Informe Práctica 1: Algoritmos Voraces

Jorge Soria Romeo
872016

Alberto Francisco Solaz García
873373

February 2025

1 Introducción

En este informe analizamos la eficiencia de los 3 algoritmos que hemos implementado durante esta práctica: el algoritmo de codificación Huffman, su correspondiente algoritmo de decodificación y una variante del primero que impone una restricción en la longitud (profundidad limitada) de los códigos generados. Nuestra elección de C++ para las prácticas de Algoritmia Básica se basa en su eficiencia, rendimiento y control por parte del programador sobre el uso de los recursos del sistema.

2 Algoritmo de codificación Huffman

El algoritmo de codificación se compone de 3 fases:

- **Construcción del diccionario de frecuencias:** `extraerFrecuenciaCaracteres` lee el fichero por bloques (de 4096 B) para minimizar lecturas. Cada carácter se trata individualmente por lo que tiene coste $O(n)$.
- **Construcción del árbol de Huffman:** `obtenerCodificacion` inserta en una min-heap nodos de cada carácter con su frecuencia a un coste de $O(\log k)$ por operación, donde k es el número de caracteres distintos. Entonces la combinación de nodos tiene coste $O(k \log k)$, siendo k pequeño en la práctica.
- **Generación de la salida codificada:** Volviendo a tener que tratar cada carácter individualmente, tiene coste $O(n)$.

Por lo tanto, la complejidad global de nuestro algoritmo de codificación Huffman es de:

$$O(n + k \log k)$$

donde:

n = número de caracteres en el fichero

k = número de símbolos en el fichero.

3 Algoritmo de decodificación Huffman

La decodificación se realiza en 2 fases:

- **Reconstrucción del árbol:** `leerArbol` reconstruye el árbol en preorden, visitando cada nodo una única vez con un coste de $O(k)$.
- **Decodificación del flujo de bits:** Nuestro algoritmo recorrde los bytes del fichero y, para cada bit, baja un nivel del árbol con una operación de comparación, con un coste de $O(n \times h)$, siendo h pequeño en la práctica.

Por lo tanto, la complejidad global de nuestro algoritmo de decodificación Huffman es de:

$$O(k + n \times h)$$

donde:

$$k = \text{número de símbolos en el fichero}$$

$$n = \text{número de caracteres en el fichero}$$

$$h = \text{altura del árbol de Huffman.}$$

4 Algoritmo de codificación Huffman con profundidad (longitud) limitada

- **Ajuste de frecuencias:** La función `limitar` extrae y ordena los pares {carácter, frecuencia} con un coste de $O(k \log k)$ y realiza ajustes (incrementos y decrementos) para cumplir la desigualdad de Kraft, con un coste de $O(k)$.
- **Construcción del árbol de Huffman con longitud limitada:** Aplicamos el algoritmo package-merge, que construye el árbol respetando la restricción de longitud, con un coste de $O(kL)$.
- **Generación de la salida codificada:** Finalmente, se recorre el fichero para generar la salida codificada, lo que tiene un coste de $O(n)$.

Por lo tanto, la complejidad global de nuestro algoritmo de codificación Huffman con longitud limitada es:

$$O(n + kL)$$

donde:

$$n = \text{número de caracteres en el fichero.}$$

$$k = \text{número de símbolos en el fichero}$$

$$L = \text{longitud máxima permitida para un código.}$$

5 Pruebas

Para probar la corrección y completitud de nuestros algoritmos (codificación estándar, codificación con profundidad limitada y decodificación) se ha usado una serie de ficheros de texto convencionales y ficheros binarios de prueba.

Se ha creado un *script* para *Bash* que compila los ficheros fuente y crea el ejecutable de nuestro programa. Posteriormente, realiza el siguiente proceso para cada fichero de prueba:

1. Obtiene el *hash* del fichero de prueba (concretamente *hash* SHA256).
2. Codifica (de manera estándar o limitada) el fichero de prueba.
3. Decodifica el fichero codificado y obtiene su *hash*.
4. Compara los *hashes*. Si coinciden el test ha resultado exitoso.

5.1 Descripción ficheros de prueba

- **test_corto01.txt** y **test_corto02.txt**: ficheros de prueba de extensión muy pequeña. En ambos no se disminuye el número de bytes porque la sobrecarga que implica codificar el árbol óptimo es más significativa que los bits que te ahorras.
- **test_largo01.txt** y **test_largo02.txt**: ficheros de extensión muy larga. El primer fichero es una traducción de la Biblia en inglés y el segundo el Quijote en español (incluye tildes, 'ñ', etc). En ambos se reduce su tamaño en bytes casí a la mitad (se reduce más de 1000000 bytes).
- **test_caracterEsFrecuente.txt**: fichero en el que uno o varios de sus caracteres tiene una frecuencia de aparición superior al resto de caracteres en el fichero.
- **test_caracterNoFrecuente.txt**: fichero en el que uno de sus caracteres tiene muy poca frecuencia de aparición repecto al resto de caracteres en el fichero.
- **test_numAparicionesParecidas.txt**: fichero en el que todos los caracteres tienen la misma frecuencia de aparición.
- **test_binario01.bin**: fichero binario estándar.
- **test_binario02.exe**: fichero binario compilado de programa "Hola, mundo" escrito en C++.

Además hemos probado la codificación limitada para ficheros como test_caracterEsFrecuente, test_caracterNoFrecuente o test_numAparicionesParecidas. La codificación limitada no siempre aumenta el tamaño de bytes que se reduce el fichero.

fichero	tamaño original (bytes)	tamaño comprimido (bytes)	decremento total (bytes)	tiempo para comprimir (ms)	tiempo para descomprimir (ms)
corto_01	29	66	-37	0,27	0,18
corto_02	51	64	-13	0,58	0,34
largo_01	2324344	1339313	985031	733,91	323,67
largo_02	2141519	1197821	943698	839,87	469,05
caracterEsFrecuente	100	46	54	0,38	0,27
caracterEsFrecuente (limitado)	100	48	52	0,43	0,31
caracterNoFrecuente	99	66	33	0,46	0,41
caracterNoFrecuente (limitado)	99	82	17	0,64	0,31
numAparicionesParecidas	95	50	45	0,60	0,33
numAparicionesParecidas (limitado)	95	55	40	0,63	0,33
binario01	100	384	-284	1,10	0,47
binario02	16392	5000	11392	8,25	1,89

Figure 1: Resultados