

PROTECCIÓN Y COMPACTACIÓN DE ARCHIVOS

HAMMING HUFFMAN

- Materia: Teoría de la Información y la Comunicación.
- Profesor: Mario Silvestri.
- Alumnos: Morales Daniela Ariadna, Juan Sanchez.
- Fecha: 22/06/2023



ÍNDICE

INTRODUCCIÓN.....	2
IMPLEMENTACIÓN.....	3
❖ PROTECCIÓN DE ARCHIVOS.....	3
❖ DESPROTECCIÓN DE ARCHIVOS.....	6
❖ COMPACTACIÓN DE ARCHIVOS.....	13
❖ CREACIÓN TABLA HUFFMAN.....	13
❖ DESCOMPACTACIÓN DE ARCHIVOS.....	15
MANUAL DE USO.....	17
PROBLEMAS HAMMING.....	25
PROBLEMAS HUFFMAN.....	27
RESULTADOS OBTENIDOS.....	28

INTRODUCCIÓN

En el campo de la informática y las comunicaciones, la fiabilidad y eficiencia en la transmisión y almacenamiento de datos son aspectos fundamentales. Para lograr esto, se utilizan técnicas de codificación de datos que permiten detectar y corregir errores, así como comprimir la información para ahorrar espacio. Dos de las técnicas más utilizadas son el código Hamming y el código Huffman.

El código Hamming es un método de detección y corrección de errores en la transmisión de datos. Se basa en la adición de bits de paridad a los datos originales, lo que permite detectar y corregir errores causados por ruido o interferencias durante la transmisión. La implementación del código Hamming en un lenguaje de programación permite garantizar la integridad de los datos transmitidos, asegurando una mayor confiabilidad en la comunicación.

Por otro lado, el código Huffman es una técnica de compresión de datos que aprovecha la frecuencia de aparición de los caracteres en un conjunto de información para asignar códigos de longitud variable a cada uno de ellos. Los caracteres más frecuentes se representan con códigos más cortos, lo que permite reducir la cantidad de bits necesarios para almacenar o transmitir los datos. La implementación del código Huffman en un lenguaje de programación permite comprimir eficientemente la información, ahorrando espacio de almacenamiento y facilitando la transmisión de datos a través de redes.

En este trabajo, exploramos la implementación de los códigos Hamming (módulos de 32, 2048 y 65536 bits) y Huffman (binario) en un lenguaje de programación (Java específicamente) .

Analizaremos los algoritmos y estructuras de datos necesarios para llevar a cabo estas técnicas de codificación y compresión, así como su aplicación en diferentes escenarios. Además, estudiaremos las ventajas y desventajas de cada técnica, y evaluaremos su rendimiento en términos de confiabilidad y eficiencia.

A través de la implementación de estos códigos en un lenguaje de programación, seremos capaces de comprender en profundidad su funcionamiento y cómo pueden mejorar la calidad de la transmisión y almacenamiento de datos en diversos contextos. Al finalizar este trabajo, estaremos en condiciones de utilizar estas poderosas herramientas para garantizar la integridad y eficiencia en la manipulación de la información en aplicaciones prácticas de la vida real.

IMPLEMENTACIÓN

❖ PROTECCIÓN DE ARCHIVOS

• Estructuras utilizadas para almacenar los bits:

ESTRUCTURA	DESCRIPCIÓN
BitSet	Es una clase que representa un conjunto de bits o banderas (flags). Simula un arreglo de bits.

• Explicación del funcionamiento del programa en cuanto a la protección del archivo:

1. Definición de constantes:

- Se definen constantes para los diferentes tamaños de Hamming, como Hamming 32, Hamming 2048 y Hamming 65536.
- También se definen las cantidades de bits de control y de información para cada tamaño de Hamming.

2. Protección del archivo:

- El método "protegerArchivo" recibe como parámetros el tipo de Hamming, el nombre del archivo a proteger y la cantidad de errores a introducir.

- Se establece el nombre del archivo protegido según el tipo de Hamming seleccionado y la extensión del archivo original.

- Se crea el archivo protegido y se llama al método "protegerCadenas" para realizar la protección.

3. Proceso de protección:

- El método "protegerCadenas" recibe como parámetros el archivo original, el archivo protegido, el tipo de Hamming y la cantidad de errores a introducir.

- Se determinan los tamaños de bits de control e información correspondientes al tipo de Hamming seleccionado.

- Se crea una matriz de generación según los tamaños de bits de control e información.

- Se leen los datos del archivo original en bloques de 8 bits.

- Se concatenan los bits de información en un BitSet principal.

- Si el BitSet principal tiene suficientes bits de información para formar un bloque completo, se codifica y escribe en el archivo protegido.

- Si quedan bits de información sin procesar al final del archivo, se realiza el proceso de codificación y se escribe en el archivo protegido.

4. Proceso de codificación:

- El método "codificar" recibe como parámetros un BitSet de entrada, el flujo de salida, el tipo de Hamming, la cantidad de errores y la matriz de generación.

- Se crea un BitSet de salida con el tamaño correspondiente al tipo de Hamming.

- Se reparten los bits de información del BitSet de entrada en el BitSet de salida según el tipo de Hamming.

- Se calculan los bits de control utilizando la matriz de generación y se asignan al BitSet de salida.
- Se calcula el último bit de control (paridad) y se asigna al BitSet de salida.
- Si se especifica una cantidad de errores a introducir:
 - Si es igual a 1, se aplica un algoritmo de introducción de error a los bits de salida.
 - Si es igual a 2 y no se ha introducido un doble error antes, se decide aleatoriamente si introducir un doble error o aplicar un solo error.
 - Si ya se ha introducido un doble error antes, se aplica un solo error.
- Se escribe el BitSet de salida en el archivo protegido.

Se utilizan múltiples funciones auxiliares durante la ejecución del código, no vale la pena ahondar en detalle el funcionamiento de ellas ya que es muy básico. Estas se encuentran algunas en las clases “FuncionesAuxiliares” y otras en “Archivos”.

- **Función que introduce 1 o 2 errores:**

- Introducción de errores: Para introducir errores en la clase “Funciones Error” lo que se hace es muy simple, primero tenemos una función que nos retorna una posición aleatoria para introducir el error y recibe como parametro el numero de modulo de hamming para dar una posición entre 0 y ese numero.

```
public static int lugarDeError(int hamming) {  
    int num = new Random().nextInt(hamming - 1);  
    return num;  
}
```

Luego la función que es utilizada es “*algoritmoDeIntroduccionDeError*” la cual lo que realiza es dada la cantidad de errores que se quieren introducir y dada una probabilidad (para simular ruido blanco) introduce el error (el flip es complementar lo q habia, un 0 por 1 o 1 por 0) en la posición que retorna “*lugarDeError*”

```
public static void algoritmoDeIntroduccionDeError(int i;
    if(cant==1){
        i = new Random().nextInt(10);
        if(i==0){
            output.flip(lugarDeError(hamming));
        }
    }else{
        if(true){
            output.flip(lugarDeError(hamming-1));
            output.flip(lugarDeError(hamming-1));
        }
    }
}
```

❖ DESPROTECCIÓN DE ARCHIVOS

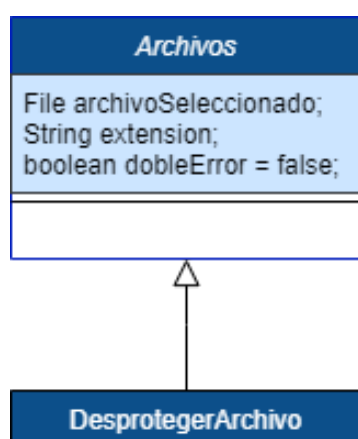
- Estructuras utilizadas para almacenar los bits:

ESTRUCTURA	DESCRIPCIÓN
BitSet	Es una clase que representa un conjunto de bits o banderas (flags). Simula un arreglo de bits.
byte[]	Arreglo de bytes (cada posición contiene un byte).

`byte[][]`

Arreglo bidimensional de bytes (cada posición contiene un arreglo de bytes).

• Clases:



Contiene todas las funciones necesarias para desproteger el archivo, más otras funciones de debug que se utilizaron para detectar algunos problemas que se mencionan más abajo

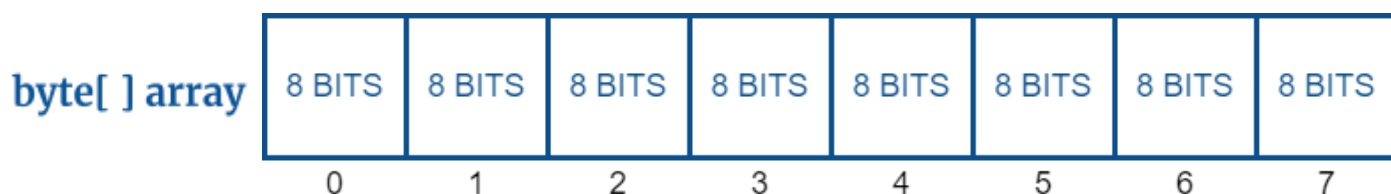
La clase DesprotegerArchivo contiene una única función para controlar la selección del archivo con una extensión válida. (.HAX, .HEX). *

* Para una mejor implementación del lenguaje orientado a objetos, las funciones relacionadas a desproteger el archivo deberían encontrarse en la clase `DesprotegerArchivo`, pero por cuestiones de tiempo quedó así, sin embargo es algo que hay que tener en cuenta en futuros proyectos.

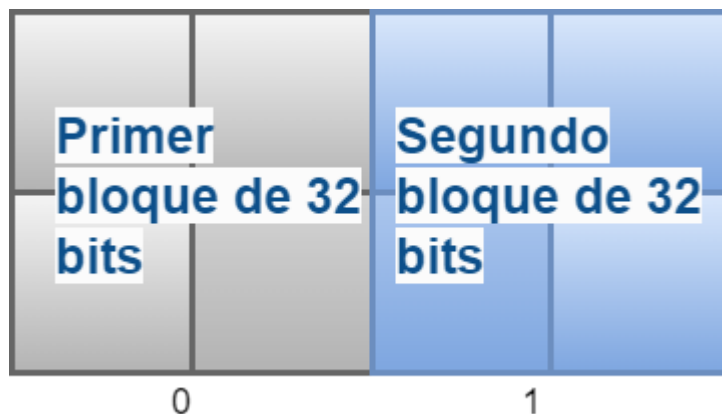
• Explicación del funcionamiento del programa en cuanto a la desprotección del archivo:

1. Se selecciona el archivo luego de los correspondientes controles. El archivo es leído y cada byte del mismo se guarda en un arreglo de bytes, por lo que el tamaño del

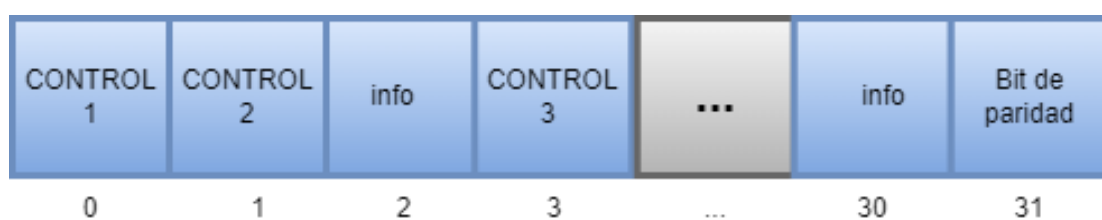
arreglo va a ser igual al tamaño del archivo en bytes. Suponiendo que el archivo protegido pesa exactamente 8 bytes y se está usando un módulo de 32 bits, el arreglo se vería de esta forma:



2. A partir de la extensión del archivo, se definen las siguientes variables, las cuales serán pasadas por parámetro a las funciones encargadas de desproteger el archivo:
 - a. **bloque**: Es el módulo en bytes que se va a utilizar.
 - b. **info**: cantidad de bits de información para cada bloque.
 - c. **controles**: cantidad de controles para cada bloque, contando el bit de paridad.
 - d. **nuevaExtension**: La extensión para el archivo que se va a generar.
3. Se decodifica el archivo: para esto,
 - i. Se pasa el arreglo de bytes a un arreglo bidimensional de bytes, donde el tamaño de cada elemento es el tamaño del bloque de hamming, de manera que cada elemento del arreglo bidimensional es un bloque de hamming.
 - ii. Luego de esto, mediante un bucle se lee cada bloque de hamming, el cuál es convertido en un bitSet, ya que de esta manera podemos acceder a las posiciones que necesitamos para desproteger el archivo y corregirlo en caso de que sea necesario



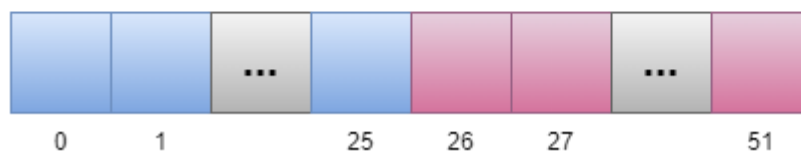
bitSet



- iii. Si es necesario se corrige el bloque y luego, del bitSet anterior se obtienen los bits de información, los cuales se van guardando en otro bitSet. Es decir, tomo los bits de información, los guardo en un bitSet “final”, y luego los próximos se van concatenando en dicho bitSet.
- Dado que es un módulo de 32, hay 26 bits de información.



bitSet final



- iv. Una vez obtenido el bitSet final con todos los bloques de información, este bitSet se pasa nuevamente a un arreglo de bytes. Ya que de ahí se leerá la información byte por byte para escribirse en el nuevo archivo como un carácter ascii utilizando la convención utf-8.

- Función que detecta y corrige 1 error y detecta 2 errores en un bloque:

➤ Detección y corrección de 1 error:

La función getBloqueCorregido, recibe cada bitSet con los bits de información del paso iii del punto anterior, lo corrige y luego ya está listo para ser concatenado. Para obtener el síndrome y poder corregir, utiliza 3 bucles anidados:

```
for(int a=1; a < bitsControl; a++){ //cantidad de controles
    acum=false;

    for(int b=cant[pos];b<tamBloque-1;b += paso[pos+1]){

        for(int c=b; c< (b+(cant[pos]+1)); c++){

            if(bloqueEntrada.get( bitIndex: c)) {
                acum ^= true;
            }else{
                acum ^= false;
            }
        }
        if(acum){
            syndrome += "1";
        }else{
            syndrome += "0";
        }
        pos++;
    }
}
```

- ```
for(int a=1; a < bitsControl; a++)
```

 : El primer bucle cuenta la cantidad de controles que se van a obtener para el síndrome, esta variable es recibida por parámetro. Comienza en 1 ya que hay que despreciar el bit de paridad .

- ```
for(int b=cant[pos]; b<tamBloque-1; b += paso[pos+1]) {
```

 :

El segundo bucle lo que hace es saltar al comienzo de cada cadena de bits a los que se le debe hacer el xor. Por ejemplo, para S2, se deben tomar 2 no, 2 sí, 2 no y así sucesivamente. Entonces el bucle saltaría a estas posiciones:

1	0	1	1	0	0	1	1	1	1	1	0	0	0	0	1	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Para s3 tomo de 4 en 4, así que saltaría a estas posiciones

1	0	1	1	0	0	1	1	1	1	1	0	0	0	0	1	1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

- ```
for(int c=b; c< (b+(cant[pos]+1)); c++) {
```

 : El tercer bucle se encarga de hacer el xor para los n bits consecutivos dependiendo del síndrome, empezando por la posición que le indica el bloque anterior.

Para los ejemplos anteriores:

**S2**

|   |     |   |   |   |     |   |   |   |     |    |    |    |     |    |    |    |
|---|-----|---|---|---|-----|---|---|---|-----|----|----|----|-----|----|----|----|
|   | XOR |   |   |   | XOR |   |   |   | XOR |    |    |    | XOR |    |    |    |
| 1 | 0   | 1 | 1 | 0 | 0   | 1 | 1 | 1 | 1   | 1  | 0  | 0  | 0   | 0  | 1  | 1  |
| 0 | 1   | 2 | 3 | 4 | 5   | 6 | 7 | 8 | 9   | 10 | 11 | 12 | 13  | 14 | 15 | 16 |

**S3**

|   |   |   |     |   |   |   |   |   |   |    |     |    |    |    |    |    |
|---|---|---|-----|---|---|---|---|---|---|----|-----|----|----|----|----|----|
|   |   |   | XOR |   |   |   |   |   |   |    | XOR |    |    |    |    |    |
| 1 | 0 | 1 | 1   | 0 | 0 | 1 | 1 | 1 | 1 | 1  | 0   | 0  | 0  | 0  | 1  | 1  |
| 0 | 1 | 2 | 3   | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11  | 12 | 13 | 14 | 15 | 16 |

El síndrome es acumulado en un string, colocando 0 para cuando el resultado es false, y un 1 para true. Luego esta cadena es invertida y convertida a entero, dándome así la posición donde se encuentra el error. Se hace uso de la función “flip” que provee bitSet para poder invertir el bit de esa posición.

### ➤ Detección de 2 errores:

Luego de hacer la corrección anterior, se realiza un xor entre todos los bits del bitSet para obtener el bit de paridad. Si el bit de paridad es distinto de 0, significa que tiene un doble error. Para esto, se setea en true el atributo `dobleError` de la clase `Archivos`, indicando que este

contiene un doble error. Este atributo nos sirve para notificar que no se pudo corregir el archivo.

## ❖ COMPACTACIÓN DE ARCHIVOS

### 1. Selección del archivo a compactar:

- Se muestra una ventana de selección de archivo.
- Se filtran los archivos para que solo se puedan seleccionar archivos de texto con extensiones ".txt" o ".docx".
- Si se selecciona un archivo con una extensión incorrecta, se muestra un mensaje de error.
- Se guarda la ruta del archivo de entrada seleccionado.
- Se establece el nombre y la ruta del archivo codificado resultante con extensión ".huf".

### 2. Codificación del archivo:

- Se crea una tabla de Huffman basada en las probabilidades de los caracteres en el archivo.
- Se genera un archivo adicional que contiene información necesaria para decodificar el archivo comprimido, como el tamaño original y la tabla de Huffman.
- Se realiza la codificación del archivo original utilizando la tabla de Huffman.
- Los bits codificados se escriben en un archivo de salida.

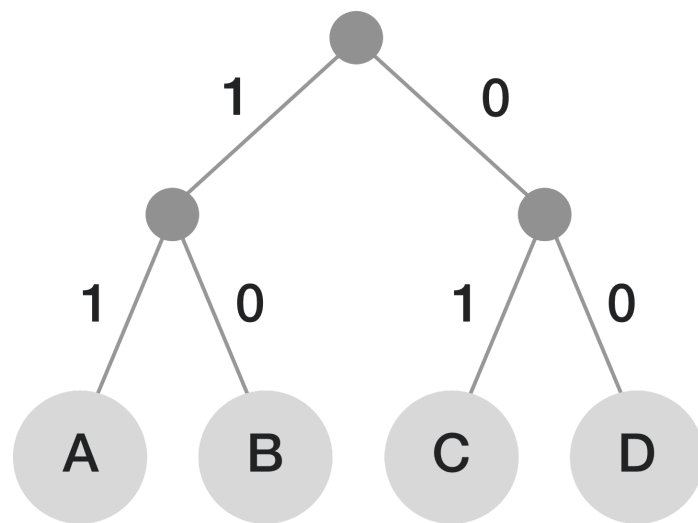
### 3. Compactación del archivo:

- Se llama a la función de codificación con el contenido del archivo original y su tamaño.
- El archivo resultante se guarda en el archivo codificado especificado.

## ❖ CREACIÓN TABLA HUFFMAN

La clase `TablaHuffman` contiene la implementación de un algoritmo para crear una tabla de códigos Huffman a partir de una lista de elementos. Aquí está el resumen de lo que hace la clase:

1. La clase tiene un método estático llamado `MaketablaHuffman` que recibe una lista de objetos.
2. Dentro de este método, se crea una cola de prioridad llamada `pq` para los nodos del árbol de Huffman.
3. Luego, se recorre la lista de objetos y se crea un nodo `HuffmanNode` para cada elemento, con su probabilidad y carácter correspondiente. Estos nodos se agregan a la cola de prioridad.
4. A continuación, se construye el árbol de Huffman combinando los nodos de menor probabilidad hasta que solo quede un nodo en la cola de prioridad.
5. Una vez construido el árbol, se obtiene el nodo raíz.
6. Se crea un mapa llamado `huffmanTable` para almacenar la tabla de códigos Huffman.
7. Se llama al método `buildHuffmanTable` para construir la tabla de Huffman, pasando el nodo raíz, una cadena vacía y el mapa `huffmanTable`.
8. El método `buildHuffmanTable` es recursivo y se encarga de recorrer el árbol de Huffman para asignar los códigos Huffman a los caracteres correspondientes en la tabla.
9. Si un nodo es una hoja, se agrega el código Huffman al carácter en la tabla.
10. Si un nodo no es una hoja, se llama recursivamente al método para los hijos izquierdo y derecho, agregando "0" al código para el hijo izquierdo y "1" para el hijo derecho.
11. Finalmente, se devuelve la tabla de códigos Huffman.



## ❖ DESCOMPACTACIÓN DE ARCHIVOS

### 1. Descompresión del archivo:

- El método ``descompactar()`` se encarga de realizar la descompresión del archivo codificado.
- Llama al método ``decodificacion()`` para realizar la decodificación del mensaje codificado.
- Obtiene el archivo codificado a partir de ``FilesClass.abrirMensajeCodificado()``.
- Llama al método ``decodificacion()`` pasando el mensaje codificado y el archivo de tabla de Huffman.

### 2. Selección del archivo para descomprimir:

- El método ``SelectArchivo()`` se encarga de abrir un cuadro de diálogo para que el usuario seleccione el archivo a descomprimir.
- Filtra los archivos por extensión y valida que se seleccione un archivo con extensión ".huf" (o con extensiones alternativas "DC1", "DC2", "DC3", "DE1", "DE2", "DE3").
- Actualiza los nombres de archivo para el archivo codificado, el archivo decodificado y el archivo de tabla de Huffman utilizando métodos como ``getExtensionFiles()`` y ``setArchivoCodificado()``.

### 3. Decodificación del mensaje:

- El método ``decodificacion()`` se encarga de realizar la decodificación del mensaje codificado utilizando la tabla de Huffman.
- Lee el diccionario de Huffman del archivo de tabla de Huffman utilizando el método ``leerDiccionario()``.
- Itera sobre el mensaje codificado byte a byte y realiza la decodificación bit a bit.



- Utiliza un diccionario invertido para buscar los caracteres correspondientes a los bits decodificados.

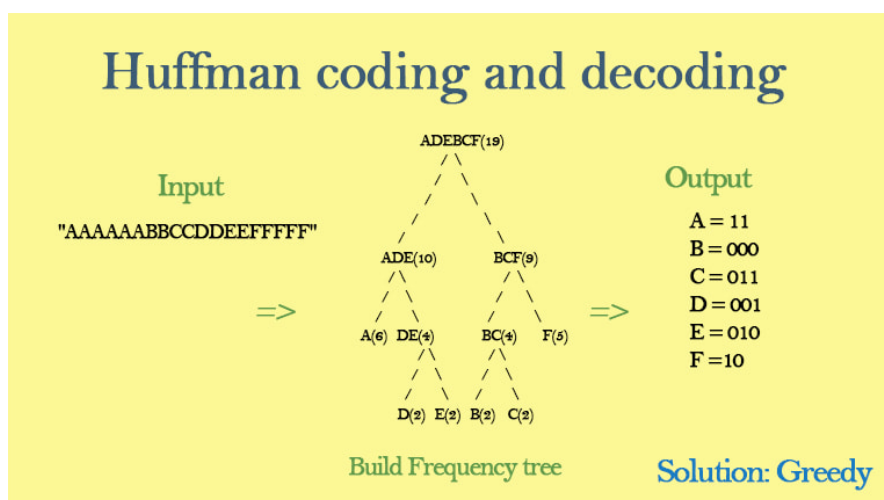
- Escribe los caracteres decodificados en el archivo decodificado.

#### 4. Lectura del diccionario de Huffman:

- El método `leerDiccionario()` se encarga de leer el diccionario de Huffman de un archivo.
- Utiliza un `BufferedReader` para leer el archivo línea por línea.
- Lee el tamaño del diccionario (número de caracteres) de la primera línea y lo almacena en la variable `size`.
- Lee las líneas restantes que contienen las entradas del diccionario y las divide en pares clave-valor utilizando el carácter `"=`".
- Almacena las entradas del diccionario en un mapa, donde las claves son los caracteres y los valores son los códigos de Huffman correspondientes.

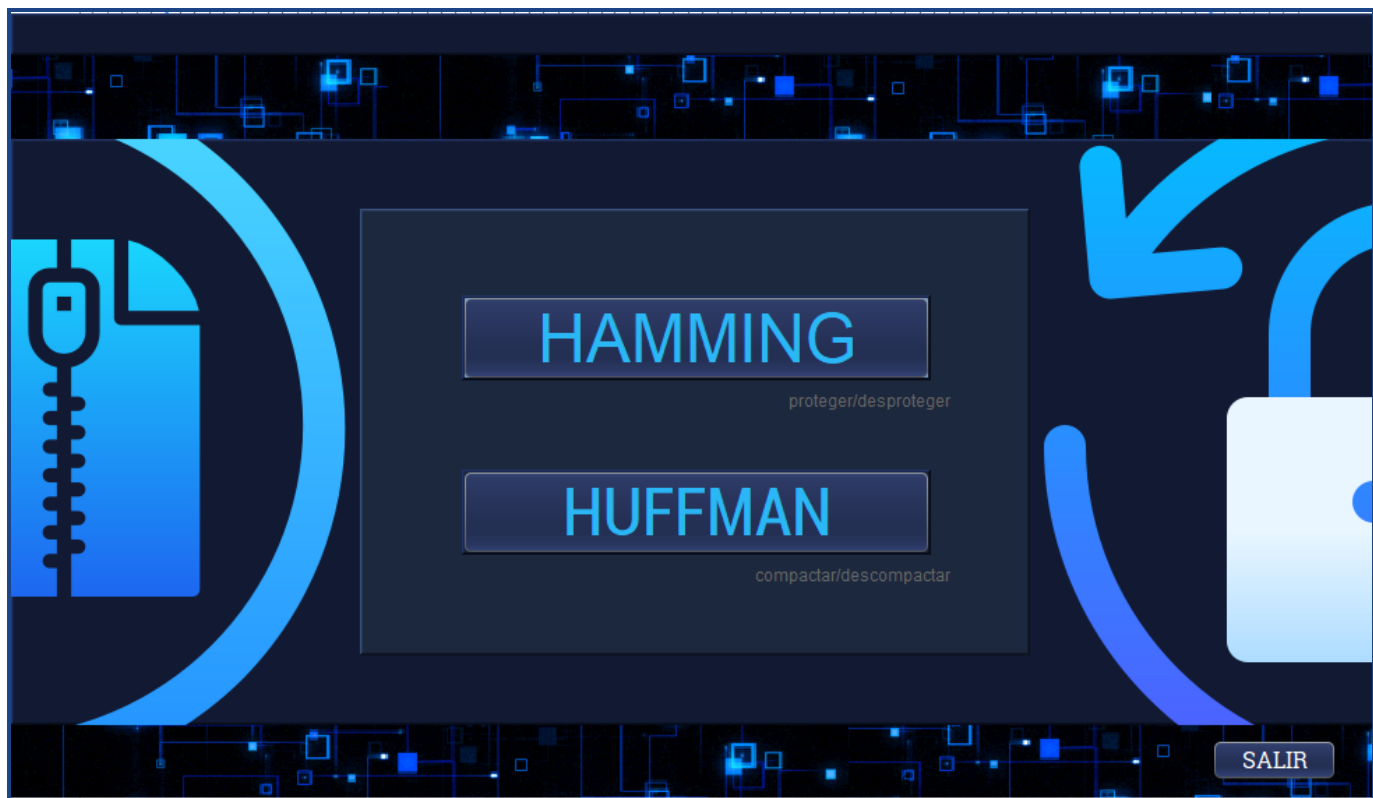
#### 5. Inversión del diccionario de Huffman:

- El método `invertirDiccionario()` se encarga de invertir el diccionario de Huffman.
- Recibe como entrada el diccionario original y crea un nuevo diccionario invertido.
- Intercambia las claves y los valores del diccionario original para crear el diccionario invertido. Ejemplo de cómo funciona:



## MANUAL DE USO

### ● MENÚ PRINCIPAL



Provee las siguientes opciones:

1. **HAMMING**: Nos envía al menú de protección-desprotección con Hamming.
2. **HUFFMAN**: Nos envía al menú de compactación-descompactación con Huffman.
3. **SALIR**: El programa termina y se cierra la interfaz, este botón está disponible en todas las interfaces para poder finalizar el programa en cualquier momento y en cualquier ventana sin necesidad de volver al menú principal.

- MENÚ DE HAMMING

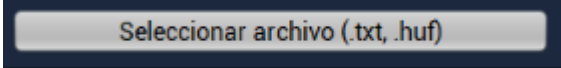


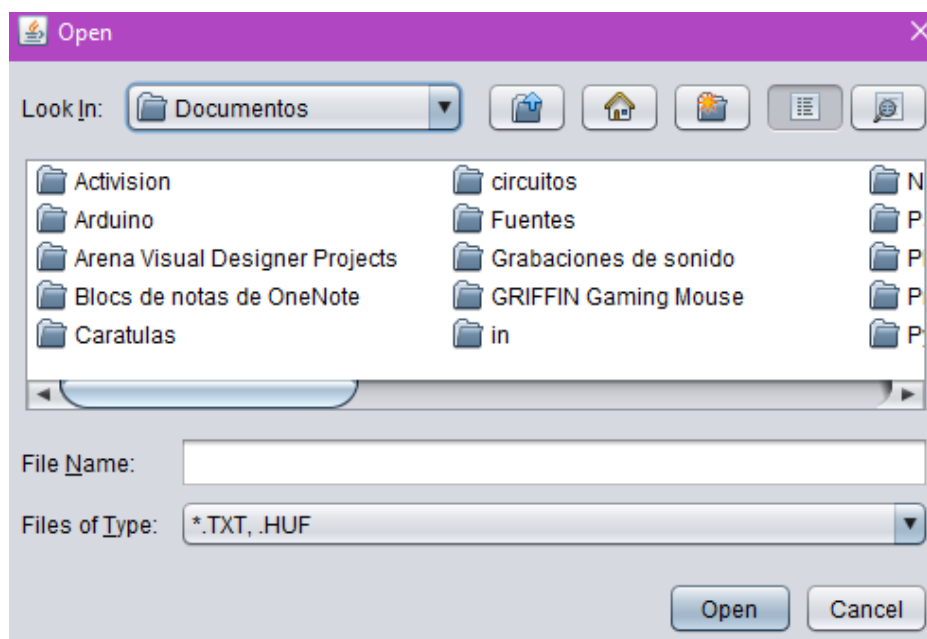
Provee las siguientes opciones:

1. **PROTEGER UN ARCHIVO**: Nos envía a la interfaz que nos permite PROTEGER UN ARCHIVO.
2. **DESPROTEGER UN ARCHIVO**: Nos envía a la interfaz que nos permite DESPROTEGER UN ARCHIVO.
3. **VOLVER**: Este botón está presente en todas las interfaces excepto en la del menú principal, y nos permite volver a la interfaz anterior.

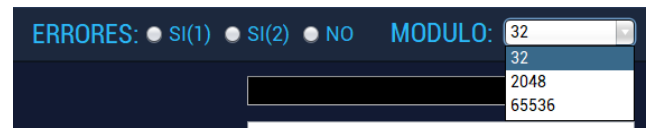
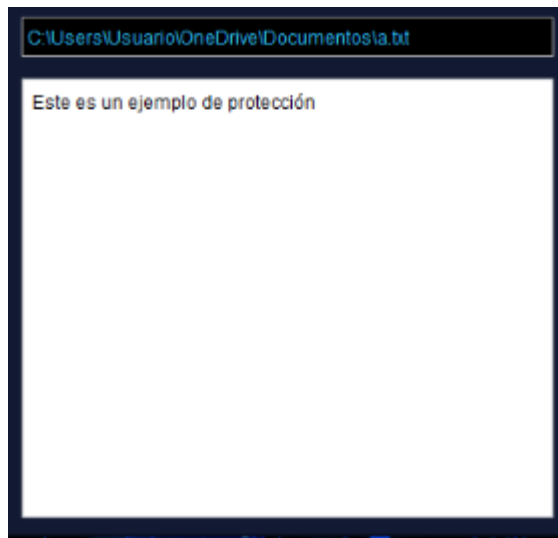
## • INTERFAZ DE PROTECCIÓN DE ARCHIVOS



- : Nos permite seleccionar un archivo con extensión .txt o .huf para poder protegerlo. Al presionarlo se abrirá el explorador de archivos el cuál filtra los archivos con la extensión que necesitamos:



- Una vez seleccionado un archivo válido (ya que en otro caso se muestra un mensaje de error).  
Podremos visualizar la ruta del archivo y su contenido en el panel izquierdo:



- Una vez realizada la configuración se desbloqueará el botón de proteger:

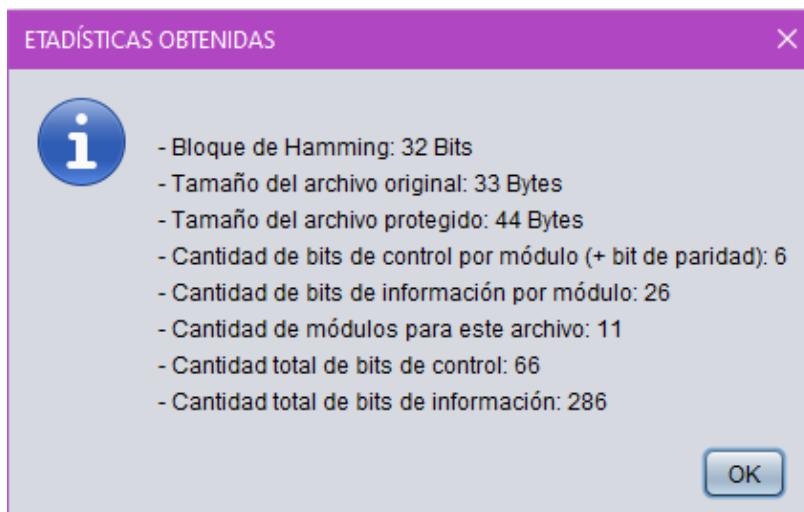


- Para poder protegerlo, debemos indicar si se quiere introducir errores (1 o 2) o no, y en qué módulo queremos realizar la protección. (32, 2048 o 65536 bits):

Al presionar el botón de **PROTEGER**, podremos visualizar el archivo protegido y su ruta, en el panel derecho.

VER ESTADÍSTICAS

Además, se desbloqueará el botón . Al presionarlo nos mostrará una ventana con las estadísticas obtenidas durante la protección del archivo:



## • INTERFAZ DE DESPROTECCIÓN DE ARCHIVOS



- El funcionamiento es el mismo que el de la interfaz de protección, sólo que aquí podremos seleccionar archivos que ya estén protegidos (.HAX, .HEX).
- En caso de que el archivo seleccionado sea .HEX, debemos indicar si queremos corregir los errores del archivo.

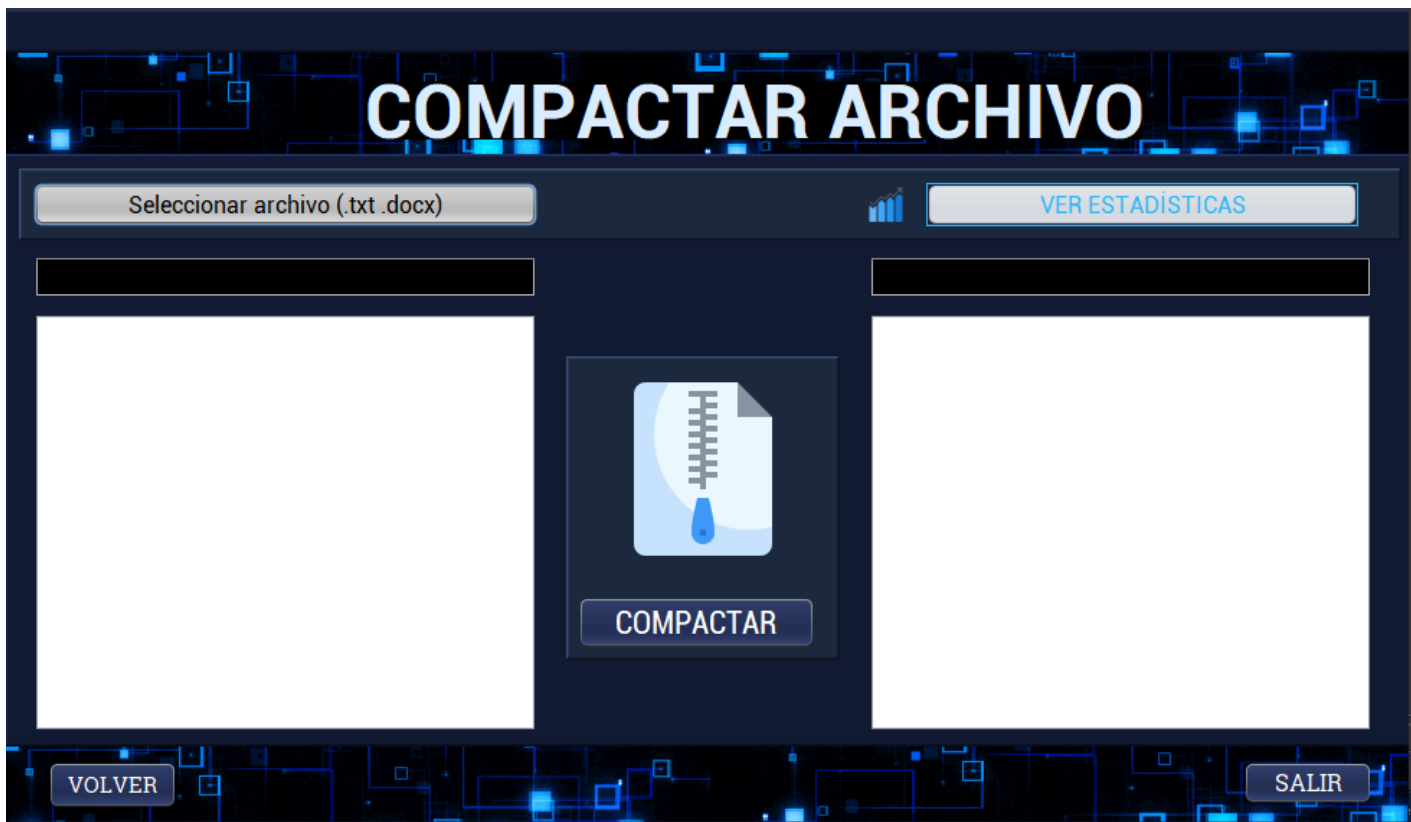
- **MENÚ DE HUFFMAN**



Provee las siguientes opciones:

1. **COMPACTAR UN ARCHIVO**: Nos envía a la interfaz que nos permite COMPACTAR UN ARCHIVO.
2. **DESCOMPACTAR UN ARCHIVO**: Nos envía a la interfaz que nos permite DESCOMPACTAR UN ARCHIVO.

- INTERFAZ DE COMPACTACIÓN DE ARCHIVOS



- Idem al funcionamiento de protección-desprotección, aquí se admiten archivos con extensión .txt y .doc. Una vez compactado el archivo se desbloquea el botón de **VER ESTADÍSTICAS**.

- INTERFAZ DE DESCOMPACTACIÓN DE ARCHIVOS

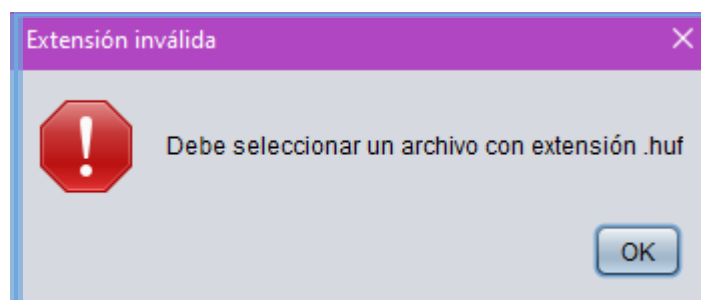
- El funcionamiento es exactamente el mismo que el de la interfaz **COMPACTAR**. Sólo que aquí se admiten archivos con extensión .huf





- **CONTROLES:**

1. Aunque la ventana JFileChooser ya contiene filtros, si se fuerza la selección de un archivo con una extensión inválida, aparecerá el siguiente mensaje indicando la extensión correcta (en este ejemplo es huf):



2. Si oprime el botón

**COMPACTAR/DESCOMP**

**ACTAR** o

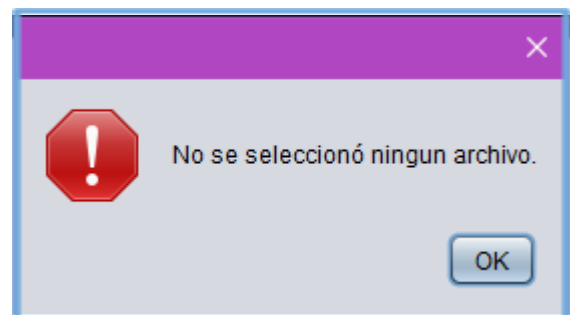
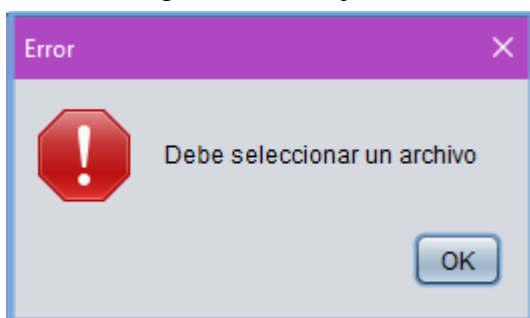
**PROTEGER/DESPROTEG**

**ER** cuando aún no ha

seleccionado ningún

archivo, aparecerá el

siguiente mensaje:



3. Si cancelamos la selección  
de un archivo:

## PROBLEMAS/CONSIDERACIONES

### PROBLEMAS HAMMING

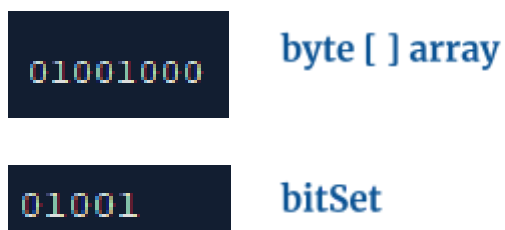
- El principal problema que tuvimos y que rápidamente pudimos solucionar investigando en la web era como poder implementar la protección a nivel de bits en un lenguaje de programación, ya que el dato más pequeño de estos es un byte, lo solucionamos utilizando la clase mencionada en el trabajo llamada “Bitset”.
- Otro problema y su posterior solución era como poder leer un archivo por mediante bytes y escribirlo mediante los mismos en vez de leer caracteres y escribirlos, esto lo logramos gracias a las clases de inputstream y outputstream de Java.

1. Cuando se lee un flujo de entrada y se almacena en un arreglo de bytes, este almacena cada byte en little endian. Por ejemplo, la palabra “Hola”, se quedaría almacenada de la siguiente manera:



**Solución:** Invertir el arreglo para poder tratarlo, y luego volver a invertirlo al momento de escribir en el archivo, ya que las funciones proporcionadas ya tienen en cuenta que la información se almacena en little-endian.

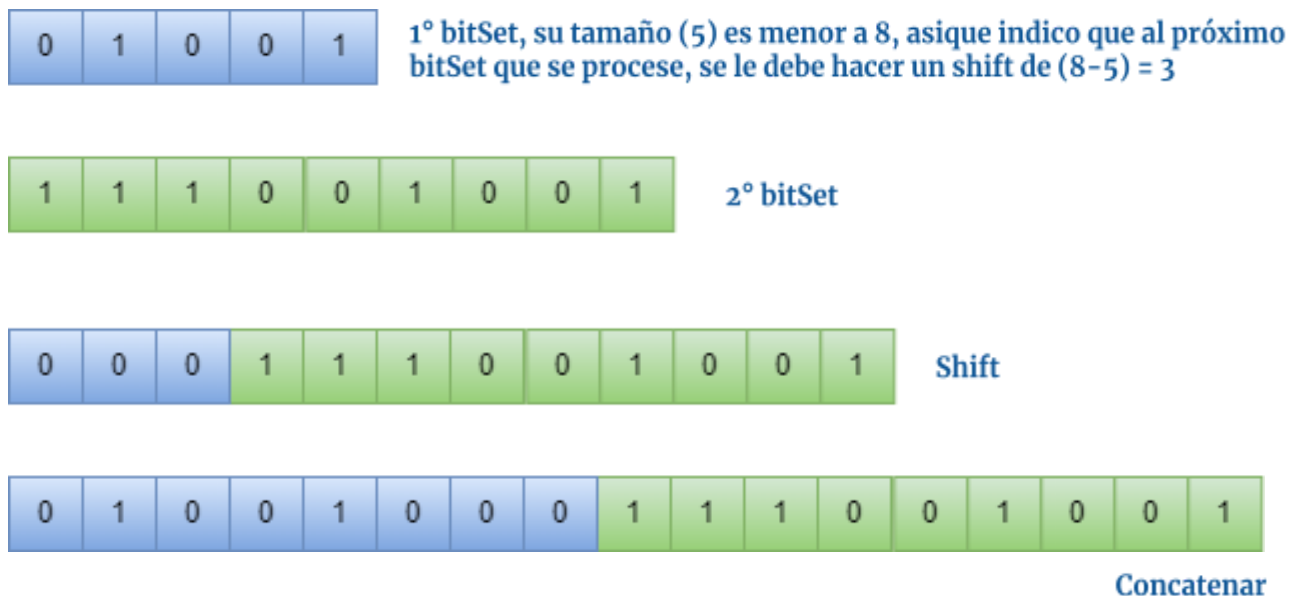
2. bitSet solo lee hasta el último flag en 1 y los demás bits en 0 los ignora, lo que generaba un problema a la hora de concatenar los bitSet, ya que se perdía la información. Por ejemplo, si el arreglo contiene el carácter “H”, y este lo paso a un bitSet, la información sería la siguiente:



**Solución:** Se recibe por parámetro **bitsInfo** la cantidad de bits de información que debería tener el bitSet, por ejemplo para un bloque de hamming de 32 bits, se debería tener 26 bits de

información ya que los demás son de control. Entonces, si calculamos el tamaño del bitSet y este es menor a **bitsInfo**, se calcula la diferencia para ver cuantos flags en 0 ignoró y luego, guardamos este valor para que cuando llegue el próximo bitSet a procesar, se le haga un shift a la derecha con esa cantidad de 0, de esta manera al concatenarlos se “recupera” esa información.

Siguiendo el ejemplo del carácter “H”, la cantidad de bits de información debería ser 8, ya que es lo que ocupa un carácter ascii.



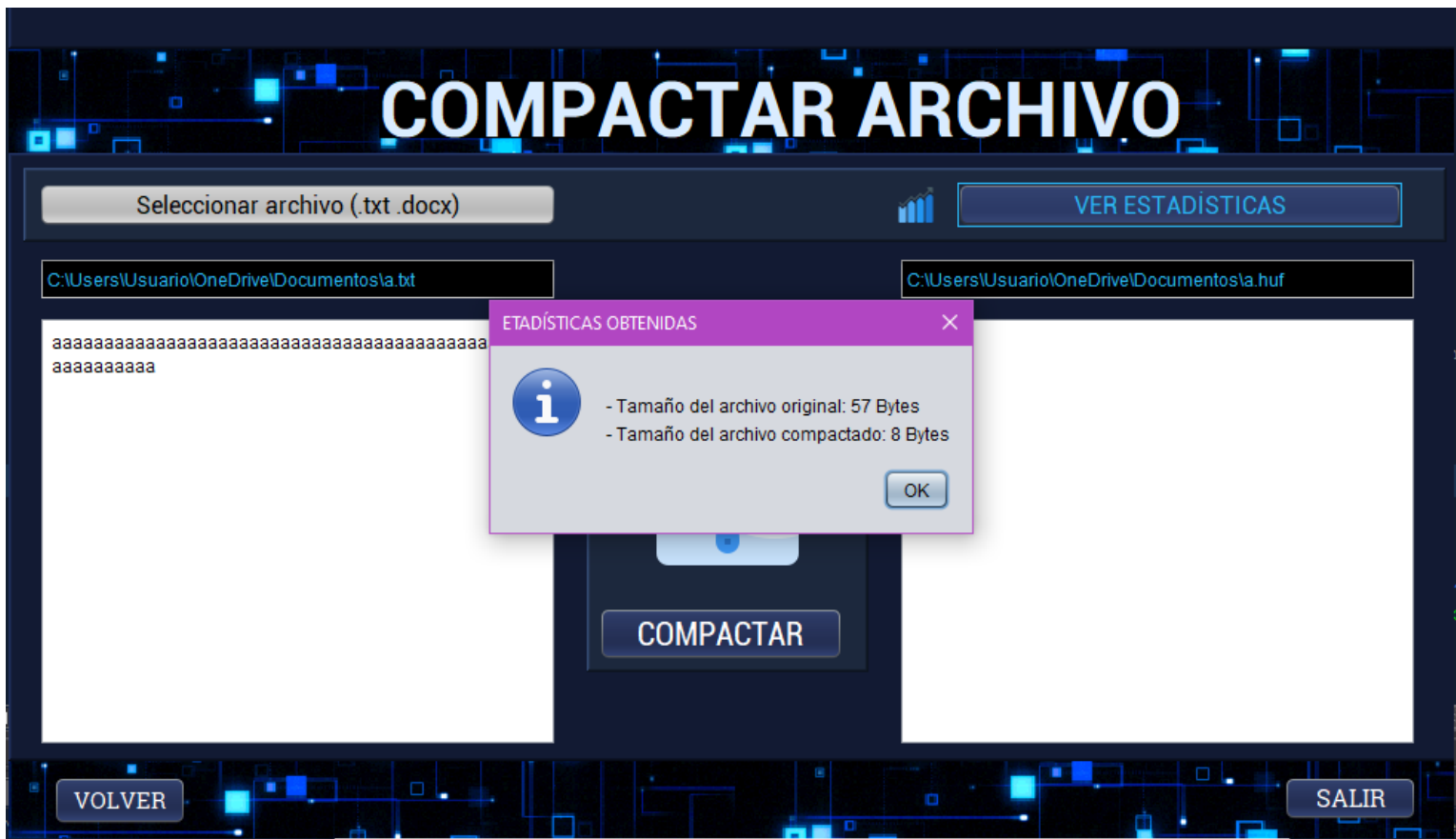
## PROBLEMAS HUFFMAN

- El principal problema fue cómo implementar correctamente la parte de la codificación huffman, ya que utilizando algunos tipos de datos o estructuras, muchas veces sucedia un `NullPointerException`, lo arreglamos utilizando una clase creada llamado Huffman code para poder implementar un nodo de huffman con su probabilidad, caracter y como máximo 2 nodos hijos y luego una cola de prioridad en la cual la prioridad era la probabilidad.

- Otro problema fue la implementación del diccionario/mapa de huffman para compactar el archivo, la solución fue utilizar la clase Map de Java la cual nos permite armar pares clave/valor y era justo lo que necesitábamos.

## RESULTADOS OBTENIDOS

- Cuándo sucede el mejor de los casos (que todos los caracteres sean iguales) podemos ver el poder de compresión de Huffman, por ejemplo pasando de 57 bytes a 8 bytes



- Cuándo sucede el peor de los casos (que todos los caracteres sean diferentes y sean muchos) podemos ver que la compresión de Huffman no ayuda, pasando de 281 bytes a 8811 bytes



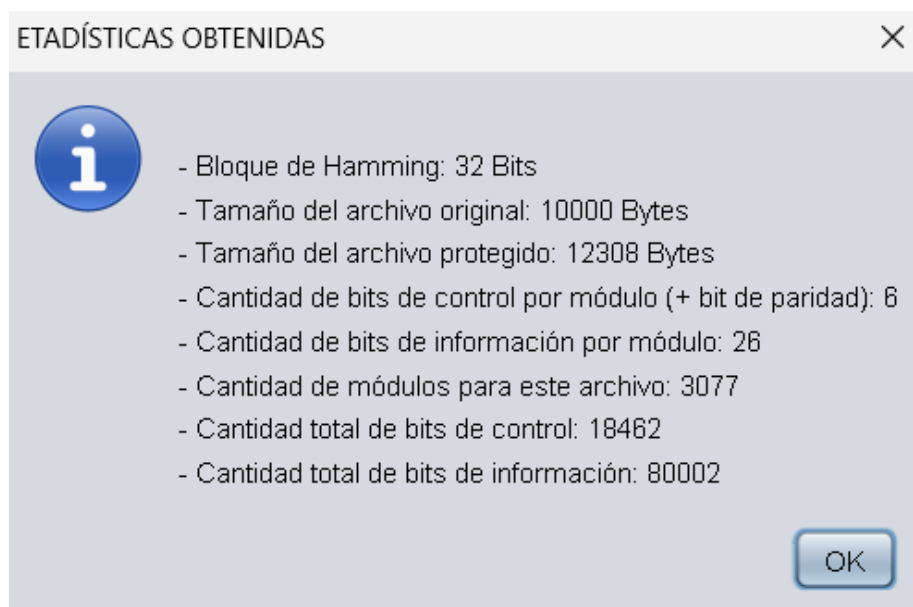
- Ahora vemos un caso en Hamming, donde por ejemplo un archivo de 100 bytes protegido con bloques de 32 bits aumenta a 124 bytes y vemos alguna de sus estadísticas.

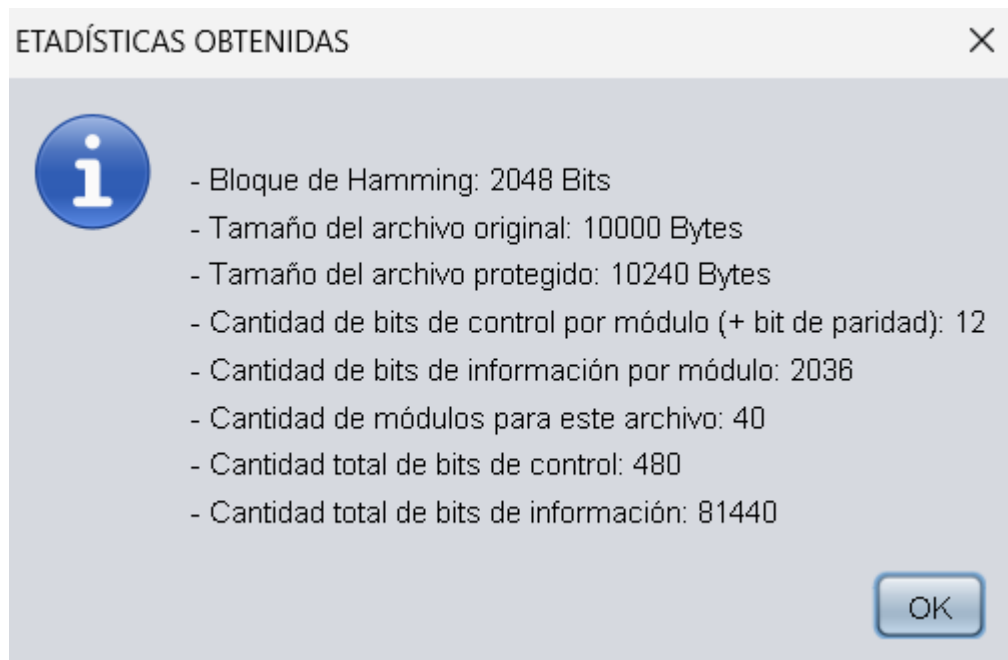


- Ahora vemos un caso extremo peor en Hamming, donde por ejemplo un archivo de 100 bytes protegido con bloques de 65536 bits aumenta a 8192 bytes teniendo muchos bits en desuso y aumentando considerablemente su tamaño



Por ende, es importante saber de antemano el tamaño del archivo a proteger para elegir la mejor cantidad de bloques para agregar la menor cantidad de información al protegerlo. Si es un archivo muy chico nos conviene siempre bloques pequeños, y si es un archivo grande bloques grande. Veamos el ejemplo de para un archivo grande (10000 bytes) cuanto agrega en bloque de 32 y de 2048.





Mientras que en el de bloque de 32 bits añadió 2308! bytes en el de 2048 solamente 240 bytes. Demostrando lo que habíamos planteado.



## CONCLUSIÓN

El trabajo presentado implementa dos algoritmos de compresión y corrección de errores: Código Hamming y Código Huffman. Ambos algoritmos son ampliamente utilizados en el campo de la compresión de datos y la detección y corrección de errores en la transmisión de información.

El Código Hamming se utiliza para la detección y corrección de errores en la transmisión de datos. Este código permite garantizar la integridad de los datos transmitidos y recuperar la información original incluso si se producen errores en la transmisión.

Por otro lado, el Código Huffman es un algoritmo de compresión sin pérdida que se basa en asignar códigos de longitud variable a los símbolos en función de su frecuencia de aparición. Este enfoque te permite reducir el tamaño de los archivos sin perder información, lo que resulta en un ahorro significativo de espacio de almacenamiento.

En resumen, el trabajo demuestra la implementación exitosa de dos algoritmos clave en el campo de la compresión y corrección de errores. La combinación de Código Hamming y Código Huffman permite garantizar la integridad de los datos transmitidos y lograr una alta eficiencia en la compresión de archivos. Estos algoritmos son fundamentales en la optimización y transmisión de datos en diversas aplicaciones, como la transmisión de archivos, la compresión de imágenes y videos, y la comunicación en redes.

## **BIBLIOGRAFÍA**

- Hamming, R. W. (1986) *Coding and Information Theory*, 2da. Edición.
- Manual Web (2023). *Tutorial Java*. [Consultado el 21 de junio de 2023]  
[.https://www.manualweb.net/java/](https://www.manualweb.net/java/)
- Java Plataform. *Class BitSet*. [Consultado el 21 de junio de 2023]  
<https://docs.oracle.com/javase/8/docs/api/java/util/BitSet.html>