(DES)CODIFICACIÓN DE HUFFMAN

Algoritmo de Huffman

César Hernández Rodríguez
Universidad de Valladolid

1 CONTENIDO

2 INTRODUCCIÓN

Para presentar el código de Huffman primero hay que saber quién es Huffman. David A. Huffman fue un pionero en el campo de la informática, estudió ingeniería eléctrica y obtuvo doctorado en el MIT donde estuvo también en el cuerpo docente, fundó el departamento de ciencia informática. Contribuyó al área de las tecnologías incluyendo la teoría de información y codificación y creó el ya mencionado código Huffman o codificación Huffman mientras era estudiante del doctorado en el MIT.

La codificación Huffman es un esquema de compresión para la codificación de textos variables sin pérdidas, es un algoritmo voraz. Un algoritmo voraz (*greedy algorithm)* es una estrategia de búsqueda sencilla y eficaz de encontrar una solución óptima a un problema.

3 IDEA PRINCIPAL DEL PROBLEMA

La idea es encontrar una técnica para la compresión de datos usando una tabla de códigos de longitud variable para codificar cada símbolo dentro de los datos. Este código se asignará dependiendo de la frecuencia con la que aparezca el símbolo dentro de los datos dando símbolos con menos bits a los símbolos que aparezcan más veces en los datos y así ocupar menos. La idea es poder crear un diccionario por cada entrada de texto donde las letras que

aparezcan con mayor frecuencia dentro de ese texto tenga una longitud en bits más corta que la que tendrían si no codificamos.

Carácter	Frecuencia	Código
Espacio	8	00
E	6	100
N	3	1100
O	3	1110
U	3	0100
A	2	0101
D	2	1010
F	2	1011
L	2	0110
M	2	0111
S	2	11010
B	1	110110
H	1	110111
J	1	111100
P	1	111101
R	1	111110
T	1	111111

Tabla 1. Ejemplo de codificación con el texto "ESTO ES UN EJEMPLO DE UN ARBOL DE HUFFMAN"

Esta codificación da una representación única a cada símbolo, dando lugar a un código prefijo (cada cadena de bits es particular y no es prefijo de otro símbolo), siendo el método de compresión más eficiente de este tipo ya que consigue la salida más pequeña, es la que requiere el mínimo número medio de bits por símbolo. Un ejemplo sería (según la Tabla 1) para codificar por el lenguaje ASCII una E se necesitarían 8 bits (1000101) es decir, un byte, mientras que después de realizar la codificación cada E estaría determinada por 3 bits (100). Por lo que con este planteamiento tienes una notable mejora por cada E que se encuentre en el texto, al ser una cadena tan corta esta E aparecerá muchas veces ya que tendrá una frecuencia alta y la ganancia será considerable.

4 CÓMO FUNCIONA EL ALGORITMO

Para explicar cómo funciona el algoritmo usaré unas imágenes de la UCP (Universidad Politécnica de Cataluña).

El algoritmo se basa en:

1. Sacar las frecuencias de cada símbolo.
2. Construir un álbum binario con los símbolos, basado en las frecuencias.
3. Asignar el código a cada símbolo
4. Codificar el texto
5. Decodificar el texto

Para empezar, tenemos un fichero ene l que aparecen los caracteres *'a'*, *'b'*, *'c'*, *'d'*, *'e'*, *'f'*, repetidos un total de 100000 lo que nos da la siguiente tabla de frecuencias:

Caracteres	a	b	c	d	e	f
Frecuencia(en miles)	45	13	12	16	9	5

Tabla 2.- Frecuencias de cada carácter en el fichero con 100000 caracteres.

La idea es poder reducir el espacio que ocupa cada símbolo utilizando un código binario. La idea de Huffman es utilizar una longitud variable para cada carácter que va a depender de la frecuencia que tengan, cuanta más frecuencia menor será la longitud. Hay que tener en cuenta que ningún código ha de ser prefijo de otro.

Caracteres	a	b	c	d	e	f
Código	0	101	100	111	1101	1100

Tabla 3.- Codificación por el algoritmo de Huffman

Codificación: aabacd = 0·0·101·0·100·111 = 001010100111

Como observamos al ver los códigos no hay ningún código que se repite ni ningún código completo es el inicio de otro código.

Primero ordenamos los nodos según las frecuencias de menor a mayor:

Caracteres	f	e	c	b	d	a
Frecuencia(en miles)	5	9	12	13	16	45

Tabla 4.- Frecuencias de cada carácter en el fichero con 100000 caracteres ordenadas.

La forma de sacar esta codificación es gracias a un árbol binario. Este árbol binario se saca juntando los nodos con menor frecuencia dando como resultado un nuevo nodo intermedio con la frecuencia de ambos nodos sumada, para poder sacar los nodos deseados ordenadores en cada iteración la lista por las frecuencias y así sacas siempre los nodos con frecuencias más pequeñas. Para nuestro ejemplo, las frecuencias más pequeñas son *e* y *f*, la primera iteración en nuestro árbol sería:

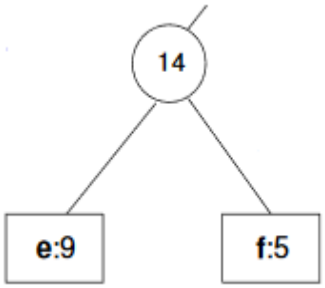

Ilustración 1.- Primer paso del árbol binario

Este proceso se seguiría añadiendo el nodo intermedio con frecuencia 14 a la lista de nodos, quedando una lista del estilo de:

Caracteres	c	b	n.i.	d	a
Frecuencia(en miles)	12	13	14	16	45

Tabla 5.- Frecuencias de cada carácter en el fichero con 100000 caracteres ordenadas tras la primera iteración.

Si seguimos este proceso hasta que solo nos quede un nodo en la lista, el árbol final para nuestro ejemplo es:

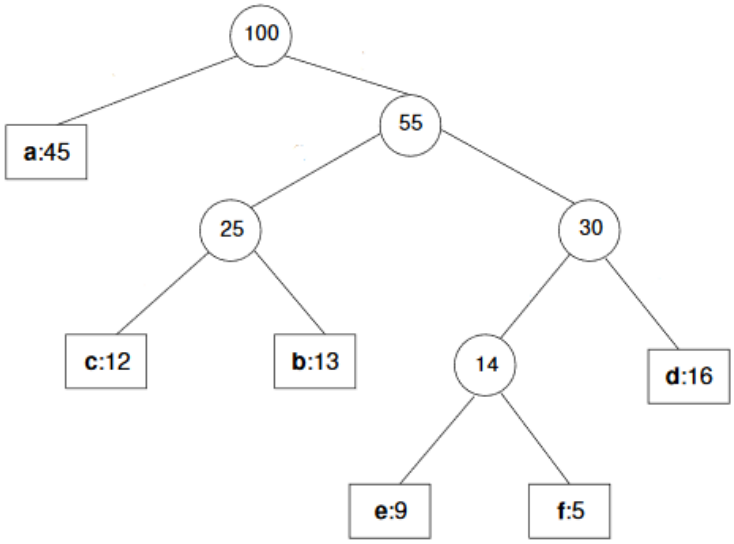

Ilustración 2.- Árbol binario completo del ejemplo

Este árbol, el nodo que nos queda en la lista tiene guardado los caminos hacia cualquier nodo del árbol.

Para terminar de completar este árbol hay que poner en cada arista del árbol un 0 o un 1 para poder codificar cada nodo, para ello haremos que el hijo de la derecha se le asignará el 0 y al hijo derecho se le asignará el 1. El árbol terminado nos queda así:

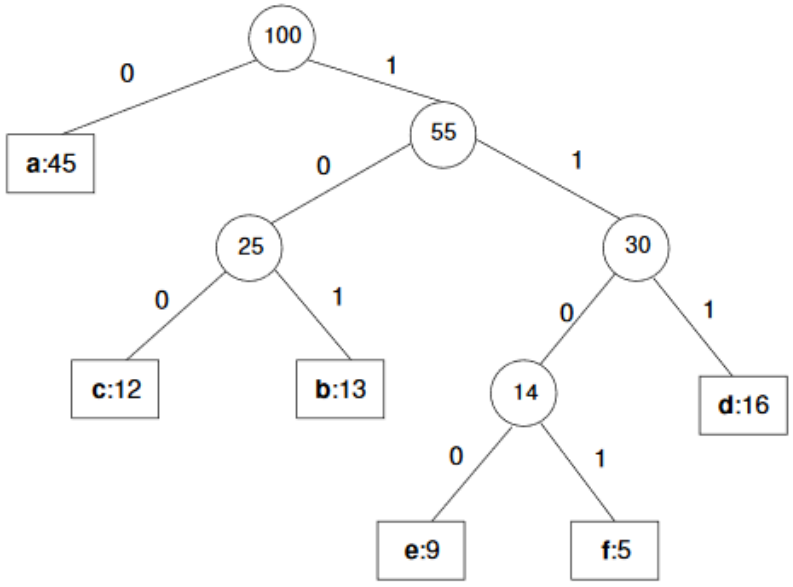

Ilustración 3.- Árbol completo terminado con la codificación de cada nodo

Si llamamos a este árbol binario T que corresponde prefijo entonces el número de bits necesarios para codificar un fichero, B(T). Para cada carácter *c* diferente del conjunto C que son los caracteres que aparecen dentro del fichero:

- F(c) es la frecuencia de *c* en la entrada.
- $d_t(c)$ es la profundidad de la hoja *c* en el árbol T.

El número de bits requeridos es: $B(T) = \sum_{c \in C} f(c) \cdot d_T(c)$

El algoritmo de Huffman obtiene una codificación prefijo óptima, es decir, consigue el min(B(T)).

Según los pasos expuestos antes para terminar con este algoritmo faltaría codificar el texto con los códigos sacados del árbol binario, para nuestro caso tendríamos como solución al problema la codificación expuesta en la Ilustración 3 que daría como resultado la Tabla 3.

Luego habría que transcribir el texto inicial con esa codificación, esta parte se haría cogiendo cada letra del texto y escribirla con al código asignado dando como resultado una retahíla de 1s y 0s que estarán en el mismo orden que el texto inicial.

Por último se puede querer descodificar el texto codificado, para ello al ser una codificación prefijo habría que ir buscando si una cadena del inicio del texto codificado coincide con alguna codificación de los caracteres hecha anteriormente, si coincide, extraemos esa parte del texto y la traducimos a carácter y volvemos a repetir este paso hasta que no quede texto codificado.

5 PSEUDOCÓDIGO

/*Sacamos de un texto las frecuencias de cada carácter y cada carácter para obtener un listado de parejas del estilo (frecuencia, 'carácter') que estarán ordenadas de menor a mayor frecuencia*/

```
Huffman(Texto T)        Entrada: n letras
        Por cada letra de T hacer
                prob[letra] = 0                          } O(n)
        Por cada letra de T hacer
                prob[letra] = prob[letra] +1             } O(n)
        ordenar(prob)   Timsort
```

/*Construimos un árbol binario y se van cogiendo los nodos de 2 en 2 que tengan las frecuencias más pequeñas y se juntan en un nuevo nodo intermedio, añadiendo este nodo nuevo creado a la lista y volviendo a ordenar. Esto nos devuelve el listado de parejas de los caracteres (sin las frecuencias) en forma de árbol*/

```
        Mientras longitud(lista) sea más grande que 1 hacer
                primeros = coger_dos_primeros(lista)
                sumar_frec = primeros[0] + primeros [1]
                añadir(lista,(suma_frec,primeros)
                ordenar(lista)   Timsort
```

} O(cte)

/*Cogemos la lista con los caracteres en forma de árbol y creamos una lista del estilo lista[carácter] = código), se necesita definir una función ya que es una función recursiva*/

```
Función asignar_codigos(lista_nodos, camino)
        Si nodo es nodo_hoja
                codigo[nodo] = (camino)
        sino
                Asignar_codigos(lista_nodo[0], camino+"0")
```

} O(cte)

```
        Asignar_codigos(lista_nodo[1], camino+"1")
    Return codigo
```

/*Para codificar el texto inicial con los códigos sacados antes cogemos el texto y vamos carácter a carácter cambiando cada carácter por su código*/

```
texto_cod = ""
para letra en T hacer
        aux = codigo[letra]
        texto_cod = texto_cod + aux
```

} O(n)

/*El paso de decodificar, vamos comprobando si texto_cod empieza por un código de los guardados en el diccionario, si empieza, cambiamos el codigo por el carácter enlazado y borramos esa parte del texto_cod, este bucle se repite hasta que la palabra texto_cod no le queden números. */

Desde longitud de texto_cod sea más grande de 1 hacer

Para cada código en diccionario hacer

Si texto_cod empieza por código entonces

texto = texto + código

} O(cte)

} O(n)

El coste de estos bucles anidados es O(n)

6 CONSTRUCCIONES FUNDAMENTALES

El código está hecho en Python 3.0.

Función *get_frecuecnias(str),* le pasamos la cadena de texto que queramos y obtenemos un diccionario con todos los caracteres que tenga el texto y las frecuencias de cada carácter. Usamos la función del módulo collections, Counter, esta función nos devuelve las veces que se repite cada carácter de un string.

```
def get_frecuencias(cadena):
    prob = {}
    for letra in cadena:
        prob[letra] = 0
    for letra in cadena:
        prob[letra] += 1
    return prob
```

Función *ordenarFreq (diccionario),* le pasamos el diccionario del método anterior y este nos devuelve un listado de tuplas del tipo (freq,'carácter') ordenador por frecuencias de menor a mayor.

```
def ordenarFrec (dict) :
    letra = dict.keys()
    tuplas = []
    for let in letra:
        tuplas.append((dict[let],let))
    tuplas.sort()
    return tuplas
```

Función *const_arbol(lista*), le pasamos la lista sacada del método anterior y nos devuelve una lista de tuplas que nos da la información de cómo está construido el árbol, que nodos se han juntado y cuales son hijos derechos e izquierdos.

```
def const_arbol(lista) :
    while len(lista) > 1:
        primeros = tuple(lista[0:2])
        resto  = lista[2:]
        combFrec = primeros[0][0] + primeros[1][0]
        lista  = resto + [(combFrec,primeros)]
        lista.sort(key=primero)
    return lista[0]
```

Función *asginar_codes(lista*), le pasamos la lista de nodos (sin las frecuencias, solo los caracteres), esta función es una función recursiva donde se va dando el código a cada nodo, devolviendo un diccionario del estilo ('carácter' : código), para poder recorrer el árbol la función es recursiva mientras se va guardando el recorrido que seguimos para saber cuántos 0s o 1s tiene cada nodo y en qué orden.

```
def asignar_codes (nodo, camino='') :
    if type(nodo) == type(""):
        codes[nodo] = camino
    else:
        asignar_codes(nodo[0], camino+"0")
        asignar_codes(nodo[1], camino+"1")
    return codes
```

Función *codificar(diccionario,texto)* para codificar necesitamos el texto que queremos codificar y el diccionario donde tenemos guardados los códigos de codificación, cada letra del texto se va codificando una a una y se juntan en el mismo orden. Devolvemos un string con el número codificado.

```
def codificar(dic,texto):
    res = ""
    for let in texto:
        codigo = dic[let]
        res = res + codigo
    print("tuuu", res)
    return res
```

La parte de decodificar el texto es la misma que codificar invirtiendo el diccionario. Las partes son las mismas, para este caso usaremos la función startswith(k) predefinida en los strings que busca la cadena k en el inicio del texto en el que se aplica.

```
def huffmanDecode (dictionary, text):
    res = ""
    while len(text)>1:
        for k in dictionary:
            if text.startswith(k):
                res += dictionary[k]
                text = text[len(k):]
    return res
```

7 COSTE

El coste total de este algoritmo como se puede ver en el pseudocódigo es O(n), siendo n el tamaño de entrada del texto que vayamos a codificar. Este estudio del coste hay que comentar que hay varios bucles señalados en la parte del pseudocódigo que tienen un coste constante y hay que señalar esta parte ya que el coste de esos buces son O(m), siendo m el número de caracteres diferentes que tenga el texto, si nos basamos en el abecedario español tenemos 27 letras más los signos de puntuación, más el espacio, el salto de línea o posibles acentos este número no va a pasar de 40 por lo que m no es una función asintótica como puede es n por lo que el coste de esas partes puede ser como mucho 40 por lo que es constante.

Comentar que una de las partes que podría traernos problemas es el ordenar a la hora de crear el árbol binario, Python, el lenguaje utilizado para construir este código usa el algoritmo Timsort, que tiene un coste de O(m log(m)), pero vuelve volviendo al párrafo anterior este coste sigue siendo constante

El coste total del algoritmo es **O(n).**

8 REFERENCIAS

https://es.wikipedia.org/wiki/Codificaci%C3%B3n_Huffman

http://bitybyte.github.io/Huffman-coding/

http://bitybyte.github.io/Descomprimiendo-datos-Huffman/

http://ramon-gzz.blogspot.com/2013/04/codificacion-huffman.html

https://www.geeksforgeeks.org/huffman-coding-greedy-algo-3/

https://www.geeksforgeeks.org/efficient-huffman-coding-for-sorted-input-greedy-algo-4/

https://www.xatakaciencia.com/matematicas/algoritmo-de-huffman

https://es.slideshare.net/ChihchengYuan/huff-48039244

https://docs.python.org/3/library/heapq.html

http://pepgonzalez.blogspot.com/2013/04/clase-metodos-codificacion-compresion.html

http://www.cs.upc.edu/~mabad/ADA/curso0708/GREEDY.pdf