MEMORIA PRÁCTICA 1

ALGORITMIA BÁSICA

CÓDIGO DE HUFFMAN

AUTORES:	NIAs:
Victor Manuel Lafuente	747325
José Manuel Romero	740914

INDICE

Resumen

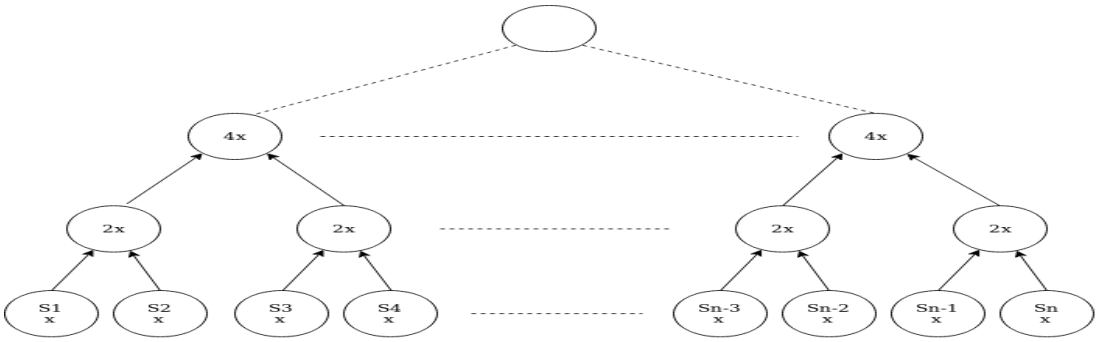
Para la realización de la práctica se ha empleado, tal como se se sugirió en clase, el algoritmo de Huffman. La implementación se ha realizado para codificar 256 tipos de símbolos (correspondientes a los 256 valores posibles por Byte). El árbol necesario para desarrollar el mencionado algoritmo, se ha apoyado en un montículo, el cual se prefirió implementar desde cero para garantizar que fuese lo más eficiente posible, evitando posibles herencias con otros *TAD*. Por supuesto, se ha verificado el correcto funcionamiento de este TAD con pruebas específícas. Se ha implementado también el *TAD* 'Trie' cuya finalidad será construir el árbol necesario para el algoritmo de Huffman que, dada su naturaleza, se ha implementado con un recorrido específico para dicho algoritmo, tal como se detalla más adelante. Una última consideración necesaria que afecta directamente a la eficiencia del código, tal como se hablará más adelante, es el uso de una *Look up table*, que almacena para cada posible *byte* (índice) su equivalente.

Cómo resultado se ha obtenido un programa que es capaz de recodificar y recuperar la información original de cualquier tipo de fichero y que además la recodificación, quitando ciertos casos que se comentarán más adelante, se consigue reducir el tamaño (y por tanto comprimir) del fichero.

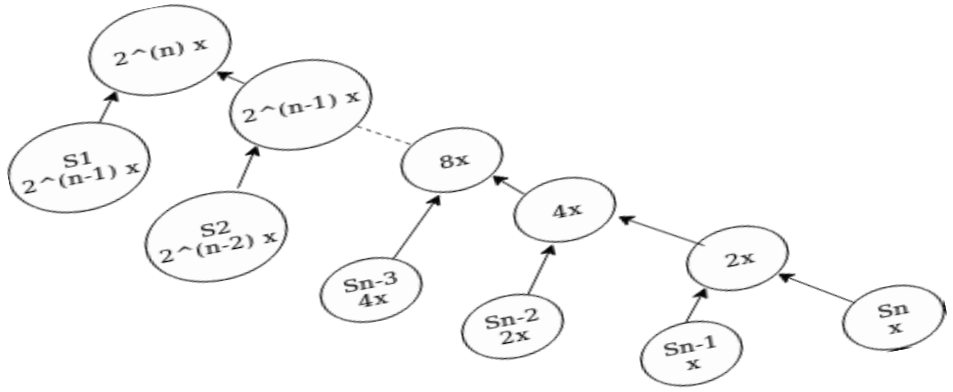
Posibles árboles generados

Tal como se estudió en clase (y sirviendo como breve resumen y contextualización), el algoritmo de Huffman se basa en juntar los dos nodos con menor peso (siendo peso el número de repeticiones) a un mismo padre cuyo peso será la suma de sus hijos, generando así secuencias de bits más largas para aquellos nodos poco empleados y secuencias más cortas para aquellos nodos más empleados, consiguiendo así una reducción del tamaño del fichero. Tal como se va a ver a continuación, no siempre se consigue generar una codificación que consiga reducir el tamaño del fichero.

Uno de los peores casos posibles, es la codificación de un fichero cuyo contenido siga una distribución uniforme o lo que es lo mismo, aparecen los 255 símbolos exactamente el mismo número de veces. Esto provocaría que se generara un árbol binario completo, con 256 hojas de altura $\log_2(256)$, o lo que es lo mismo, se haría un gasto de tiempo en tratar la codificación para transformar cada símbolo de 8 bits en otro de 8 bits, lo que generará un fichero del mismo tamaño más la constante de almacenar en la cabecera del fichero lo necesario para la recuperación del fichero (detallada en su respectiva sección).



Otro posible caso es que se genere un árbol cuasi-degenerado (ya que no es posible por la naturaleza del mismo problema que se genere completamente degenerado). Este caso se puede dar si el histograma (gráfica donde se refleja el número de ocurrencias por cada símbolo) tiene la forma de una función logarítmica en base dos, es decir, dado un conjunto de símbolos ordenados por ocurrencias, cada símbolo tiene el doble de ocurrencias que su anterior y la mitad de su siguiente símbolo. Si se da este caso en un fichero, la codificación de cada símbolo oscila entre 1 bit para el símbolo ocurrente y 255 bits para el menos.



Para este caso en particular, se puede calcular la tasa de compresión del fichero generado de la siguiente forma:

$$tasa\ compresión = \frac{tamaño\ fichero\ generado}{tamaño\ fichero\ original} = \frac{255x + \sum_{i=1}^{255} (2^{i-1} x (256 - i))}{8x + \sum_{i=1}^{255} (2^{i-1} 8x)} = 0.25\ ;\ \forall\ x \in \mathbb{N}$$

Dónde:

- ***x*** es el número de repeticiones del símbolo menos ocurrente
- 2^{i-1} es el factor multiplicativo del símbolo ‘i’ (recordemos que en este es proporcional)
- ***8*** bits por cada símbolo del fichero original
- ***(256 - i)*** número de bits necesarios para codificar el símbolo ‘i’

Por simplificación, no se ha considerado para el cálculo el espacio necesario para almacenar la tabla de símbolos.

Con la ecuación dada se puede ver que este tipo situaciones resultan un tanto idóneas (y poco frecuentes) ya que la tasa de compresión es elevada. No obstante, a la hora de codificar los símbolos en el algoritmo, ha sido necesario emplear un *TAD* el cual pueda tener longitud variable entre 1 a 255.
El resto de posibles casos, son intermedios entre estos dos árboles.

Consideraciones de implementación

El algoritmo se ha implementado en C++, ya que es un lenguaje de programación cuya eficiencia en tiempo y memoria resulta a día de hoy poco cuestionable al ser un lenguaje compilado y que trabaja con punteros, otorgando al programador mucha libertad de acceso a los datos y, por tanto, la eficiencia que se puede llegar a obtener depende en gran medida de los programadores.

Se ha implementado el algoritmo para recodificar cada Byte del fichero original (256 símbolos en total). Se ha considerado que con 256 símbolos distintos el algoritmo puede comprimir la mayoría de ficheros de manera eficiente, a excepción (tal como se ha visto anteriormente) de aquellos que conforman una distribución uniforme de símbolos, caso que ,estadísticamente, es extraño de ver.

Entrando en los detalles de la implementación, para almacenar el número de ocurrencias de cada símbolo se ha hecho mediante un vector de 256 elementos (inicialmente todos a 0), cuyo acceso a cada elemento es constante. Por otra parte, se ha empleado otro vector del mismo tamaño a modo de *Look Up Table* (*LUT*) para almacenar en cada posición ‘i’ de la tabla, la codificación del símbolo ‘i’. De esta forma, obtener la codificación de cada símbolo es también constante.

Tal como se ha demostrado anteriormente, el tamaño máximo de codificación son 255 bits, por lo que se ha decidido hacer el vector *LUT* de tipo *strings*. Si bien es cierto que el tipo de dato ‘string’ no es primitivo su manejo resulta dinámico y muy versátil, lo que facilita la implementación y depuración del código y dado que el tamaño máximo es conocido, constante y no pocos bytes, no se ha visto necesario buscar posibles alternativas más eficientes en memoria o en tiempo de acceso.

Por último, y no menos importante, se ha decidido almacenar al comienzo del fichero comprimido el vector de ocurrencias debido a su sencillez, ya que el tamaño de datos a almacenar no es muy elevado, y además, la reconstrucción del árbol en tiempo es considerablemente buena. Esto también simplifica el código ya que, si se requiriese de modificar en algún aspecto el árbol, este será robusto al emplear el mismo algoritmo. Dicho vector de ocurrencias se almacena en el fichero como tipo carácter con objeto de poder observar el contenido del fichero resultante. Este tipo de dato no es óptimo para la tarea desempeñada (almacenar datos para la reconstrucción del árbol), pero hemos preferido mantenerlo dado que para fichero de tamaño elevados el resultado no varía y así poder mantener dicha introspección.

Desarrollo del algoritmo

Sea '**B**' el número de **Bytes en el fichero original**, '**S**' el número de **símbolos distintos** en el fichero original.

El algoritmo de compresión comienza con una lectura completa del fichero y contabilizando las ocurrencias de símbolos en su vector. Esto genera un coste en tiempo O(B) y en memoria O(1), ya que el vector de ocurrencias es siempre constante (256).

Una vez leído el fichero completo se ordenan los símbolos con el algoritmo de HeapSort de menor a mayor ocurrencias. Este algoritmo se ha implementado de manera que, se insertan los símbolos leídos en el montículo (O(S*log(S)), generando así una cola con prioridad. Una vez ordenado, se puede generar el árbol. Dicho árbol tendrá exactamente 2S-1 nodos (contando hojas), y debido que por cada nodo generado hay que extraerlo del montículo (con coste log(S)) y reinsertarlo (también coste log(S)), el coste en tiempo de generar el árbol es de 2S*2(log(S)), por tanto O(S*log(S)), resultando así la generación de la cola con prioridad y el árbol completo.

Por tanto, dado que el número **B** es mayor o igual que **S**, el coste en tiempo durante la compresión es de **O(B)** y en memoria es de orden **O(S)** debido al tamaño del árbol.

Por otra parte la descompresión la descompresión se basa en leer las frecuencias, volver a generar el árbol y leer el resto del fichero para poder recuperar la información original. Dicha recuperación se basa en ir recorriendo, por cada bit leído, el árbol hacia el hijo izquierdo o derecho (Dependiendo de si es un 0 o 1), lo que hace un coste **O(b)**, siendo b el número de bits a decodificar, y la complejidad en memoria es de **O(S)** por el tamaño del árbol.

Pruebas Realizadas

Se han realizado numerosas pruebas con distintos ficheros, tanto de texto como binarios, con los que observar el correcto funcionamiento del programa. En particular, y siendo este conjunto de pruebas el adjunto a este trabajo, se han implementado en *Octave* dos pequeños scripts, de manera que se generen 9 ficheros cuyo contenido son los 256 símbolos, cuyas ocurrencias siguen aproximadamente una distribución normal de media aleatoria, pudiendo ir el número de ocurrencias entre 0 a N, dónde N es un número arbitrario que dependerá del tamaño deseado del fichero, tal como se podrá observar en la tabla de Resultados. El segundo script genera 2 ficheros de distribución uniforme, o lo que es lo mismo, un fichero donde todos los símbolos contienen la misma cantidad de ocurrencias. El primer fichero que se genera con este segundo script es de tamaño 2'4MB y el segundo es de 24MB. Debido al tamaño del fichero necesario para generar un árbol degenerado con una cierta cantidad considerable de símbolos es elevado, no se ha podido probar.

Por otra parte, también se han adjuntado el fichero de texto propuesto como ejemplo (Quijote de tamaño 2.1MB), para verificar que el marco de lo obtenido es correcto. También se han adjuntado otros dos ficheros extraídos de la misma página con distintos tamaños (uno de 800KB y otro de 3'2MB).

Se ha automatizado el proceso de comparativa con un script en *bash* que compila, genera una copia de cada fichero, la comprime, descomprime y la compara con el comando *'diff'* con la copia original.

Resultados

Fichero	Distribución del contenido	Tamaño original	Tamaño comprimido	Nº de símbolos	Tiempo Compresión	Tiempo Descompresión
prueba1.pr	Gaussiano	2,3M	1,2M	251	0m2.727s	0m1.445s
prueba2.pr	Gaussiano	2,6M	1,5M	251	0m3.320s	0m1.724s
prueba3.pr	Gaussiano	2,6M	1,7M	249	0m3.653s	0m1.817s
prueba4.pr	Gaussiano	2,9M	1,9M	250	0m3.893s	0m2.048s
prueba5.pr	Gaussiano	4,9M	3,0M	253	0m6.351s	0m3.319s
prueba6.pr	Gaussiano	5,9M	3,4M	251	0m7.464s	0m3.854s
prueba7.pr	Gaussiano	12M	6,8M	251	0m15.164s	0m8.036s
prueba8.pr	Gaussiano	24M	14M	256	0m30.698s	0m15.672s
prueba9.pr	Gaussiano	28M	16M	252	0m35.271s	0m18.272s
pruebaUnifor1.pr	Uniforme	2,4M	2,4M	256	0m3.760s	0m2.032s
pruebaUnifor2.pr	Uniforme	24M	24M	256	0m37.421s	0m20.253s
quijote2M.txt	-	2,1M	1,2M	104	0m2.684s	0m1.500s
quijote800K.epub	-	849K	852K	256	0m1.288s	0m0.719s
quijote3M.mobi	-	3,4M	3,2M	256	0m5.177s	0m2.902s

Tal como se puede observar en la tabla, los ficheros con distribución normal, tienen en media una tasa de compresión de 0.59, o lo que es lo mismo, el fichero resultante es poco mayor de la mitad del tamaño original. Tal como se preveía, el tamaño de las distribuciones uniformes se mantiene (aunque sabemos que será un pelín mayor el comprimido debido a la cabecera de equivalencias de símbolos).

Tal como se preveía, comparando los resultados obtenidos por *prueba1.pr* y *quijote2M.txt*, cuyos tamaños son significativamente similares, El tiempo de ejecución es sobretodo proporcional al tamaño del fichero de entrada. Esta proporción se observa muy bien comparando con los ficheros de mayor tamaño.

También se puede observar que con los ficheros de distribución uniforme, el tiempo de ejecución es un poco mayor que ficheros equivalentes de distribución normal. Esto es debido a que el fichero resultante también es mayor.