Teoría de la Información

Trabajo Práctico Especial 2

UNIVERSIDAD NACIONAL DEL CENTRO DE LA PROVINCIA DE BUENOS AIRES
FACULTAD DE CIENCIAS EXACTAS

Grupo 13:

- Schuenemann, Tomas Ezequiel. (tschuenemann@alumnos.exa.unicen.edu.ar) LU: 249222
- Viduzzi, Franco. (fviduzzi@alumnos.exa.unicen.edu.ar) LU: 249942

FACULTAD DE CIENCIAS EXACTAS
UNIVERSIDAD NACIONAL DEL CENTRO DE LA PROVINCIA DE BUENOS AIRES

Ejercicio - Trabajo Práctico Especial

Introducción:

Para este informe que trata sobre la compresión de archivos por el método de Huffman semi-estático, primero mostraremos los dos archivos donde se muestra el signo, la frecuencia con la que aparece y su codificación mediante el algoritmo.
Luego vamos a comparar los resultados obtenidos con el archivo original para después concluir sobre estos mismos.

Enunciado:

1) Se dispone de dos señales de audio almacenadas en archivos de texto plano (link1 y link2). Se requiere implementar computacionalmente los siguientes ítems y obtener para cada señal:

a) la distribución de probabilidades de los valores de la señal (considerada como una fuente sin memoria) y su codificación (el conjunto de códigos asociados a los valores de la señal) según el método de Huffman. Almacenarlos en un archivo.
b) la longitud total en bits de la señal codificada con Huffman y comparar respecto del tamaño del archivo original.
c) el rendimiento del código de Huffman obtenido en a) respecto del valor de la entropía de la señal. Explicar de qué manera se podría mejorar dicho rendimiento.

a)_

Los archivos con la probabilidad y la representación dado el Algoritmo de Huffman se encuentran en los siguientes enlaces:

Enlace 1: https://drive.google.com/file/d/1pCYQM3IyHM6ESwl26aESvmuBxEuq7jV3/view?usp=sharing

Enlace 2: https://drive.google.com/file/d/1_0kKzp32AZ1dY-uF2zRj9vDj39eoNJSk/view?usp=sharing

También se puede ver el código en Java en el siguiente enlace a replit:

Enlace 3: https://replit.com/@Tomas-Ezequiel1/TE2SchuenemannViduzzi#GestorArchivos.java

Mirando los resultados vemos que tenemos probabilidades bastante cercanas entre sí, y sumado a la enorme cantidad de símbolos esto hace que se tengan que usar varios bits para codificar el símbolo con más probabilidad y solo unos pocos más para el de menor probabilidad. Esta observación se puede ver en el archivo Beethoven que el símbolo más probable (18640 con proba 5/1000) ocupa 8 bits mientras que el menos probable (58870 con proba 1/1000) ocupa 10 bits, y también en el archivo L-gante el símbolo con mayor probabilidad (590 con proba 41/1000) ocupa 5 bits y el de menor (1580 con proba 1/1000) ocupa 10 bits, aquí se aprecia mejor la diferencia de probabilidades. De esta observación interpretamos que las longitudes medias serán cercanas a la entropía y que esta última va a ser un valor relativamente grande, pero el rendimiento del archivo Beethoven será mayor al de L-gante debido a la menor diferencia entre probabilidades y la entropía de Beethoven va a ser más grande ya que la cantidad de símbolos es muy superior.

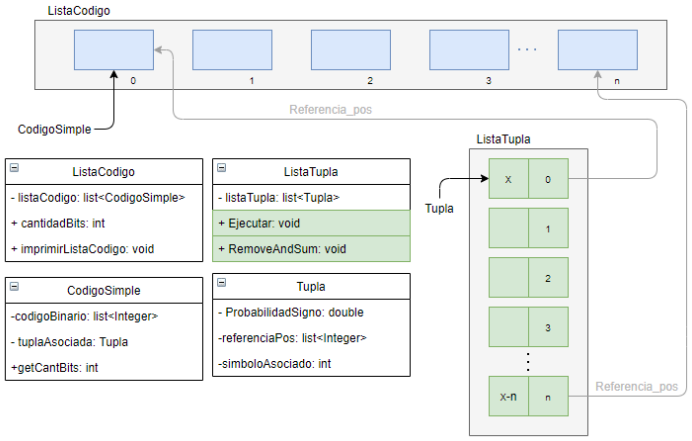
Explicación Pseudocódigo:

Primero generamos, desde el archivo, una matriz de 2xn, siendo n la cantidad de símbolos distintos, una fila para guardar el símbolo y otra para guardar la cantidad de apariciones de dicho símbolo. Para calcular n, recorremos el archivo línea por línea, y en una lista agregamos cada símbolo que no esté repetido junto con la cantidad de apariciones, con lo cual ya tenemos la distribución de probabilidades.

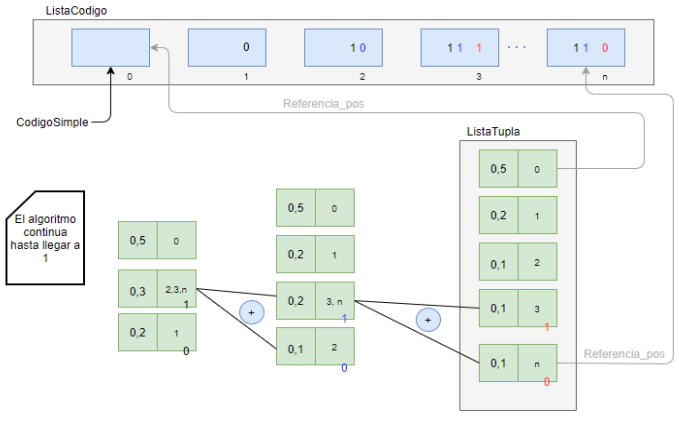
A partir de la matriz con los símbolos, generamos una lista de tuplas con una tupla por cada símbolo de la matriz. Esta tupla consiste de un símbolo, su probabilidad y una lista de referenciados, esta lista guarda las posiciones de *ListaCodigo,* donde se encuentra *CodigoSimple.* Ordenamos las tuplas por probabilidad y luego por cada tupla generamos una lista para guardar el código binario. Ahora por cada tupla generada, tomamos las dos últimas (ya que están ordenadas de mayor a menor), de la primera tupla, le agrego un bit 0 a cada referenciado, y de la segunda tupla, agrego un 1 a cada referenciado. Sumamos las probabilidades de las dos tuplas y creamos una nueva tupla a partir de ese valor. A esa nueva tupla le agregamos como referencia las posiciones que está referenciado en *ListaCodigo*, a medida que el algoritmo se ejecuta las posiciones en las nuevas tuplas se van acumulando. Luego removemos las tuplas de la lista. Este procedimiento se va a repetir hasta que quede solo una tupla en la lista, con lo cual para obtener el Código de Huffman se deberá recorrer

la lista de códigos generada por cada tupla. (*ejecutar()*)

(Estructura del código)



(Ejemplo de ejecución)



En el ejemplo de ejecución podemos ver 2 iteraciones, para entender cómo es que se acumulan las “Referencia_Pos” para luego ir propagando los 0 y 1 en *ListaCodigo* y agregar el valor en *CodigoSimple.*

Pseudocódigo de Huffman:

```
List<Tupla> listTuplas; //Tuplas cargadas con simbolo, cantidad de apariciones y
referenciados, estos últimos
 // están inicialmente vacíos. Las tuplas están ordenadas por cantidad de apariciones
ejecutar(ListaCodigo listaCodigo){
    while (listTuplas.size() > 1)
          removeAndSum(listaCodigo);
}
removeAndSum(ListaCodigo listaCodigo){
    index = listTuplas.size() -1;
    Tupla aux = listTuplas.get(index);
    Tupla aux2 = listTuplas.get(index-1);
    // Coloco 0 y 1 dependiendo de las posiciones referenciadas
    for (i = 0; i < aux.getReferenciaPos().size(); i++) {
        posCodigo = aux.getReferenciaPos().get(i);
        listaCodigo.agregarSimboloCodigo(posCodigo, 0);
    }
    for (int j = 0; j < aux2.getReferenciaPos().size(); j++) {
        posCodigo2 = aux2.getReferenciaPos().get(j);
        listaCodigo.agregarSimboloCodigo(posCodigo2, 1);
    }
    //Sumo las tuplas
    double sum = aux.getProbabilidadSigno() + aux2.getProbabilidadSigno();
    //creo una nueva tupla
    Tupla nuevaTupla = new Tupla(sum);
    // Agrego las nuevas pociciones a las tuplas
    nuevaTupla.addPosicionesTupla(aux);
    nuevaTupla.addPosicionesTupla(aux2);
    listTuplas.remove(index);
    listTuplas.remove(index-1);
    // Agrego la nueva tupla de manera ordenada, con esto no altero la consistencia
    insertarOrdenado(nuevaTupla);
}
```

b)_

Archivo L-gante

Tamaño original (en disco): 12 KB → 12288 bytes → 97824 bits

Tamaño comprimido: 6703 bits

→ 6703/97824 = 0.0685210 = 6.85% del tamaño del archivo original

Archivo Beethoven

Tamaño original (en disco): 12 KB → 12288 bytes → 97824 bits

Tamaño comprimido: 9591 bits

→ 9591/97284 = 0.0985876 = 9.85% del tamaño del archivo original

Para calcular el tamaño de archivo original analizamos lo que ocupa en disco, en un sistema operativo Windows 10, y luego lo pasamos a bits. En el caso del comprimido lo que hacemos es la sumatoria del producto de la frecuencia de cada símbolo y su longitud en codificación de Huffman. Por último hacemos el cociente entre los tamaños para ver qué cantidad de veces está el archivo comprimido en el original.

c)_ Rendimiento = Entropía / Longitud = H / L

Archivo L-gante
Entropia: 6.669716829526338
long Media: 6.702999999999994
→ *Rendimiento* = 0,9950345859364923

Archivo Beethoven
Entropia: 9.585063625675984
long Media: 9.590999999999871
→ *Rendimiento* = 0,9993810474065387

Como podemos observar, las Longitudes Medias se acercan bastante a la entropía esto es así ya que, como mencionamos anteriormente, las probabilidades están distribuidas casi uniformemente, por lo que la longitud promedio se acerca bastante al valor teórico promedio y dicho valor, en ambos casos, es relativamente grande. Además, la Entropía del archivo L-gante es mucho menor a la de Beethoven esto se debe a la cantidad de símbolos diferentes que hay en cada archivo, esto se traduce en tener que hacer más preguntas binarias para llegar a cada elemento.

Si quisiéramos acercarnos todavía más a la entropía lo que podemos hacer es extender la fuente, y como trabajamos a partir de una fuente sin memoria, si extendemos lo suficiente estas fuentes llegaríamos al valor teórico de la entropía, pero esto sería muy costoso.

Conclusión:

Pudimos notar que el tamaño del archivo original es más grande que el del archivo comprimido, esto se debe a que el sistema operativo asigna el espacio según dependiendo de la codificación de los símbolos en el archivo. Por lo que siempre va a ser a tender a ser mayor, esto nos da a entender que la representación de los números en el archivo.txt el S.O asigna la misma cantidad de bytes para representar.
Análogamente hablando del archivo comprimido, también depende de la cantidad de los símbolos que se encuentren en el archivo y su respectiva frecuencia de aparición, dado que si un símbolo aparece demasiadas veces según el algoritmo de Huffman no será necesaria una cantidad grande de bits, como se puede ver en los dos enlaces 1 y 2, los símbolos con más frecuencia tienen una longitud más corta de código. De esto concluimos que al aparecer símbolos con grandes frecuencias harían falta hacer menos preguntas binarias para determinar el símbolo que es.