ALGORITMIA BÁSICA

Práctica 1 - Código Huffman

Nicolás de Rivas Morillo (843740)
Cristina Embid Martínez (842102)

Resumen

En esta práctica se ha llevado a cabo el diseño e implementación de un algoritmo voraz que tiene como entrada un fichero, construye el árbol y compacta el fichero. Además también se realiza la descompactificación. La implementación está basada en el código de Huffman explicado en las clases de teoría. Hemos decidido utilizar C++ con un diseño orientado a objetos.

Estructura de los metadatos

Para decodificar hemos tenido que introducir en el fichero codificado una serie de metadatos que nos indicarán cómo reconstruir el árbol y hasta dónde interpretar como códigos huffman.

1 Byte
Número de códigos

1 Byte	1 Byte	Longitud Bytes
Original	Longitud	Código Huffman con relleno

…

1 Byte
Relleno final

El número de códigos debe caber en un byte, puesto que codificamos un lenguaje de (2^8) por lo tanto tendrá como máximo ese número de códigos, solo tenemos que tener en cuenta el caso de que todos los bytes posibles harán overflow del byte y volverá a 0, es decir, en este caso 0 = 256.
La longitud de código también debe caber en un byte puesto que la longitud máxima será (2^8-1).
Finalmente después de todos los códigos huffman encontramos el relleno final del texto (Para escribir bytes exactos) que es a lo sumo 7.
Otras estructuras de los metadatos son posibles pero de esta manera (guardando longitudes) podemos reducir la cantidad de bits de relleno que introducimos en total.

Análisis de las pruebas realizadas

En todas las pruebas realizadas el objetivo era que tras pasar por ambos procesos de nuestro programa (Codificación y decodificación) el fichero original fuese byte a byte igual al original. Además en las llamadas con el parámetro *-L* hemos comprobado la longitud de los códigos.

Prueba nº1:
La prueba más básica posible, el fichero contiene la cadena “abcd”. El objetivo de esta era detectar los errores más básicos que podíamos haber cometido.

Prueba nº2:
Otra prueba bastante sencilla pero en este caso con diferentes frecuencias de los carácteres. Con esta prueba buscamos errores básicos en el caso de frecuencias diferentes.

Prueba nº3:
En este caso la prueba contiene emojis de texto, que no tienen codificación ascii. Esta prueba tiene el objetivo de detectar fallos con bytes genéricos que no representan carácteres ascii.

Prueba nº4:
La siguiente se trata de un solo carácter en este caso 'ñ', repetido cuatro veces. El objetivo de esta prueba es captar errores al construir el árbol al tratar de solo un carácter y además que se trataba de la letra eñe.

Prueba nº5:
Esta prueba contiene todo el abecedario castellano, puesto que debemos asegurar que nuestro programa codifique todos los textos en castellano.

Prueba nº6:
La siguiente prueba se trata de un texto científico, esta es especialmente interesante puesto que (tal y como construimos el árbol) llega un punto que los códigos huffman se alinean formando el carácter EOF en mitad del archivo, lo cual no ayudó a detectar que no podemos finalizar la lectura cuando leemos un carácter EOF.

Prueba nº7:
En esta se trata de El Quijote, un caso límite que debíamos probar, es importante ya que contiene una gran cantidad de carácteres pero principalmente es un fichero muy extenso.

Prueba nº8:
Esta prueba es simplemente un fichero vacío, caso que tratamos como excepcional y dejamos el fichero codificado completamente vacío.

Prueba nº9:
Como indica el enunciado es un caso extremo, dos carácteres aparecen muchas veces y otros 6 (Incluyendo el EOF) aparecen únicamente una vez.

Prueba nº10:
El caso extremo inverso al anterior, dos carácteres aparecen una vez mientras que el resto aparecen muchas.

Prueba nº11:
Esta prueba contiene una aparición de cada byte posible, desde *0x00* hasta *0xFF*. El objetivo es probar todos los bytes posibles que va a poder leer nuestro sistema.

Análisis de tiempos y eficiencia

En cuanto a los tiempos, en casi todas nuestras pruebas son despreciables (menos de 10ms), la prueba más larga (prueba7) es la que más tarda, unos 2 segundos para decodificar y 0.25 segundos para codificar con un tamaño de 2141520 bytes.
En cuanto al espacio ahorrado, hay de todo, ya que algunas de las pruebas son las situaciones menos favorables para el algoritmo mientras que otras son la situación óptima. En un texto largo y escrito en castellano (Ejemplo de texto que se plantea como uso normal) vemos reducciones de un 50% como puede ser en la prueba7 (El Quijote).