Memoria Práctica 1

Algoritmia básica

David Jiménez Omeñaca (825068)
Carlos Giralt Fuixench (831274)

Escuela de
Ingeniería y Arquitectura
Universidad Zaragoza

Ingeniería Informática
Universidad de Zaragoza
Zaragoza,España
2023-2024

Índice

1. Introducción

La presente memoria detalla el planteamiento, la metodología, la implementación y el análisis de resultados obtenidos durante la realización de la primera práctica de la asignatura Algoritmia Básica. Su objetivo es la implementación del algoritmo de Huffman para la compresión y descompresión de ficheros, tanto de texto como binarios.

2. Metodología y recursos empleados

En primer lugar, se analizó el problema y se estudió en profundidad el algoritmo de Huffman, previamente expuesto en clase. Posteriormente, se planteó, en pseudocódigo, la solución al problema. Una vez satisfechos con nuestro planteamiento, pasamos a implementarlo. Para ello, decidimos usar el lenguaje de programación Python, pues su simplicidad de manejo se adaptaba de maravilla a la dimensión y complejidad del problema. Por otro lado, dispone de librerías muy útiles y cómodas para el tratamiento de ficheros binarios: funciones de lectura y volcado de datos, etc.

En cuanto a la codificación de los ficheros, se decidió incluir la información necesaria para la decodificación de los mismos en el propio fichero `.huf`. Para ello, decidimos almacenar la representación binaria del árbol obtenido mediante el algoritmo Huffman, pues la complejidad en cuanto a tiempo de descompresión es considerablemente inferior frente a la complejidad obtenida utilizando un diccionario cuyas parejas clave-valor contienen, para cada carácter del fichero, su representación binaria. Esto se debe a que el árbol, en sí mismo, es un autómata que permite, en tiempo lineal, encontrar la codificación original. También se precisaron bits de alineamiento ya que el algoritmo no asegura que los bits codificados sean múltiplos de 8. Para ello, se destinaron tres (3) bits para representar el número de ceros añadidos al final de la codificación.

Por otro lado, al almacenarse la codificación del árbol en el mismo fichero, es necesario reservar una serie de bits para indicar el tamaño del mismo. Tras realizar acotaciones del tamaño del árbol, se decidió reservar treinta y dos (32) bits para indicar el tamaño en **bytes** de la codificación. Nótese que con esta codificación se pueden asignar un máximo de 2^{32} bytes o 4GB.

Un diagrama del contenido del fichero se puede ver en A

Una vez implementada nuestra solución, planteamos un conjunto de casos de prueba para comprobar el correcto funcionamiento de nuestro código.

Finalmente, se realizó un análisis temporal de nuestra solución, para el cual se utilizó la herramienta **gnuplot**.

3. Pseudocódigo de la solución propuesta

Como se ha comentado anteriormente, se desarrolló pseudocódigo para las partes críticas del algoritmo. Dicho código puede verse en detalle en B.

4. Análisis temporal

Tras implementar los programas, se realizó un análisis temporal de cada función. Los órdenes de cada función se pueden consultar en el código fuente de la práctica. Se exponen a continuación las dos variables que se tendrán en cuenta:

- m, que es el número de caracteres **distintos** de la cadena a codificar. Esta es proporcional (por una constante) al número de nodos del árbol implementado. Además, como se está trabajando con codificación ASCII con caracteres imprimibles, se puede suponer que m es también una constante. Esto se debe a que, como hay una cantidad finita de caracteres, cualquier función creciente $f(m)$ cumple que:

$$f(m) \leq f(127) \quad \forall m \in [33, 127]$$

- n es el número de caracteres **totales** de la cadena. A diferencia de la anterior, esta no esta acotada, pues el tamaño de los ficheros es arbitrario. El orden de esta variable será la que realmente se analizará temporalmente.

A partir de esto, se pueden obtener los tiempos:

Función	**Orden de tiempo**
ParsearFichero	$\Theta(n \cdot m)$
Huffman	$\Theta(n \cdot m + m \cdot \log(m))$
CrearCodificacion	$\Theta(m)$
DecodificarCadenaConArbol	$\Theta(n)$

Cuadro 1: Tabla resumen de tiempos

Por lo tanto, podemos concluir que, una vez fijado el número de caracteres, ambos algoritmos (codificación y decodificación) son lineales en la longitud del fichero.

Para comprobar esto, se ha escrito un script de **Python** para medir el tiempo de ejecución de la codificación y decodificación. Además, dicho script se complementa con otro de **gnuplot** para poder crear y visualizar gráficas a partir de estos datos.

Las gráficas obtenidas de las pruebas temporales realizadas se pueden apreciar en C

De ellas, se puede concluir que:

- Como ya se había mencionado, la tendencia de ambas es lineal.
- Las constantes $O(n) = a \cdot m + b$ de la decodificación son, en general, más grandes, aumentando con el número de caracteres. Esto se debe a que la decodificación maneja cadenas de **bits**. De este modo, lo que antes era un carácter en la codificación, se puede convertir en múltiples. Por lo tanto, como se puede ver en las gráficas, las rectas se distanciarán más cuantos más caracteres se codifiquen.
- Los picos se pueden deber a un pico en las **prestaciones** del ordenador donde se realizaron las pruebas.

5. Resultados y conclusiones

En esta primera práctica, se han retomado técnicas usadas en la asignatura "Estructuras de datos y algoritmos", como la definición de tipos abstractos de datos (**TAD**) o el planteamiento de soluciones utilizando pseudocódigo. Los resultados obtenidos coinciden con lo esperado en cuanto a que los tiempos de codificación y decodificación crecen **linealmente** con el número de caracteres distintos en el caso de la codificación, y con el total de caracteres del fichero en el caso de la decodificación. Estamos muy satisfechos con el trabajo realizado, así como con su presentación y correspondiente análisis de los resultados.

A. Diagrama del contenido del fichero comprimido

Se muestra, a continuación, un esquema del contenido de los ficheros `.huf`:

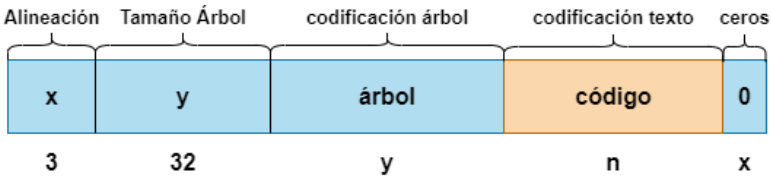

Figura 1: Codificación del archivo `.huf`

Nótese que la sobrecarga (S) viene dada por:

$$S = \frac{23 + x + y(n,m)}{23 + x + y(n,m) + n} = 1 - \frac{n}{23 + x + y(n,m) + n}$$

Por lo tanto, el algoritmo no es eficiente para n pequeños, mas cuando $n \longleftarrow \infty$, tenemos que la sobrecarga es nula. Recalcar que el tamaño del árbol también depende del fichero, mas como se ha comentado, está "parcialmente acotado".

B. Pseudocódigo de la solución propuesta

Se muestra, a continuación, el pseudocódigo de correspondiente a la solución propuesta:

Comenzamos con una función encargada de crear un diccionario (o mapa hash), que recoge los diferentes caracteres y su frecuencia de aparición en un fichero.

```
1  funcion ParsearFichero(F:fichero) devuelve (M:Mapa{caracter,entero})
2  variables M:Mapa{caracter,entero}, char:caracter
3  principio
4      M := crearVacio(M)
5      mientrasQue no finFichero(F) hacer
6          leer(char)
7          si esta(c,M) entonces
8              M[char] := M[char] + 1;
9          sino
10             anadir(M,{c,1})
11         fsi
12     fmq
13     devuelve M
14 fin
```

Utilizando esta función, diseñamos otra encargada de crear un árbol binario óptimo para la codificación de un fichero, utilizando el algoritmo de **Huffman**.

```
1  funcion Huffman(F:fichero) devuelve arbol
2  variables Q:colaPri; i,fx,fy,fz:entero; z,x,y,aux:arbol;
       M:Mapa{clave=>caracter,valor=>entero}
3  principio
4      crearVacia(Q)
5      M := ParsearFichero(F)
6      para todo {clave,valor} en M hacer
7          aux := creaArbol(raiz=>clave, hijoIzq=>nil, hijoDer=>nil)
8          insertar(Q,<aux,valor>)
9      fpara;
10
11     para i:=1 hasta |M|-1 hacer
12         <x,fx> := primero(Q) ; borra(Q);
13         <y,fy> := primero(Q) ; borra(Q);
14         fz := fy+fx
15         z := creaArbol(raiz=>fz,hijoIzq=>x,hijoDer=>y)
16         insertar(Q,<z,fz>)
17     fpara;
18     <z,fz> := primero(Q);borra(Q)
19     devuelve z
20 fin
```

Una vez tenemos el árbol, podemos codificar cadenas de texto. Para ello, vamos a asignar, a cada carácter, una cadena de bits. Con tal propósito, usamos una función recursiva que, al desplazarse por el árbol, asigna un 0 al hecho de pasar al hijo izquierdo de un nodo, y un 1 si se pasa al derecho.

```
1  funcion CrearCodificacion(a:arbol) devuelve
       (M:Mapa{clave=>caracter,valor=>cadena})
2  principio
3      devolver CrearCodificacionRec(a,"")
```

```
fin

funcion CrearCodificacionRec(a:arbol,c:cadena) devuelve
    (M:Mapa{clave=>caracter,valor=>cadena})
variables MIzq,MDer:Mapa{clave=>caracter,valor=>cadena
principio
    si arbol == nil entonces
        devolver {}
    sino_si esHoja(arbol) entonces
        devolver Mapa{arbol.raiz,c}
    sino
        MIzq = CrearCodificacionRec(a.hijoIzq,c+"0")
        MDer = CrearCodificacionRec(a.hijoDer,c+"1")
        devolver unirMapa(MDer,MIzq)
fin
```

Usando este diccionario, ya podríamos codificar nuestro fichero de texto. Nos centramos, ahora, en su decodificación. Para ello, creamos una función que, dada una cadena de ceros y unos y un árbol binario, devuelva la cadena decodificada. El funcionamiento es el siguiente, dada una cadena de ceros y unos, vamos recorriendo el árbol hasta encontrarnos con un nodo hoja. De este modo, el coste del algoritmo es **lineal**.

```
funcion decodificarCadenaConArbol(s:cadena, a:arbol) devuelve cadena
variables auxArbol:arbol, auxCadena:cadena
principio
    auxArbol = a
    para cada c en cadena:
        si esHoja(auxArbol) entonces
            auxCadena = auxCadena + a.raiz
            auxArbol = a
        fsi

        si c = "0" entonces
            auxArbol = auxArbol.hijoIzq
        sino
            auxArbol = auxArbol.hijoDer

    auxCadena = auxArbol + a.raiz

    devuelve auxCadena

fin
```

C. Gráficos obtenidos en el análisis temporal del algoritmo

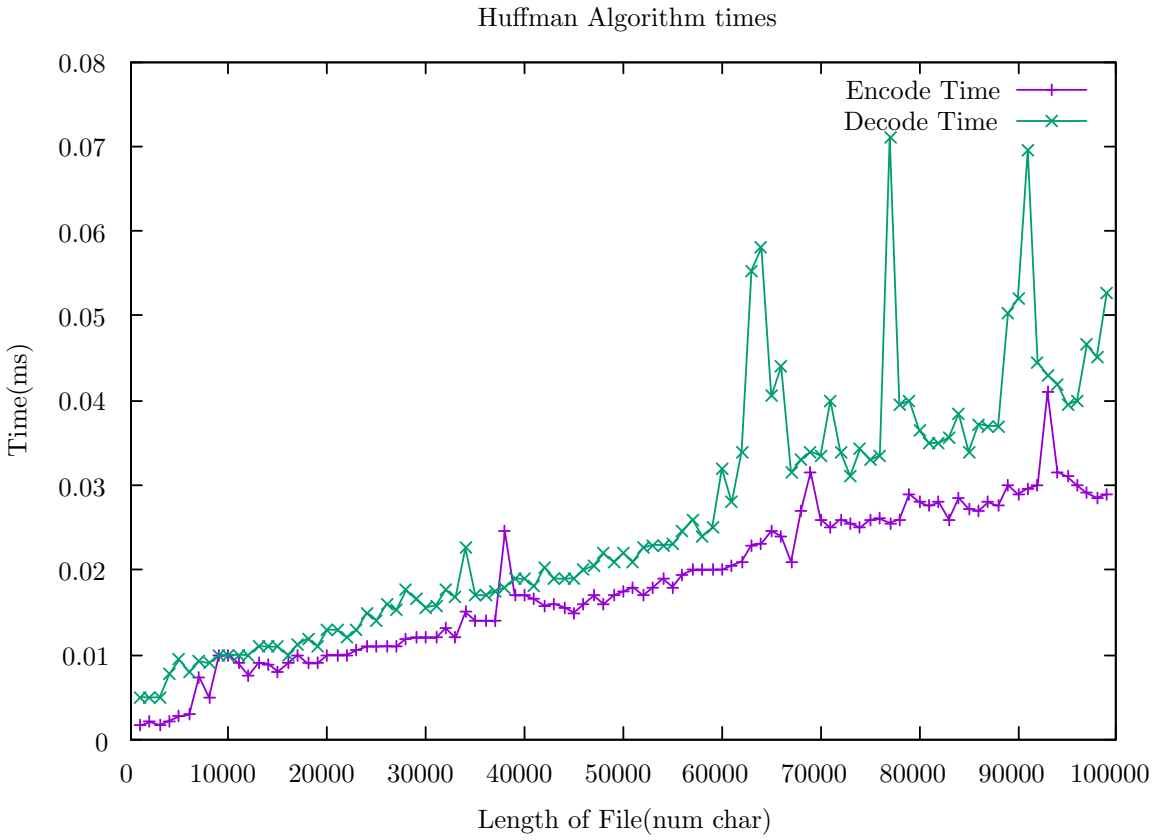

Figura 2: Prueba de tiempos con un solo carácter en el intervalo $[0, 10^5]$

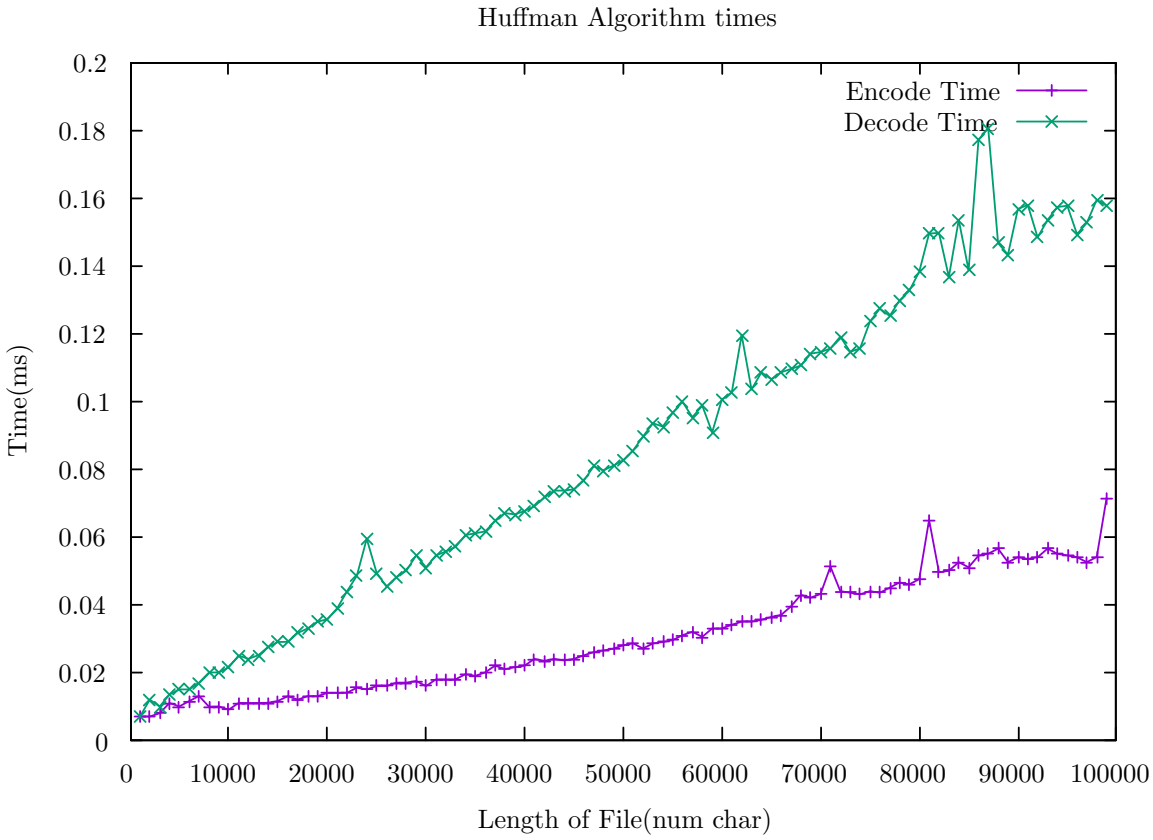

Figura 3: Prueba de tiempos con 93 caracteres en el intervalo $[0, 10^5]$.

D. Pruebas

A fin de comprobar el correcto funcionamiento de nuestro código, se diseñó una batería de pruebas. Dichas pruebas consisten en la compresión y descompresión de 4 ficheros de texto para, posteriormente, comparar el contenido de los ficheros originales con el contenido de los ficheros obtenidos tras la descompresión. A contiuación se detallan los contenidos de los ficheros de prueba:

1. **test0.txt:** fichero de texto vacío, que no contiene ningún caracter.
2. **test1.txt:** fichero de texto que contiene un único caracter: **a**.
3. **test2.txt:** fichero de texto que contiene una gran cantidad de apariciones de cada caracter ASCII (imprimible y no imprimible)
4. **test3.txt:** fichero de texto que contiene caracteres ASCII imprimibles. Aparecen letras en mayúscula y minúscula, así como saltos de línea.
5. **test4.txt:** fichero de texto que contiene un fragmento de una noticia sobre las declaraciones de Donald Trump acerca del cambio climático.

E. bitArray.py

Como parte del código entregado se encuentra el fichero `bitArray.py`, que contiene la definición e implementación de la clase **bitArray** para la manipulación y gestión de bytes y su correspondiente representación binaria. La implementación de dicha clase se inspira en el repositorio https://github.com/scott-griffiths/bitstring/issues.