Algoritmia básica - Practica 1 (2016), pp. 1–4
Manuel Lagunas Arto

Compactador de ficheros basado en el código Huffman

Manuel Lagunas[1]

[1]Universidad de Zaragoza, Algoritmia básica

Abstract

El siguiente documento contiene la información referente a la práctica 1 de la asignatura de Algoritmia básica. El objetivo de esta es desarrollar un compactador de ficheros basado en el código de Huffman. Las siguientes lineas exponen la estrategia seguida para su desarrollo asi como las decisiones tomadas. El código ha sido desarrollado por el propio autor de este documento. Así como la idea de mejora del algoritmo.

1. Construir el árbol

Para poder construir el árbol de donde poder obtener los códigos de los diferentes carácteres leidos por el fichero de entrada ha sido necesario desarrollar una nueva estructura de datos [Bro]. Esta estructura recibe el nombre de *Node* y almacena el **peso** (número de ocurrencias en el fichero de entrada), el **carácter** que representa, tan solo los nodos que se encuentren sin hijos (nodos finales) serán los que representen un carácter. Tambien almacenará punteros a sus hijos **izquierdo** y **derecho**, así como a su nodo **padre**.

Una vez se tiene la estructura donde almacenar el árbol que nos permitirá obtener los nodos es momento de construirlo. Para ello se lee la entrada del fichero y se ordena (por carácter) y agrupa de manera que la tarea de contar sea realizada de manera sencilla con coste $O(n \log n)$ al ordenar la cadena de carácteres de entrada. Se crearan nodos sin hijos con dichos valores y pesos que serán almacenados en un monticulo (el coste del cambio de una lista al monticulo será $O(n)$ [Pyt] el monticulo nos devolverá los valores con menor peso cada vez que ejecutemos un *pop*. El algoritmo a seguir para poder obtener el árbol será:

```
mientras lista.size > 1:
        pop dos elementos (izq y dch);
        nuevo nodo suma de pesos izq + dch;
        nodo hijos izq y dch;
        añadir a la lista nodo;
```

Figure 1: *Algoritmo para obtener el árbol*

Tras finalizar el algoritmo de la figura 1, el montículo tendrá solamente un objeto, dicho objeto nodo será la raiz del árbol que tendra como hijos subárboles que recorridos recursivamente llegarán a las hojas finales, donde se encuentrán los nodos que representan los carácteres.

2. Comprimir el fichero

Con el árbol almacenado en una estructura de datos, ahora se tienen que obtener los códigos para cada carácter.

2.1. Obtener los códigos

En este paso, lo que se hace es dar un peso a cada rama del árbol. Si la rama conecta con el hijo izquierdo será 0, si conecta con el hijo derecho será 1. Para obtener el código de cada carácter habrá que recorrer de las hojas finales (nodo con un carácter almacenado) a raiz, almacenando los pesos que vayamos obteniendo que serán un número binario que es el código del carácter en cuestión. Para despues conseguir el código de cada carácter en $O(1)$ lo almacenamos en una tabla hash de tal manera que las claves serán los carácteres y los valores los códigos de estos. Para poder construir el mensaje.

2.2. Comprimir el mensaje

Para comprimir el mensaje será tan sencillo como hacer un bucle que recorra carácter a carácter la cadena de entrada dada por el fichero accediendo a la tabla hash que mapea los carácteres del fichero con el código obtenido.

```
map = hash map inicializado;
resultado = '';
para char en input:
        resultado += map[char];
```

Figure 2: *Algoritmo para comprimir el contenido del fichero*

2.3. Codificar el árbol

Ademas de incluir el propio fichero codificado será necesario escribir una representación del árbol usado, pues de otra forma sería

imposible conocer los carácteres o volver a re-crearlo. Para la codificación del árbol se sigue un algoritmo sencillo que lo recorrera recursivamente escribiendo un 0 si ese nodo es una hoja no final y un 1 si la hoja es final (representa un caracter) seguido de un byte que representa el carácter. El resultado final será una cadena binaria que habrá que decodificar.

2.4. Mensaje final

Una vez tenemos el árbol codificado, el mensaje (o fichero) comprimido debemos añadir algo más de información que nos permita poder decodificarlo de manera correcta. Se añadirá ademas un byte al inicio que marcará el número de bits necesarios para completar el ultimo byte, pues Python, no es capaz de escribir bits por en ficheros. Además seguido se añadira otro byte que marcará el número de nodos finales que tendra nuestro árbol, para asi poder hacer la decodificación correctamete. El contenido global codificado será como se representa en la siguiente figura:

$$m = [nbitsfinal] + [header] + [mensajecomprimido]$$

$$header = [nnodosfinales] + [arbolcodificado]$$

Figure 3: *Representación de la estructura del fichero comprimido*

3. Descomprimir el fichero

Para le descompresión del fichero se necesitará el mismo árbol que el usado para comprimirlo. Además de conocer datos como el número de nodos finales de este o los bits finales usados para completar el último byte. Los pasos a realizar han de realizarse en el orden indicado.

Lo primero a hacer será almacenar el numero de bits para completar el último byte, para ello basta con leer un byte del fichero de entrada.

3.1. Recodificar el árbol

El seguiente punto a realizar será volver a construir el árbol de tal manera que sea casi identico (solo varían los pesos) que al que se tenía para la compresión. Se leera un byte para conocer el número de nodos finales del árbol y despues de eso se llamará a un algoritmo recursivo que, conociendo que el árbol se ha codificado dando prioridad a las ramas izquierdas, lo reconstruira. El algoritmo será el siguiente: Primero si no existen nodos se creara uno nuevo teniendo en cuenta si la entrada es 0 o 1 (nodo final o no), y se llamará recursivamente de nuevo con la nueva raiz y avanzando uno en el vector de carácteres. Si ya existe un nodo entonces se comprueba si este tiene hijos, si tiene ambos se vuelve al nodo padre con una llamada recursiva y se repite el proceso, si solo tiene nodo izquierdo (por la prioridad izquierda dada en la codificación) se creará un nuevo nodo teniendo en cuenta si representa un carácter o no (caracter de la entrada 1 o 0, respectivamente), en caso de que no tenga hijos se creara el nodo y se añadira al hijo izquierdo. Si la entrada ha sido uno y se ha creado un nuevo nodo, el número de nodos finales se reduce en uno. La recursividad se irá repitiendo hasta que el número de nodosFinales sea cero que se devolverá la raiz.

Tras este algoritmo tendremos creado un árbol con la misma estructura que el de la compresión pero con distintos pesos, ya que no son de interes porque es posible crearlo sin ellos. El algoritmo en pseudocódigo se puede ver en los anexos.

3.2. Descodificar el mensaje

Con el árbol ya creado el paso siguiente es inmediato, para poder volver a conseguir los carácteres hay que recorrer el árbol de acuerdo a la entrada obtenida [Sig]. Si se tiene un 0 se avanzará en la rama izquierda, si se tiene un 1 se avanzará en la rama derecha. Si se llega a un nodo sin hijos, significa que este será un nodo final y por tanto representará un caracter. Ese carácter es el obtenido para los n bits leidos. Con repetir este proceso hasta que se llegue al final se obtendrá el mensaje descomprimido. En este punto hay que tener en cuenta el número de bits usados para llenar el último byte, asi pues, la representación del último byte podrá ser distinta y no necesariamente con 8 bits. El resultado de este paso tendrá complejidad $O(n)$ en el tamaño de la entrada recibida por el fichero.

4. Mejoras en la ejecución

Esta parte presenta ciertas mejoras implementadas para reducir el tiempo necesario para comprimir ficheros de tamaño considerable: $>$ 80000 bytes. A la hora de descomprimir se ha creado un tabla hash conteniendo un mapeo código ->carácter ordenada en tamaño ascendiente (priorizando elementos que más ocurrencias tienen) en cuanto a longitud del código. De esta manera el tiempo de ejecución disminuye si el fichero tiene un tamaño considerable. Esto se debe a que la tabla permite accesos con coste $O(1)$.

El algoritmo crea la tabla conforme va explorando el árbol y descomprimiendo el fichero, a cada nuevo carácter que encuentra, la tabla tendrá una entrada nueva que podrá ser accedida en la siguiente iteración. De esta manera se puede comprobar si x bits de la entrada pertenecen a una llave de la tabla hash, que en caso afirmativo devolverá su contenido. Este algoritmo provoca que el comienzo sea más lento, dado que la tabla estará llenandose, y hará que funcione más rápido en el momento que la tabla contenga un conjunto de claves considerable.

5. Resultados obtenidos

En la sección siguiente se muestra una tabla de tiempos y bytes, para comparar distintos ficheros y distintos algoritmos.

En la primera tabla tenemos comparaciones de tamaños de compresión. En el caso del $test_1$ se observa que el tamaño del comprimido es prácticamente igual al original, esto se debe a que codificar la cabecera (el árbol) conlleva un espacio considerable respecto al total del fichero inicial. El $test_2$ se trata de un fichero con numerosas repeticiones de caracteres (una frase repetida todo el rato) que por tanto hace que la penalización por almacenar la cabecera se disminuya, consiguiendo una compresion de alrededor del 52 % (notar que en la tabla el porcentaje es: porcentaje del comprimido sobre el original). Tanto $test_3$, $test_4$ y $test_6$ son ficheros con textos corrientes de diferentes tamaños, ambos tienen porcentajes de

Test	original	comprimido	porcentaje	tiempo
test1	69	65	0.9420	0.00079
test2	2850	1382	0.4849	0.00337
test3	21040	11925	0.5667	0.04806
test4	252723	134484	0.5321	5.56854
test5	152	181	1.1907	0.001052
test6	552721	302833	0.5478	29.846

Table 1: *Comparativa del tamaño de la compresión, y tiempos para distintos ficheros de prueba (tiempos medios para 5 repeticiones)*

Test	algOriginal	algMejora
test2	0.010841	0.01372
test3	0.3424	0.3200
test4	69.7636	12.2737
test6	472.1747	59.06

Table 2: *Comparativa de tiempos de descompresión en ficheros con tamaño considerable (tiempos medios de 5 repeticiones)*

compresion similares, pues como se menciona anteriormente son textos corrientes sin repeticiones de carácteres provocadas, aunque se observa cierta penalización en el de menor tamaño; el $test_3$. Por último, $test_5$ se trata de un fichero pequeño con no muchos carácteres repetidos, lo que provoca una gran penalización de la compresión al guardar la cabecera haciendo que el resultado sea peor que el fichero inicial.

La segunda tabla muestra tiempos en comparacion de ambos algoritmos. Se observa como en ficheros de tamaño reducido como $test_2$ se produce un empeoramiento en el tiempo, dado que el tiempo usado en construir la tabla hash no es recuperado una vez se ha construido. Con ficheros de un tamaño medio, como $test_3$ el algoritmo empieza a mostrar mejoras sin ser realmente considerables. Cuando los ficheros tienen un conjunto de bytes considerable, la mejora es sustancial tal y como puede observarse para $test_4$ y $test_6$. Esto se debe a que una vez la tabla ha sido construida las decodificaciones restantes que queden se resolveran en tiempo $O(1)$ por cada carácter y no en $O(n)$ como pasa en el caso de tener que buscar en el árbol.

References

[Bro] BROWN B.: *Text Compression with Huffman Coding*. https://www.youtube.com/watch?v=ZdooBTdW5bM. 1

[Pyt] PYTHON SOFTWARE FOUNDATION: *Heap queue algorithm*. https://docs.python.org/2/library/heapq.html#heapq.heapify. 1

[Sig] SIGGRAPH: *A quick tutorial on generating a huffman tree*. https://www.siggraph.org/education/materials/HyperGraph/video/mpeg/mpegfaq/huffman_tutorial.html. 2

6. Anexo

```
func construir(arbolCodificado, nodosFinales, nodo = nil, raiz = nil):
 si nodosFinales > 0:
  si arbolCodificado[0] == 0
   si nodo == nil
    n = nuevo nodo
    construir(arbolCodificcado[1..x], nodosFinales, n, n)
   sino
    si nodo.izq y nodo.dch
     construir(arbolCodificcado[0..x], nodosFinales, n.padre, raiz)
    sino
    n = nuevo nodo con padre nodo
    si nodo.izq
     nodo.dch = n
    sino
     nodo.izq = n
    construir(arbolCodificcado[1..x], nodosFinales, n, raiz)
  si arbolCodificado[0] == 0
   si nodo == nil
    n = nuevo nodo con char
    construir(arbolCodificcado[1..x], nodosFinales, n, n)
   sino
    si nodo.izq y nodo.dch
     construir(arbolCodificcado[0..x], nodosFinales, nodo.padre, raiz)
    sino
     n = nuevo nodo con padre nodo y char
    si nodo.izq
     nodo.dch = n
    sino
     nodo.izq = n
    construir(arbolCodificcado[1..x], nodosFinales, nodo, raiz)
 sino
  devolver raiz
endFunc
```

Figure 4: *Algoritmo para obtener el árbol*