



## Trabajo Práctico 1

### CODIFICACIÓN DE HUFFMAN

En este trabajo práctico vamos a implementar un *compresor* de datos. El mismo va a funcionar para cualquier tipo de archivo y es *sin pérdida*, es decir, desde el archivo comprimido podemos reconstruir exactamente el original (a diferencia de la compresión *con pérdida*, por ejemplo el formato JPG para imágenes, que puede perder algunos detalles).

## 1. Introducción

El compresor se basa en la codificación de Huffman. La idea de la misma es analizar la frecuencia con la que aparecen los caracteres en el texto de entrada, y optimizar a aquellos que son más frecuentes. Mediante un algoritmo (que detallamos adelante) se le asigna una secuencia de bits a cada caracter, con las siguientes propiedades:

- Los caracteres que aparecen más frecuentemente tienen secuencias más cortas. Esto es lo que nos hace ahorrar espacio.
- Ninguna secuencia es prefijo de otra, es decir, tenemos lo que se llama un *código de prefijos*. Esto es necesario para que, desde el archivo comprimido, podamos identificar exactamente los caracteres representados en el mismo.

Veamos un pequeño ejemplo. Supongamos que sólo existen los caracteres 'A', 'B', 'C' y 'D' y que nuestra entrada es la cadena. Una posible codificación, similar a la codificación ASCII, es asignarle a cada una una secuencia de 2 bits. Por ejemplo, es la siguiente:

$$A \mapsto 00 \quad B \mapsto 01 \quad C \mapsto 10 \quad D \mapsto 11$$

Codificar es tan fácil como reemplazar cada caracter por su código:

BBBAABDBBABCAB  
↓  
01 01 01 00 00 01 11 01 01 00 01 10 00 01  
↓  
0101010000011101010001100001

Luego, podemos fácilmente recuperar el texto original tomando de a 2 bits.

0101010000011101010001100001  
↓  
01 01 01 00 00 01 11 01 01 00 01 10 00 01  
↓  
BBBAABDBBABCAB

Esto es una codificación *de ancho fijo*. Podemos, en cambio, usar una codificación *de ancho variable*, e intentar usar códigos más chicos para símbolos más frecuentes, y entonces ahorrar espacio. Por ejemplo, una codificación óptima para el ejemplo anterior resulta ser:

$$'A' \mapsto 01 \quad 'B' \mapsto 1 \quad 'C' \mapsto 001 \quad 'D' \mapsto 000$$

Para comprimir el texto, reemplazamos cada caracter por su código:

BBBAABDBBABCAB  
↓  
1 1 1 01 01 1 000 1 1 01 1 001 01 1  
↓  
1110101100011011001011

Esta cadena de bits tiene longitud 22, mientras que la anterior tenía una longitud de 28. Para recuperar el texto original desde esta secuencia, buscamos repetidamente con cuál caracter empieza. Por ejemplo, el primer caracter debe ser una ‘B’, ya que es único caracter cuya codificación comienza con 1.

Aquí se ve por qué tenemos que evitar que la codificación de un caracter sea prefijo de alguna otra: en ese caso no podríamos separar la cadena (ej. si tuviéramos que 0 y 00 son códigos de caracteres distintos, no podríamos distinguir el primer símbolo).

1110101100011011001011  
↓  
1 1 1 01 01 1 000 1 1 01 1 001 01 1  
↓  
BBBAABDBBABCAB

## 2. El Algoritmo de Huffman

El primer problema consiste en *cómo* conseguir una buena codificación de prefijos. Es fácil encontrar *una* codificación de prefijos para cualquier cantidad de símbolos, simplemente podemos asignarles los códigos 0, 10, 110, 1110, ..., en sucesión. Es fácil ver que ninguno es prefijo de otro. Incluso podemos hacerlo en orden para asignarle el 0 al símbolo más frecuente, y así en sucesión. Sin embargo, esto no es necesariamente una buena idea. Si nuestra cadena original fuera ‘AABBCCDD’, la codificación de ancho fijo usaría 16 bits, mientras que siguiendo esta idea tenemos la codificación<sup>1</sup>:

‘A’  $\mapsto$  0      ‘B’  $\mapsto$  10      ‘C’  $\mapsto$  110      ‘D’  $\mapsto$  111

Con esa codificación, el resultado de codificar es:

AABBCCDD  
↓  
0 0 10 10 110 110 111 111  
↓  
001010110110111111

con una longitud total de 18 bits, mayor que la original, y las pérdidas son cada vez mayores a medida que agregamos símbolos.

---

<sup>1</sup>Incluso hacemos una pequeña optimización y a ‘D’ le asignamos el 111 en vez del 1110. Siempre podemos hacer esto con el último símbolo.

Entonces, tenemos que encontrar una *buena* codificación de prefijos. David Huffman encontró en 1952 un algoritmo que garantiza conseguir la codificación *óptima*<sup>2</sup> [Huf52]. El algoritmo consiste en:


- Computar las frecuencias de cada símbolo. Es decir, cuántas veces aparece cada símbolo en la entrada.
- Crear, para cada símbolo  $s$  de frecuencia  $w$ , un árbol binario consistiendo de una única hoja  $s$ . Se le asigna a este árbol el “peso”  $w$ .
- Mientras haya más de un árbol, tomamos dos de menor peso y los unimos en uno solo mediante un nuevo nodo. El peso del nuevo árbol es la suma de los dos pesos originales.

Veamos un ejemplo, partiendo de la frase “licenciatura en ciencias de la computacion” (tomando al espacio ‘ ’ como un símbolo). Las frecuencias son:

(‘ ’, 5) (‘a’, 5) (‘c’, 6) (‘d’, 1) (‘e’, 4) (‘i’, 5) (‘l’, 2) (‘m’, 1)  
(‘n’, 4) (‘o’, 2) (‘p’, 1) (‘r’, 1) (‘s’, 1) (‘t’, 2) (‘u’, 2)



El algoritmo comienza entonces tomando dos árboles de peso mínimo, por ejemplo los correspondientes a ‘s’ y ‘r’. Los unimos en un árbol de peso 2.

(‘ ’, 5) (‘a’, 5) (‘c’, 6) (‘d’, 1) (‘e’, 4) (‘i’, 5) (‘l’, 2) (‘m’, 1)  
(‘n’, 4) (‘o’, 2) (‘p’, 1) (‘s’ ‘r’, 2) (‘t’, 2) (‘u’, 2)




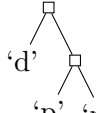
Seguimos uniendo, por ejemplo a ‘p’ y ‘m’.

(‘ ’, 5) (‘a’, 5) (‘c’, 6) (‘d’, 1) (‘e’, 4) (‘i’, 5) (‘l’, 2) (‘p’ ‘m’, 2)  
(‘n’, 4) (‘o’, 2) (‘s’ ‘r’, 2) (‘t’, 2) (‘u’, 2)



Ahora, unimos ‘d’ con el árbol de ‘p’ y ‘m’.

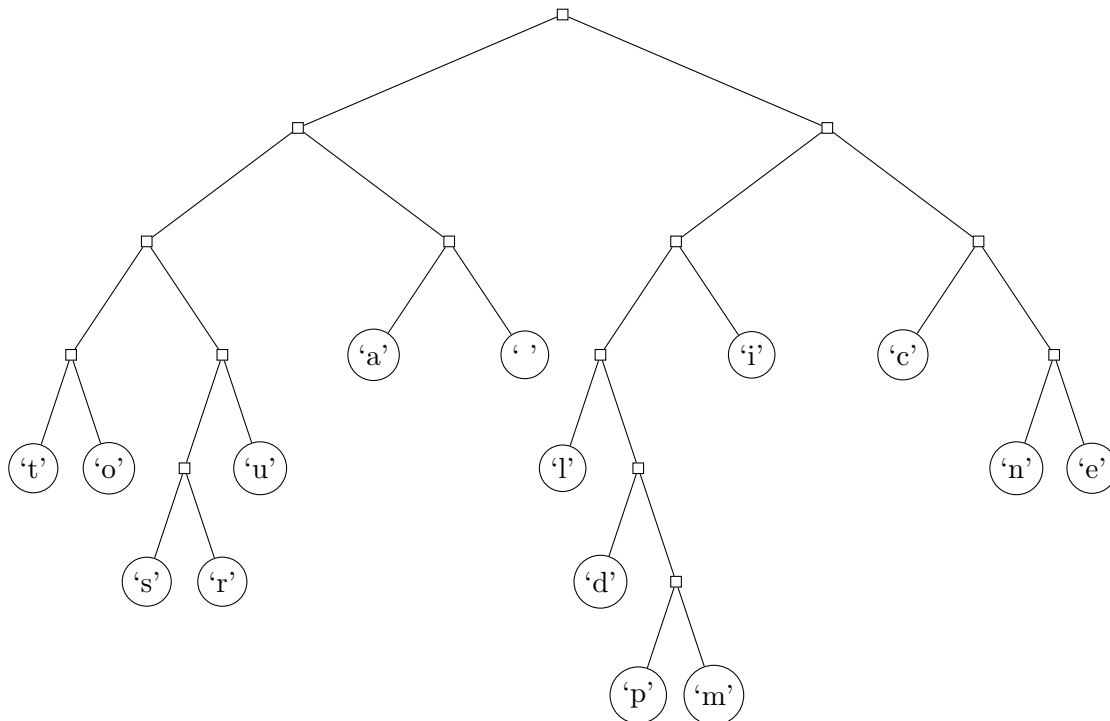
(‘ ’, 5) (‘a’, 5) (‘c’, 6) (‘e’, 4) (‘i’, 5) (‘l’, 2) (‘d’ ‘p’ ‘m’, 3)  
(‘n’, 4) (‘o’, 2) (‘s’ ‘r’, 3) (‘t’, 2) (‘u’, 2)



Seguimos este procedimiento hasta tener un sólo árbol. Una posibilidad es terminar con el árbol siguiente:

---

<sup>2</sup>Siempre que hablemos de codificaciones símbolo-a-símbolo. Otros códigos o métodos de compresión (como los que usa **gzip**) siguen otras ideas y pueden lograr mejores resultados.



'c'	$\mapsto$	011	'a'	$\mapsto$	010	'c'	$\mapsto$	110
'd'	$\mapsto$	10010	'e'	$\mapsto$	1111	'i'	$\mapsto$	101
'l'	$\mapsto$	1000	'm'	$\mapsto$	100111	'n'	$\mapsto$	1110
'o'	$\mapsto$	0001	'p'	$\mapsto$	100110	'r'	$\mapsto$	00101
's'	$\mapsto$	00100	't'	$\mapsto$	0000	'u'	$\mapsto$	0011

## 4. Descompresión

Para descomprimir, partimos desde una secuencia de bits, y tenemos que encontrar los símbolos que la generan. Iterativamente, tenemos que traducir algún prefijo (de longitud desconocida) de la secuencia a un símbolo. Para hacer esto, nos paramos en la raíz del árbol y seguimos el siguiente procedimiento:

- Si el primer bit es un cero, nos movemos hacia el hijo izquierdo. Si no, nos movemos hacia el hijo derecho.
- Avanzamos en la cadena (es decir “consumimos” el bit).
- Si llegamos a una hoja del árbol, ese es el símbolo que buscamos.
- Si no llegamos a una hoja, repetimos.

Por ejemplo, si la cadena de bits es 110100... y seguimos el árbol de la Sección 2, encontramos que el primer caracter es ‘c’ (110), y nos queda todavía 100.. como el resto de la entrada.

Al terminar ese procedimiento, habremos decodificado un símbolo de la cadena. Repetimos esto hasta haber consumido toda la cadena de bits y terminamos.

## 5. Serializando el Árbol

Nos falta un ingrediente para implementar el compresor/descompresor: tenemos que comunicar de alguna manera cuál es la codificación. Si el compresor escribe solamente un archivo con la codificación símbolo-a-símbolo, el descompresor no tiene manera de saber cuales eran los símbolos originales.

Necesitamos entonces una forma de guardar el árbol en un archivo para que pueda ser leído por el descompresor.

### 5.1. Serializando la forma

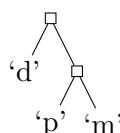
Hay muchas formas de lograr esto, para distintos tipos de árboles. En nuestro caso, tenemos árboles algo particulares, ya que:

- No tenemos valores en los nodos, sólo los tenemos en las hojas.
- Cada nodo tiene 0 hijos (las hojas), o 2 (el resto). Ningún nodo tiene 1 hijo. Se dice que el árbol está *lleno* (*full*).

Esto nos permite tener una codificación relativamente simple. Primero, nos preocupamos por codificar la *forma* del árbol, luego pensaremos en los valores que contienen las hojas. Seguiremos el siguiente procedimiento para codificar la forma de un árbol  $T$ .

- Si  $T$  es una hoja, imprimimos 1.
- Si  $T$  es un nodo, imprimimos 0, luego la codificación de su hijo izquierdo, y luego la codificación de su hijo derecho.

Con este procedimiento, la forma del árbol

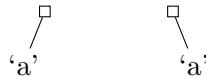


se codifica como 01011.

Para *parsear* un árbol (generarlo a partir de su texto), seguimos el siguiente procedimiento:

- Consumimos un caracter de la entrada
- Si es un 1, el árbol es una hoja, terminamos.
- Si es un 0, el árbol es un nodo. Mediante dos llamados recursivos parseamos el subárbol izquierdo y el subárbol derecho, y armamos el árbol completo con ellos.

¡Notar que esta codificación sólo sirve para árboles llenos! Los dos árboles siguientes (no llenos) no pueden distinguirse de esta manera.



## 5.2. Serializando los valores de las hojas

Una vez que serializamos la forma del árbol, debemos representar los valores de cada hoja. Dado que cada valor ocupa exactamente 1 byte (son caracteres) y que conocemos la cantidad de hojas (exactamente 256), podemos simplemente guardarlos (*en orden*) uno después del otro, inmediatamente después de la serialización de la forma del árbol.

Para reconstruir el árbol, primero creamos un árbol de la forma correcta sin ningún valor en sus hojas. Luego, hacemos un recorrido en orden consumiendo caracteres del archivo y guardándolos en las hojas.

Este será el contenido del archivo `.tree` que usa el descompresor.

## 6. Consignas y Cuestiones de Implementación

Implemente un compresor/descompresor siguiendo los procedimientos indicados arriba. El mismo debe poder ser corrido como:

```
$ huff C f.txt
```

Para comprimir al archivo `f.txt`, generando los archivos `f.txt.hf` con el texto comprimido y un archivo `f.txt.tree` con la serialización del árbol. Luego, se debe poder correr

```
$ huff D f.txt.hf
```

Para generar al archivo `f.txt.dec` con resultado de la descompresión (leyendo `f.txt.hf` y `f.txt.tree`).

Recuerde seguir las convenciones para el código estipuladas en comunidades ([link](#))

Tiene libertad para elegir las estructuras de datos internas. Su programa debería poder descomprimir cualquier texto comprimido y conseguir *exactamente* el archivo original.

No es necesario que el compresor efectivamente *disminuya* el tamaño de todo archivo al comprimir (incluso sin considerar al árbol). De hecho, es fácil de demostrar que esto es imposible para cualquier compresor sin pérdida. Sin embargo, sí debería funcionar bien en archivos con mucha redundancia, por ejemplo, el código fuente del mismo compresor.

En su implementación, considere la eficiencia de todos los componentes, incluyendo: la obtención del código óptimo, la codificación y la decodificación.

## 6.1. Código Provisto

Se proveen los archivos `io.c` e `io.h` que se encargan de las partes engorrosas de la escritura/lectura de cadenas de bits.

- `readfile` lee de un archivo y devuelve un buffer (alocado con `malloc()`) con sus contenidos. También setea en `*len` la longitud del mismo. **Nota:** los archivos pueden contener caracteres nulos (`'\0'`), por lo cual en general no pueden usarse funciones de cadenas como `strlen`.
- `writefile` escribe un buffer a un archivo.
- `explode` convierte un buffer a su representación como una cadena de bits. Normalmente, explota cada carácter a 8 nuevos caracteres `'1'` o `'0'`, y agrega un carácter nulo `'\0'` al final. También remueve el padding (ver adelante).
- `implode` convierte una cadena de bits (una string de C compuesta solamente de `'1'` y `'0'`) a bytes, poniendo 8 bits por byte. Dado que la longitud de la cadena puede no ser múltiplo de 8, esta función agrega un *padding* para que sí lo sea. El mismo es removido luego por `explode`.

La idea de `implode/explode` es que no tengan que preocuparse por manipular bits. Su código debería solamente trabajar con cadenas de `'1'` y `'0'`.

**Importante:** Si tenemos un `char c` y queremos acceder a la posición correspondiente a `c` en un array `A`, no es correcto hacer `A[c]`. El tipo `char` es (usualmente) con signo, y algunos caracteres son entonces negativos, haciendo que ese acceso no sea correcto. Para evitar este problema, podemos usar un cast a un tipo sin signo, ejemplo: `A[(unsigned)c]`.

## 6.2. Otras Referencias

Puede consultar a los libros [Cor+09] y [BB88] para más información. Este último puede conseguirse en español.

## Referencias

- [BB88] Gilles Brassard y Paul Bratley. *Algorithmics: Theory & Practice*. USA: Prentice-Hall, Inc., 1988. ISBN: 0130232432.
- [Cor+09] Thomas H. Cormen y col. *Introduction to Algorithms, Third Edition*. 3rd. The MIT Press, 2009. ISBN: 0262033844.
- [Huf52] David A. Huffman. "A Method for the Construction of Minimum-Redundancy Codes". En: *Proceedings of the IRE* 40.9 (1952), págs. 1098-1101. DOI: 10.1109/JRPROC.1952.273898.