

Compresor de archivos Greedy basado en el algoritmo Huffman

Miruna Andreea Gheata, Rafael Adrián Gil Cañestro

Resumen—En este documento se muestra la implementación de la práctica *Implementar un compressor d'arxius basat en un codi de Huffman de manera ávida*. Se explicará la estructuración del proyecto, el patrón de programación empleado para el desarrollo, la solución propuesta, los distintos problemas encontrados durante el proceso, un manual de usuario y, finalmente, unas conclusiones.

Index Terms—Compresor, Huffman, algoritmo Greedy, código de Huffman

1. INTRODUCCIÓN

En este artículo se detallará por capítulos la implementación de un programa en Java que comprima archivos mediante *códigos de Huffman*.

En el capítulo 2 se presentará el problema propuesto, al igual que los requerimientos definidos que se tienen que cumplir.

En el capítulo 3 se expondrá la solución propuesta, mencionando a grandes rasgos las técnicas y patrones principales que se han adoptado para resolver el problema: el *patrón MVC*, el *DDD*, la *concurrentia*; y mencionando aspectos importantes de la implementación.

En el capítulo 4 se especificarán las tecnologías utilizadas, es decir, el entorno de programación, librerías destacadas, etc.

En el capítulo 5 se explicará la implementación del **Modelo**, separado en Dominio (la definición de los datos) e Infraestructura (las operaciones que se realizan sobre estos datos).

En el capítulo 6 se explicará la implementación de la **Vista**, que se ha separado en varias clases, cada una representando un elemento gráfico.

En el capítulo 7 se explicará la implementación del **Controlador**, explicando los métodos que contiene.

En el capítulo 8 se calculará el **Coste computacional del programa**.

El capítulo 9 corresponde a una **Guía de usuario** en la que se definen los pasos a seguir para poder ejecutar el programa y unos ejemplos de ejecución.

En el capítulo 10 se exponen los **Problemas encontrados** a lo largo de la implementación del programa.

Y, finalmente, el capítulo 11 contiene las **Conclusiones** obtenidas tras realizar este proyecto.

2. DEFINICIÓN DEL PROBLEMA

Se debe realizar un programa mediante el cual se puedan comprimir y descomprimir archivos, utilizando como algoritmo de compresión el *algoritmo de Huffman*.

Para que la compresión sea significativa los archivos deben contener o bien un texto extenso, o bien un gran número de caracteres repetidos, ya que la frecuencia de los caracteres es importante en la *codificación de Huffman*.

Los códigos de Huffman pueden tener un **tamaño fijo o variable**. Usar uno u otro implica una gran diferencia en el tamaño del archivo resultante de la compresión. Si por ejemplo se tiene un fichero con 3 caracteres A, B y C, y con una frecuencia de 3000, 1300 y 400 respectivamente, los códigos Huffman asociados podrían ser:

$$A = 00, B = 01, C = 10$$

$$2 * 3000 + 2 * 1300 + 2 * 400 = 9400 \text{ bits}$$

Con un código variable, dado que A es el más frecuente, los códigos podrían ser:

$$A = 0, B = 01, C = 10$$

$$1 * 3000 + 2 * 1300 + 2 * 400 = 6400 \text{ bits}$$

Se puede observar la gran diferencia de tamaño que se obtiene al utilizar el segundo formato.

En esta práctica se ha elegido los códigos de tamaño variable, por lo que se obtiene una mejor compresión ya que los elementos que se analizan (en este caso los **bytes del archivo**), tendrán asociado un código de Huffman de la longitud óptima en función de su frecuencia.

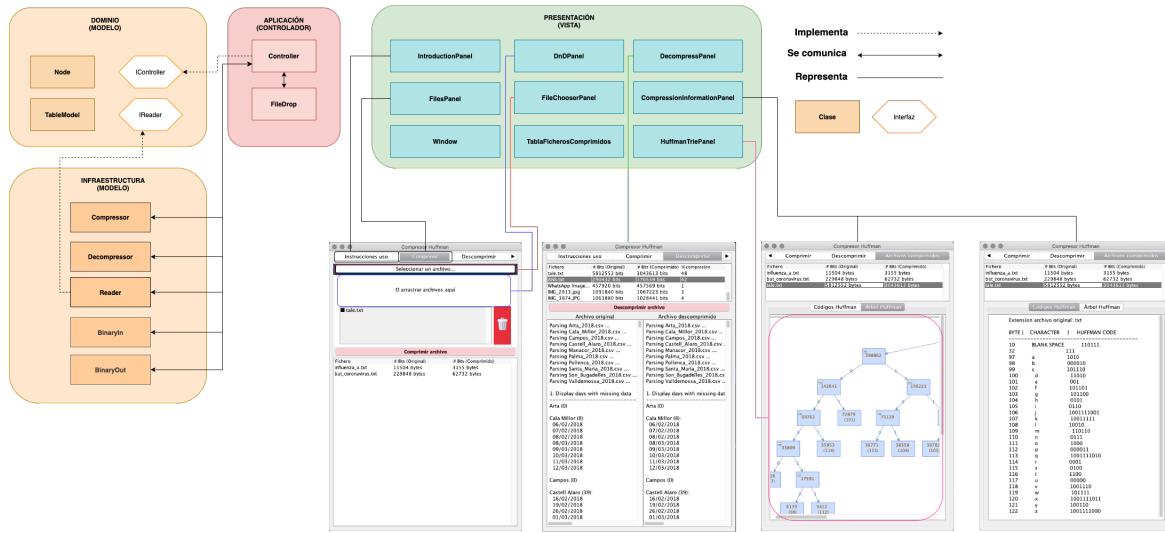


Figura 1: Estructura MVC del proyecto

3. SOLUCIÓN PROPUESTA

La solución propuesta hace uso de distintos patrones, técnicas y formas de programación y de estructuración de proyectos.

Algoritmo ávido: también conocido con el nombre de *greedy*, es un algoritmo que se basa en elegir la opción óptima en cada paso local con la esperanza de llegar a una solución general óptima. Las decisiones se toman en función de la información inmediatamente disponible, sin importar las consecuencias futuras. Usado especialmente en problemas de optimización (Prim, Kruskal).

Codificación Huffman: algoritmo ávido usado en la compresión de datos. Tipicamente se puede llegar a comprimir desde un 20% hasta un 90% de los datos, dependiendo de las características de los datos que se están comprimiendo. El algoritmo de Huffman utiliza una tabla en la que se almacena el número de veces (la frecuencia) que aparece cada byte en el archivo, y se usará con el fin de encontrar una forma óptima de representar cada byte como una cadena binaria.

La compresión de Huffman se realiza normalmente con archivos de texto extensos, y los códigos de Huffman se obtienen a partir de los caracteres del mismo.

En esta práctica no se mirarán los caracteres sino **los bytes del archivo**, por lo que se pueden comprimir todo tipo de archivos (PDF, Word, MP3...). Sin embargo, los resultados de la compresión puede que no sean notables ya que muchos de estos archivos los bytes aparecen con la misma frecuencia (por ejemplo en el caso de los PDF).

Para la selección de los archivos se ha implementado dos maneras distintas:

- **Mediante un selector de archivos:** se podrá elegir un archivo para subir desde el ordenador mediante

una ventana.

- **Arrastrando un archivo o más:** se podrán elegir uno o varios archivos y comprimir a la vez.

Una vez comprimido un archivo se puede visualizar la información asociada a la compresión: los códigos de Huffman generados y el árbol de Huffman resultante.

También se ha implementado la **descompresión de los archivos** de manera que se vuelva al archivo original a partir del comprimido.

3.1. Patrón MVC y enfoque DDD

Este proyecto está estructurado basándose en el diseño guiado por el dominio, en adelante DDD, ya que se complementa muy bien con MVC. El DDD no es una tecnología ni una metodología, es un enfoque que proporciona una visión estructural del sistema. DDD separa el *Modelo* en tres partes: *Dominio*, *Aplicación* e *Infraestructura*.

La capa de *Aplicación* es responsable de coordinar la *Infraestructura* y la capa de *Dominio* para hacer una aplicación útil. Por lo general, la capa de *Aplicación* usará la *Infraestructura* para obtener los datos, consultaría el *Dominio* para ver qué se debería hacer y luego volvería a usar la *Infraestructura* para realizar las operaciones pertinentes para obtener los resultados deseados. [1]. Por lo tanto,

- La capa de *Aplicación* se correspondería al *Controlador*
- la de *Dominio* e *Infraestructura* al *Modelo*
- la de *Presentación* a la *Vista*

3.2. Estructuración DDD

A nivel interno el programa está dividido en 5 carpetas (*Presentación*, *Aplicación*, *Dominio*, *Infraestructura* y *Utils*, en la que se guardan elementos que se usan en todo el proyecto) y un fichero Main.java.

3.3. Concurrency

El programa contiene distintos elementos de carácter concurrente con el fin de agilizar los cálculos y no bloquear la GUI. De esta manera, se tiene:

- Las operaciones de compresión y decompresión se realizan mediante hilos de tipo `ExecutorService`:

```
ExecutorService e = Executors.newSingleThreadExecutor();
e.submit(() -> comprimirFichero(nombre));
```

- La compresión de un archivo se realiza con una única instancia de la clase `Compressor`. En el caso de que se quieran comprimir varios archivos a la vez, se crearán tantos `Threads` (instancias de la clase `Compressor`) como archivos haya utilizando `ExecutorService`.

3.4. Aspectos a destacar

- Se ha implementado un *Drag and Drop* para poder seleccionar archivos.
- Se tiene una clase `Constantes` en la que se guardan todas las variables finales (nombres, títulos, dimensiones, etc.) a modo que sean visibles para todas las clases del proyecto. De esta manera se consigue centralizar todas las variables para que su manipulación sea más sencilla (creación, modificación, acceso, etc.).
- Se ha implementado un gestor de archivos de manera que si se añaden archivos repetidos a la lista de archivos por comprimir se podrá elegir si se quiere reemplazar un archivo, todos los archivos o guardar el anterior. Las demás funcionalidades están explicadas en el *Manual de usuario* [9.1].
- El programa **tiene memoria**, de manera que si en una ejecución previa se han comprimido ya archivos estos seguirán apareciendo en las pestañas de *Descomprimir* y *Archivos comprimidos*.

4. TECNOLOGÍAS UTILIZADAS

4.1. Lenguaje de programación utilizado

La implementación de este proyecto se ha hecho utilizando como lenguaje de programación Java.

4.2. Entorno de programación

La práctica se ha desarrollado en el IDE **IntelliJ IDEA**. Cabe destacar que si se intenta ejecutar el proyecto el Netbeans no compilará debido a que IntelliJ utiliza carpetas extra (como la de `.idea`) para poder compilar el proyecto. Por lo tanto, para ejecutar el proyecto se debe utilizar necesariamente IntelliJ.

Al igual que Netbeans, IntelliJ ofrece una ayuda a la hora de programar elementos gráficos, por lo que se ha facilitado todo el tema del diseño de las ventanas.

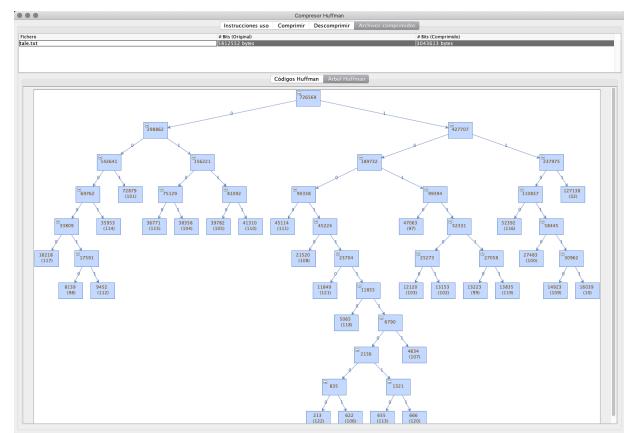


Figura 2: Ejemplo de árbol Huffman creado para la compresión

4.3. Librerías destacadas

`JGraphX` - utilizada para representar el árbol de Huffman resultante [2]. Dicha librería se encuentra en un `JAR` dentro de la carpeta del proyecto al mismo nivel que la carpeta `src`. En caso de que el proyecto no compile, **SE DEBE ENLAZAR EL JAR CON EL PROYECTO** (leer manual de usuario [9.1]).

5. IMPLEMENTACIÓN DEL MODELO

El *Modelo* es:

- el elemento responsable de mantener la información con la que el sistema opera: los nodos que compondrán el árbol Huffman y las tablas que contendrán los archivos comprimidos.
- encargado de enviarle la información a la *Presentación* (Vista) para que esta pinte por pantalla la información necesaria. Por otro lado, las peticiones de actualización o manipulación de los datos llegan al *Modelo* desde el *Controlador*.

Las funciones del *Modelo* están divididas en dos partes. El *Dominio* se encarga mantener, enviar y modificar la información de los arboles y los archivos resultado de compresión y descompresión.

Por el otro lado, la *Infraestructura* es la encargada de realizar las operaciones con los datos, en este caso la implementación del compresor y decompresor Huffman, los elementos de I/O (lectura y escritura de ficheros binarios), la creación de los árboles, de los ficheros de texto con los resultados comprimidos y de los nuevos ficheros descomprimidos.

5.1. Dominio

Al ser el encargado de mantener y brindar la información, tiene que contener la información sobre los objetos que se van a usar y las interfaces que usamos

para facilitar los futuros cambios.

Por lo tanto, podemos encontrar las definiciones de las distintas estructuras de datos que se usarán para poder representar los elementos del problema:

- **Node**, para la creación del árbol Huffman
 - **TableModel**, para organizar la información de los archivos
1. **Node** Los atributos de la clase són:
 - **byteRepresentado** En el caso de ser un nodo hoja, representará el carácter que se ha codificado.
 - **frecuencia** Es la suma de las frecuencias de los hijos y en el caso de no tener es la frecuencia del carácter que se ha codificado.
 - **rightNode** Es el hijo derecho del nodo
 - **leftNode** Es el hijo izquierdo del nodo

Los métodos definidos son:

- **setRightNode**: Inicializa el nodo hijo de la derecha (aresta con valor 1)
- **setLeftNode**: Inicializa el nodo hijo de la izquierda (aresta con valor 0)
- **isLeaf**: Comprueba si el nodo es hoja, mirando si tiene hijos
- **compareTo**: Compara la frecuencia de dos nodos mediante su resta

2. **TableModel** Su constructor crea una tabla con las columnas que entran por parámetro

- **addRowToModel** Crea y añade un nuevo objeto a la tabla con la información que le entra por parámetro.

Por otro lado encontramos la carpeta *Interficies*, la cual contiene las dos interficies que usamos en el programa:

- **IController** Usada por el controlador conteniendo todas las funciones para servir de puente de comunicación entre el Modelo y la Vista.
- **IReader** Usada por la clase *Reader* y contiene todas las operaciones de lectura que se realizan en el programa.

5.2. Infraestructura

La infraestructura es la encargada de leer los archivos, codificarlos creando un árbol, crear el archivo comprimido y después con la ayuda del árbol descomprimirlo. La codificación se hace siguiendo el algoritmo de Huffman,

el cuál va asignando un código variable a cada byte dependiendo de su frecuencia. De esta manera se crea el árbol asociado de Huffman.

Las clases pertenecientes a la *Infraestructura* son:

1. **Compressor**

Es la clase encargada de comprimir los ficheros y escribir la información en las respectivas carpetas para los árboles y códigos Huffman. Los atributos de la clase són:

- **controller** Para tener acceso a las funciones del controlador.
- **bytes** Contiene los bytes del archivos a descomprimir
- **file** Archivo que se va a descomprimir

Los métodos pertenecientes a esta clase son:

void comprimir(): comprime el fichero mediante la creación el árbol Huffman y los códigos Huffman y los ficheros en los que se guardará

void encode(Node root, String str, Map<Byte, String> huffmanCode): Recorre el árbol Huffman, mientras no llegue a las hojas concatena el valor de la aresta y al llegar a la hoja lo guarda.

Map<Byte, Integer> crearTablaFrecuencias(): Calcula la frecuencia de los bytes del fichero original.

PriorityQueue<Node> crearArbolHuffman(Map<Byte, Integer> freq): Mediante colas de prioridad que almacenan los nodos, va creando el árbol Huffman

void writeTrie(String outputType, Node x): Con métodos de la clase BinaryOut escribe el árbol Huffman en el fichero

void writeTriePrivate(String outputType, Node x): Método recursivo que va escribiendo el árbol.

void writeHuffmanCodes(String outputType, String extension): Escribe la información de la compresión en un fichero

void writeCompressedFile(String outputType): Escribe el resultado de la compresión en el fichero.

2. **Decompressor**

Es la clase encargada de descomprimir los archivos. Los atributos de la clase són:

- **controller** Para tener acceso a las funciones del controlador.
- **root** Raíz del árbol Huffman que se usará para descodificar.
- **file** Archivo que se debe descomprimir.
- **extension** Extensión del archivo original.

Los métodos de la clase són:

void descomprimir(): Lee el fichero comprimido y recorre el árbol hasta llegar a un nodo hoja. Entonces coge el byte que representa y lo escribe en el fichero descomprimido. El recorrido según el resultado de la lectura, si lee un 0 visitará al nodo hijo de la izquierda y en caso de ser 1 visitará el de la derecha.

3. Reader

Los métodos de la clase són:

byte[] getBytes(File file): Inicializa el nodo hijo de la derecha (arista 1).

Future<StringBuilder>getFileContent(String path): Inicializa el nodo hijo de la izquierda (arista 0).

Future<Node>readTrieFromFile(String fileName): Comprueba si el nodo es hoja, mirando si tiene hijos

Future<Object[]>getOrigiCompBytes(String path): Compara la frecuencia de dos nodos mediante su resta.

Future<String>getPathArchivoOriginal(String path): Devuelve la ruta del archivo original

Node readTrieFromFilePrivate(Node node, BinaryIn bIn): Devuelve el nodo raíz del árbol.

4. BinaryIn

El constructor de la clase crea un *BufferedInputStream* para la lectura del fichero que le entra por parámetro.

void fillBuffer(): Lee un byte del fichero

boolean isEmpty(): Comprueba si el Stream está vacío

boolean readBoolean(): Devuelve el siguiente byte como booleano

char readChar(): Devuelve el siguiente byte como carácter

int readInt(): Devuelve el siguiente byte como entero

long readLong(): Devuelve el siguiente byte como long

5. BinaryOut

El constructor de la clase crea un *BufferedOutputStream* para la escritura del fichero que le entra por parámetro.

void writeBit(boolean x): Escribe el bit que entra por parámetro en el fichero.

void writeByte(int x): Escribe un byte del fichero dependiendo del valor que entra por parámetro (0-256).

void clearBuffer(): Escribe en el fichero los bits restantes que queden en el Buffer.

void flush(): Limpia el buffer.

void close(): Limpia y cierra el Stream.

void write(boolean x): Escribe el byte especificado en el Stream.

void write(byte x): Escribe los 8-bit en el Stream.

void write(int x): Escribe los 32-bit en el Stream.

void write(char x): Escribe el carácter como 8-bit en el Stream.

void write(char x, int r): Escribe el carácter como r-bits en el Stream.

void write(String s): Escribe el String como 8-bit en el Stream.

void write(String s, int r): Escribe el String como r-bits en el Stream.

6. IMPLEMENTACIÓN DE LA VISTA

La *Vista* es la responsable de representar gráficamente el modelo. Como mínimo, tiene que ser capaz de tener un sistema de selección de archivos para comprimir y una manera de visualizar los resultados de la compresión. Con el fin de llevar esto a cabo, se ha dividido la representación de cada requisito en distintas clases:

1. CompressionInformationPanel: panel que contiene

- toda la información de los archivos comprimidos. En el se encuentra:
- Una *tabla* con todos los archivos comprimidos, diferenciados por archivos nuevos (salen en negrita) y archivos comprimidos anteriormente.
- Cuando se seleccione una de las filas de la tabla se cargará la información asociada al archivo que se encuentra en la fila:
- Pestaña de código Huffman*: aparecerán los códigos Huffman asociados a cada byte, su frecuencia *y*, en caso de que se trate de archivos de texto, el carácter asociado. Además, también se muestran los bits originales, los bits tras la compresión, la extensión del archivo original y la ruta absoulta del archivo original.
 - Pestaña de Árbol Huffman*: se mostrará el árbol Huffman asociado a la compresión. Los nodos del árbol son colapsables, es decir, se pueden esconder los hijos de cada nodo.
2. *DecompressPanel*: panel que contiene los elementos asociados a la descompresión de un archivo. En este se muestra:
- La tabla de archivos comprimidos, diferenciados entre los nuevos (en negrita) y los que se han comprimido en ejecuciones anteriores.
- La descompresión de los archivos es distinta dependiendo del formato del archivo original. En el caso de que se trate de un archivo de texto (extensión txt), se tiene:
- Panel en el que saldrá el contenido del fichero de texto original.
 - Panel en el que saldrá el contenido del fichero descomprimido.
- De esta manera se puede apreciar la correcta descompresión del archivo.
- Si se trata de otro tipo de archivo (imagen, audio, pdf, etc.) se abrirá el archivo descomprimido.
3. *DnDPanel*: panel que sirve como región en la que el usuario puede arrastrar archivos para que se puedan comprimir.
4. *FileChooserPanel*: selector de archivos mediante el cual se puede seleccionar un archivo para comprimir.
5. *FilesPanel*: panel que contiene todos los elementos necesarios para la compresión de los archivos. Los atributos de esta clase son:
- JPanel selectedFilesPanel*: muestran los ficheros seleccionados para la compresión.
 - JPanel deleteFilePanel*: permite eliminar un archivo de la lista de archivos seleccionados arrastrando dicho archivo encima de este.
 - JPanel comprimirFicherosPanel*: panel que comprende los dos elementos anteriores.
 - TablaFicherosComprimidos archivosComprimidos*: *tabla* que contiene los archivos nuevos comprimidos. Estos archivos salen en negrita y se diferencian en las demás tablas que salen en el programa.
 - HashMap<File, JLabel> labels*: conjunto de ficheros y labels mediante el cual se tiene constancia de todos los ficheros que hay en la vista. Los ficheros se visualizan por pantalla mediante su label. Es necesario identificaro cada fichero y su label para después ser capaz de detectar qué fichero se quiere eliminar cuando se arrastre el label en el *deleteFilePanel*.
 - JProgressBar progressBar*: barra de progreso que se muestra mientras se están comprimiendo los archivos.
 - HighlightButton comprimirArchivoButton*: botón que inicializa el proceso de compresión de los archivos.
- Cuando el usuario seleccione archivos que ya había seleccionado anteriormente para comprimir, saldrá una *JDialog* con un mensaje. Según el total de archivos repetidos que haya (uno o más), se podrá seleccionar o bien reemplazar el archivo que se ha detectado, o bien reemplazar todos los archivos.
- boolean reemplazarArchivo*: sirve para saber cuando el usuario ha seleccionado que sólo se reemplace el archivo que se está mirando ahora.
 - boolean reemplazarArchivos*: sirve para saber si el usuario ha seleccionado que se reemplacen todos los archivos repetidos.
6. *HuffmanTriePanel*: panel que contiene el árbol Huffman asociado a un archivo comprimido.

7. **IntroductionPanel**: ofrece una breve explicación del funcionamiento del programa. Está implementado como un JEditorPane por la facilidad de representar elementos tipo HTML y de esta manera estilizar el contenido.

8. **TablaFicherosComprimidos**: tabla que contiene los archivos comprimidos. En el proyecto existen varias tablas, pero comparten el mismo modelo. Existen 2 modelos de tablas:

- a) **tableModelNewArchivos**: este modelo sólo contiene los elementos nuevos comprimidos. Los elementos que pertenezcan a este modelo aparecerán en negrita.
- b) **tableModelTotalArchivos**: este modelo contiene todos los archivos comprimidos, tanto los de otras ejecuciones como los nuevos.

Cuando se añade un nuevo fichero para comprimir se actualizan ambas tablas.

9. **Window**: ventana principal que recoge todos los elementos gráficos. La aplicación se ha dividido en 4 pestañas distintas:

- a) **Pestaña de introducción**: breve manual de usuario
- b) **Pestaña de compresión**: ventana en la que se seleccionan los archivos para comprimir
- c) **Pestaña de descompresión**: ventana en la que se seleccionan archivos ya comprimidos para descomprimirlos
- d) **Pestaña de información**: ventana en la que se puede visualizar la información de los archivos ya comprimidos: los códigos de Huffman para cada byte y el árbol resultante.

Se han utilizado distintas clases para poder implementar de manera satisfactoria los paneles mencionados anteriormente. Dichas clases se encuentran en la carpeta de *Utils*, y son descritas a continuación:

1. **CellRenderer**: clase mediante la cual se renderizan (se visualizan) las celdas de la tabla TablaFicherosComprimidos. Es la encargada de poner el texto en negrita cuando los archivos comprimidos sean nuevos.
2. **DropTargetListener**: clase que implementa la funcionalidad del elemento en el cual se pueden arrastrar cosas (*Drag 'n Drop*), más concretamente la funcionalidad que tiene el `deleteFilesPanel`. Tiene un único método:

```
@Override
public void drop(DropTargetDropEvent evt) {
    try {
        Transferable tr = evt.getTransferable();
        evt.acceptDrop(java.awt.dnd.DnDConstants.ACTION_COPY_OR_MOVE);
        // Get a useful list
```

```
JLabel fileList = (JLabel) tr.getTransferData(jLabelFlavor);
// Eliminar fichero de la lista de archivos seleccionados para comprimir
controller.deleteFile(new File(fileList.getName()));
// Mark that drop is completed.
evt.getDropTargetContext().dropComplete(true);
} catch (Exception e) {
    e.printStackTrace();
    evt.rejectDrop();
}
```

3. **FileLabel**: clase label de un archivo. Dicha clase implementa DragGestureListener y DragSourceListener para que se pueda arrastrar y que además sea detectado por los elementos que implementan DnD.
4. **FoldableTree**: clase que implementa el árbol Huffman.
5. **HighlightButton**: clase que implementa los botones de la aplicación.
6. **JLabelTransferable**: clase que se utiliza para que los labels de los ficheros sean reconocibles por el DnDPanel, y que además detectan la información del label.

7. IMPLEMENTACIÓN DEL CONTROLADOR

El *Controlador* es el elemento que monitoriza y controla las comunicaciones entre la Vista y el Modelo.

Los atributos que se encuentran en el controlador son:

1. **ExecutorService executorService**: hilo que ejecutará la compresión y decompresión de los ficheros. Para cada archivo se crea un hilo, por lo que si hay varios archivos seleccionados su compresión se hará en paralelo.
2. **Map<String, Node>rootNodes**: colección de los nodos raíz (del árbol Huffman) asociados a los ficheros comprimidos.
3. **HashMap<String, BinaryOut>binaryOutFiles**: colección de instancias `BinaryOutFiles` asociados a los archivos de *salida* del proyecto. Los ficheros que se escriben mediante esta clase son el que contiene códigos Huffman y el que contiene el árbol asociado al fichero. Se ha decidido usar esta clase debido a que proporciona una "API" para poder escribir elementos en vez de bytes (booleans, Strings, etc.), y a la hora de crear los archivos es más fácil de visualizar.
4. **HashMap<String, FileOutputStream>outFiles**: colección de instancias `FileOutputStream` asociados a los archivos de *salida* del proyecto. Los ficheros que se escriben mediante esta clase son los del fichero codificado.
5. **HashMap<String, Boolean>fileIsNew**: conjunto de booleans que marca los ficheros comprimidos si son nuevos o no. Esto se usa para diferenciar los ficheros nuevos de los antiguos

- cuando se representan en la tabla que los contiene.
6. **AtomicInteger filesCompressed:** variable que se usa para saber cuándo se han comprimido todos los ficheros seleccionados.
 7. **int totalFiles:** variable que indica cuántos ficheros se están comprimiendo en total.

El controlador implementa la interfaz **IController** que tiene los siguientes métodos:

```
void addArchivosPorComprimirAPanel(File file, int bytesOriginales, int bytesComprimidos): : se llama desde el Compressor cuando se ha acabado de comprimir un archivo para que se añada una nueva fila con la información correspondiente al TableModel. En la tabla se guarda el nombre del archivo, los bits que ocupaba el archivo original y los bits que ocupan el archivo comprimido resultante.
```

```
void addFileRoot(Node node, String fileName): : se llama desde el Compressor cuando se ha creado el árbol de Huffman de un archivo. Se guarda el nodo raíz dentro de rootNodes. Si ya existía una raíz para ese archivo se reemplaza.
```

```
void addFiles(File[] selectedFiles):
```

guarda los ficheros seleccionados por el usuario para que se puedan visualizar y manipular dentro del FilesPanel.

```
JComponent addTrieToPanel(String fileName): : crea el gráfico del árbol Huffman asociado al fichero. Se crea una nueva instancia de HuffmanTriePanel y se retorna el componente gráfico resultante.
```

```
void closeBinaryOutputFile(String outputFile): : se cierra el archivo de salida asociado y se elimina la instancia de la lista de binaryOutFiles.
```

```
void closeOutputFile(String binaryOutputFile):  

void closeBinaryOutputFile(String outputFile): : se cierra el archivo de salida asociado y se elimina la instancia de la lista de outFiles.
```

```
void comprimirFicheros(Set<File>files):
```

se crean tantos hilos como ficheros por comprimir haya y se inicia el proceso de compresión.

```
void createBinaryOutputFile(String outputType, String path): crea una nueva instancia de la clase BinaryOutputFile que será la que se encargará de escribir el texto en el archivo output indicado. Se añade dicha instancia a la lista de binaryOutFiles.
```

```
void createOutputFile(String outputType, String path): crea una nueva instancia de la clase FileOutputStream que será la que se encargará de escribir el texto en el archivo output indicado.
```

Se añade dicha instancia a la lista de outFiles.

void deleteFile(File file): se borra el fichero de la lista de ficheros del panel, y se elimina de la Vista.

void descomprimirFichero(String nombre, File file): se crea un hilo y se inicia el proceso de descompresión. Cuando se acaba, se añade el contenido del archivo original y del comprimido al panel.

Object[] getOriginalAndCompressedBytes(String path): retorna los bits originales, los bits comprimidos y la extensión del archivo original que se encuentra dentro del fichero de huffmanCodes asociado al fichero especificado.

String getPathArchivoOriginal(String path): retorna la ruta absoluta del fichero original.

void initRootNodes(): se encarga de comprobar si hay archivos dentro de la carpeta compressed, ya que si hay significa que hay archivos comprimidos de ejecuciones anteriores. En caso de que haya se crean los nodos raíz de cada uno y se añaden a la lista de rootNodes, y se añaden tantas filas como archivos haya en el TableModel que contendrá todos los ficheros.

ArrayList<File>listFilesForFolder(File folder): retorna el número de archivos existentes dentro de una carpeta.

StringBuilder readFileContent(String fileName): retorna el contenido (en texto) del fichero especificado.

void replaceProgressBarFilesPanel(): se encarga de reemplazar la barra de progreso por el botón de comprimir.

void resizePanels(int width, int height): sirve para redimensionar los paneles que contendrán el archivo original y comprimido cuando la ventana cambie de dimensión. De esta manera los paneles que los contiene siempre ocuparán lo mismo dentro de la ventana.

void write(String outputFile, boolean bool): escribe mediante una instancia de BinaryOut un booleano al fichero de salida.

void write(String outputFile, byte b): escribe mediante una instancia de BinaryOut un byte al fichero de salida.

void write(String outputFile, byte[] bytes): escribe mediante una instancia de FileOutputStream un booleano al fichero de salida.

void write(String outputFile, int integer): escribe mediante una instancia de BinaryOut

un entero al fichero de salida.

void write(String outputFile, String string): escribe mediante una instancia de BinaryOut un String al fichero de salida.

8. COSTE COMPUTACIONAL

El coste computacional del algoritmo de compresión de Huffman es de

$$O(n * \log n)$$

donde n es el número de caracteres/bytes únicos dentro del fichero.

9. GUÍA DE USUARIO

En esta guía se explicarán los requerimientos del proyecto, las funcionalidades del gestor de ficheros implementado, los pasos a seguir para poder utilizar el programa y, por último, se mostrarán varios ejemplos de ejecución.

9.1. Requerimientos de proyecto

Para la creación de los gráficos de los árboles Huffman se utiliza una librería concreta. Por lo consecuente, el proyecto tiene una dependencia a un archivo .jar que se encuentra dentro de la carpeta lib del proyecto. Este archivo .jar se debe enlazar con el proyecto para que pueda funcionar, ya que la ruta es absoluta para el directorio del proyecto (varía con cada ordenador.) El .jar es jgraphx-master.jar. Los pasos para enlazar el .jar son los siguientes [6]:

1. Abrir el proyecto en IntelliJ
2. Compilar el proyecto y comprobar si funciona
 - a) Lo más seguro es que no [6a]
 - b) Si funciona, ejecutar programa y ver si va bien
3. Si no funciona, hacer click izquierdo encima del proyecto y seleccionar Open Module Settings [6b]
4. Navegar hasta la pestaña de Libraries [6c]
 - a) Seleccionar el '+' del menú de la izquierda
 - b) Añadir una nueva librería del tipo Java
 - c) Seleccionar jgraphx-master.jar de la carpeta ./lib/ y pulsar Apply [6d, 6e, 6f]
5. Repetir el proceso anterior pero dentro de la pestaña de Global Libraries
 - a) Saldrá una ventana preguntando si se desea reemplazar la librería anterior. Pulsar OK y Apply. [6g]
6. Recomilar el proyecto y ya debería funcionar [6h]

9.2. Funcionalidades del gestor de ficheros

La gestión de los ficheros es muy importante para esta práctica dado que ofrece la mayoría de funcionalidad que presenta. Los ficheros que se gestionan son:

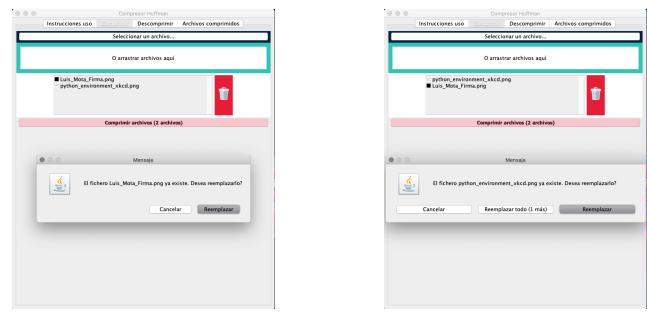
- Los ficheros comprimidos (carpeta compressed)
- Los ficheros de los árboles de Huffman (carpeta huffmanTrees)
- Los ficheros de los códigos de Huffman (carpeta huffmanCodes)
- Los ficheros descomprimidos (carpeta decompressed)

9.2.1. Compresión de ficheros en batches

Existe la posibilidad de comprimir varios ficheros a la vez, tanto eligiendo a través del selector de ficheros como arrastrando los ficheros en la región correspondiente.

9.2.2. Sustitución de archivos ya seleccionados

Los ficheros que se seleccionan para comprimir se van guardando en una lista hasta que se pulsa el botón de comprimir (momento en el los archivos se van eliminando a medida que se van comprimiendo). Si se añaden ficheros nuevos a la lista que ya existían se preguntará por pantalla si se desean reemplazar. Se pueden reemplazar tanto ficheros individuales [3a] como en batches [3b].



(a) Un archivo

(b) Varios archivos

Figura 3: Mensaje al añadir archivos ya existentes en la lista de archivos por comprimir

9.2.3. Carga de ficheros ya comprimidos

El programa tiene memoria, de manera que se guarda toda la información de todos los ficheros que se van comprimiendo en cada ejecución. Cuando se levanta la aplicación, primero se mira si hay elementos dentro de la carpeta compressed, ya que si existen es que hay elementos comprimidos anteriores.

Los ficheros ya comprimidos son cargados a continuación y se podrá visualizar su información de compresión, y también de podrán descomprimir.

9.2.4. Visualización de los ficheros descomprimidos

Los ficheros que se descomprimen se visualizan al acabar la descompresión. Dependiendo del tipo de fichero, la visualización puede ser:

1. Si es de texto (.txt), aparecerán en la ventana el fichero original y el fichero comprimido

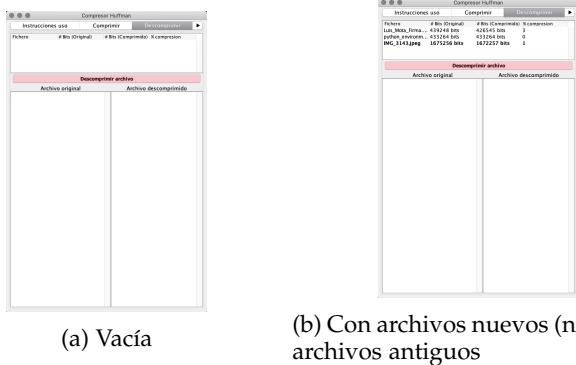


Figura 4: Tabla de archivos comprimidos

- Si es de otro tipo (imagenes, documentos, audios, videos, etc.) se abrirá el archivo correspondiente.

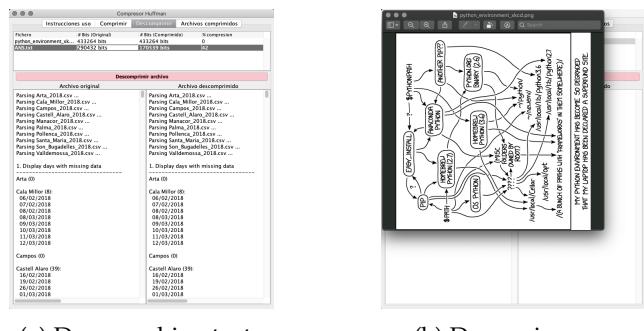


Figura 5: Ejemplos de descompresión de archivos

De esta manera se puede comprobar la correcta descompresión.

9.3. Ejecución del programa

RECOMENDACIÓN DE EJECUCIÓN: Ejecute el programa dos veces (cerrando entre cada vez) para poder ver la funcionalidad de carga de ficheros ya comprimidos (en la segunda y demás ejecuciones se podrá apreciar).

Para poder **comprimir un archivo** se deben seguir los siguientes pasos:

- Seleccionar la pestaña de Comprimir
- Seleccionar uno o varios archivos mediante el selector de archivos o arrastrándolos en la ventana
- Pulsar el botón de Comprimir
- Al acabar la compresión los ficheros resultantes de guardarán en las carpetas correspondientes (ya se menciona dicho proceso en las secciones anteriores).

Para poder **descomprimir** un fichero se debe:

- Selecinciar la pestaña de Descomprimir.
- Seleccionar uno de los archivos de la tabla.
- Pulsar botón de Descomprimir archivo.
- Una vez finalizada la descompresión se guardará el fichero descomprimido dentro de la carpeta decompressed.

- El fichero descomprimido aparecerá en la ventana o se abrirá dependiendo del formato del archivo ([\[5\]](#)).

Para poder visualizar la **información de la compresión de un archivo** se debe:

- Seleccionar la pestaña de Archivos comprimidos.
- Seleccionar un archivo de la tabla de archivos comprimidos.
- Para ver el **árbol Huffman** pulsar en la pestaña de Árbol Huffman.
- Para ver los **códigos Huffman** asociados a cada byte del fichero pulsar en la pestaña de Códigos Huffman.

10. PROBLEMAS ENCONTRADOS

Durante la implementación de esta práctica no se han encontrado problemas que valga la pena mencionar.

11. CONCLUSIÓN

La compresión de archivos puede ser intuitiva si se tiene el algoritmo adecuado, y puede ser muy eficiente si se optimiza el código (como se ha visto en el caso de utilizar los códigos de Huffman de longitud variable).

La implementación de esta práctica ha sido muy exhaustiva, ya que se han implementado un gran número de funcionalidades en la parte del *back*. Esto nos ha dado una visión de cómo pueden ser, a grandes rasgos, el desarrollo de las distintas aplicaciones que usamos.

REFERENCIAS

- [1] Algoritmo de Huffman <https://algs4.cs.princeton.edu/55compression/Huffman.java.html>
- [2] Teoría de compresión de datos <https://algs4.cs.princeton.edu/55compression/>
- [3] JGraphX <https://github.com/jgraph/jgraphx>
- [4] Documentación de la librería JGraphX https://jgraph.github.io/mxgraph/docs/manual_javavis.html#1.3
- [5] Ejemplo árbol con nodos colapsables <https://stackoverflow.com/questions/47972193/how-to-create-a-hierarchical-tree-with-collapse-nodes-in-jgraphx>

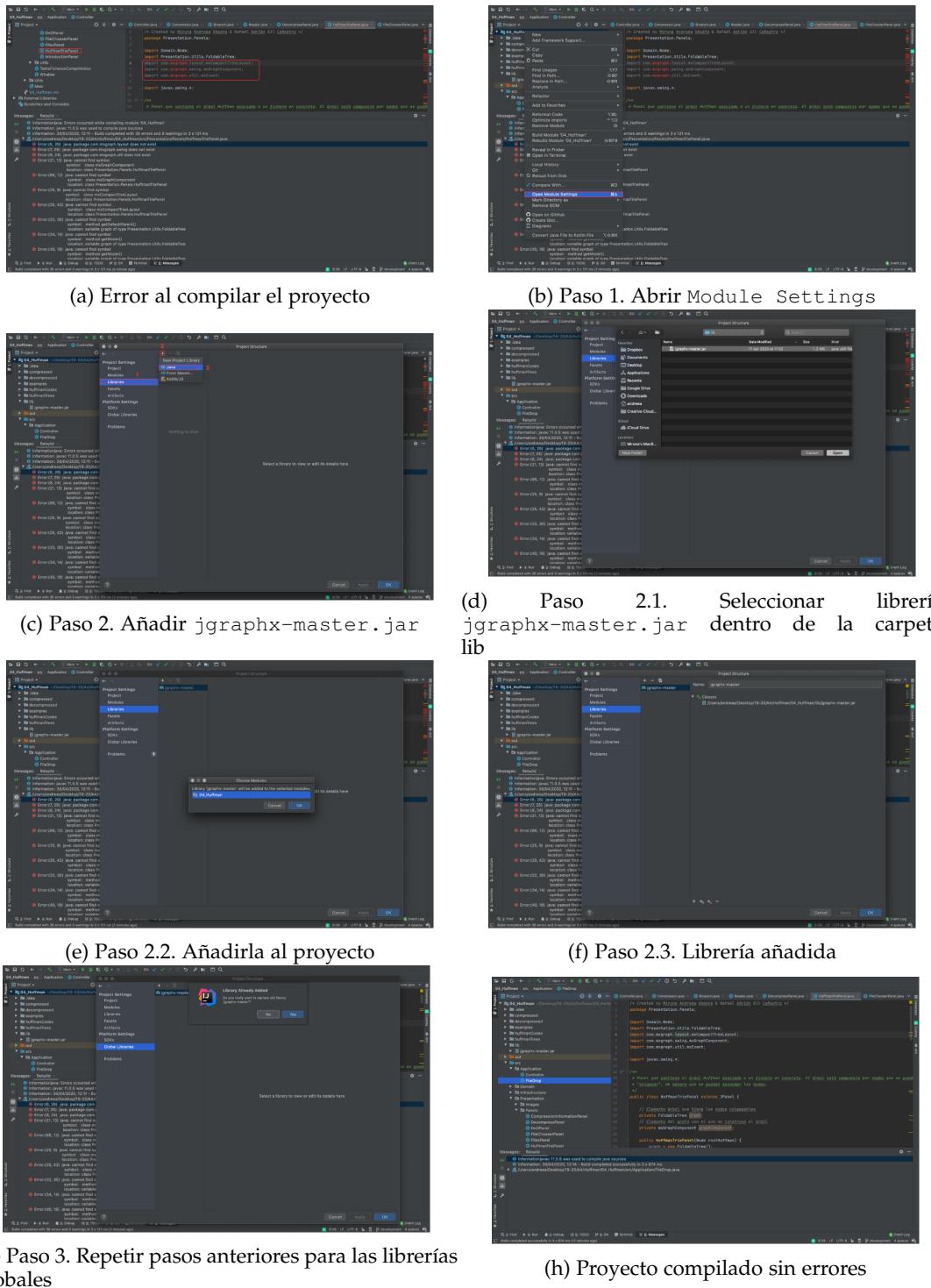


Figura 6: Pasos para seguir cuando proyecto no compila por culpa de que no encuentra la librería jgraphx-master.jar