Ficha 11

El Algoritmo de Huffman

1.) Introducción general al problema de la compresión.

Un *compresor de datos* es un programa que toma como entrada un archivo de *m* bytes de longitud, y produce *otro* archivo cuyo tamaño *n* es menor que *m*, pero sin pérdida de información con relación al archivo original. Del mismo modo, un *descompresor* es capaz de tomar un archivo comprimido como fue generado por el compresor, y producir el archivo orginal. Normalmente los productos comerciales (como *WinZip*) que hoy se encuentran en el mercado informático son *suites* que incluyen las funciones de compresión y descompresión en el mismo programa de gestión.

Las ventajas de la compresión son obvias: un archivo comprimido ocupa menos lugar en el disco que el original. Además, si se pretende enviar el archivo por medio de servicios web como el mail, el tiempo de envío o de bajada del archivo será menor que el del archivo original.

El problema básico de la compresión es producir una secuencia binaria más corta que la empleada por el archivo original en su representación de bytes. Clásicamente, cada caracter que se usa en una computadora se representa internamente con una cadena de 8 bits estandarizada en base a patrones que hoy son indiscutibles: la tabla *ASCII* es quizás el estandar más conocido hoy en día, y el formato *Unicode* es otro (aunque este último usa dos bytes por caracter).

La codificación ASCII es un típico ejemplo de *codificación binaria de longitud fija*: a cada caracter o símbolo que se quiere codificar se le asocia una cadena de bits de la misma longitud *t* (que en ASCII es *t = 8*) Así, si un mensaje o texto tiene un total de *x* caracteres, la conversión a binario de ese mensaje insumirá indefectiblemente un total de *t * x* bits. Así, el mensaje *AABACCD* que contiene siete caracteres, se llevará un total de 7 * 8 = 56 bits.

Está claro que la principal ventaja de la codificación binaria de longitud fija es la *sencillez* del proceso de codificación / decodificación: producir la cadena binaria es tan directo como buscar cada caracter en la tabla y reemplazarlo por su cadena de bits asociada. Y la decodificación también es directa: dada la cadena binaria total del mensaje, sabemos que cada *t* bits (8 bits en el caso ASCII) tenemos un caracter original, que también se recupera desde la tabla.

Las técnicas tradicionales de compresión se basan en atacar este concepto. La idea es que a cada caracter del alfabeto del mensaje se le asocie una *cadena de bits de longitud variable*. Y quizás las primeras ideas aportadas en ese sentido fueron las de *David Huffman*, en 1952. Desde entonces, las técnicas derivadas de sus ideas se conocen como el *Algoritmo de Huffman*, el cual ha sido estudiado y mejorado sistemáticamente desde su planteamiento original.

2.) El Algoritmo de Huffman.

Tomemos el mensaje que sirvió de ejemplo en el capítulo anterior: *AABACCD.* Podemos ver que si se quiere producir una codificación binaria de este mensaje que ocupe menos lugar que la que ocuparía con ASCII, no parece muy sensato que a cada caracter se le asocie una cadena de longitud fija, igual para todos. El problema es muy obvio: sólo el caracter *A* aportará 24 de los 56 bits que ocupará el mensaje traducido a ASCII. Y entre *A* y *C* se llevarán 40 bits, que es mas de la mitad del total de 56...

La idea planteada por Huffman es directa, y da pie un algoritmo basado en una *estrategia ávida*: asigne a los caracteres que más aparecen, una cadena de bits *más corta* que la que le asigne a los que aparecen más. En otras palabras: los caracteres del alfabeto del mensaje deben tener asociada una cadena de bits cuya longitud sea inversamente proporcional a la frecuencia de aparición de ese caracter en el mensaje. Así, si un caracter aparece muchas veces, el impacto de su frencuencia mayor disminuye en el largo del mensaje por aportar individualmente menos bits. En estas circunstancias resulta obvio que algunos símbolos serán codificados con menos bits y otros con más, y lo que se pretende es *minimizar la longitud de codificación promedio* entre todos los símbolos.

A modo de ejemplo, suponga que de algún modo se ha determinado que a cada símbolo del alfabeto del mensaje anterior le corresponda la cadena de bits que muestra la siguiente tabla en la columna *b* (en la cual también mostramos la frecuencia de aparición de cada símbolo, y la tabla se ordena por frecuencias):

Símbolo	Frecuencia	Probabilidad de aparición	Cadena fija de 2 bits posible (a)	Cadena de bits de longitud variable propuesta (b)
A	3	3/7 = 0.428	00	0
C	2	2/7 = 0.286	01	10
B	1	1/7 = 0.143	10	110
D	1	1/7 = 0.143	11	111

De esta forma, el mensaje *AABACCD* quedaría codificado en la forma siguiente, usando las secuencias de bits de la columna *b* de la tabla:

AABACCD = 0011001010111

y puede verse que la longitud total del mensaje es de 13 bits. Es cierto que la comparación es injusta, pues sólo se están considerando cuatro caracteres y en ese caso no harían falta los 8 bits de la tabla ASCII para codificarlos. Sería suficiente con dos bits para representar a los cuatro símbolos (columna *a* de la tabla)... pero es curioso que aún en ese caso el mensaje codificado tendría 14 bits... uno más que el que obtuvimos con la tabla mostrada.

Note que si se usan secuencias de bits de longitud fija *k* para todos los símbolos, estonces la longitud promedio de codificación en bits es igual a *k* (si usamos 2 bits por simbolo, la longitud promedio de bits por símbolo es 2…) Pero ¿cuál es la longitud de codificación promedio por símbolo si se usan secuencias de longitud variable? Para hacer ese cálculo, las frecuencias de aparición de cada símbolo deben convertirse en probabilidades. La tabla anterior surge de un mensaje de 7 símbolos, y la columna *frecuencia* de esa tabla indica cuántas veces cada símbolo aparece en el mensaje. Por lo tanto, la probabilidad de aparición de cada símbolo es igual a su frecuencia individual, dividida 7 (la longitud total

del mensaje). Con estos elementos, la longitud de codificación promedio por símbolo *LP* puede calcularse como:

LP = ∑(probabilidad del símbolo i * bits del símbolo i)
LP = $\sum(p_i * b_i)$

Para el ejemplo, y en base a la tabla propuesta de probabilidades y secuencias de bits, tendríamos una longitud de codificación promedio de:

LP = 0.428 * 1 + 0.286 * 2 + 0.143 * 3 + 0.143 * 3
LP = 0.428 * 1 + 0.286 * 2 + (0.143 + 0.143) * 3
LP = 0.428 + 0.572 + 0.286
LP = 1.858 bits por símbolo

Está claro que esta técnica lleva a codificaciones binarias que en total reducen la cantidad de bits para codificar un mensaje, pero siempre y cuando los símbolos en ese mensaje tengan distribuciones de frecuencias diferentes: si todos los símbolos del mensaje aparecen con distribución uniforme o con frecuencias muy similares, el algoritmo que presentaremos no puede distinguir entre "mejores" y "peores" formas de codificación.

El proceso de codificación (si se tiene la tabla) es directo: se reemplaza cada caracter por su cadena propuesta. Pero ahora la decodificación no es tan simple: la cadena binaria debe analizarse bit a bit, y proceder de la siguiente forma: si el bit que se toma corresponde a un símbolo, se toma ese símbolo y se salta al bit siguiente. Si el bit analizado no corresponde a un símbolo, se toma el siguiente bit y ahora se busca si algún símbolo corresponde a los dos bits tomados. Y así se sigue, hasta decodificar todos los bits.

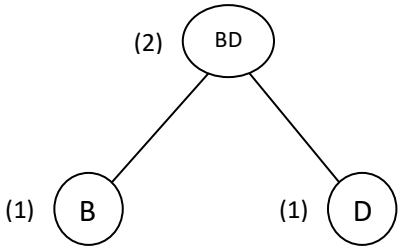
Pero lo anterior supone un potencial problema: ¿qué pasaría si en la tabla anterior el código asignado a la C fuera 01 en lugar de 10? En principio, podría pensarse que no hay problema alguno pues la longitud de la cadena asignada no ha cambiado. Pero entonces una secuencia como 01110 sería imposible de decodificar: no quedaría claro si se trata de una A seguida de una D y luego una A, o si se trata de una C seguida de una B. Con la tabla originalmente propuesta, esos problemas no se presentan. Entonces queda claro que la técnica de codificación propuesta no sólo debe asignar cadenas de longitud inversamente proporcional a la frecuencia del símbolo, sino que debe evitar que la cadena binaria asociada a un símbolo nunca se use como prefijo para formar la cadena binaria de otro... En el ejemplo, si la C se asocia con 01, entonces la cadena de la A (0) es prefijo para formar la cadena de la C, y allí comienzan los problemas.

¿Cómo lograr la codificación binaria de cada símbolo en las condiciones exigidas? La idea es usar un *árbol binario estricto*: un árbol binario en el cual cada nodo es una hoja sin hijos, o es un nodo con dos hijos (ningún nodo tiene un sólo hijo). El proceso de construcción del árbol propuesto tiene que garantizar que las dos condiciones se cumplan. Tal proceso se conoce como *Algoritmo de Huffman*, y produce el árbol que se conoce como *Árbol de Huffman*.

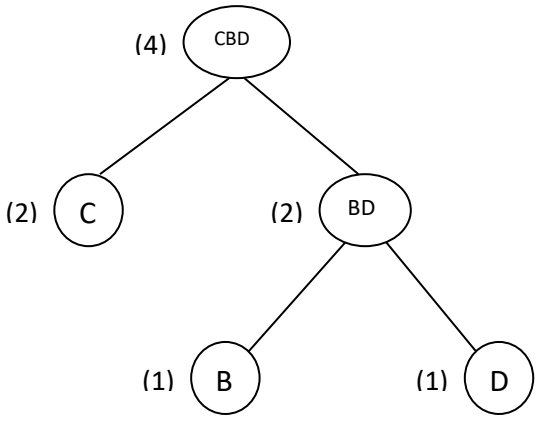
Esencialmente, el algoritmo de Huffman toma como entrada una tabla de símbolos, de forma que se conoce la *probabilidad de aparición* de cada uno o bien se conoce directamente su *frecuencia de aparición*. El objetivo del algoritmo es producir un *árbol binario estricto*, en el cual cada símbolo de la tabla original sea una hoja, pero de tal forma que ese árbol *no admita problemas de prefijos* y que además, *garantice que la longitud promedio de bits usada para codificar todos los símbolos de la tabla sea mínima*.

Como se sugirió, el algoritmo de Huffman es esencialmente un *algoritmo ávido*: se basa en aplicar sistemáticamente una regla que parece correcta, sin posibilidad de vuelta atrás ni de revisión o rebalance. Hemos sugerido en una ficha anterior que los algoritmos ávidos tienen la ventaja de ser generalmente simples de comprender e implementar, pero también tienen la desventaja de requerir que se pruebe que la regla aplicada es correcta, y eso no suele ser tan simple. En el caso del algoritmo de Huffman, el proceso constructivo del árbol garantiza que la regla es correcta, aunque se pueden hacer pruebas de corrección formales más exhaustivas.

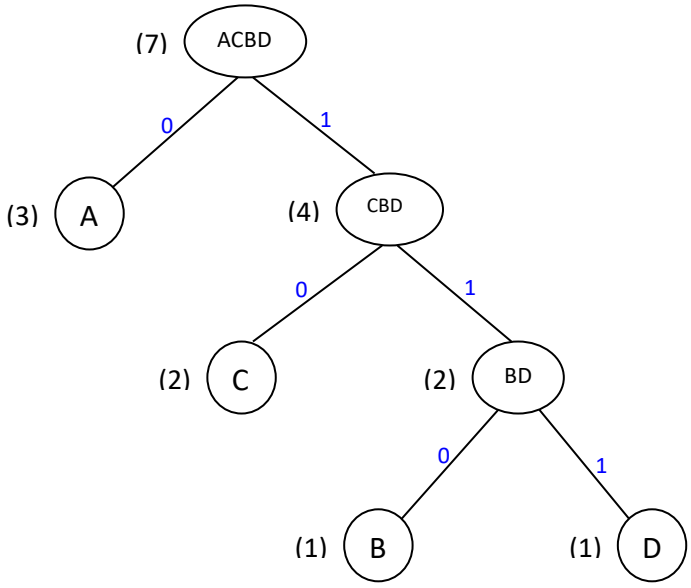
La idea es la siguiente: se parte de los dos símbolos de menor frecuencia (la B y la D en nuestro caso). En el árbol, cada símbolo del alfabeto original entra como un *nodo sin hijos*. Por lo tanto, la B y la D entran como dos hojas, y se asocia como padre de ellas a un tercer nodo que se considera como la unión de ambos símbolos (BD). En cada nodo se almacena la frecuencia del símbolo contenido (en la gráfica se colocó ese número entre paréntesis al lado de cada nodo), y si se trata de un nodo unión se toma una frecuencia igual a la suma de las frecuencias de sus hijos:



Los símbolos de la tabla que ya entraron al árbol (B y D) ya no se tienen en cuenta. Ahora se considera que quedan el resto de los símbolos originales (A y C) más el símbolo unión que se acaba de formar (BD). De entre esos tres símbolos se vuelven a tomar los dos de menor frecuencia (C y BD en este caso). Como C es un símbolo del alfabeto original, entra al árbol como una hoja. El símbolo unión BD ya está en el árbol y simplemente se une con C produciendo un nuevo símbolo unión CBD con frecuencia 4:



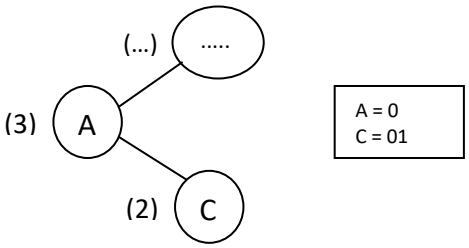
Sólo quedan la A de la tabla de símbolos originales, y el símbolo unión CBD que se acaba de formar. Como sólo quedan esos dos, es obvio que son los de menor frecuencia y se unen a su vez, produciendo el árbol final. El paso que sigue, es asignar a cada rama del árbol un peso o ponderación: si se trata de una rama izquierda se le da un valor 0 y si se trata de una rama derecha se le asocia un valor 1 (en la gráfica estos pesos se colocaron en azul al lado de cada enlace):



Una vez construido el árbol, los símbolos se codifican de la siguiente forma: se parte de la hoja del símbolo que se quiere codificar, y se va subiendo por el árbol nivel por nivel. Cada vez que se sube por una rama izquierda se agrega un 0 en la secuencia binaria de ese símbolo, pero de derecha a izquierda (la secuencia binaria se construye de atrás hacia delante pues el árbol se está recorriendo de abajo hacia arriba). Y cada vez que se sube por una rama derecha, se asocia un 1 a la secuencia. Así, la secuencia asignada a la B termina con 0 (enlace <B, BD>), luego sigue con 1 (enlace <BD, CBD>) y tiene un 1 al principio (debido al enlace <CBD, ACBD>). Procediendo de esta forma, surge la tabla con la que comenzamos esta discusión.

El proceso garantiza que las condiciones exigidas para garantizar que la codificación / decodificación no sea ambigua se cumplan ambas: como el árbol se construye comenzando con los símbolos de menor frecuencia, estos quedan en niveles del árbol más alejados de la raiz que los símbolos de mayor frecuencia, por lo que el recorrido desde su hoja hasta la raiz tiene más enlaces y por lo tanto más bits que los de mayor frecuencia.

Por otra parte, los símbolos originales del alfabeto entran al árbol como hojas (nodos sin hijos) y esto es lo que garantiza que la secuencia binaria asociada a un símbolo *no será* usada como prefijo para formar la secuencia de otro: sólo si un símbolo fuera hijo de otro compartirían la misma subsecuencia de bits para ascender hacia la raiz:



3.) Implementación del Algoritmo de Huffman.

La primera idea de implementar el árbol con una estructura basada punteros debería ser reconsiderada de inmediato: tal implementación haría díficil poder acceder directamente a una hoja, y bastante incómodo el recorrido del árbol hacia arriba. Estas dos circunstancias nos hacen pensar en un arreglo en lugar de un árbol de punteros tradicional. La idea del arreglo se ve apoyada por el hecho que el árbol de Huffman es binario estricto y se conoce de antemano el número *h* de hojas que tendrá. Si este es el caso, la cantidad total *n* de nodos del árbol es:

$$n = 2 * h - 1$$

Por lo tanto, siempre se puede conocer cuantos nodos necesitará el árbol y se puede entonces dimensionar un arreglo de acuerdo a ese tamaño. Cada componente del arreglo contendrá un objeto de la clase *HuffmanTreeNode* que contendrá a su vez un atributo para la frecuencia del nodo representado, el índice del padre de ese nodo en el árbol (lo cual permite la navegación hacia arriba en el árbol), un indicador booleano para saber si el enlace con el padre es una rama izquierda o derecha y de allí saber si se agrega un cero o un uno al subir, y dos atributos más con los índices de los hijos (que son necesarios al decodificar, y que valen –1 si el nodo es una hoja):

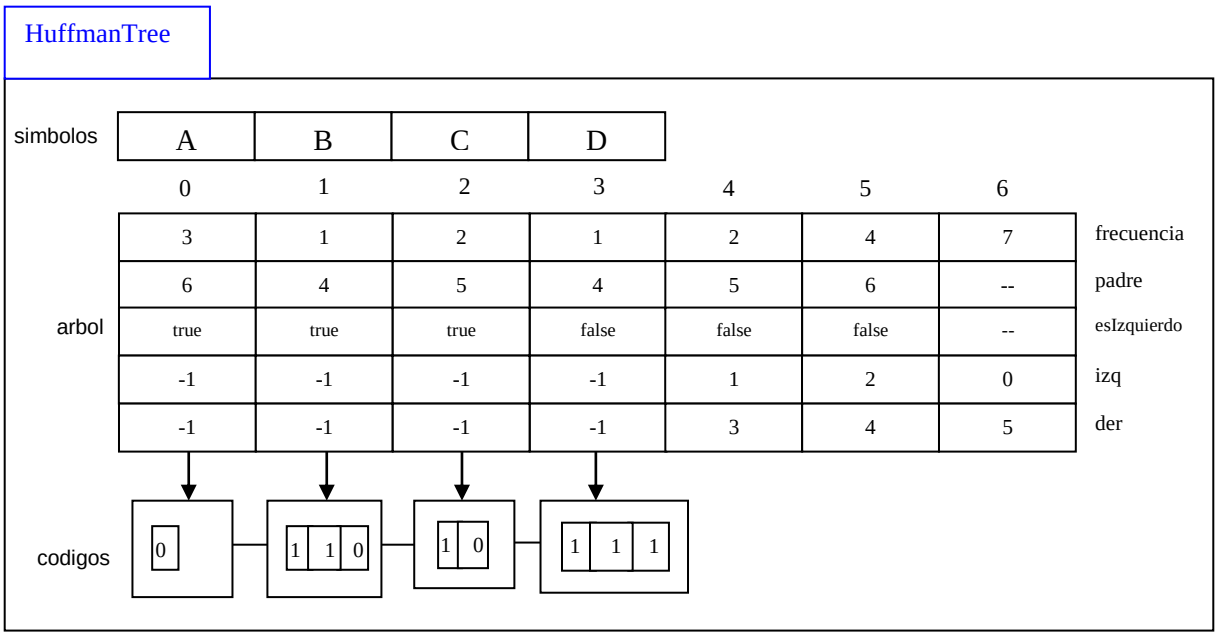
```
public class HuffmanTreeNode
{
        private int  frecuencia;    // frecuencia del signo representado
        private int  padre;         // indice del padre dentro del arreglo que soporta al árbol
        private boolean esIzquierdo;   // indica si este nodo es hijo izquierdo o no de su padre

        private int izq, der;       // para la fase de decodificación, son los índices de los hijos
        ...
}
```

La clase principal del modelo es *HuffmanTree*, la cual contiene al arreglo que representa al árbol y algunas estructuras más. Por lo pronto, también contiene un arreglo para almacenar los símbolos del alfabeto original, que no es necesario que se almacenen en el arreglo que representa al árbol pues este es más largo que la cantidad de símbolos y quedarían elementos sin usar (los símbolos unión no necesitan guardarse en ninguna parte, aunque sí su frecuencia).

La clase *HuffmanTree* dispone de un método *makeTree()* que arma el árbol llenando el arreglo que lo contiene. Para ir armando el árbol este método requiere poder tomar los dos símbolos de menor frecuencia, y para ello utiliza una *cola de prioridad* (en este caso implementada como un *Heap ascendente*). Cada elemento almacenado en esa cola de prioridad es un objeto de la clase Frequency, en el cual se almacena la frecuencia del símbolo que se trata, y el índice que ese símbolo tiene en el arreglo que representa al árbol.

Otro método importante en la clase *HuffmanTree* es el método *makeCodes()* que a partir del árbol ya creado, obtiene los códigos de Huffman de cada símbolo. Esos códigos binarios deben ser almacenados en alguna parte para que el proceso se realice una sola vez, y no tener que volver a calcular todos los códigos cada vez que se los necesite. Para ello, la clase *HuffmanTree* cuenta con un atributo de tipo *ArrayList*, en la cual se guardan a su vez otros objetos ArrayList que contienen valores de tipo byte: la idea es que por cada uno de los *h* símbolos originales, exista un *ArrayList* en el que cada elemento sea un 1 o un 0, y de allí que se requiera una lista que contenga a su vez a todos esos *ArrayList*.

Por lo tanto, si se quiere representar el árbol construido en la página anterior para el mensaje *AABACCD*, las estructuras de datos podrían verse como sigue (sin la cola de prioridad, cuyo manejo es muy dinámico y sirve sólo a los efectos de ir construyendo el árbol):



Se deja para los alumnos la tarea de analizar el código fuente del proyecto que se anexa, en el cual se implementan todas estas ideas.