



Escuela Superior de Cómputo.
Instituto Politécnico Nacional, México.



Práctica 9: Estrategia Greedy.

Blancas Pérez Bryan Israel
orionmunecaycanica@gmail.com

Resumen: Implementar un algoritmo basado en la estrategia Greedy. El algoritmo a implementar, será el "algoritmo de Huffman" para la construcción de códigos de Huffman", utilizado para la compresión de archivos.

Palabras Clave: Huffman, Código Huffman, Algoritmos Greedy, Complejidad Computacional.

1. Introducción

Algoritmos Greedy.

También conocidos como Algoritmos Voraces, los algoritmo Greedy, son aquellos que, para resolver un problema determinado, sigue un esquema de resolución consistente en elegir la solución óptima en cada paso local, con la esperanza de llegar a la solución general óptima. Gracias a esta forma de resolución del problema, los algoritmos greedy son algoritmos muy fáciles de diseñar. [1]

Es decir, contrario a la programación dinámica, un algoritmo voraz, nunca tomará en cuenta la decisión que se tomó previamente, a este tipo de algoritmos sólo les interesa elegir la mejor solución en el momento actual, o dicho de otra forma, sólo les interesa elegir la mejor opción en esa etapa de la resolución del problema.

2. Conceptos Básicos

Codificación Huffman

La codificación Huffman es un algoritmo utilizado para la compresión de datos. Fue desarrollado por David A. Huffman mientras estudiaba su doctorado en la MIT, en 1952. Esta codificación hace uso del algoritmo Huffman, el cual es un algoritmo voraz.[2]

Vista general del funcionamiento del algoritmo Huffman.

El algoritmo trabaja a partir de un conjunto de símbolos con sus respectivos pesos (repeticiones) dado. El algoritmo consiste en la creación de un árbol binario que tiene cada uno de los símbolos del alfabeto de entrada como hoja, y construido de manera que no sea ambiguo, para después asignar códigos de distinta longitud de bits a cada uno de los símbolos. Una cosa que vale la pena resaltar, es que mientras mas común (aparece con más frecuencia) sea el símbolo, menor es la cantidad de bits que se le asigna, de tal forma que el símbolo con más repeticiones, es también, el que menor cantidad de bits tiene asignado en la codificación.

Pasos del algoritmo:

1. A partir de la tabla de frecuencias de los símbolos, crear una lista con cada tupla de la tabla, y ordenar la lista crecientemente considerando la frecuencia del símbolo.
2. Convertir cada elemento de la lista en un árbol.
3. Fusionar todos los árboles en uno solo:
 - 3.1. Con los dos primeros árboles formar un nuevo árbol, cada uno de los árboles originales en una rama.
 - 3.2. Sumar las frecuencias de cada rama en el nuevo elemento árbol.
 - 3.3. Insertar el nuevo árbol en el lugar adecuado de la lista según la suma de frecuencias obtenida.
4. Para asignar el nuevo código binario de cada carácter sólo hay que seguir el camino adecuado a través del árbol. Si se toma una rama cero, se añade un cero al código, si se toma una rama uno, se añade un uno.
5. Se recodifica el fichero según los nuevos códigos.

Pasos para la decodificación del archivo:

1. Cargar la codificación a la memoria.
2. Empezar a leer bits del archivo binario.
3. Leer bits de uno en uno, y checar si la cadena de bits está en la codificación.
4. Si se encuentra la cadena de bits en la codificación, escribir el símbolo correspondiente en el archivo de salida.

3. Experimentación y Resultados

Ejercicio 1.

Implementar el algoritmo de la codificación de Huffman con las siguientes condiciones:

Para la codificación:

Entrada: Un archivo de texto con extensión .txt (original.txt) a codificar.

Salida: se generaran tres archivos .txt.

Frecuencias.txt: Mostrará la tabla de frecuencias de los caracteres que aparecen en el archivo original.txt.

Codificacion.txt: Mostrará los caracteres que aparecen en el archivo original.txt y la codificación que le corresponde.

Archivo_codificado.txt: Mostrará el archivo codificado.

Para la decodificación:

Entrada: **Archivo_codificado.txt** generado en la codificación y los archivos necesarios para su decodificación.

Salida: Mostrar la información decodificada (**archivo_decodificado.txt**).

Pseudocódigo del algoritmo:

```
1  #Funciones Auxiliares
2
3  def existLetter(letters , char):
4      k=0
5      for node in letters:
6          if (node[1]==char):
7              return k
8          k+=1
9      return -1
10
11 def getFrecuency(i):
12     letters=[]
13     k=0
14     for line in i:
15         for char in line:
16             k=existLetter(letters , char)
17             if (k==-1):
18                 letters.append([1 , char])
19             else:
20                 aux=letters[k]
21                 aux[0]+=1
22                 letters[k]=aux
23     return letters
24
25 #Algoritmo de Huffman
26
27 def encode(letters):
28     heap=[[frecuence , [char , '']] for frecuence , char in letters]
29     heapify(heap)
30
31     while (len(heap)>1):
32         x=heappop(heap)
33         y=heappop(heap)
34
35         for i in x[1:]:
36             i[1]='0'+i[1]
37         for i in y[1:]:
38             i[1]='1'+i[1]
39
40         heappush(heap , [x[0]+y[0]]+x[1:]+y[1:])
41
42     return heappop(heap)
```

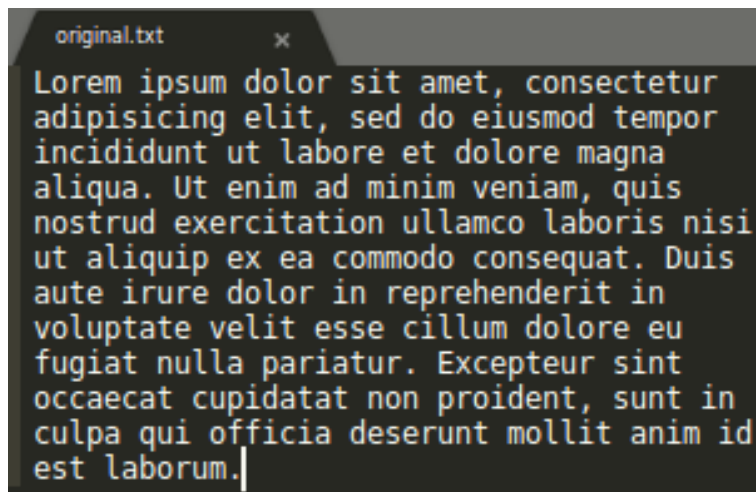
Función **getFrequency** retorna, en forma de lista, la frecuencia de los símbolos del archivo introducido; hace uso de la función **existLetter** para saber si un símbolo ya había sido tomado en cuenta, para así aumentarle el contador de la frecuencia, o por el contrario, agregar un nuevo elemento a la lista.

La función **encode**, es la implementación del algoritmo de Huffman, descrito en la sección anterior. Esta función realiza exactamente los mismos pasos que el algoritmo descrito con anterioridad. Los métodos utilizados, pertenecen a una librería llamada **heapq**, disponible en lenguaje python, la cual nos brinda una forma alternativa de implementar las estructuras de datos necesarias para el algoritmo (árbol binario), mediante una cola de prioridades.

Ejecución del Algoritmo.

Input.

La figura 1, muestra el contenido del archivo **original.txt**:

A screenshot of a text editor window titled 'original.txt'. The text inside is a Lorem Ipsum placeholder text, displayed in a monospaced font with syntax highlighting. The text is as follows:

```

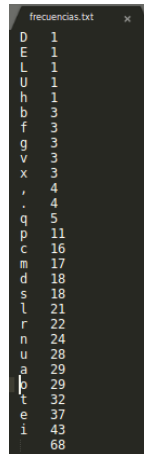
Lorem ipsum dolor sit amet, consectetur
adipisicing elit, sed do eiusmod tempor
incididunt ut labore et dolore magna
aliqua. Ut enim ad minim veniam, quis
nostrud exercitation ullamco laboris nisi
ut aliquip ex ea commodo consequat. Duis
aute irure dolor in reprehenderit in
voluptate velit esse cillum dolore eu
fugiat nulla pariatur. Excepteur sint
occaecat cupidatat non proident, sunt in
culpa qui officia deserunt mollit anim id
est laborum.

```

Figura 1: Original.txt.

Output.

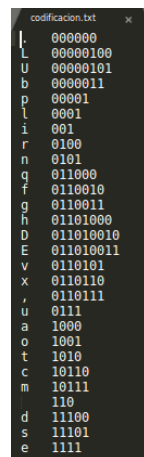
La figura 2, muestra la tabla de frecuencias generada por el algoritmo en el archivo **frecuencias.txt**:



D	1
E	1
L	1
U	1
h	1
b	3
f	3
g	3
v	3
x	3
,	4
.	4
q	5
p	11
c	16
m	17
d	18
s	18
l	21
r	22
n	24
u	28
a	29
o	29
t	32
e	37
i	43
	68

Figura 2: frecuencias.txt.

La figura 3, muestra la tabla de codificación generada por el algoritmo en el archivo **codificacion.txt**, cabe resaltar que como se muestra en la figura 2, los símbolos con más repeticiones (i, *espacio*), son los que tiene menor número de bits en su correspondiente código:



	000000
L	00000100
U	00000101
b	0000011
p	00001
l	0001
i	001
r	0100
n	0101
q	011000
f	0110010
g	0110011
h	01101000
D	011010010
E	011010011
v	0110101
x	0110110
,	0110111
u	0111
a	1000
o	1001
t	1010
c	10110
m	10111
	110
d	11100
s	11101
e	1111

Figura 3: codificacion.txt.

La figura 4, muestra la codificación generada por el algoritmo en el archivo `archivo_codificado.txt`:

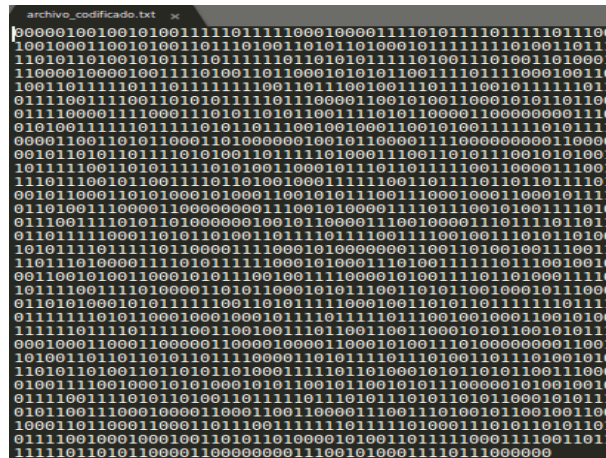


Figura 4: `archivo_codificado.txt`.

Cabe aclarar que, con motivos de visibilidad para el humano, el archivo resultante imprime ya sea '0' o '1', pero en formato ASCII, por lo que el archivo resultante es más *pesado* que el original. Ejemplo: en la tabla de codificación se muestra que el símbolo *i*, tiene un código asignado igual a '001', el cual representa únicamente 3 bits, menor a el byte necesario para expresar *i* en código ASCII, sin embargo, al momento de imprimir en el archivo **archivo_codificado.txt**, es necesario imprimir un bit en formato ASCII, para que sea legible al humano, lo que representa que cada bit terminará *pesando* un byte. Para arreglar eso, mediante programación he obtenido el peso de la compresión, contando el número de *bits* ('0','1') en el archivo resultante y haciendo la conversión a bytes. La figura 5, muestra lo anterior.

```
bryan@bryan-ubuntu:~/Documentos/Análisis de Algoritmos/practica9/codigo$ python huffman.py
Tam del archivo original: 446 bytes
Tam del archivo codificado: 231.25 bytes
```

Figura 5: Tamaño de la compresión.

Pseudocódigo del algoritmo de decodificación:

```
1 def decode(encode , i ) :
2     word=""
3     txt=""
4     for line in i:
5         for char in line:
6             word+=char
7             k=findEncode( encode , word)
8             if k!=-1:
9                 node=encode[k]
10                txt+=node[0]
11                word=""
12
13     return txt
```

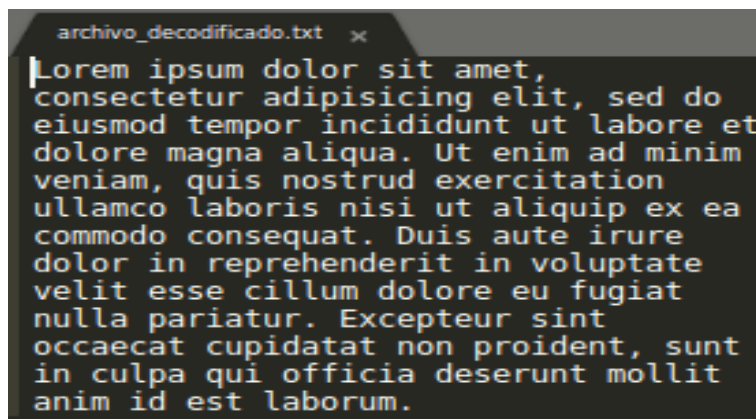
La función **decode**, decodifica el archivo original comparando cada cadena de bits, con la codificación cargada a partir del archivo **codificacion.txt**. El algoritmo va formando cadenas de bits hasta que alguna coincidencia sea encontrada en la tabla de codificación, de ser así, coloca el símbolo correspondiente en la cadena de retorno, la cual al final de la ejecución del algoritmo, contendrá el contenido del texto original.

Input.

Archivo **archivo_codificado.txt** y archivo **codificacion.txt**.

Output.

La figura 6, muestra el contenido del archivo **archivi_decodificado.txt**.



```
archivo_decodificado.txt x
Lorem ipsum dolor sit amet,
consectetur adipisicing elit, sed do
eiusmod tempor incididunt ut labore et
dolore magna aliqua. Ut enim ad minim
veniam, quis nostrud exercitation
ullamco laboris nisi ut aliquip ex ea
commodo consequat. Duis aute irure
dolor in reprehenderit in voluptate
velit esse cillum dolore eu fugiat
nulla pariatur. Excepteur sint
occaecat cupidatat non proident, sunt
in culpa qui officia deserunt mollit
anim id est laborum.
```

Figura 6: archivo_decodificado.txt.

4. Conclusiones

Esta práctica se me hizo muy interesante, sobretodo, porque la implementación del algoritmo de Huffman no es nada sencilla. Las estructuras de datos implementadas son un tanto complejas y el procedimiento del algoritmo tiene cierto grado de complejidad. Tuve problemas al momento de realizar la función del algoritmo de Huffman, pero lo pude solucionar implementando el módulo de python llamado *heapq*, el cual por medio de una cola de prioridad, me permitió emular los árboles binarios que se requieren para el algoritmo. Desde mi punto de vista, la parte más sencilla de la práctica fue decodificar el archivo, ya que la codificación hecha previamente no permite la ambigüedad en la compresión.

Sin duda los algoritmo Greedy son una gran herramienta para poder alcanzar la solución óptima de un problema, sin embargo, no siempre se pueden aplicar, ya que hay problemas en los que es necesario buscar un enfoque dinámico para poder dar con la solución óptima.

5. Bibliografía

- [1] https://es.wikipedia.org/wiki/Algoritmo_voraz
- [2] https://es.wikipedia.org/wiki/Codificación_Huffman