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 líneas exponen la estrategia seguida para su desarrollo así como las decisiones tomadas. El código ha sido desarrollado por el propio autor de este documento.

1. Construir el árbol

Para poder obtener los códigos de huffman de cada carácter del fichero de entrada es necesario almacenar el número de ocurrencias de estos, para, posteriormente, poder crear el árbol que permita reconstruir cada código para cada carácter. Para ello es necesario crear una nueva estructura de datos que almacene información relativa al árbol a crear, así como información relativa a los caracteres. Esta estructura recibe el nombre de *Node* y almacena el **peso** (número de ocurrencias en el fichero de entrada), el **carácter** y punteros a sus hijos **izquierdo** y **derecho**, así como a su nodo **padre**.

Una vez creados los objetos donde almacenar el árbol que nos permitirá obtener los códigos 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 caracteres de entrada. Se crearán nodos sin hijos para cada carácter con sus pesos que serán almacenados en un montículo (el coste del cambio de una lista al montículo será $O(n)$ [Pyt] que devuelve 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 Ecuación 1, el montículo tendrá solamente un objeto Nodo, dicho Dodo será la raíz del árbol que tendrá como hijos subárboles, que, recorridos recursivamente llegarán a las hojas finales, donde se encuentran los nodos que representan los caracteres.

2. Comprimir el fichero

Con el árbol almacenado en una estructura de datos, ahora, se tienen que obtener los códigos para cada carácter realizando un recorrido y asignando pesos de acuerdo a la rama seguida.

2.1. Obtener los códigos

Para obtener los códigos se tiene que dar un valor a cada rama seguida dependiendo de si conecta con el hijo derecho o izquierdo. Si está conecta con el hijo izquierdo el valor asignado es 0, si conecta con el hijo derecho su valor es 1. Para obtener el código de cada carácter habrá que recorrer de las hojas finales (nodo con un carácter almacenado) a raíz, almacenando los pesos que vayamos obteniendo que serán un número binario que es el código del carácter en cuestión. Para después poder acceder al código de cada carácter en $O(1)$ se almacena en una tabla hash de tal manera que las claves sean los caracteres y los valores los códigos generados.

2.2. Comprimir el mensaje

Para realizar la compresión del mensaje es necesario un bucle que recorra carácter a carácter la cadena de entrada dada por el fichero accediendo a la tabla hash que mapea los caracteres 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

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

imposible conocer los caracteres o volver a re-crearlo. Para la codificación del árbol se sigue un algoritmo sencillo que lo recorrerá recursivamente escribiendo un 0 si ese nodo es una hoja no final y un 1 si la hoja es final (representa un carácter) 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 y el mensaje (o fichero) comprimido es necesario añadir una cabecera de información que nos permita poder decodificar el árbol de manera correcta. Se añadirá, además de la cabecera, un byte al inicio que marcará el número de bits necesarios para completar el último byte del fichero comprimido, pues Python, no es capaz de escribir bits en ficheros, siendo necesario escribir como mínimo un carácter, es decir un byte. Además seguido a el primer byte con los bits para completar el fichero, se añadirá otro byte que marcará el número de nodos finales que tendrá nuestro árbol, para así poder hacer la decodificación correctamente. 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, el contenido entre [] representa 1 solo byte, en caso contrario podrá contener más de 1.*

2.5. Problemas de eficiencia

Dado el tiempo necesario para poder comprimir un fichero de 1MB de tamaño se realizaron mediciones, dividiendo el proceso de compresión en 3 subprocesos que son: crear el árbol, comprimir el texto y escribirlo, obteniendo tiempos cercanos al milisegundo (medidos en la máquina personal) en los dos primeros mientras que el tercero era cercano al minuto. Esto era debido al uso de una operación en Python con la siguiente sintaxis $vector[num:]$ cuya función es obtener el vector desde el indice *num* hasta su final, cuyo su coste asintótico es $O(k)$ siendo k el número de elementos entre *num* y el final haciendo que sea una solución ineficiente. Esta función se usaba para eliminar la parte del vector que ya se había usado y así avanzar para realizar cálculos con la siguiente, en cuenta se ha optado por evitar esta operación y sustituirla por índices que marquen el punto del vector en el que se han de realizar los cálculos.

3. Descomprimir el fichero

Para le descompresión del fichero se necesitará el mismo árbol que el usado para comprimirlo, el número de nodos finales de este y los bits finales usados para completar el último byte. Los pasos a realizar han de llevarse a cabo en el orden indicado.

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

3.1. Recodificar el árbol

El siguiente punto a realizar será volver a construir el árbol de tal manera que sea casi idéntico (solo varían los pesos y no se modifican los códigos) que el que se tenía para la compresión. Se leerá un byte para conocer el número de nodos finales del árbol, en este punto hay que tener en cuenta que se el fichero tiene 256 caracteres diferentes se escribirá un número con 9 bits, por ello se escribe el número de caracteres diferentes menos 1 y posteriormente se leerá el número más uno. Tras conocer el número de nodos finales del árbol se llamará a un algoritmo recursivo que, conociendo que el árbol se ha codificado dando prioridad a las ramas izquierdas, lo reconstruye. El algoritmo será el siguiente: Primero si no existen nodos se creará 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 raíz y avanzando uno en el vector de caracteres. 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 (carácter de la entrada 1 o 0, respectivamente), en caso de que no tenga hijos se creará el nodo y se añadirá 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 función recursiva se irá repitiendo hasta que el número de nodos finales sea cero que se devolverá la raíz.

Tras este algoritmo tendremos creado un árbol con la misma estructura que el de la compresión pero con distintos pesos, ya que estos no son de interés pues es posible recrear la estructura 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 caracteres 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 carácter. Ese carácter es el obtenido para los n bits leídos. 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, así 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.

3.3. Problemas de eficiencia

Al igual que con la compresión del fichero existen problemas con las funciones nativas de Python para manipular ficheros. De nuevo se realizan mediciones de tiempos en 3 subprocesos de la descompresión: formatear a binaria la entrada del fichero, construir el árbol y descomprimir. Mientras que el formateo es rápido, la creación del árbol y la descompresión se tratan de procesos más lentos, siendo necesario modificar las copias de parte del vector por índices que accedan a la zona necesaria para realizar los cálculos, reduciendo drásticamente los tiempos de ejecución.

4. Resultados obtenidos

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

En la 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 compresión 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 compresión similares, pues como se menciona anteriormente son textos corrientes sin repeticiones de caracteres 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 caracteres repetidos, lo que provoca una gran penalización de la compresión al guardar la cabecera haciendo que el resultado sea peor que el fichero inicial.

Test	original	comprimido	porcentaje	tiempoC	tiempoD
test1	69	65	0.9420	0.0008511	0.000729
test2	2850	1382	0.4849	0.003133	0.011094
test3	21040	11925	0.5667	0.01603	0.048255
test4	252723	134484	0.5321	0.1131	0.49135
test5	152	181	1.1907	00.00114	0.0008289
test6	552721	302833	0.5478	0.2454	1.0759

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

References

[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

5. 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*