None:
            return None

        if valor[0] < nodo.valor[0]:
```

```
126             nodo.iz = self._eliminar(nodo.iz, clave, valor)
127         elif valor[0] > nodo.valor[0]:
128             nodo.dr = self._eliminar(nodo.dr, clave, valor)
129         else:
130             # En caso de igualdad, compara el segundo elemento de la tupla
131             if valor[1] < nodo.valor[1]:
132                 nodo.iz = self._eliminar(nodo.iz, clave, valor)
133             elif valor[1] > nodo.valor[1]:
134                 nodo.dr = self._eliminar(nodo.dr, clave, valor)
135             else:
136                 # Caso 1: Nodo sin hijos o con un solo hijo
137                 if nodo.iz is None:
138                     return nodo.dr
139                 elif nodo.dr is None:
140                     return nodo.iz
141 
142                 # Caso 2: Nodo con dos hijos
143                 # Encontrar el sucesor inorden (mínimo en el subárbol derecho)
144                 nodo_minimo = self._encontrar_minimo(nodo.dr)
145                 nodo.clave = nodo_minimo.clave
146                 nodo.valor = nodo_minimo.valor
147                 nodo.dr = self._eliminar(nodo.dr, nodo_minimo.clave)
148 
149         return nodo
150 
151 
152 
153     def encontrar_minimo(self):
154         """ Encuentra el nodo con el valor mínimo en el árbol."""
155         return self._encontrar_minimo(self.raiz)
156 
157 
158     def _encontrar_minimo(self, nodo):
159         """ Función auxiliar para encontrar el nodo mínimo"""
160 
161         while nodo.iz is not None:
162             nodo = nodo.iz
163         return nodo
164 
165 
166 
167     def nodos(self):
168         """ Devuelve la cantidad total de nodos en el árbol."""
169 
170         return self._nodos(self.raiz)
171 
172 
173     def _nodos(self, nodo):
174         """Función auxiliar de la funcion nodos"""
175 
176         if nodo is None:
177             return 0
178         else:
179             izq = self._nodos(nodo.iz)
180             dcha = self._nodos(nodo.dr)
181             nod = 1 + izq + dcha
182             return nod
183 
184 
185 
186     def gen_dic_cod(self):
187         """
188         Genera un diccionario con el código de cada hoja (caracter),
```

```
189         utilizando un recorrido inorden.
190         """
191         dic_cod = {}
192         self._gen_dic_cod(self.raiz, "", dic_cod)
193         return dic_cod
194 
195 
196     def _gen_dic_cod(self, nodo, codigo_actual, dic_cod):
197         """Función auxiliar de la funcion gen_dic_cod"""
198         if nodo is not None:
199             # Si el nodo es una hoja, agregar el código al diccionario
200             if nodo.iz is None and nodo.dr is None:
201                 dic_cod[nodo.clave] = codigo_actual
202 
203             # Recursión a la izquierda (0)
204             self._gen_dic_cod(nodo.iz, codigo_actual + "0", dic_cod)
205             # Recursión a la derecha (1)
206             self._gen_dic_cod(nodo.dr, codigo_actual + "1", dic_cod)
207 
208 
209 
210     def inorden(self):
211         """ Imprime los nodos del árbol en orden inorden."""
212         self._inorden(self.raiz)
213 
214 
215     def _inorden(self, nodo):
216         """Función auxiliar para el recorrido inorden."""
217         if nodo:
218             self._inorden(nodo.iz)
219             print(f"Clave: {nodo.clave}, Valor: {nodo.valor}")
220             self._inorden(nodo.dr)
221 
222 
223 
224 # Funciones para resolver la práctica 1.
225 
226 def es_ruta_de_archivo(s):
227     """
228     Verifica si la cadena s corresponde a una ruta de archivo válida.
229 
230     Parámetros:
231     - s (str): La cadena a verificar.
232 
233     Return:
234     - bool: Si s es una ruta válida o no.
235     """
236 
237     return path.isfile(s)
238 
239 
240 def leerFichero(nombre):
241     """
242     Lee el contenido de un archivo y lo devuelve como un string.
243 
244     Parámetros:
245     - nombre (str): El nombre del archivo a leer.
246 
247     Return:
248     str: El contenido del archivo.
249     """
250 
251     with open(nombre, 'r',encoding="utf8") as file:
```

```
            contenido = file.read()
    return contenido


def contenido(palabra):
    """
    Si la entrada es una ruta de archivo, devuelve el contenido de un archivo.
    De lo contrario, devuelve la misma cadena.

    Parámetros:
    - palabra (str): La ruta de archivo o cadena.

    Return:
    str: El contenido del archivo o la cadena original.
    """

    if es_ruta_de_archivo(palabra):
        contenido = leerFichero(palabra)
    else:
        contenido = palabra
    return contenido


def contar_frecuencias(cadena):
    """
    Cuenta las frecuencias y las primeras apariciones de los caracteres en una
    cadena.

    Parámetros:
    - cadena (str): La cadena de entrada.

    Return:
    tuple: Dos diccionarios, uno con las frecuencias y otro con las frecuencias
    y las primeras apariciones de cada caracter.
    """
    frec = {}
    primera_ap = {}
    contador = 0

    for carac in cadena:
        if carac in frec:
            frec[carac] += 1
        else:
            frec[carac] = 1
            primera_ap[carac] = contador

        contador += 1
    resultado = {carac: (frec[carac], primera_ap[carac]) for carac in frec}
    return resultado, frec


def arbol_frecuencias(dic):
    """
    Crea un árbol binario de busqueda de pares a partir de un diccionario.

    Parámetros:
    - dic (dict): El diccionario de frecuencias.

    Return:
    ArbolDePares: Un árbol binario de búsqueda de frecuencias.
    """

    arbol = ArbolDePares()
```

```
315     for clave in dic:
316         arbol.insertar(clave, dic[clave])
317     return arbol
318 
319 
320 def plantar(nodo, ab_izq, ab_dch):
321     """
322     Crea un nuevo árbol uniendo un nodo con dos subárboles.
323 
324     Parámetros:
325     - nodo (Nodo): El nodo raíz del nuevo árbol.
326     - ab_izq (ArbolDePares o None): Subárbol izquierdo.
327     - ab_dch (ArbolDePares o None): Subárbol derecho.
328 
329     Return:
330     ArbolDePares: El nuevo árbol.
331     """
332 
333     nuevo_arbol = ArbolDePares()
334     nuevo_arbol.raiz = nodo
335     if ab_izq is None:
336         nuevo_arbol.raiz.iz = None
337     else:
338         nuevo_arbol.raiz.iz = ab_izq.raiz
339 
340     if ab_dch is None:
341         nuevo_arbol.raiz.dr = None
342     else:
343         nuevo_arbol.raiz.dr = ab_dch.raiz
344     return nuevo_arbol
345 
346 
347 def add_dic_ab(min1,dic, iz, dr):
348     """
349     Agrega un nuevo árbol al diccionario.
350 
351     Parámetros:
352     - min1 (Nodo): Nodo del árbol a agregar.
353     - dic (dict): Diccionario de árboles.
354     - iz (ArbolDePares o None): Subárbol izquierdo.
355     - dr (ArbolDePares o None): Subárbol derecho.
356 
357     Return:
358     dict: Diccionario actualizado.
359     """
360 
361     if min1.clave not in dic:
362         newNodo = Nodo(min1.clave, min1.valor)
363         newArbol = plantar(newNodo, iz, dr)
364         raizD = newArbol.raiz.clave
365         dic[raizD] = newArbol
366     return dic
367 
368 
369 def dic_arboles(arbol):
370     """
371     Construye un diccionario de árboles a partir de un árbol original. Esta
372     funcion escoge los dos valores con frecuencias mas pequeñas, crea un
373     arbol con esos nodos y lo añade al diccionario. 'Idea detras del algoritmo
374     de Huffman'
375 
376     Parámetros:
377     - arbol (ArbolDePares): El árbol original.
```

```
    Return:
    dict: Diccionario de árboles.
    """

    dic = {}

    if arbol.nodos() == 1:
        min1 = arbol.encontrar_minimo()
        dic = add_dic_ab(min1, dic, None, None)

    else:
        while arbol.nodos() != 1:

            min1 = arbol.encontrar_minimo()
            arbol.eliminar(min1.clave, min1.valor)

            min2 = arbol.encontrar_minimo()
            arbol.eliminar(min2.clave, min2.valor)

            newchar = min1.clave + min2.clave
            newfrec = min1.valor[0] + min2.valor[0]
            newprior = min(min1.valor[1], min2.valor[1])
            newNodo = Nodo(newchar, (newfrec,newprior))
            arbol.insertar(newNodo.clave, newNodo.valor)


            dic = add_dic_ab(min1, dic, None, None)
            dic = add_dic_ab(min2, dic, None, None)
            dic = add_dic_ab(newNodo, dic, dic[min1.clave], dic[min2.clave])

    return dic


def crear_palabra(dic_cambio, codif_Hauf):
    """
    Crea una palabra a partir de una codificación Huffman.

    Parámetros:
    - dic_cambio (dict): Diccionario de cambio.
    - codif_Hauf (str): La codificación Huffman.

    Return:
    str: La palabra decodificada.
    """

    palabra = ""
    codif = ""
    for i in range(len(codif_Hauf)):
        car = codif_Hauf[i]
        codif += car
        if codif in dic_cambio:
            palabra += dic_cambio[codif]
            codif = ""
    return palabra

def codigo_Huffman(string):
    """
    Genera el diccionario de códigos Huffman y las frecuencias de caracteres.

    Parámetros:
    - string (str): Muestra con la que se crea el codigo de Huffman.

```

```
    Return:
    tuple: Un diccionario de códigos Huffman, otro con las frecuencias y otro
    con las frecuencias y la primera vez que aparece cada caracter.
    """

    texto = contenido(string)
    frec, frec_SR = contar_frecuencias(texto)
    arbol_frec = arbol_frecuencias(frec)
    diccionario = dic_arboles(arbol_frec)

    ultimo = list(diccionario.keys())[-1]
    arbol = diccionario[ultimo]
    dic_traduccion = arbol.gen_dic_cod()

    return dic_traduccion, frec, frec_SR


def codificar_palabra(texto, dic_traduccion):
    """
    Codifica una palabra utilizando un diccionario de traducción.

    Parámetros:
    - texto (str): La palabra a codificar.
    - dic_traduccion (dict): Diccionario de traducción de caracteres a códigos.

    Return:
    str: La palabra codificada.
    """

    codificacion = ""
    for i in texto:
        codificacion += dic_traduccion[i]

    return codificacion


def decodificacion(codif_Hauf,dic_trad):
    """
    Decodifica una cadena Huffman utilizando un diccionario de traducción.

    Parámetros:
    - codif_Hauf (str): La cadena codificada con Huffman.
    - dic_trad (dict): Diccionario de traducción caracteres a códigos.

    Return:
    str: La cadena decodificada.
    """

    # Cambio las claves y los valores del diccionario.
    dic_cambio = dict(zip(dic_trad.values(), dic_trad.keys()))
    palabra = crear_palabra(dic_cambio, codif_Hauf)

    return palabra


def longitudMedia_entropia(palabra, codigo):
    """
    Calcula la longitud media, la entropía y la longitud total y el error de
    un string utilizando un código dado.

    Parámetros:
    - palabra (str o dict): La palabra o diccionario de frecuencias.
    - codigo (dict): El código de cada caracter.
```

```
504
505         Return:
506         tuple: Una tupla con la longitud total, entropía, longitud media y error.
507         """
508
509         if isinstance(palabra, dict):
510             frec = palabra
511             num_caract = sum(list(frec.values()))
512         else:
513             frec_CR, frec = contar_frecuencias(palabra)
514             num_caract = len(palabra)
515
516         lista_claves = list(frec.keys())
517         longitud = 0
518         entropia = 0
519         error = 0
520
521         for i in lista_claves:
522             frecI = frec[i]
523             long = len(codigo[i])
524             longitud += frecI*long
525             prob = frecI/num_caract
526             error += (log2(e*prob))**2
527             entropia -= prob * log2(prob)
528
529         error = 1/(len(lista_claves))**2 * sqrt(error)
530         longitud_media = longitud / num_caract
531         return longitud, entropia, longitud_media, error
532
533
534 def comprobar_1TS(l,h):
535         """
536         Comprueba el Primer Teorema de Shanon.
537
538         Parámetros:
539         - l (float): Longitud media de un codigo binario usando Huffman.
540         - h (float): Entropía del sistema.
541
542         Return:
543         bool: Si se cumple el Primer Teorema de Shanon con esos valores.
544         """
545
546         return h <= l and l < h + 1
547
548
549 def longitudCBU(string):
550         """
551         Calcula la longitud de un string despues de usar la codificacion binaria
552         usual (ASCII).
553
554         Parámetros:
555         - string (str): La cadena de caracteres que pasamos a binario usual y
556                         medimos su longitud.
557
558         Return:
559         int: La longitud total en bits.
560         """
561
562         long = 0
563         codif = ''
564         for i in string:
565             binario = bin(ord(i))
566             long += len(binario) - 2
```

```
        codif += str(binario)[2:]
    return long, codif


def comprobarL(long,long_CBU):
    """
    Comprueba si la longitud total de la codificacion usando el codigo de
    Huffman es menor que usando el codigo binario usual.

    Parámetros:
    - long (int): La longitud de la cadena usando Huffman.
    - long_CBU (int): La longitud de la cadena usando binario usual.

    Return:
    bool: Si la longitud usando Huffman es menor que usando binario usual.
    """

    return long <= long_CBU



# Soluciones

if __name__ == "__main__":

    # Apartado 1

    path1 = 'GCOM2024_pract1_auxiliar_esp.txt'
    path2 = 'GCOM2024_pract1_auxiliar_eng.txt'

    S_Es, frecEs, frecEs_SR = codigo_Huffman(path1)
    S_En, frecEn, frecEn_SR = codigo_Huffman(path2)

    long_Es, h_Es, l_Es, er_Es = longitudMedia_entropia(frecEs_SR, S_Es)

    long_En, h_En, l_En, er_En = longitudMedia_entropia(frecEn_SR, S_En)


    comprobacionEs = comprobar_1TS(l_Es,h_Es)
    comprobacionEn = comprobar_1TS(l_En,h_En)


    print(f'i) El codigo Huffman binario español: S_Es = {S_Es}')
    print('')
    print(f'El codigo Huffman binario inglés: S_En = {S_En}')
    print('')


    print(f'La longitud media en L(S_Esp) es {l_Es}')
    print(f'La longitud media en L(S_Eng) es {l_En}')


    print('')
    print("Por último, comprobamos si se satisface el Primer Teorema de " +
          "Shannon. Para ello hay que comprobar que H(C) <= L(C) < H(C) + 1")
    print('')
    print(f"Para el sistema español: {comprobacionEs}. Tenemos que " +
          f"H(C) = {h_Es} <= L(C) = {l_Es} < H(C) + 1 = {h_Es + 1}.")
    print('')
    print(f"Para el sistema inglés: {comprobacionEn}. Tenemos que " +
          f"H(C) = {h_En} <= L(C) = {l_En} < H(C) + 1 = {h_En + 1}.")
    print('')
    print(f'El error del calculo de la entropia para S_Es es {er_Es}.' +
```

```
630             f'y para S_En, {er_En}.')
631
632
633     # Apartado 2
634
635     palabra = 'Lorentz'
636
637     codif_Es = codificar_palabra(palabra, S_Es)
638     long_Es, h_Es, l_Es, er_Es = longitudMedia_entropia(palabra, S_Es)
639     long_CBU_Es, codif_BU = longitudCBU(palabra)
640     comprobacion_Es = comprobarL(long_Es,long_CBU_Es)
641
642     codif_En = codificar_palabra(palabra, S_En)
643     long_En, h_En, l_En, er_En = longitudMedia_entropia(palabra, S_En)
644     long_CBU_En,codif_BU = longitudCBU(palabra)
645     comprobacion_En = comprobarL(long_En,long_CBU_En)
646
647
648     print('')
649     print('')
650     print(f"ii) La codificacion de la palabra {palabra} usando el código " +
651           "español obtenido en el apartado anterior es:")
652     print(f'S_Es : {codif_Es}')
653     print('')
654     print(f"La codificacion de la palabra {palabra} usando el código " +
655           "inglés obtenido en el apartado anterior es:")
656     print(f'S_En : {codif_En}')
657     print('')
658
659     print("Por último vamos a comprobar que es más eficiente que el código " +
660           "binario usual. Para ello comprobamos que la codificacion de " +
661           f"Lorentz usando el codigo binario usual (ASCII) es: {codif_BU}")
662     print('')
663     print(f'Codificación de Huffman español : {long_Es} bits')
664     print(f'Binario usual: {long_CBU_Es} bits')
665     print(f"¿Es más eficiente?: {comprobacion_Es}. Hay una diferencia de " +
666           f"{long_CBU_Es - long_Es} bits.")
667     print('')
668     print(f'Codificación de Huffman ingles : {long_En} bits')
669     print(f'Binario usual: {long_CBU_En} bits')
670     print(f"¿Es más eficiente?: {comprobacion_En}. Hay una diferencia de " +
671           f"{long_CBU_En - long_En} bits")
672
673
674     # Apartado 3
675
676     palabra_Es = decodificacion(codif_Es, S_Es)
677     palabra_En = decodificacion(codif_En, S_En)
678
679     print('')
680     print('')
681     print("iii) La decodificación de los códigos obtenidos en el " +
682           "apartado ii) son:")
683     print('En español, decodificamos:')
684     print(f'{codif_Es} y obtenemos {palabra_Es}')
685     print('En inglés, decodificamos:')
686     print(f'{codif_En} y obtenemos {palabra_En}')
```

Programa 1: Código de Huffman.py