

Práctica 4 - Codificación de archivos

Jayme Said Barbosa Hernández, Guillermo Carreto Sánchez, Angel Miguel Sánchez Pérez, IEE

IPN

(Instituto Politécnico Nacional, Guillermo Valle 11, Tlaxcala de Xicohténcatl)

Resumen— Este programa puede ser implementado a texto, imágenes o videos, se basa en la implementación de un árbol binario, asignando codificaciones más pequeñas a su valor normal en ASCII para reducir el peso del archivo original, se les asignan valores binarios a cada carácter en función del número de repeticiones que presenta a lo largo del archivo, y con la codificación correspondiente de cada carácter se pueden tomar dos caminos, el primero y tal vez el más sencillo es crear un diccionario para cada carácter con su valor correspondiente haciendo más simple el proceso de codificación, y la segunda opción de rastrear el recorrido dentro del árbol binario a cada carácter para poder asignarle su codificación correspondiente, aunque este proceso implica un número mucho mayor de operaciones o instrucciones.

I. INTRODUCCIÓN

Este documento incluye el proceso de elaboración de un programa de codificación de archivos de texto.

Este reporte incluye cuatro procesos distintos:

1° Tabla de frecuencias.

2° Lista enlazada.

3° Generación de árbol.

4° Generación de códigos.

Esta técnica asigna a cada Byte de datos del archivo una cantidad más pequeña de bits, con la finalidad de usar la menor cantidad de bits posible. La cantidad de bytes en un archivo puede variar donde la codificación provocará que aquellos bytes que repetidamente tendrán codificaciones más cortas y contrariamente aquellos que casi no aparecen en el archivo utilizarán combinaciones de bits más grandes con la posibilidad de incluso ser mayores de 8 bits.

II. TEORÍA

De manera general el primer paso es la elaboración de una tabla de frecuencia de los datos o letras que aparecen en el archivo inicial.

Lo siguiente es crear una lista ligada a partir de la tabla de frecuencia en donde cada nodo lleva el carácter, su ocurrencia o frecuencia y su apuntador, iniciando con los datos menos frecuentes a los de mayor frecuencia.

Posteriormente se genera un árbol binario con la lista ligada tomándose conjuntos de dos elementos yendo de los elementos de la izquierda (menor ocurrencia) a los de la derecha (mayor ocurrencia), este proceso será continuo hasta que la lista se haya llegado al final de la lista.

El último paso es el proceso de generar códigos a cada uno de los caracteres siendo los que están ubicados en las hojas del árbol, para esto se seguirá el camino desde la raíz general a cada una de las hojas asignando 0 o 1 de acuerdo a qué lado se vaya avanzando, derecha o izquierda respectivamente.

Dentro de esta misma práctica se realiza el proceso de decodificación del archivo codificado como método de comprobación de que el programa funciona adecuadamente, para este proceso se hace nuevamente el uso del árbol generado al inicio del programa, donde nos posicionamos en la raíz y se recorre bit por bit el camino que indica la codificación de cada carácter sabiendo que si se lee un 0 es avanzar a la izquierda y un 1 para avanzar a la derecha, al final se llegará a una hoja del árbol con su carácter correspondiente y se sustituye en un nuevo archivo de texto en el cual va el resultado de la decodificación.

A continuación en el desarrollo se mostrarán cada una de las funciones y estructuras usadas en el programa junto con una explicación de cómo funciona cada una para poder entender más a detalle los procesos de ejecución.

III. DESARROLLO

1° estructura:

```
typedef struct nodo {
    char caracter;
    int frecuencia;
    struct nodo* izquierda;
    struct nodo* derecha;
} Nodo;
```

Descripción: La estructura `Nodo` está diseñada para representar un nodo dentro de un árbol binario utilizado en este caso para la compresión de datos. Dentro de cada instancia de esta estructura, se almacena un único caracter que puede ser una letra, símbolo o cualquier otro tipo de dato representable por un `char`. Además, se registra la frecuencia con la que ese caracter específico aparece en el texto original, lo cual es crucial para determinar la ubicación y el peso del nodo en el árbol binario. Adicionalmente, cada nodo tiene dos punteros: `izquierda` y `derecha`, que apuntan a otros nodos en el árbol. Estos punteros representan, respectivamente, el hijo izquierdo y el hijo derecho del nodo actual, lo que permite la construcción y navegación en el árbol binario.

2º estructura:

```
typedef struct lista {
    Nodo* nodo;
    struct lista* siguiente;
} Lista;
```

Descripción: La estructura `Lista` define un nodo dentro de una lista ligada, una estructura de datos lineal donde cada elemento apunta al siguiente. Cada nodo de la lista, representado por la estructura `Lista`, contiene un puntero `nodo` que apunta a un nodo en el árbol binario, definido previamente por la estructura `Nodo`. Esto permite que la lista ligada almacene y mantenga un conjunto ordenado de nodos del árbol. Además, cada nodo de la lista tiene un puntero `siguiente` que señala al siguiente nodo en la secuencia, facilitando la navegación y operaciones en la lista, como la inserción o eliminación de elementos.

1º Función:

```
Lista* crear_lista() {
    return NULL;
```

```
}
```

Descripción: La función `crear_lista` es una función auxiliar que se encarga de inicializar una lista ligada vacía. Al ser invocada, esta función simplemente devuelve un puntero nulo (`NULL`). En la estructura `Lista`, un puntero nulo indica que la lista está vacía y no contiene ningún elemento. Esta función es útil al iniciar el programa o al necesitar reiniciar la lista en algún punto, proporcionando un punto de partida o referencia para operaciones posteriores de inserción o modificación en la lista.

2º Función:

```
void insertar_ordenado(Lista** lista, Nodo* nodo) {
    Lista* nuevo = (Lista*)malloc(sizeof(Lista));
    nuevo->nodo = nodo;
    nuevo->siguiente = NULL;
    if (*lista == NULL || nodo->frecuencia <
        (*lista)->nodo->frecuencia) {
        nuevo->siguiente = *lista;
        *lista = nuevo;
    } else {
        Lista* temp = *lista;
        while (temp->siguiente != NULL &&
            temp->siguiente->nodo->frecuencia < nodo->frecuencia) {
            temp = temp->siguiente;
        }
        nuevo->siguiente = temp->siguiente;
        temp->siguiente = nuevo;
    }
}
```

Descripción: La función `insertar_ordenado` añade un nodo a una lista ligada, asegurando que la lista permanezca ordenada por la frecuencia del nodo. Si el nodo tiene una frecuencia menor que el primer elemento o si la lista está vacía, se inserta al inicio. De lo contrario, se busca la posición adecuada y se inserta manteniendo el orden.

3º Función:

```
Nodo* extraer_minimo(Lista** lista) {
```

```

Nodo* temp = (*lista)->nodo;
Lista* eliminar = *lista;
*lista = (*lista)->siguiente;
free(eliminar);
return temp;
}

```

Descripción: La función `extraer_minimo` obtiene y elimina el nodo con la menor frecuencia de una lista ligada ordenada. Se guarda el nodo con la menor frecuencia (el primer nodo de la lista). Se actualiza el inicio de la lista para que apunte al siguiente nodo. Se libera la memoria del nodo eliminado. Y por último se devuelve el nodo guardado previamente.

4° Función:

```

Nodo* construir_arbol(Lista** lista) {
    while (*lista && (*lista)->siguiente) {
        Nodo* izquierda = extraer_minimo(lista);
        Nodo* derecha = extraer_minimo(lista);

        Nodo* nuevo = (Nodo*)malloc(sizeof(Nodo));
        nuevo->caracter = '\0';
        nuevo->frecuencia = izquierda->frecuencia +
derecha->frecuencia;
        nuevo->izquierda = izquierda;
        nuevo->derecha = derecha;

        insertar_ordenado(lista, nuevo);
    }
    return extraer_minimo(lista);
}

```

Descripción:

La función utiliza un bucle `while` que se ejecuta mientras la lista no esté vacía y tenga al menos dos elementos (nodos).

En cada iteración del bucle, se extraen los dos nodos con las menores frecuencias de la lista utilizando la función `extraer_minimo`.

Se crea un nuevo nodo (nuevo) que representa un nodo interno del árbol de Huffman. Este nodo tiene un carácter nulo ('\0') ya que no representa un carácter individual, sino un nodo interno. La frecuencia del nuevo nodo es la suma de las frecuencias de

los nodos extraídos (izquierda y derecha). Los hijos izquierdo y derecho del nuevo nodo se establecen como los nodos extraídos.

El nuevo nodo se inserta nuevamente en la lista de manera ordenada por frecuencia utilizando la función `insertar_ordenado`.

Este proceso se repite hasta que la lista tenga un solo elemento, que será la raíz del árbol de Huffman.

Finalmente, la función devuelve el único nodo restante en la lista, que representa la raíz completa del árbol de Huffman.

5° Función:

```

void imprimir_codigos_recursivo(Nodo* raiz, int cod[], int
index, FILE* archivo_salida) {
    if (raiz->izquierda) {
        cod[index] = 0;
        imprimir_codigos_recursivo(raiz->izquierda, cod, index +
1, archivo_salida);
    }

    if (raiz->derecha) {
        cod[index] = 1;
        imprimir_codigos_recursivo(raiz->derecha, cod, index +
1, archivo_salida);
    }

    if (!raiz->izquierda && !raiz->derecha) {
        fprintf(archivo_salida, "%c: Frecuencia = %d, Codigo = ",
raiz->caracter, raiz->frecuencia);
        for (int i = 0; i < index; i++) {
            fprintf(archivo_salida, "%d", cod[i]);
        }
        fprintf(archivo_salida, "\n");
    }
}

```

Descripción:

La función utiliza un enfoque recursivo para recorrer el árbol de Huffman.

Si el nodo actual tiene un hijo izquierdo (`raiz->izquierda`), se asigna el valor 0 al arreglo `cod[index]` y se llama

recursivamente a la función `imprimir_codigos_recursivo` con el hijo izquierdo y el índice incrementado en 1.

Si el nodo actual tiene un hijo derecho (`raiz->derecha`), se asigna el valor 1 al arreglo `cod[index]` y se llama recursivamente a la función `imprimir_codigos_recursivo` con el hijo derecho y el índice incrementado en 1.

Si el nodo actual no tiene hijos (es una hoja), se imprime en el archivo de salida el carácter, su frecuencia y el código Huffman asociado.

Se utilizan las funciones `fprintf` para escribir en el archivo de salida. La información se presenta en el formato: `"%c: Frecuencia = %d,Codigo = %s\n"`, donde `%c` es el carácter, `%d` es la frecuencia y `%s` es la representación binaria del código Huffman.

6° Función:

```
void imprimir_codigos(Nodo* raiz, FILE* archivo_salida) {
    int cod[100];
    imprimir_codigos_recursivo(raiz, cod, 0, archivo_salida);
}
```

Descripción:

La función inicializa un arreglo `cod` de tamaño 100 para almacenar temporalmente los bits del código Huffman.

Llama a la función `imprimir_codigos_recursivo` con el nodo raíz, el arreglo `cod`, un índice inicial de 0 y el archivo de salida proporcionado como parámetros.

7° Función:

```
void liberar_arbol(Nodo* raiz) {
    if (raiz) {
        liberar_arbol(raiz->izquierda);
        liberar_arbol(raiz->derecha);
        free(raiz);
    }
}
```

Descripción:

La función comprueba si el nodo raíz no es nulo (`if (raiz)`).

Si el nodo raíz no es nulo, la función realiza las siguientes operaciones de manera recursiva:

Llama a `liberar_arbol` con el hijo izquierdo del nodo actual (`raiz->izquierda`).

Llama a `liberar_arbol` con el hijo derecho del nodo actual (`raiz->derecha`).

Libera la memoria del nodo actual utilizando la función `free(raiz)`.

8° Función (Prototipo):

```
void codificar_cadena_recursivo(Nodo* raiz, char caracter, int cod[], int* index, FILE* archivo_salida);
```

Descripción:

La función `codificar_cadena_recursivo` realiza una codificación recursiva de un carácter específico en el árbol de Huffman.

Utiliza un enfoque de recorrido en el árbol basado en el carácter deseado y actualiza el arreglo `cod` con los bits correspondientes al código Huffman del carácter.

Esta función es utilizada en el proceso de codificación de cadenas completas durante la ejecución del programa principal.

9° Función:

```
void codificar_cadena(Nodo* raiz, char* cadena, FILE* archivo_salida) {
    Escribir(archivo_salida, "Cadena codificada: ")
    Para i desde 0 hasta Longitud(cadena) - 1 hacer
        caracter := cadena[i]
        cod: arreglo de enteros de tamaño 100
        index := 0
        codificar_cadena_recursivo(raiz, caracter, cod, index,
        archivo_salida)

    Escribir(archivo_salida, "\n")
}
```

Descripción:

La función se encarga de recorrer toda la cadena de texto del documento, leyendo cada carácter para asignarle un valor binario. Luego, dentro de la función, se llama a otra función encargada de codificar el carácter en binario. Esto se hace según el tipo de letra y las letras que ya han sido evaluadas previamente.

10° Función:

```
void codificar_cadena_recursivo(Nodo* raiz, char caracter, int
Si raiz es diferente de nulo Entonces
    Si raiz.caracter es igual a caracter Entonces
```

Para i desde 0 hasta (*index) - 1 hacer

 Escribir(archivo_salida, cod[i])

Fin Para

 Escribir(archivo_salida, " ")

Retornar

Fin Si

cod[(*index)] := 0

(*index) := (*index) + 1

 codificar_cadena_recursivo(raiz.izquierda, caracter, cod,
index, archivo_salida)

(*index) := (*index) - 1

cod[(*index)] := 1

(*index) := (*index) + 1

 codificar_cadena_recursivo(raiz.derecha, caracter, cod,
index, archivo_salida)

(*index) := (*index) - 1

Fin Si

}

Descripción:

La función es llamada desde la función anterior, y se realiza de manera recursiva porque está recorriendo el árbol. Mientras la raíz no sea nula, se verificará que el carácter de la raíz sea igual al que se está buscando. En este caso, se escribirá su respectiva codificación binaria. Si no lo es, la función recorrerá sus hijos hasta encontrarlo. Una vez encontrado, se codificará.

11° Función:

void escribir_bits(Nodo* raiz, char* cadena, FILE*
archivo_salida_bits) {

 Para i desde 0 hasta Longitud(cadena) - 1 hacer

 caracter := cadena[i]

 cod: arreglo de enteros de tamaño 100

 index := 0

 codificar_cadena_recursivo(raiz, caracter, cod, dirección
de index, archivo_salida_bits)

 Fin Para

}

Descripción:

La función se encarga de recorrer toda la cadena de texto del documento, leyendo cada carácter para asignarle un valor binario. Luego, dentro de la función, se llama a otra función encargada de codificar el carácter en binario. Esto se hace según el tipo de letra y las letras que ya han sido evaluadas previamente.

12° Función:

void decodificar_bits(Nodo* raiz, FILE* archivo_entrada_bits,
FILE* archivo_salida_texto) {

 nodo_actual := raiz

 bit := leer_bit(archivo_entrada_bits)

 Mientras bit no sea igual a EOF Hacer

 Si bit es igual a '0' Entonces

 nodo_actual := nodo_actual.izquierda

 Sino Si bit es igual a '1' Entonces

 nodo_actual := nodo_actual.derecha

 Fin Si

 Si nodo_actual.izquierda es nulo y nodo_actual.derecha es
nulo Entonces

 // Llegamos a una hoja, escribimos el caracter en el
archivo de salida

 Escribir(archivo_salida_texto, nodo_actual.caracter)

 // Reiniciamos para empezar desde la raíz en la próxima
iteración

 nodo_actual := raiz

 Fin Si

 bit := leer_bit(archivo_entrada_bits)

Fin Mientras

}

Descripción:

La función lo que hace es leer el carácter de la cadena, lo guarda en una variable llamada 'bit', y ese valor se compara. Si es un 1, se avanza a la derecha, y si es un 0, se avanza a la izquierda. Si este nodo ya no tiene hijos, significa que se ha llegado al resultado final y ese será su valor cuando se busque.

13° Función:

```

int main() {
    nombreArchivoEntrada: arreglo de caracteres de tamaño 100
    nombreArchivoSalida: arreglo de caracteres de tamaño 100

    LeerNombreArchivo("entrada", nombreArchivoEntrada)
    LeerNombreArchivo("salida", nombreArchivoSalida)

    archivo_entrada, archivo_salida :=
AbrirArchivos(nombreArchivoEntrada, nombreArchivoSalida,
"r", "w")

    Si archivo_entrada o archivo_salida es nulo Entonces
        ImprimirError("Error al abrir archivos")
        Retornar 1
    Fin Si

    cadena := LeerCadena(archivo_entrada)
    CerrarArchivo(archivo_entrada)

    frecuencia, lista := CalcularFrecuenciaYCrearLista(cadena)
    raiz := ConstruirArbol(lista)

    ImprimirCodigos(raiz, archivo_salida)
    CodificarCadena(raiz, cadena, archivo_salida)

    archivo_salida_bits := AbrirArchivo("bits.txt", "w")
    Si archivo_salida_bits es nulo Entonces
        ImprimirError("Error al abrir archivo de bits")
        Retornar 1
    Fin Si

    EscribirBits(raiz, cadena, archivo_salida_bits)
    CerrarArchivo(archivo_salida_bits)

    Imprimir("Proceso completado. Resultados guardados en ",
nombreArchivoSalida)

    archivo_entrada_bits := AbrirArchivo("bits.txt", "r")
    archivo_salida_texto :=
AbrirArchivo("texto_decodificado.txt", "w")

```

```

    Si archivo_entrada_bits o archivo_salida_texto son nulos
Entonces
    ImprimirError("Error al abrir archivos")
    Retornar 1
Fin Si

    DecodificarBits(raiz, archivo_entrada_bits,
archivo_salida_texto)
    LiberarArbol(raiz)

    CerrarArchivos(archivo_salida, archivo_entrada_bits,
archivo_salida_texto)
    Retornar 0
}

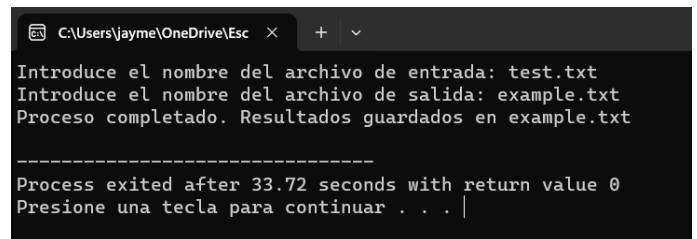
```

Descripción:

La función principal invoca todas las funciones mencionadas en orden, de modo que primero se solicita al usuario que ingrese los nombres de los archivos de entrada y salida. Luego, se abren los archivos de entrada y salida. A continuación, se lee la cadena del archivo de entrada y se calcula la frecuencia de los caracteres en la cadena. Se construye un árbol binario a partir de la frecuencia de los caracteres, se imprimen los códigos binarios en el archivo de salida y se codifica la cadena, escribiéndola en el archivo de salida. Posteriormente, se crea un archivo de bits que contiene la codificación binaria de la cadena. Se realiza la decodificación de los bits y se escribe el resultado en un archivo de texto decodificado. Finalmente, se liberan los recursos y se cierran los archivos.

IV. RESULTADOS

Una vez ejecutado el programa, deberemos completar dos campos. El primero debe contener la cadena de caracteres que deseamos codificar. El segundo archivo se generará con ese nombre, indicando la tabla de frecuencias correspondiente.



```

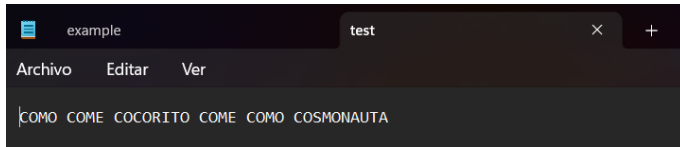
C:\Users\jajame\OneDrive\Esc x + v
Introduce el nombre del archivo de entrada: test.txt
Introduce el nombre del archivo de salida: example.txt
Proceso completado. Resultados guardados en example.txt

-----
Process exited after 33.72 seconds with return value 0
Presione una tecla para continuar . . . |

```

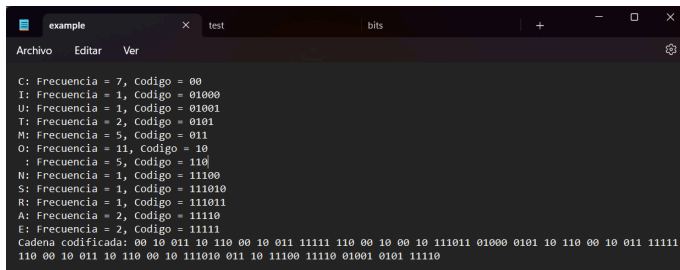
Una vez concluido eso, veremos que se generan tres archivos: uno llamado "example.txt", otro llamado "bits.txt" y el que convertirá la cadena del archivo de bits a cadena normal llamado "texto_decodificado.txt". La cadena usada para este ejemplo fue la asignada en la práctica.

test.txt:



```
COMO COME COCORITO COME COMO COSMONAUTA
```

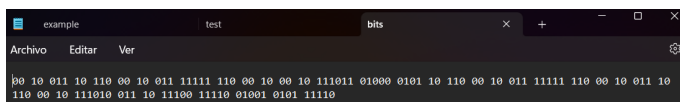
Vista del archivo example.txt:



```
C: Frecuencia = 7,Codigo = 00
I: Frecuencia = 1,Codigo = 01000
U: Frecuencia = 1,Codigo = 01001
T: Frecuencia = 2,Codigo = 0101
M: Frecuencia = 5,Codigo = 011
O: Frecuencia = 11,Codigo = 10
: Frecuencia = 5,Codigo = 110
N: Frecuencia = 1,Codigo = 11100
S: Frecuencia = 1,Codigo = 111010
R: Frecuencia = 1,Codigo = 111011
A: Frecuencia = 2,Codigo = 11110
E: Frecuencia = 2,Codigo = 11111
Cadena codificada: 00 10 011 10 110 00 10 011 11111 110 00 10 00 10 111011 01000 0101 10 110 00 10 011 11111 110 00 10 011 11110 110 00 10 111010 011 10 11100 11110 01001 0101 11110
```

Se apreciará el código de cada letra con su respectiva frecuencia y, al final, la cadena en bits para confirmar que está correctamente escrita.

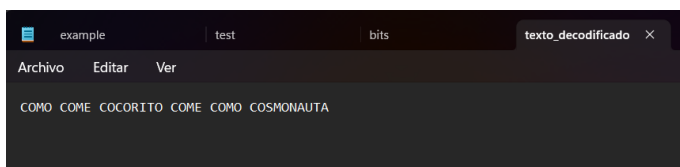
bits.txt:



```
00 10 011 10 110 00 10 011 11111 110 00 10 00 10 111011 01000 0101 10 110 00 10 011 11111 110 00 10 011 11110 110 00 10 111010 011 10 11100 11110 01001 0101 11110
```

Solo la cadena de forma codificada

texto_decodificado.txt:



```
COMO COME COCORITO COME COMO COSMONAUTA
```

Aquí está la decodificación del código, donde se compara el orden binario y se escribe su equivalencia en caracteres.

V. CONCLUSIONES

A través del análisis y codificación de un conjunto de caracteres, el código elaborado logra generar representaciones

comprimidas que minimizan el espacio requerido para almacenar la información original. La estructura y organización meticulosa del programa evidencian una comprensión clara de los principios del algoritmo, así como una habilidad técnica para su implementación práctica. Además, la inclusión de funcionalidades específicas para la codificación, decodificación y manejo de archivos destaca la importancia de considerar el flujo completo de trabajo al trabajar con técnicas de compresión.

En conclusión, esta práctica no solo refuerza los conocimientos teóricos sobre algoritmos de compresión, sino que también ilustra su relevancia y aplicabilidad en escenarios reales de manejo de datos. Además, hay que destacar la capacidad de reducir significativamente el tamaño de los datos sin pérdida de información, lo cual es esencial en áreas como la transmisión de datos, el almacenamiento en dispositivos con recursos limitados y la optimización de recursos en sistemas de cómputo.