**Teoría de la información**

**Trabajo Práctico N°3: Compresión**

Juan Ignacio Basualdo

Juan Ignacio Olave

Introducción:

Para este trabajo, aplicamos los fundamentos aprendidos en clase para comprimir sin pérdida un archivo de texto en uno binario, utilizando algoritmos de Huffman y luego descomprimir para recuperar el archivo original.

Explicaremos a lo largo del informe paso a paso como nuestro programa logra comprimir y descomprimir y como logra calcular la entropía, la longitud media, la tasa de compresión, el rendimiento y la redundancia.

Dentro de la descripción paso a paso, habrá enlaces numerados (por ejemplo (1) o (2)) los cuales serán desarrollados aparte, en “explicación de elecciones”, para mantener simple la explicación del algoritmo.

Compresión:

Para comprimir el archivo, nuestro programa se basa en el algoritmo de Huffman, es decir, calculará la frecuencia de cada letra y a partir de esa información, creará el árbol de Huffman por caracter, para luego crear un diccionario y comprimir en archivo binario.

Se realiza de la siguiente manera:

1. Lectura del archivo de texto y obtención de frecuencias:

Se procesa el archivo, y se va leyendo caracter a caracter, por cada uno se aumenta las componentes de índice de valor del caracter en tabla ASCII de un vector de *short int*[*(1)*](#Variable_short_int)de 256 componentes inicializado en 0. Por ejemplo, si se lee el carácter “a”, se incrementa la componente 97 del vector en 1.

El resultado luego de procesar el archivo de texto, es un vector con valor 0 en cada letra que no salió, y la cantidad de apariciones en las que, si salieron, obteniendo un vector de acceso directo por caracter para saber la frecuencia de cada uno, es decir, si queremos saber cuántas veces salió la “a”, consultamos el valor de abecedario[a].

1. Creación del árbol de Huffman:

Una vez obtenido el abecedario, se crea un nodo de Huffman por cada carácter que haya salido en el archivo de texto, el cual tendrá 4 componentes: Su probabilidad de aparición (frecuencia de letra / frecuencia total), su caracter (letra “a”), y dos punteros a nodo que será inicialmente NULL. Todos los nodos son almacenados en un vector ordenado por probabilidad de aparición, de mayor a menor.

Se crea un nuevo nodo que apunta a los dos últimos del vector, teniendo como frecuencia la suma de los dos nodos hijos, se agrega al vector, se sacan los hijos y se reordena. Se repite este proceso hasta tener solo 1 nodo en el vector. El nodo resultante es **el árbol de Huffman.**

1. Creación de la tabla de Huffman:

Se crea una tabla de 2 columnas (carácter y código), y tantas filas como caracteres hayan aparecido en el archivo de texto.

Se recorre el árbol de Huffman, completando la tabla, si se avanza por el hijo izquierdo se agrega un 0 al código, si avanza por el hijo derecho se agrega un 1, si no hay hijos, se agrega el carácter a la fila de la tabla y se empieza de nuevo, recorriendo todo el árbol, agregando todos los caracteres y su código a la tabla.

Es importante mencionar que, en la tabla, el código es un *string* de caracteres, y cada 1 y 0 corresponde a un carácter (cada 1 y 0 no son bits, sino bytes) [(2)](#explicacion_string).

1. Creación del archivo comprimido:

Lo primero que se agrega al archivo comprimido es un *header* de 512 bytes, el cual es el primer vector obtenido, el abecedario de *short int* de 256 componentes que contiene la frecuencia de cada letra [(3)](#explicacion_header).

Luego se va leyendo el archivo de texto original, buscando el carácter leído en la tabla de Huffman, se obtiene el código que corresponde y se va agregando de a 1 bit en un buffer de 1 byte de tamaño. Al completar los 8 bits en el buffer, se escribe en el archivo comprimido y se sigue hasta terminar de leer el archivo de texto.

Descompresión:

1. Recuperación de la tabla de Huffman:

Se lee el *header de 512 bytes* y se obtiene el vector abecedario que se había creado cuando se había analizado el archivo de texto. Con el abecedario, se crea el árbol y tabla de Huffman de igual manera que fue creado en los pasos 2 y 3 de la compresión.

1. Lectura del archivo comprimido y creación del archivo de texto original:

Se tiene un buffer de 32 bits (4), el cual se carga el primer bit del archivo binario luego

del *header* y se verifica si existe ese código en la tabla de Huffman, si no, se agrega el siguiente bit y se vuelve a verificar. Una vez encontrado una coincidencia, se escribe el caracter en el archivo de texto y se reinicia el buffer, esto es posible debido a que el código es instantáneo. Se repite este proceso hasta leer todo el archivo binario.

Análisis de archivos:

**Tasa de compresión:**

Para calcular la tasa de compresión, se hace la comparación de ambos archivos de la siguiente manera:

Según la teoría de la información, la compresión de texto sin pérdida es alrededor del 50%, es decir que se espera una tasa de compresión de alrededor de 0.50. En nuestro programa se obtuvo 0.53 para la mayoría de los archivos.

**Rendimiento:**

Según el primer teorema de Shannon, la longitud media del mejor código compacto e instantáneo debe ser igual o muy cercano al valor de la entropía del archivo, por lo que el rendimiento se calcula como la relación entre la entropía y la longitud media:

Lo ideal es que la relación esté cercana al 100%. En nuestro programa, el rendimiento nos dio entre 97% y 99%.

**Redundancia:**

La redundancia es el opuesto del rendimiento:

Lo ideal es que la relación esté cercana al 0%. En nuestro programa, el rendimiento nos dio entre 1% y 3%.

Explicación de elecciones:

(1) La variable utilizada es de tipo *short int* ya que los archivos de texto provistos por la cátedra tienen una frecuencia máxima de caracteres de alrededor de 12000, por lo que una variable de 16 bits sobra para representarlas. El vector que almacena el abecedario, irá luego incluido como *header* en el archivo comprimido, por lo tanto, debía ser lo más chico posible.

En caso que el programa escale y se analicen archivos de textos mucho mas extensos, se puede modificar la variable para almacenar más valores.

(2) Se podría haber utilizado bits para representar cada letra, pero en este caso tendríamos que haber usando tablas de longitud variable, lo cual incrementaba mucho la creación del programa. Por otro lado, la tabla es utilizada solo como intermediaria y se puede crear a partir del vector abecedario (*header*), por lo que no hay ningún motivo por el cual pensar en ahorrar espacio.

(3) Elegimos usar este vector de header ya que ocupa siempre 512 bytes, mientras que el árbol de Huffman y la tabla ocupan mucho mas al ser estructuras complejas. Se podría haber creado la tabla de Huffman con el código en binario en vez de String (mencionado en (2)), pero esto dificulta mucho la creación del código del programa, además de hacer el header de longitud variable, también dificultando la lectura/escritura.