PRÁCTICAS TXÓN. DATOS MULTIMEDIA

INTRODUCCIÓN A LA COMPRESIÓN: HUFFMAN Y ENTROPÍA

1. Introducción

El objetivo de esta práctica es doble, por un lado ser capaz de entender el concepto de entropía, y por otro conocer los fundamentos del algoritmo de codificación de datos Huffman. Para lograr estos objetivos, en primer lugar realizaremos una implementación del algoritmo de Huffman y del cálculo de la entropía, y en segundo lugar codificaremos diversos conjuntos de datos, observando los resultados para ver la relación existente entre el índice de compresión obtenido por Huffman y la estimación óptima ofrecida por la entropía.

Para el desarrollo de la práctica, se proporciona un código fuente incompleto, que contiene una implementación de Huffman que se deberá analizar y completar. Esta implementación se ha escrito en lenguaje ANSI C++, y la interacción con el usuario se realiza por medio de paso de comandos en una ventana de sistema, por lo que su compilación se puede realizar en cualquier entorno de desarrollo C++, tanto Windows como Linux.

2. Codificación Huffman

En compresión, buscamos ser capaces de representar un conjunto de símbolos (o cadena de símbolos) obtenidos a partir de un cierto alfabeto, usando el menor número de bits posible, pero preservando en todo momento la capacidad de descomprimir o decodificar la información. En general, el sistema que realiza el proceso directo lo llamamos compresor o codificador, mientras que el que reconstruye los datos originales (o una aproximación a ellos si realizamos compresión con pérdidas) lo llamamos descompresor o decodificador.

El algoritmo de codificación/compresión Huffman se propuso en 1952 como una forma sencilla y óptima de mapear cada símbolo de un alfabeto con un código (*codeword*) de longitud óptima. De esta forma, para comprimir cada símbolo de la cadena, simplemente debemos usar el código que se ha calculado mediante Huffman. Para conseguir esta asignación óptima, los símbolos se representan con códigos cuya longitud es inversamente proporcional a la probabilidad del símbolo. De esta forma, los símbolos menos probables se representan con códigos más largos, y los más probables con códigos más cortos.

El proceso de asignación de códigos se lleva a cabo mediante la construcción de un árbol binario, desde las hojas hacia la raíz, de manera que los nodos hoja son los símbolos del alfabeto. En la construcción del árbol, los nodos menos probables se unen sucesivamente para formar otro nodo de mayor probabilidad, de forma que cada uno de los enlaces añade un bit al código de los símbolos que estamos juntando. Este proceso termina cuando sólo se dispone de un nodo, de forma que éste representa la raíz del árbol.

Para mejorar su comprensión, veamos un ejemplo sobre este proceso. Supongamos que queremos codificar la siguiente cadena de símbolos:

$$S=\{aabaacc\}$$

que usa el alfabeto

$$A=\{a, b, c\}$$

La probabilidad de cada uno de los símbolos vendrá dada por las siguientes expresiones:

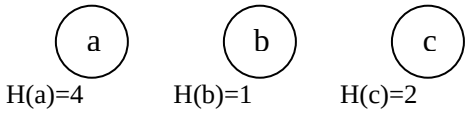
$$P(a)=4/7 \qquad P(b)=1/7 \qquad P(c)=2/7$$

Como es lógico, el símbolo “a”, que se repite mucho, nos interesa representarlo con el menor número de bits posibles.

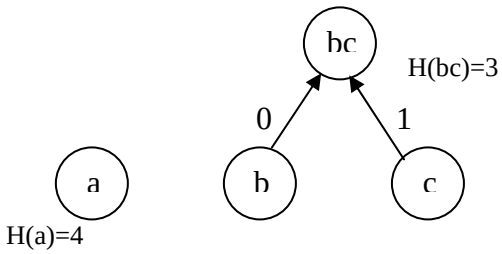
El primer paso del algoritmo será plantear un grafo no conexo de nodos que representan cada uno de los símbolos del alfabeto, junto con su probabilidad asociada. Para mejorar su comprensión y facilitar el cálculo, en lugar de usar directamente las probabilidades P(a), P(b), P(c), usaremos una cuenta de repeticiones, a modo de histograma, de forma que

$$H(a)=4 \qquad H(b)=1 \qquad H(c)=2$$

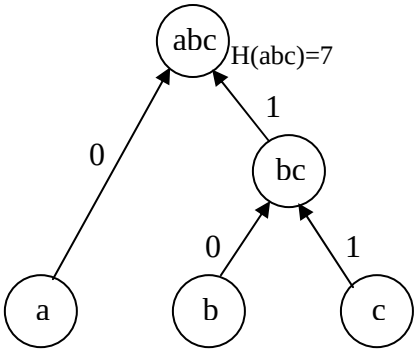
El grafo inicial será el siguiente:



El primer paso será juntar los nodos menos probables en un nuevo nodo, asignando un bit distinto a cada uno de los enlaces. El grafo resultante es el siguiente:



A continuación nos quedan dos nodos por unir, repetimos la misma operación y obtenemos ya el árbol final.



Para obtener los códigos a usar en la codificación, simplemente debemos recorrer el árbol de la raíz a cada una de las hojas, asignando a cada símbolo el código resultante de

unir las etiquetas asociadas a cada uno de los enlaces que se han recorrido. De esta forma, los códigos finales en el ejemplo son los siguientes:

C(a)=0 C(b)=10 C(c)=11

Y por tanto la cadena original S={aabaacc} quedaría codificada como sigue

C(aabaacc)= 0 0 10 0 0 11 11

El resultado final de la compresión es que hemos empleado 10 bits para codificar los 7 símbolos originales, así que se han usado 10/7 = 1,43 bits por símbolo.

Un inconveniente del proceso de decodificación Huffman es que es necesario disponer del árbol a partir del que se codifican los datos. Por lo tanto, no es suficiente con almacenar la cadena final, sino que también hay que comunicar al decodificador las probabilidades de la fuente (o el histograma asociado a los símbolos), de forma que el decodificador sea capaz de reconstruir el árbol (otra alternativa sería transmitir directamente la tabal de *codewords*). Este proceso no sería necesario si la fuente que estamos codificando tuviera unas características estadísticas bien conocidas a priori, tanto por el codificador como por el decodificador.

El decodificador, una vez ha reconstruido el árbol, puede decodificar la cadena original fácilmente. Para ello, debe recorrer el árbol desde la raíz hacia las hojas, usando los bits de la cadena codificada para avanzar en el recorrido hacia las hojas. Así, a partir de C(aabaacc), el primer bit es un 0, lo que nos dará directamente el símbolo “a”, ya que se llega desde la raíz directamente a una de las hojas. Con el siguiente 0 ocurre lo mismo. A continuación partimos nuevamente de la raíz, pero tenemos un 1, con lo que nos situamos en el nodo “bc”, y necesitamos leer un bit más, en este caso un 0, para llegar a una hoja, y decodificar así el símbolo “b”. Este proceso continúa mientras tenemos bits que leer.

3. Entropía

La entropía es un concepto que representa los límites de la codificación basada en la entropía, en la que se codifican los datos sin necesidad de conocer la naturaleza de los mismos. Huffman es un ejemplo de codificación basada en la entropía. La entropía denota el mínimo número de bits por símbolo necesarios para representar una cadena. Es un índice que denota la cantidad de información que existe en una fuente de datos (la cadena a codificar).

Aunque el cálculo de la entropía general no se puede calcular, de forma práctica se suele emplear la entropía de primer orden como una aproximación. Esta entropía viene definida como sigue:

$$H = \sum_{a_i \in A} P(a_i) \log_2 \frac{1}{P(a_i)}$$

siendo a_i cada uno de los símbolos del alfabeto A.

En el ejemplo del punto anterior, la entropía se puede calcular como

$$H = P(a)\log_2\frac{1}{P(a)} + P(b)\log_2\frac{1}{P(b)} + P(c)\log_2\frac{1}{P(c)}$$
$$H = \frac{4}{7}\log_2\frac{7}{4} + \frac{1}{7}\log_2 7 + \frac{2}{7}\log_2\frac{7}{2} = 1{,}38$$

Fíjate que aunque la entropía nos dice que los datos se pueden codificar usando 1,38 bits por símbolo, el resultado final usando Huffman es que han sido necesarios 1,43 bits por símbolo, ya que la entropía indica una cota inferior de los bits por símbolo necesarios.

4. Código fuente

El programa que implementa Huffman está formado por cuatro módulos/ficheros principales:

- codificador.cpp: obtiene los datos del usuario por parámetro (o si no se indica ningún dato, se asignan valores por defecto) y llama a las rutinas necesarias para realizar la codificación Huffman.
- Huffman.cpp: este módulo es el encargado de implementar Huffman.
- Histograma.cpp: sirve para realizar una cuenta de símbolos en la cadena original, y calcular así las probabilidades necesarias.
- FichBits.cpp: es un sencillo módulo para leer/escribir en disco bits sueltos, o palabras de cualquier longitud.

Veamos cada uno de estos módulos con algo más de detalle.

4.1 FichBits.cpp

Este módulo se ofrece totalmente implementado. El codificador lo usará para escribir en fichero cada uno de los códigos que genere. Para esto, previamente se debe inicializar pasándole el nombre del fichero que debe crear, por medio de la función

```
int InicializaEscritura(char *nombre);
```

Posteriormente, cada una de las palabras generadas se pueden guardar en disco por medio de la función

```
void EscribePalabra(int nbits, int palabra);
```

a la que se le pasa como primer parámetro la longitud del código, y como segundo parámetro el código a almacenar. Como alternativa, si sólo deseamos escribir un bit se puede usar la función `void EscribeBit(int bit);`

Finalmente, se debe finalizar el proceso, cerrando así el fichero creado y liberando los recursos ocupados, por medio de una llamada a la función

```
void FinalizaEscritura();
```

Para realizar el proceso de lectura del fichero, que será necesario en la decodificación, usaremos las funciones `int InicializaLectura(char *nombre)` y `void`

`FinalizaLectura()` para inicializar y terminar el proceso de lectura, y la función `int LeeBit()`, que devuelve un bit leído del fichero.

4.2 Histograma.cpp

Este módulo está formado por tres variables globales, y siete rutinas que operan sobre ellas. Las variables son:

- `unsigned int *Histograma`: Vector que contiene el histograma. Es una cuenta de símbolos, de forma que en la posición 0 se encuentra el número de veces que aparece el primer símbolo del alfabeto (la 'a'), en la posición 1 tenemos las veces que aparece el segundo símbolo ('b'), etc.
- `int longCadena`: Indica la longitud de la cadena a codificar.
- `int tamAlfabeto`: Indica el tamaño del alfabeto (por defecto, nuestro alfabeto será desde la 'a' hasta la 'e', así que será de cinco símbolos distintos).

De las rutinas podemos destacar:

- `void InicializaHistograma(int longitudAlfabeto, int longitudCadena, char *cadena)`: Sirve para crear el histograma, y calcularlo a partir de los datos que se obtienen por el tercer parámetro (cadena), cuya longitud viene indicada por el segundo parámetro (longitudCadena).
- `int LeeHistograma(int pos)`: Con esta función obtenemos el número de veces que aparece el símbolo "`pos`" en la cadena. La usaremos para realizar la construcción del grafo inicial para el árbol de Huffman.
- `float CalculaEntropia()`: Calcula la entropía a partir de la información en el histograma (variable `Histograma`)

El resto de rutinas de este módulo pueden ser consultadas en el fichero fuente, y se usan para finalizar el histograma (liberar recursos), para guardar el histograma en disco, para leerlo a partir del disco, y para imprimir el histograma por pantalla.

4.3 Huffman.cpp

Es aquí donde se realiza la implementación del codificador y decodificador Huffman.

4.3.1 Construcción del árbol Huffman

Una parte fundamental del proceso de codificación y decodificación es la construcción del árbol, que nos va a permitir obtener los códigos Huffman en el codificador, y decodificar la cadena original siguiendo el árbol en el decodificador. Para la construcción y el recorrido de este árbol, hemos definido en Huffman.h la estructura de tipo `TNodo`, que representa un nodo del árbol. Su definición es la siguiente:

```
typedef struct nodo
{
  struct nodo *infDer;
  struct nodo *infIzq;
  struct nodo *der;
  unsigned int cuenta;
  unsigned char simbolo;
```

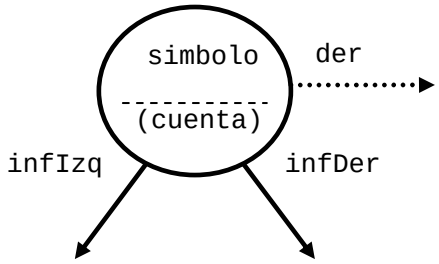
```
} TNodo;
```

La información sobre el propio nodo viene dada por los atributos `cuenta` y `simbolo`, que representan el número de veces que aparecen los símbolos representados por ese nodo en la cadena a codificar (es decir, la probabilidad de ese nodo), y el propio símbolo que se representa. Fíjate que el campo símbolo sólo es realmente necesario cuando el nodo representa a una hoja, sin embargo la cuenta debe actualizarse continuamente.

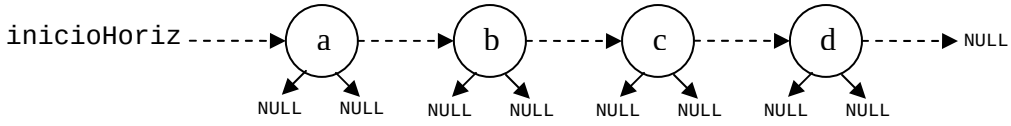
Los atributos `infDer` e `infIzq` se usan para indicar cuáles son los nodos hijo derecho e hijo izquierdo del nodo actual, formando de esta forma el árbol. Observa que estos atributos para los nodos hoja apuntarán siempre a NULL.

El último de los atributos, `der`, se emplea durante la construcción del árbol, manteniendo una lista enlazada horizontal que indica cuáles son los nodos que se pueden juntar para seguir construyendo el árbol. Para entender el objetivo de esta lista será necesario recordar en algo más de detalle la construcción de los árboles. Recuerda que inicialmente el grafo está formado por nodos independientes. A continuación, juntamos los dos nodos menos probables del grafo y formamos así un nuevo árbol. A partir de este momento, los nodos que se encuentran en las hojas de este árbol ya no son susceptibles de ser juntados, sino que únicamente se considera el nodo raíz de este nuevo árbol. De esta forma, fíjate que estamos juntando los nodos raíz de los árboles para hacer árboles mayores. Por tanto, es necesario mantener una lista de cuáles son los nodos raíz de los árboles, y que son susceptibles de ser unidos para crear árboles mayores.

A continuación veremos un ejemplo gráfico para facilitar la comprensión de estas estructuras. Un nodo aislado lo vamos a representar gráficamente como sigue:

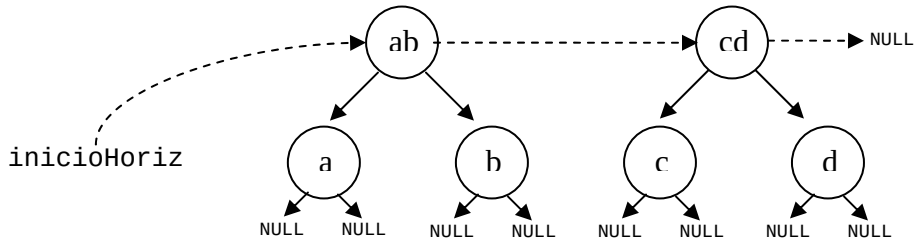


Un ejemplo de situación inicial, con un alfabeto de cuatro símbolos, sería la siguiente:

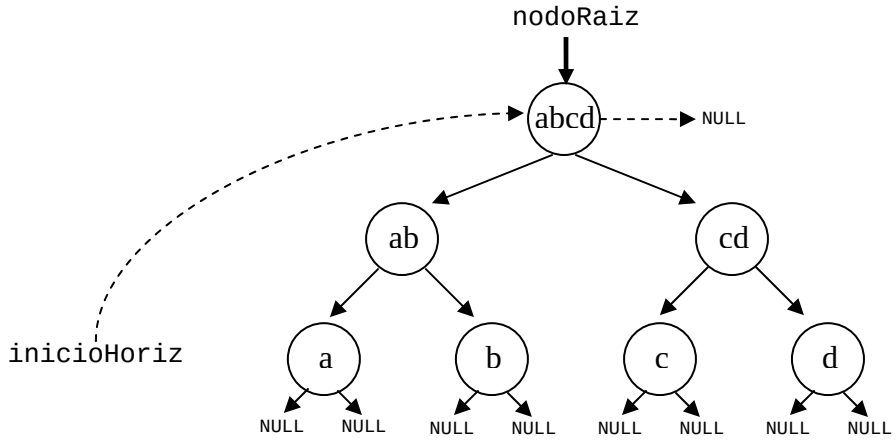


Observa como todos los nodos inicialmente forman árboles aislados, y sus punteros a descendientes apuntan a NULL. Además se ha formado la lista horizontal (con el puntero `der`) para identificar la raíz de cada árbol. En esta lista, el último de los nodos apuntaría a NULL. Para su localización y manipulación, es necesario indicar cuál es el primer elemento de esta lista. Este primer elemento vendrá identificado por la variable global de Huffman.cpp, `inicioHoriz`.

A continuación mostramos un ejemplo del estado del árbol a medio construir



La lista horizontal sigue apuntando a la raíz de los subárboles que se han construido (en este caso dos). Finalmente, el árbol quedaría como sigue



Este último ejemplo nos sirve para introducir otra variable global del módulo Huffman.cpp, la variable `nodoRaiz`, que debe apuntar a la raíz del árbol de Huffman una vez construido. Este puntero será la forma que tenemos para referenciar el árbol para su posterior recorrido.

La función que se encarga de construir el árbol de Huffman es

```
TNodo *ConstruyeArbol()
```

Esta función tiene dos partes. En primer lugar se construye el grafo inicial, formado por un nodo por cada símbolo del alfabeto. En segundo lugar, se realiza el proceso de construcción del árbol. Para ello, se utiliza las funciones

```
TNodo *SacaMenor()
void JuntaNodo(TNodo *Nodo1, TNodo *Nodo2)
```

La primera de ellas saca el nodo con menor índice de aparición (por tanto, con menor probabilidad) de la lista horizontal, y devuelve un puntero a ese nodo. La segunda función añade un nuevo árbol a la lista horizontal, formado por la unión de los dos nodos pasados por parámetro (y de sus descendientes).

La construcción del árbol consistirá en sacar sucesivamente los dos nodos menores e ir juntándolos, hasta que no quede ningún nodo en la lista horizontal. Finalmente, la función debe devolver el nodo raíz del árbol construido.

Para liberar la memoria ocupada por el árbol, se usará la función `void DestruyeArbol(TNodo *Raiz)`, que recorre el árbol de manera recursiva, liberando la memoria ocupada por cada uno de sus nodos.

4.3.2 Construcción del vector de códigos

Una vez se ha formado el árbol, hay que usarlo para construir los códigos asociados a cada símbolo del alfabeto (la tabla de *codewords*), realizando un recorrido del propio árbol. Esto se ha implementado de manera recursiva, de la raíz a las hojas, por medio de la función:

```
void ConstruyeCodigos(TNodo *Raiz, unsigned int codigo,
                                   unsigned char longcodigo)
```

En el recorrido del árbol, esta función asigna un "1" a cada rama derecha y un "0" a cada rama izquierda (estos valores se pueden cambiar modificando los `#define RAMA_DER` y `RAMA_IZQ`). Una vez se llega a las hojas, se asigna al símbolo representado por esa hoja el código correspondiente, que se ha generado en el camino hacia la hoja.

Después de ejecutar esta función, los códigos quedan almacenados en el vector `unsigned int *codigos`, que está definido como variable global. En este vector, la posición cero mantiene el código del primer símbolo ('a'), la posición uno la del segundo, etc. Para saber cual es la longitud de cada uno de los códigos, es necesario consultar otro vector, que se define como `unsigned int *longCodigos`.

4.3.3 Codificación

Antes de poder usar Huffman para codificar los datos, es necesario llamar a la función

```
int InicializaHuffmanCod(int longitudAlfabeto,
                  unsigned int longCadena, char *cadena, char *nombre)
```

indicando por parámetro (1) la longitud del alfabeto a usar, (2) la longitud de la cadena, (3) la propia cadena a codificar, y (4) el nombre con el fichero que va a contener la cadena codificada. Es importante destacar que la cadena a codificar debe contener un vector de bytes que representan con un 0 al primer símbolo ('a'), con un 1 al segundo símbolo ('b'), etc., así que previamente se debe haber normalizado la cadena usando la función `ConvierteCadena()`(ver módulo codificador.cpp).

Básicamente en la inicialización de Huffman: (a) se inicializa el fichero de escritura, (b) se calcula el histograma a partir de los datos de la cadena, (c) se construye el árbol de Huffman, (d) se guarda el histograma en disco para que el decodificador lo tenga disponible, (e) se construye la tabla de códigos (*codewords*) a usar en la codificación a partir del árbol, y (f) se libera la memoria usada por el árbol.

Una vez creada la tabla de códigos, para codificar un símbolo de la cadena es suficiente con llamar a la función

```
void CodificaSimbolo(int simbolo);
```

que escribe en disco el código asociado al símbolo que se desea codificar.

Finalmente, para terminar el proceso de codificación, y liberar los recursos asociados, hay que llamar a la función `void FinalizaHuffmanCod()`.

4.3.4 Decodificación

Como en el proceso de codificación, antes y después de empezar la decodificación hay que llamar a las funciones `InicializaHuffmanDecod(...)` (para reservar recursos y construir el árbol de Huffman) y `FinalizaHuffmanDecod()` (para liberar recursos).

Para decodificar cada símbolo, a partir de la información previamente codificada, que se lee progresivamente de disco, hay que usar la función

```
Int DecodificaSimbolo()
```

que debe recorrer el árbol desde la raíz a las hojas, usando los bits que se leen del fichero. Para realizar este proceso se puede usar la función auxiliar

```
TNodo *Descendiente(TNodo *nodo, unsigned char bit)
```

que devuelve el hijo izquierdo o derecho del nodo que se pasa por parámetro. Con el parámetro bit se especifica si se desea obtener el hijo izquierdo o el derecho.

4.3.5 Otras funciones

Finalmente, en el módulo Huffman.cpp, se definen las funciones `void ImprimeCodigos()` para imprimir por pantalla los códigos Huffman una vez generados, y `float CalculaBitsPorSimbolo()`, que devuelve el número de bits por símbolo que se han empleado en la codificación, usando para ello una cuenta de bits (variable global `int cuentaDeBits`) y una cuenta de símbolos (variable global `int cuentaDeSimbolos`).

4.4 codificador.cpp

Este es el módulo principal (*main*) del programa. En primer lugar se realiza la lectura de parámetros por consola, asignando valores por defecto si no se especifican (variables globales `cadenaPorDefecto, ficheroPorDefecto` y `opcionPorDefecto`).

Posteriormente, mediante la función `int ConvierteCadena(...)`, comprueba que todos los valores de la cadena a codificar estén dentro del alfabeto que se está usando, que viene definido entre ‘a’ y ‘e’ (este último valor se puede cambiar modificando `MAX_SIMBOLO`), y convierte la cadena a codificar, de una representación en código ASCII a una representación más adecuada para el módulo Huffman, en la que la ‘a’ se representa con un 0, la ‘b’ con un 1, etc.

Después, si la opción seleccionada es “codificar”, se inicializa la codificación Huffman, codificando posteriormente uno a uno los símbolos de la cadena de entrada, de la siguiente forma

```
for (f=0;f<nsimbolos;f++)
    CodificaSimbolo(cadena[f]);
```

Para, en último lugar, finalizar la codificación Huffman.

Si la opción es seleccionada es “decodificar”, se inicializa la decodificación, se decodifica la cadena, símbolo a símbolo, mostrándose por pantalla de la siguiente forma

```
for (f=0;f<nsimbolos;f++)
    printf("%c",DecodificaSimbolo()+'a');
```

y por último se finaliza la decodificación.

Finalmente, fíjate que existe una última opción que realiza ambos procesos secuencialmente: primero la codificación y luego la decodificación de los datos.

5. Instrucciones de uso del programa

Como hemos visto en el punto anterior, el programa dispone de una serie de parámetros por defecto, de forma que para probar su funcionamiento simplemente es necesario teclear el nombre del programa en una ventana del sistema y pulsar *enter*.

Por defecto, se usa la cadena que viene definida en la variable global `cadenaPorDefecto` de codificador.cpp, y se almacena en el fichero indicado en la variable `ficheroPorDefecto`. Sin embargo, si quedemos modificar algún parámetro de la codificación, se pueden usar las opciones “–i cadena” para indicar la cadena a codificar, y “–o nombreFichero” para indicar el fichero de salida. Además, en caso de querer usar alguna de las opciones anteriores, tendremos que indicar en primer lugar que queremos codificar símbolos, usando el modificador “–c”, de la siguiente forma:

codificador –c –i abcde –o nombre2.huf

Para la decodificación habría que usar “–d”, y en caso de no querer usar el nombre de fichero por defecto, usar “–i nombreFichero”, tal y como sigue

codificador –d –i nombre2.huf

Observa que se puede realizar ambos procesos de codificación y decodificación en la misma ejecución, con la opción “–a”.

6. Trabajo a realizar

El trabajo a realizar en esta práctica es el siguiente:

1) Leer la memoria y entender cómo funciona el programa para la codificación y decodificación de datos usando Huffman.

2) Implementar las funciones que se han dejado en blanco. En concreto, hay que completar las siguientes funciones:

- De codificador.cpp:

- `int ConvierteCadena (unsigned int longcadena, char *cadenaOriginal, char *cadena):` Esta es una función muy sencilla, pero necesaria para poder trabajar posteriormente con los símbolos de una manera más cómoda.
  Principalmente recibe como parámetro una cadena indicada en "`cadenaOriginal`", cuya longitud es "`longcadena`", y que está representada en código ASCII, y se debe pasar a una representación numérica, dejando el resultado de esta conversión en el parámetro "`cadena`". Será sencillo hacer la conversión restando a cada elemento de "`cadenaOriginal`" el carácter 'a'.
  Además, esta función devuelve "0" si la conversión se ha realizado con éxito, o "1" si algún elemento de la cadena a convertir está fuera del alfabeto (es decir, es menor que 'a' o mayor que el carácter definido en MAX_SIMBOLO).

- De Histograma.cpp:

  - `void InicializaHistograma(...):` En esta función, ya se encuentra implementada la parte para reservar memoria para el histograma. Lo siguiente que debes implementar es inicializar a cero todas las posiciones del histograma, para a continuación completarlo a partir de la información que viene dada en la "`cadena`" que se lee por parámetro, y que tiene de longitud "`longitudCadena`".
    Para poder probar sin problemas que el histograma se ha implementado correctamente, puedes añadir **temporalmente** al final de esta función las instrucciones `ImprimeHistograma();exit(0);`
    Es importante que recuerdes que, en este punto del programa, los símbolos del vector "`cadena`" ya están normalizados por la función "`ConvierteCadena()`".

  - `float CalculaEntropia():` A partir del histograma, y sabiendo que el número de símbolos totales en el histograma viene dado por la variable global `longCadena`, puedes obtener la probabilidad de cada símbolo del alfabeto, y por tanto su entropía, usando para ello la expresión anteriormente indicada.
    Nuevamente, para comprobar que su cálculo es correcto puedes añadir **temporalmente** las instrucciones

    ```
    printf("\nEntropia: %f\n",CalculaEntropia());exit(0);
    ```

    al final de la función `InicializaHistograma()`.
    Si estás usando como cadena por defecto la "aabaaacceaabbbadeee", el resultado debe ser 2.037401 bits/símbolo.

- De Huffman.cpp:

  - `TNodo *ConstruyeArbol():` En esta función se proporciona ya implementada la construcción del grafo inicial a partir de la información del histograma. La segunda parte de la función es la que hay que completar, y en ella se realiza la construcción del árbol. Para esto, hay

que obtener los dos nodos de menor probabilidad (usar la función `SacaMenor()`) y juntarlos en el árbol como un solo nodo (con la función `JuntaNodo()`).
Observa que si el segundo nodo extraído con `SacaMenor()` devuelve NULL, quiere decir que ya no hay más nodos a sacar. En ese momento hay que terminar la función, devolviendo mediante la instrucción de C `return` el primer nodo leído, que resulta ser la raíz del árbol.
Para comprobar que esta función está bien implementada, es suficiente con ejecutar el programa y comprobar que la tabla de códigos que se visualiza en pantalla es correcta.

- `void CodificaSimbolo(int simbolo)`: Llegados a este punto ya se ha construido el vector de códigos (`codigos`) con sus longitudes correspondiente (`longCodigos`). Resultará muy sencillo utilizar estos dos vectores para codificar un símbolo, utilizando para ello la función `EscribePalabra`(…) del módulo FichBits.h.
  Si está bien implementado, el fichero resultante de codificar la cadena por defecto tendrá un tamaño de 24 bytes (las probabilidades también se guardan en el fichero).

- `float CalculaBitsPorSimbolo()`: Esta función devuelve el resultado de la compresión (en bits por símbolos) como el resultado de la división (en coma flotante) de los atributos globales `cuentaDeBits` y `cuentaDeSimbolos` (recuerda que estas variables deben ser actualizadas en la función `CodificaSimbolo()`).
  Para probar la corrección de esta cuenta con la cadena por defecto, el resultado debe ser 2.105263 bits/símbolo.

- `int DecodificaSimbolo()`. Esta función debe recorrer el árbol desde el nodo raíz (indicado por la variable `nodoRaiz`) hasta las hojas. A partir de este nodo raíz, se puede ir descendiendo utilizando la función `Descendiente()` y `LeeBit()` de la siguiente forma

  ```
  Nodo=Descendiente(Nodo, LeeBit());
  ```

  En el momento en el que alguno de los hijos del nodo que estamos usando para recorrer el árbol sea NULL (el `Nodo->infIzq` o `Nodo->infDer`, de manera indistinta) no tendremos que seguir descendiendo, sino que ya habremos llegado a un nodo hoja, y podremos devolver el valor del atributo "`simbolo`" de esta hoja.
  Para comprobar la corrección de la decodificación, bastará con usar el programa para decodificar una cadena previamente codificada.

3) Una vez completada la implementación, es interesante ver el siguiente ejemplo. Codifica la cadena que hay por defecto, y compara los valores de la entropía y el resultado de la compresión por Huffman, ¿cuál es menor? ¿tiene sentido?.

   A continuación codifica la cadena "abcde" y anota los valores de la entropía y el resultado de la compresión. Posteriormente codifica la cadena "aaaaabcde" y anota nuevamente los anteriores valores. Codifica ahora "aa…aabcde", con un número total de "a"s de aproximadamente 50 y anota el resultado. Repite

nuevamente este proceso con 100 y con más “a”s. ¿Hacia que valor parece converger la entropía? ¿Y el resultado de Huffman? ¿Por qué crees que se da este resultado?

7. Posibles extensiones de la práctica

A continuación proponemos una serie de ampliaciones sobre la práctica. Recuerda que puedes completar una o varias de ellas, de manera que cuantas más ampliaciones hagas, mejor puntuación obtendrás.

1) *Realiza una implementación adaptativa de Huffman, y compárala con la versión original observando la tasa de compresión y tiempo de ejecución.*

   Las versiones adaptativas de los codificadores basados en la entropía son capaces de adaptarse dinámicamente a las propiedades estadísticas del conjunto de símbolos a codificar, de manera que incluso se puede desconocer a priori las probabilidades los símbolos que forman el alfabeto (evitando la formación inicial del histograma). Así, inicialmente todos los símbolos del alfabeto pueden tener la misma probabilidad de aparición, y a medida que vamos codificando los símbolos vamos aumentando las probabilidades de aquellos símbolos que van apareciendo.

   Una forma sencilla de implementar la adaptatividad en el programa de prácticas es empezar con una cuenta de símbolos igual a cero para todos los símbolos y calcular el árbol de Huffman inicial y sus *codewords* (lógicamente nos dará unos códigos de longitud similar para todos los símbolos). A partir de aquí, por cada símbolo que codifiquemos, incrementaremos en uno la cuenta asociada a ese símbolo, y recalcularemos el árbol de Huffman para actualizar los *codewords* según las nuevas estadísticas. Observa que así no nos hace falta guardar las estadísticas de los símbolos del alfabeto, ya que el decodificador la podrá generar dinámicamente de la misma forma que hizo el codificador.

2) *Implementa un codificador Huffman de símbolos agrupados*.

   Como hemos visto en esta práctica, una de las principales limitaciones de Huffman es que necesita asignar un número entero de bits a cada símbolo (y por tanto cada símbolo se codifica con al menos un bit). Esta limitación hace que la mínima tasa de bits por símbolo sea de 1, incluso cuando las probabilidades están concentradas en muy pocos símbolos. Para resolver esto, podemos codificar agrupaciones de símbolos en lugar de símbolos sueltos.

   Una primera opción de implementación puede ser codificar los símbolos por parejas, de forma que el alfabeto a usar quede definido como todos los posibles pares obtenidos de combinar dos símbolos del alfabeto inicial. Por ejemplo, si tenemos el siguiente alfabeto de tres símbolos, A={a, b, c}, el nuevo alfabeto combinado sería A={aa, ab, ac, ba, bb, bc, ca, cb, cc}, de forma que si uno de los símbolos iniciales fuera muy probable (por ejemplo el símbolo “a”), la repetición de ese mismo símbolo (en este caso “aa”) también sería muy probable, y se le podría asignar un único bit. De esta manera, si lo vemos desde el punto de vista del alfabeto inicial, podríamos conseguir una codificación del

símbolo probable (“a”) con tan sólo medio bit. De hecho, esta es la idea que subyace en la codificación aritmética.

En esta extensión de la práctica puedes probar a agrupar distinta cantidad de símbolos, y comparar el resultado con la versión original, viendo qué sucede al variar las características estadísticas de los símbolos del alfabeto (por ejemplo, concentrando las probabilidades en uno o dos símbolos, distribuyéndolas entre todos los símbolos del alfabeto, etc).

3) *Implementa una versión completa: adaptativa y con símbolos agrupados.*

Uno de los principales inconvenientes de la codificación agrupando símbolos es que aumenta el tamaño del alfabeto, y por tanto la cantidad de información estadística que el codificador debe comunicar al decodificador para que éste sea capaz de construir el árbol de Huffman. Para evitar la necesidad de transmitir estas estadísticas puedes realizar una versión adaptativa del codificador de símbolos agrupados, tal y como se ha explicado en la primera ampliación de prácticas.

Una vez implementadas las cuatro versiones (la original y las tres extensiones propuestas), sería interesante que las compararas entre ellas en términos de eficiencia en compresión y costes computacionales. Para ello puedes usar distintos tipos de datos de entrada, como un fichero de texto (en código ASCII) o información multimedia (p.ej., una imagen sin comprimir, en formato crudo). Ten en cuenta que el compresor que has implementado es sin pérdidas (*lossless*) y de carácter general, con lo que la tasa de compresión que obtendrás con información multimedia será mucho más baja que en compresores de imágenes específicos con pérdidas (*lossy*).

4) *Estudio y uso de una implementación eficiente de Huffman adaptativo.*

En http://www.cipr.rpi.edu/~said/FastAC.html puedes encontrar una implementación eficiente de Huffman adaptativo realizada por Amir Said (un experto de reconocido prestigio en el campo de la compresión de datos). En esta ampliación os proponemos estudiar esta implementación, y evaluarla, comparándola con la implementación de Huffman básica que habéis completado en prácticas. También sería interesante realizar una comparación de esta versión de Huffman con alguna implementación de un codificador aritmético adaptativo.