Capítulo 4

Jonathan Zinzan Salisbury Vega
Universitat de les Illes Balears
Palma, España
jonathan.salisbury1@estudiant.uib.cat

Joan Sansó Pericás
Universitat de les Illes Balears
Palma, España
joan.sanso4@estudiant.uib.cat

Joan Vilella Candia
Universitat de les Illes Balears
Palma, España
joan.vilella1@estudiant.uib.cat

El espacio de almacenamiento y el tamaño de los archivos ha sido uno de los principales problemas desde el inicio de la existencia de los ordenadores. Los algoritmos de compresión, por ende, son de las herramientas más buscadas por parte de los ingenieros, ya que permiten ahorrar espacio de almacenamiento, tráfico de red, etc. Los algoritmos ávidos, en cambio, lo que hacen es ahorrar tiempo. Al elegir en cada paso la mejor opción, y al no volver atrás, brindan normalmente una solución decente, aunque no óptima, en un tiempo aceptable.

Keywords—Huffman, Compresión, Árbol, Greedy, Ávido

I. INTRODUCCIÓN.

Los algoritmos ávidos (popularmente conocidos como "greedys") conforman un paradigma de la programación que tiene un acercamiento a la solución en la que se elige la mejor opción posible disponible al momento. Por lo tanto, el algoritmo no se preocupa si la solución actual traerá consigo la solución óptima.

Con este acercamiento, el algoritmo nunca deshace su decisión más reciente. Trabaja con un método "top-down". La estrategia de búsqueda sigue una heurística[1], es gracias a esto por lo que el algoritmo puede discernir entre cual es la "mejor decisión" en ese momento. De todos los paradigmas de programación, es de los que menos dificultades plantea a la hora de diseñar una solución. Una vez diseñada la solución, se deberá comprobar la corrección y estado de las soluciones que encuentra.

Por estos motivos, suelen ser una buena opción para encontrar una "buena" solución para casos en los que computacionalmente es inaceptable utilizar un algoritmo de fuerza bruta. Típicamente estos problemas son de optimización.

Uno de los ejemplos más comunes y que, es el que se ha implementado a lo largo de esta práctica, es el Algoritmo de Huffman. Este es el algoritmo encargado de la construcción de los códigos homónimos, los cuales son utilizados para la compresión de datos.

Dicho algoritmo fue muy útil en su momento ya que permitía comprimir en gran medida los datos y por lo tanto facilitaba su transmisión. Aún a día de hoy se sigue utilizando, pero ha perdido importancia frente a otros algoritmos. Ya que este comprime más los archivos, pero a costa de un mayor tiempo de codificación y decodificación.

A lo largo de este paper se mostrará la implementación llevada a cabo de este algoritmo en la que generará un código de Huffman, para un archivo seleccionado por el usuario. Además de mostrar una serie de métricas, como tamaño pre y post compresión, como también la comparación entre la solución real con la teórica.

Finalmente, se ha implementado la compresión y descompresión de archivos. En este caso, cuando se comprime se guarda en el archivo toda la información necesaria para su posterior descompresión.

II. CONTEXTO Y ENTORNO DEL ESTUDIO

Uno de los requisitos a la hora de realizar la práctica es el uso del lenguaje de programación Java. Además, se ha dado la opción de elegir entre dos IDE (*Integrated Development Environment).*

- NetBeans
- IntelliJ

En este caso se ha escogido el IDE de NetBeans por familiaridad de uso. Además, se utiliza una herramienta de control de versiones (Git). Más específicamente su versión de escritorio *Github Desktop* por su facilidad de uso mediante interfaz gráfica.

[1] En ingeniería, una heurística es un método basado en la experiencia que puede utilizarse como ayuda para resolver problemas de diseño, desde calcular los recursos necesarios hasta en planear las condiciones de operación de los sistemas. Mediante el uso de heurísticas, es posible resolver más rápidamente problemas conocidos o similares a otros conocidos.

III. DESCRIPCIÓN DEL PROBLEMA

La codificación Huffman, tanto en ingeniería informática como en teoría de la información, es un tipo de código prefijo óptimo usado para la compresión sin pérdida de datos.

A. *Códigos prefijo*

Un código prefijo es un mecanismo de codificación. La particularidad que tiene es que: "Sea el conjunto completo de posibles valores codificados, estos no deben contener ningún valor que comience con cualquier otro valor del conjunto". Este tipo de códigos son especialmente útiles ya que, si se tiene un flujo de información constante, se puede analizar cada uno de los valores sin necesidad de saber donde empieza y acaba cada uno de ellos.

Sea sugerida la siguiente codificación, para ejemplificar la utilidad de este tipo de códigos:

$$[1, 2, 33, 34, 61]$$

Y, por ejemplo, se recibe la siguiente secuencia:

$$[16113334]$$

Empezando de izquierda a derecha el análisis sería el siguiente.

1. [**1** 6113334]: Se lee en primer lugar el 1, no puede ser otro valor que 1. Ya que, por definición de código prefijo, si 1 forma parte de la codificación, ningún otro código puede empezar por 1.
2. [1 **61** 13334]: A continuación, se hace lo mismo con los números restantes. Se leería el 6, al no estar dentro se seguiría con el 1. Entonces se analiza 61 como prefijo, al formar parte, ningún otro puede empezar por 61. La sucesión de pasos sería la siguiente:
3. [1 61 **1** 3334]
4. [1 61 1 **33** 34]
5. [1 61 1 33 **34**]

Como se puede observar, no ha habido problema a la hora de distinguir el 33 y el 34. El hecho de que ambos empiecen por 3 no importa; pueden existir prefijos compartidos, mientras que un valor no use en su totalidad como prefijo otro valor.

Para acabar de entender cómo funcionan se pondrá como ejemplo una codificación que no cumple estos requisitos:

$$[1, 3, 12, 23]$$

Y la siguiente secuencia:

$$[12323]$$

Empezando al igual que la manera anterior (de izquierda a derecha):

[**1** 2323]: La secuencia comienza con un 1, pero este 1 podría ser el propio 1 que está en la codificación o podría pertenecer al código 12. No existe manera de saberlo a no ser que se añada información adicional por parte del emisor, por lo que hay ambigüedad. Por lo que habría esatas posibles soluciones:

- [1 23 23]
- [12 3 23]

Y es por eso por lo que un código de prefijo es útil, si desea distinguir valores en una secuencia que no tiene 'marcadores' explícitos de dónde comienza y termina un valor.

B. *¿Cómo funciona la codificación Huffman?*

La idea es la de asignar códigos de longitud variable a los distintos caracteres que se reciben como input. La longitud de dichos códigos está basada en la frecuencia de aparición en el archivo a analizar. Los caracteres que tengan una mayor aparición en el mensaje (mayor frecuencia) se le asignarán los códigos de menor longitud y por ende a los menos frecuentes, los más largos. Esto se realiza de esta manera por términos de eficiencia, tanto de velocidad como de espacio. Ya que esto permitirá que sea más rápido decodificar como enviar dichos mensajes y a su vez ocuparán menos espacio.

Se podría decir que la salida de aplicar al algoritmo de Huffman es una tabla con códigos de longitud variable, la cual será utilizada para codificar los símbolos fuente (los del archivo input).

Para generar esta tabla, se ha de construir un árbol binario en el que cada uno de los nodos hoja, será un carácter con su frecuencia asociada. Después, de manera consecutiva se irán uniendo parejas de nodos que tengan una menor frecuencia, el padre de dichos nodos será la suma de sus frecuencias. Se seguirá este proceso hasta que no queden más nodos hojas por unir (ningún carácter por tratar) a ningún nodo superior.

Finalmente, se deberán etiquetar las aristas con 1's o 0's atendiendo al convenio elegido (hijo derecho e izquierdo, respectivamente, por ejemplo). El código resultante para cada uno de los caracteres es el propio de la lectura de dicho árbol hasta llegar al nodo correspondiente.

C. *Ejemplo de construcción*

Sea el siguiente String un ejemplo de mensaje a analizar.

Ilustración 1

1. El primer paso será calcular las frecuencias de dichos caracteres.

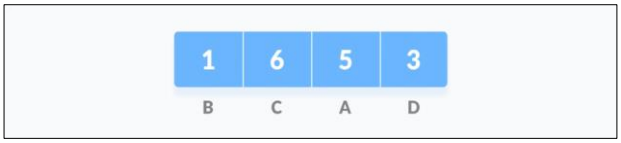

Ilustración 2

2. Una vez calculadas dichas frecuencias, se almacenarán en una cola de prioridad, quedando así los caracteres ordenados de menor a mayor frecuencia.

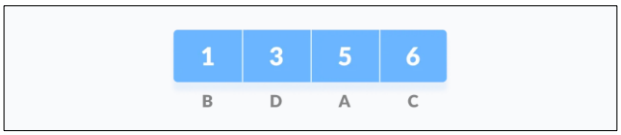

Ilustración 3

3. A continuación, se deberá hacer cada carácter un nodo hoja, con su frecuencia asociada.
4. Se crea un nodo intermedio. Se le asignan de izquierda a derecha los nodos hijo correspondientes a los nodos de menor frecuencia. Se le asigna a dicho nodo la suma de ambos nodos hijo.
5. Se borran los dos nodos de menor frecuencia.
6. Se añade el nuevo nodo intermedio.

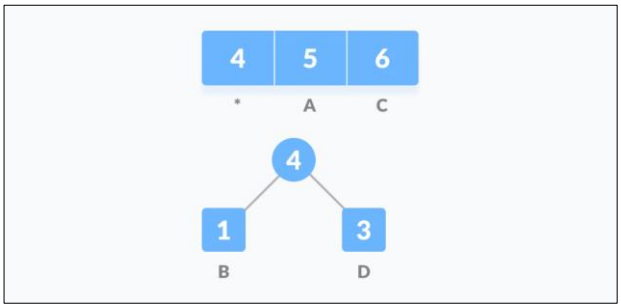

Ilustración 4

7. Se repiten los pasos tres y cinco para el resto de los caracteres.

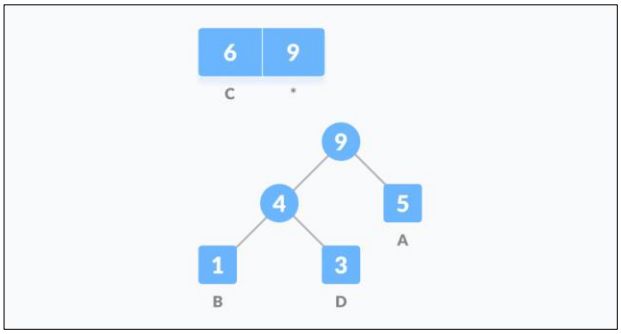

Ilustración 5

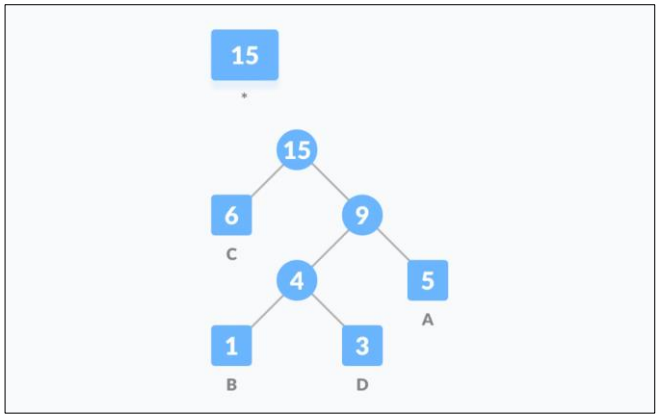

Ilustración 6

Por cada nodo no hoja, se asigna 0 para el de la izquierda y uno para el de la derecha.

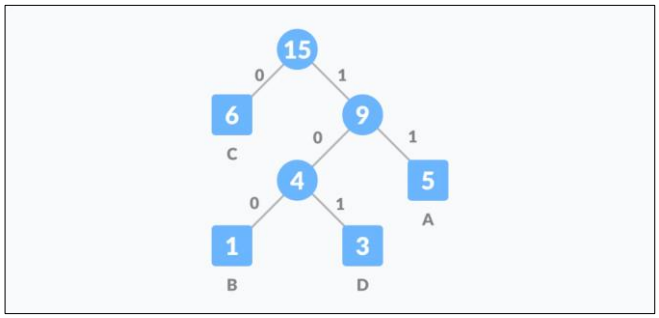

Ilustración 7

Con esto, el árbol ya estaría construido. Ahora quedaría construir la tabla de frecuencias que quedaría de la siguiente manera:

Character	Frequency	Code
A	5	11
B	1	100
C	6	0
D	3	101

Ilustración 8

Como se puede observar, los caracteres de menor frecuencia tienen códigos de menor longitud. Además, se cumple la regla de los códigos prefijo en la que un código no debe pertenecer nunca de manera completa al prefijo de otro.

En esta explicación a nivel teórico de cómo funciona el algoritmo de Huffman no se ha hecho hincapié en detalles técnicos como: uso y tamaño de buffer de lectura, estructuras de datos adicionales como HashMaps, entre otros. Estos serán explicados en el apartado de “Solución propuesta”.

D. *Entropía*

La entropía ha sido un concepto que se ha ignorado en la explicación teórica de como funciona el algoritmo de Huffman, pero tiene una gran relevancia a la hora de analizar la efectividad de este. Se podía definir como la cantidad media de información contenida por un símbolo y se expresa de la siguiente manera:

$$H(X) = \sum_{i=1}^{n} p_i \log_2 \frac{1}{p_i} = -\sum_{i=1}^{n} p_i \log_2 p_i$$

Donde X sería la variable aleatoria que toma los valores de un conjunto $\{x_1, x_2, \dots, x_n\}$ con sus respectivas probabilidades $p_1, p_2, \dots, p_n$ donde $\sum_i p_i = 1$. Se podría decir que es un concepto que expresa los límites de la compresión de la información.

Es importante recalcar que para calcular de manera correcta la entropía, es necesario conocer la distribución que siguen los datos. Pero esto es muy complicado, por lo que se suele usar la aproximación vista antes.

IV. SOLUCIÓN PROPUESTA

A. *Preámbulo*

Toda la lógica utilizada para la resolución de la codificación Huffman está realizada en la clase "Huffman" dentro de la carpeta del modelo. A diferencia de otras prácticas, ha sido conveniente utilizar más de una clase dentro del modelo. El modelo por una parte será el encargado de hacer las llamadas a funciones propias de la clase Huffman para la creación del árbol y de los códigos. Esta información la utilizará para realizar la compresión de los archivos, además la clase "Model" también realizará funciones adicionales como obtención de las frecuencias, cálculo de entropías, entre otros.

En este apartado se limitará la explicación a la clase "Huffman" como tal, ya que es la propia que resuelve el problema propuesto y a su vez es la tarea obligatoria para esta entrega. Más adelante se comentarán las funcionalidades de la clase "Modelo" propia y se aprovechará para comentar las funcionalidades añadidas de compresión y descompresión de archivos.

B. *Explicación*

Para la realización de la codificación se utilizan tres estructuras de datos:

- HashMap
- Cola de prioridad
- Nodo

El HashMap en el que se almacena por cada byte, el código de Huffman asociado. La cola de prioridad cumple el mismo propósito explicado a nivel teórico. Es una cola de nodos en la que quedarán ordenados cada uno de los bytes asociados con su frecuencia de mayor a menor (ordenados por frecuencia). Esto permitirá la creación del árbol.

Finalmente, la estructura nodo. Esta se explicará un poco más en profundidad porque es de creación propia y se ha diseñado para el cumplimiento de esta práctica.

1) *Clase nodo*

Los atributos de esta clase son:

- Value (byte)
- Frequency (int)
- isLeaf (boolean)
- left, right (Node)

Esta clase tiene dos constructores para distinguir los dos tipos de nodos que compondrán el árbol de Huffman. Por una parte, estarán los nodos hoja, que para ello se utiliza el atributo "isLeaf" y por otra estarán los nodos intermedios. El funcionamiento es exactamente el mismo que a nivel teórico. Los nodos hoja almacenan el valor y la frecuencia de aparición y los intermedios la suma de ambos hijos (izquierdo y derecho). Por esta razón los nodos hoja no se instancian con nodos hijos. Por otro lado, los nodos intermedios no almacenan ningún valor y cuando se llama al constructor, se le han de pasar los dos nodos (izquierdo y derecho) del cual proviene, el constructor es el encargado de realizar la suma de ambas frecuencias.

2) *Métodos*

Los métodos serán descritos por orden de llamada desde la clase "Model" para un seguimiento secuencial del funcionamiento.

CreateTree: Este método recibe un HashMap con las frecuencias asociadas a cada uno de los Bytes. Con esto, crea un nodo hoja por cada una de las entradas y las agrega a la cola (la cola se encarga por si sola de ordenarse). Finalmente recorre la cola cogiendo los dos nodos con menos frecuencia, crea un nodo intermedio con la suma de ambos y añade este nuevo nodo a la cola. La raíz del árbol será el nodo restante de toda la operación.

```
public void createTree(HashMap<Byte, Integer> freqs) {
    freqs.forEach((k, v) -> queue.add(new Node(k, v)));
    while (queue.size() > 1) {
        Node parent = new Node(queue.poll(), queue.poll());
        queue.add(parent);
    }
    treeRoot = queue.poll();
}
```

Ilustración 9

GenerateCodes: El método que se llama desde el modelo no es este, es "generateCode" pero se explicará este porque es el que realiza el trabajo como tal. El método acabado en singular hace una llamada a este método con la raíz del árbol y devuelve los códigos generados en la estructura de HasMap.

Este método es el que crea los códigos de Huffman por cada byte, comenzando desde la raíz y realizando un recorrido de preorden sobre el árbol.

El caso base será que el nodo sea una hoja, por lo que se hará una inserción en el Hashmap con el valor que se ha obtenido hasta llegar a ese nodo y el valor del byte. Después se realizan dos llamadas recursivas, una correspondiente al hijo izquierdo y, por ende, se le asocia un 0 a su codificación Huffman. Y otra llamada a su hijo derecho asociándole un 1 a su codificación.

```
private void generateCode(Node node, String code) {
    if (node.isLeaf) {
        codes.put(node.value, code);
    } else {
        generateCode(node.left, code + "0");
        generateCode(node.right, code + "1");
    }
}
```

Ilustración 10

V. PATRÓN MVC

Como ya se ha mencionado en la introducción, para realizar esta práctica se ha utilizado el patrón de arquitectura de software MVC.

Esta técnica consiste en separar los datos de una aplicación, la interfaz de usuario, y la lógica de control en tres componentes distintos.

- Modelo: En esta capa es en la que se trabaja con los datos, por lo que contendrá los mecanismos para acceder a la información y también actualizar su estado. Además, contiene toda la lógica del problema,
- Vista: Contiene el código que muestra la aplicación, es decir, que va a producir la visualización de la interfaz del usuario como de los resultados.
- Controlador: Es la parte que actúa como intermediario entre el modelo y la vista, gestionando el flujo de información entre ellos.

Las ventajas principales del MVC son: Escalabilidad, facilidad de tratamiento de errores y reutilización de componentes. Existen otras ventajas, pero esta arquitectura se aprovecha en mayor medida en aplicaciones web, el cual no es este caso.

La principal desventaja que existe del patrón MVC es la complejidad que añade a la programación. Ya que, para el mismo problema, hay que modificar el acercamiento para que quepa dentro de esta arquitectura. Lo que implica una mayor complejidad.

A. Descripción del modelo

Se ha diseñado la clase *Model* la cual será la encargada de almacenar todos los algoritmos y funciones auxiliares para resolver la otra parte del problema. La relacionada con la compresión y descompresión de datos, a su vez con el cálculo de estadísticas como la entropía o el cálculo del tamaño esperado.

Una de las restricciones más importantes a la hora de realizar la lectura de un archivo era que el buffer de lectura no tenga la longitud de todo el archivo. Esto debe ser así porque sino el programa deja de ser escalable, porque ante un archivo lo suficientemente grande, este se cargaría todo en memoria y el programa se quedaría congelado, con la imposibilidad de terminar dicha tarea.

Es por esta razón que al inicio de la clase se define el tamaño del buffer que se utilizará, en este caso uno de 512 bytes. Se ha elegido este tamaño ya que, después de hacer unas pruebas, brinda el mejor rendimiento sin ser demasiado grande para tener en memoria. Además, está definido como constante el tamaño en bits de un byte (8), porque dicho valor se utiliza a lo largo del programa para distintas operaciones.

Las variables que se utilizarán a lo largo de toda la clase son:

- *huffmanCodes* (HashMap)
- *frequencies* (HashMap)
- *buffer* (array de bytes)

Por sus nombres es fácil intuir para que serán utilizadas: Almacenamiento de los códigos de Huffman, almacenamiento de la frecuencia de aparición y buffer. Ambos HashMaps tienen como clave el Byte (que sería equivalente al carácter en el marco teórico) y tienen como valor el String asociado a su código y un entero que representa su frecuencia, respectivamente.

1) Métodos clave

En cuanto a compresión y descompresión de los archivos, los métodos más importantes son:

- Compress
- Decompress

Ya que se podría decir que son los métodos finales que hacen uso de todas las funciones auxiliares, es por eso por lo que son los únicos métodos públicos. Tanto los de la clase Huffman como los propios métodos auxiliares dentro de la clase *Model*. A continuación, se explicarán el funcionamiento general de dichos métodos y después se ahondarán en los métodos auxiliares que se consideren más relevantes.

Compress: Este método recibe por parámetro el archivo especificado por el usuario y sigue este conjunto de pasos:

1. Calcula las frecuencias del archivo

2. Crea el árbol de Huffman
3. Genera los códigos
4. Comprime el archivo y lo guarda en el mismo path que el archivo pasado por parámetro.

Es importante tener en cuenta que a la hora de comprimir el archivo se escribe en él, el árbol de Huffman. Es por esta razón que la clase *Node* se ha hecho Serializable. Ya que esto permite su escritura y lectura.

Decompress: Este método recibe por parámetro el archivo que se quiere descomprimir y realiza lo siguiente:

1. Crea el archivo donde se guardará la información descomprimida en el mismo path que el archivo pasado por parámetro.
2. Llama al método para descomprimir.

Como se puede observar, estos dos métodos son de muy alto nivel y por ello no profundizan en el proceso de compresión y descompresión. Por esta razón a continuación se explicarán los métodos que se encargan exclusivamente de esta tarea, dichos algoritmos son:

- compressAndWriteFile
- readAndDecompressFile

CompressAndWriteFile: Este método recibe como parámetro el archivo original, el nuevo archivo donde se guardará la información comprimida y el árbol de Huffman.

Una vez abierto los streams de lectura (del archivo original) y de escritura (del nuevo archivo), se reserva un espacio para el offset de los bits del último byte. Este será sobreescrito si el offset es distinto de 0, es decir, si el último byte necesita padding.

A continuación, se escribe el árbol de Huffman en el archivo final. Ahora, mediante un bucle se irán leyendo bytes hasta el final del archivo. Dentro del bucle se van añadiendo los códigos de Huffman al buffer del String. En caso de que la longitud del buffer no sea un múltiplo de 8, solo se procese el substring que contiene bytes completos y se deja el resto en el buffer para la próxima lectura.

Después, se realiza la conversión de un buffer de Strings a un array de bytes. Se escribe la información comprimida en el archivo de salida. Se borra la información que se ha escrito y se dejan los próximos bits para la siguiente iteración. Para acabar con las operaciones dentro del bucle, se lee el siguiente buffer.

Una vez acabado el bucle, si el último byte ha necesito padding, se escribe el offset en el último byte. Para ello, se utiliza una función auxiliar llamada *writeLastByte*.

ReadAndDecompressFile: Este método recibe como parámetro el archivo a descomprimir y el archivo donde se guardará la información descomprimida.

Una vez abierto el stream de lectura, se lee el primer byte, el cual contiene cuántos 0's se han usado para el paddding en el último byte del archivo. Después de esto, se lee el árbol de Huffman almacenado en el archivo comprimido.

A continuación, se lee la información comprimida en el buffer de bytes. Después de esto inicia el bucle donde en primer lugar se añaden los bits del byte al buffer String.

Si el número de los bytes leídos es menor que el tamaño del buffer, significa que se ha llegado al EOF (End Of File) y por esto, se tiene que quitar el padding extra de bits leídos al inicio del String binario.

Se vuelve a leer y si se llega al EOF, se sabe que el buffer String contiene el último byte del archivo, por lo que no hay que volver a leerlo. Finalmente se quita el padding extra.

B. Descripción del controlador

Se ha diseñado la clase Controller que se encarga de gestionar eventos y manejar la interacción entre el resto de las partes. Para ello se han implementado algunos métodos que servirán de listeners [2] para los botones de la vista.

Cuando el usuario utilice algún botón de la interfaz este lo notifica al método correspondiente que realiza la acción necesaria si es posible y devuelve los resultados a la vista para que lo muestre. Para que el usuario pueda disfrutar de una interfaz gráfica fluida se hace uso del concepto de Concurrencia que se explica a continuación.

1) Concurrencia

La programación concurrente es una forma de cómputo en la que el trabajo se divide en varios hilos de ejecución distintos que trabajan simultáneamente. Esto suele mejorar el rendimiento de una aplicación al poder realizar cálculos largos en un hilo aparte. Para este proyecto se ha decidido utilizar este concepto sobre el modelo para realizar los cálculos en segundo plano.

A medida que va avanzando la compresión o descompresión el hilo del Modelo avisa del progreso al

[2] Un Event Listener es un procedimiento o función en un programa que espera a que ocurra un evento. Ejemplos de un evento son el usuario haciendo clic o moviendo el ratón, presionando una tecla en el teclado.

Controlador que actualiza la barra de progreso de la vista en el hilo principal.

C. *Descripción de la vista*

Para la vista se han implementado dos maneras de seleccionar el archivo deseado.

- En primer lugar, hay un sistema de *Drag and Drop* que permite al usuario deslizar archivos sobre la interfaz y soltarlos para seleccionarlo. Esto se ha realizado utilizando la clase *DropTargetListener* de java.
- En segundo lugar, hay un botón que permite al usuario seleccionar el archivo haciendo uso del *JFileChooser* de java.

Una vez se ha seleccionado el archivo se pone visible un pequeño JPanel que muestra información adicional del archivo selecionado como por ejemplo el tamaño.

Después de que el usuario haya elegido un archivo se habilitan los dos botones de la vista para poder realizar la operación deseada sobre el archivo.

La clase *View.java* consta de cuatro métodos públicos

- *addListeners()*: que permite añadir un listener a los botones.
- *getSelectedFile():* que devuelve el archivo selecionado actual
- *setProgress():* que permite actualizar el valor de la barra de progreso.
- *showCompressionInfo():* este método crea un pequeño JDialog donde se muestra la información de la compresión realizada. Además tiene un botón que permite al usuario ver los códigos de Huffman generados durante el proceso.

VI. Estudio de la compresión en diferentes tipos de archivos

Para comprobar la eficacia del programa de compresión, se ha llevado a cabo la prueba de *Canterbury Corpus*[3]. Esta prueba consiste en aplicar el algoritmo de compresión sobre 11 archivos diferentes, con distintos formatos y cantidades de información:

Tamaño (bytes)	Tipo de información
152,089	Texto en inglés
125,179	Shakespeare
24,603	Código fuente html
11,150	Código fuente C
3,721	Código fuente LISP
1,029,744	Hoja de Cálculo Excel
426,754	Texto técnico
481,861	Poesía
513,216	CCITT test set
38,240	Ejecutable SPARC
4,227	Página del manual GNU

Tabla 1

Se ha comprimido cada uno de estos archivos y apuntado los siguientes datos:

- Tamaño del archivo original.
- Tamaño esperado de los datos comprimidos.
- Tamaño actual del archivo comprimido (que es simplemente datos comprimidos + árbol de Huffman + byte del offset)
- Tamaño del Árbol (Tamaño del archivo original – comprimido – 1)

Estos son los resultados:

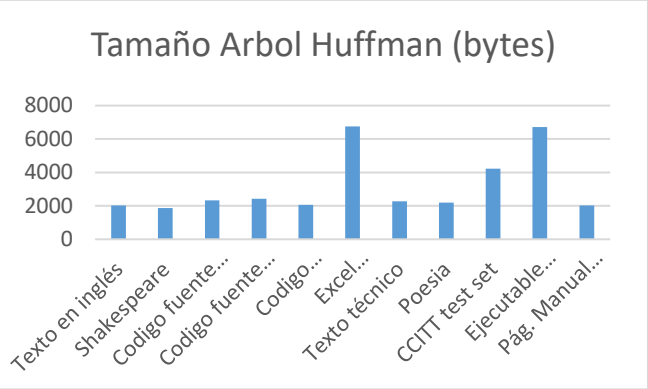

Gráfico 1

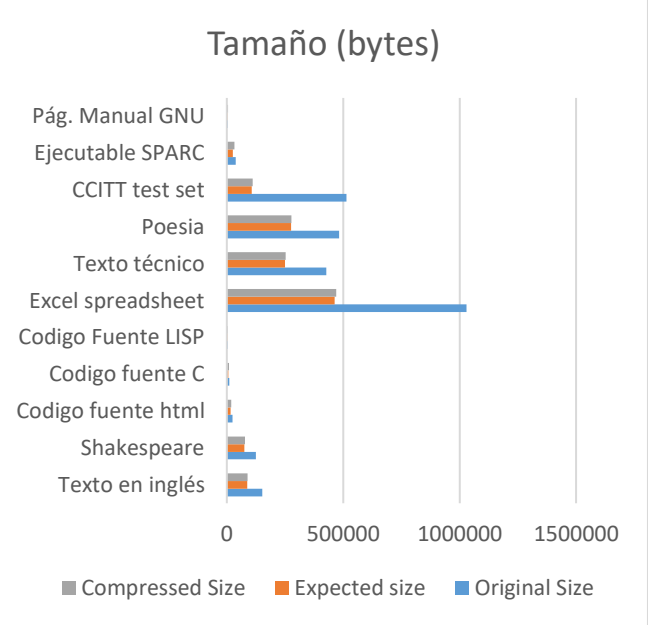

Gráfico 2

[3] https://en.wikipedia.org/wiki/Canterbury_corpus

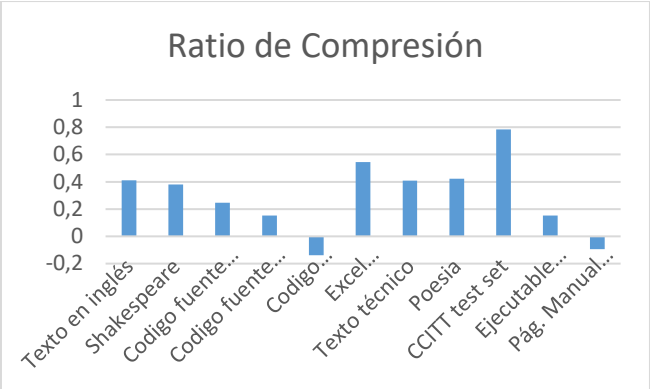

Gráfico 3

Como se puede observar, casi siempre se obtiene una buena compresión, con la excepción de en el Código fuente LISP y el Ejecutable SPARC. Después de hacer más pruebas, se ha concluido que en archivos pequeños (menor que unos 15kb), el archivo resultante será casi siempre mayor que el original. Esto es debido al tamaño del árbol del Huffman, que está escrito en el archivo en forma de Objeto Serializable de Java, ocupa unos cuantos Kilobytes, lo que hace que, al sumarle los datos comprimidos, exceda el tamaño del archivo original.

En general, en los archivos de texto la compresión resulta bastante óptima. Esto es debido a que los textos tienen una distribución no uniforme del uso de las letras, de hecho, las letras siguen la Ley de Zipf[4], donde la segunda letra más usada sale ½ de las veces que la primera, la tercera ⅓ de las veces, etc. También tener en cuenta que las mayúsculas salen mucho menos que las minúsculas, y muchos caracteres de control apenas salen.

Se puede observar cómo los archivos que tienen el árbol de Huffman más grande son el archivo de Excel y el ejecutable de SPARC, mientras que los archivos de texto tienen un árbol más pequeño. Esto tiene sentido, ya que los archivos de Excel tienen muchos más datos que los que simplemente guarda el usuario, contiene fórmulas, gráficos, datos para asegurar la consistencia, celdas vacías… en general, muchos datos binarios. El archivo SPARC, al ser un ejecutable, contiene datos binarios, así que también usará muchos bytes distintos. Mientras que los archivos de texto apenas usan la mitad del rango del Byte.

VII. Manual de usuario

A continuación, se mostrará un ejemplo de uso del programa.

Cuando se ejecuta el programa se puede observar la siguiente pantalla:

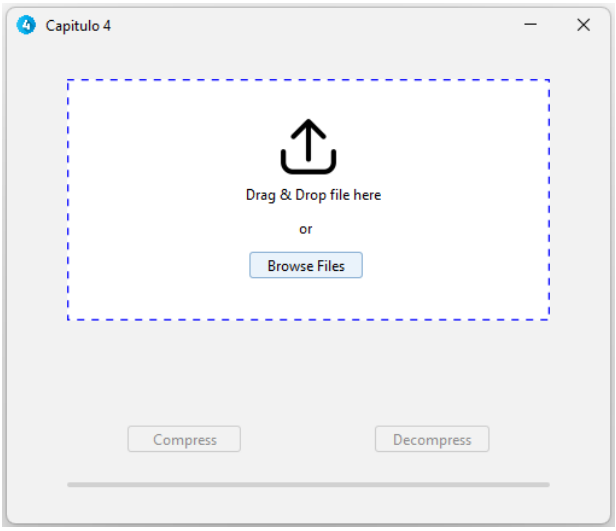

Ilustración 11

En esta pantalla se pueden distinguir dos zonas muy claras, por un lado, la parte donde se añade el archivo y, por otro lado, la parte los botones para tratar con el mismo.

Como se ha explicado en apartados anteriores, se pueden añadir los archivos de dos maneras:

- Con explorador de archivos.
- Con “Drag and Drop”

Mediante el explorador de archivos, una vez se hace click en el botón “Browse Files”. Una vez hecho esto nos aparecerá la siguiente ventana de navegación.

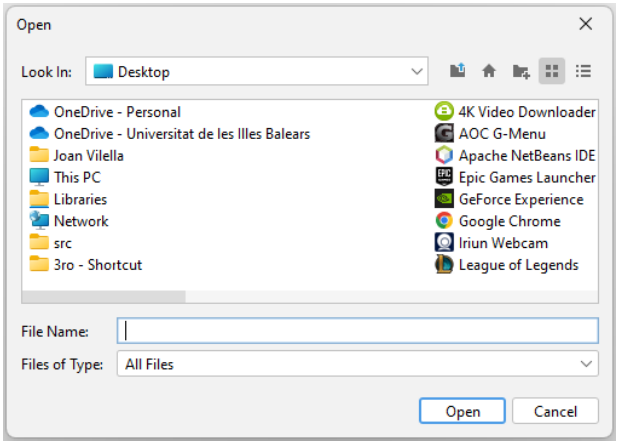

Ilustración 12

Desde aquí, se deberá buscar el archivo deseado y hacer click en “Open”.

La otra opción es arrastrar el archivo dentro del recuadro delimitado con líneas azules.

[4] https://es.wikipedia.org/wiki/Ley_de_Zipf

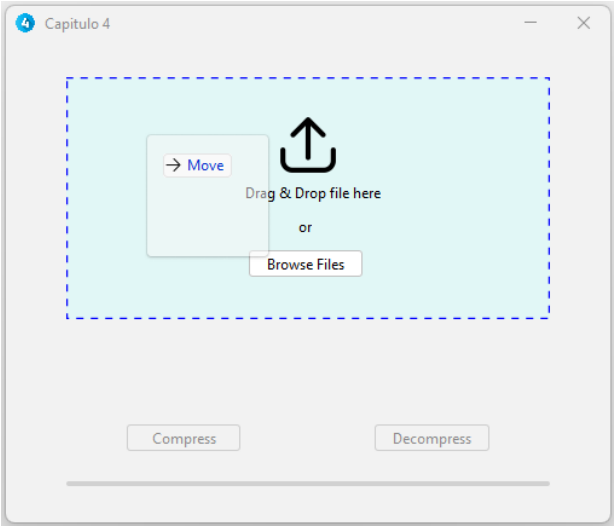

Ilustración 13

Dicho recuadro se coloreará en verde, indicando que ya se puede soltar el archivo. Una vez "cargado" el archivo, se mostrará el archivo en pantalla y quedarán habilitados ambos botones de compresión y descompresión.

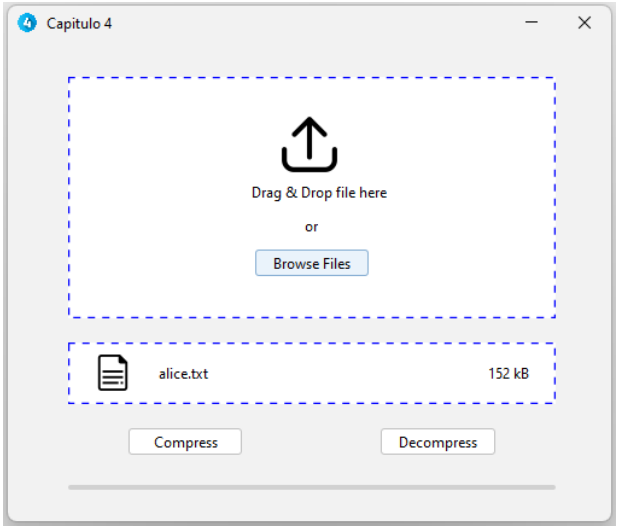

Ilustración 14

En este caso, se ha seleccionado el libro de Alicia en el país de las maravillas. Como es un texto sin comprimir, se deberá comprimir primero. Para ello se deberá hacer click en el botón "Compress".

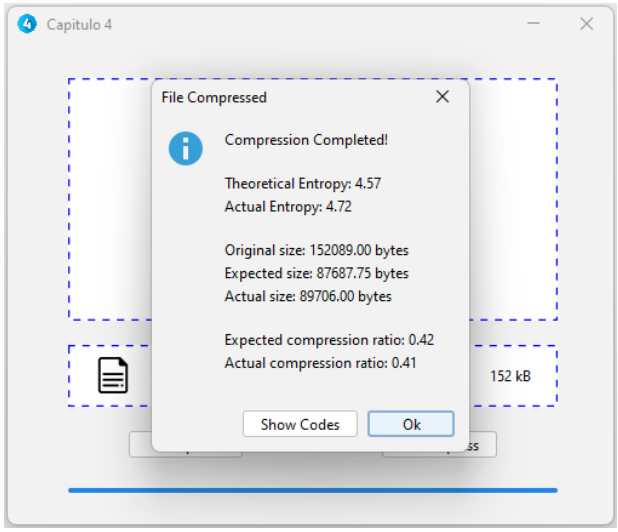

Ilustración 15

En la zona inferior se muestra el progreso del proceso de compresión. Cuando este acaba, sale una ventana emergente en la que se muestran los distintos datos del proceso de compresión. Además, se pueden ver los distintos códigos si se hace click en "Show Codes".

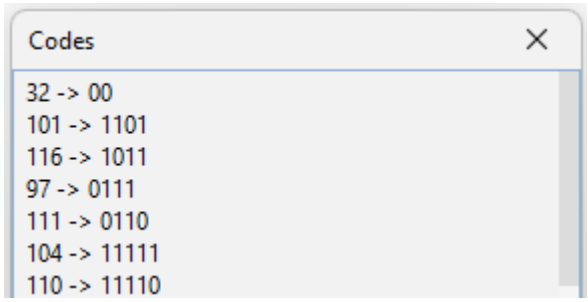

Ilustración 16

Aquí se puede observar a la izquierda el byte y a su derecha el código asociado correspondiente.

Cuando se comprime, se genera en el mismo directorio de origen del archivo seleccionado, el archivo comprimido. Con el mismo nombre pero con la terminación ".huff". Se ha optado por no quitar el formato de archivo original ya que así nos ahorramos tener que guardar que tipo de archivo era, simplemente lo podemos saber mirando el nombre.

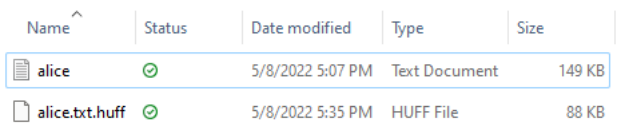

Ilustración 17

Para descomprimir este nuevo archivo y comprobar que funciona, se debería seleccionar dentro del programa y hacer click en el botón "Decompress".

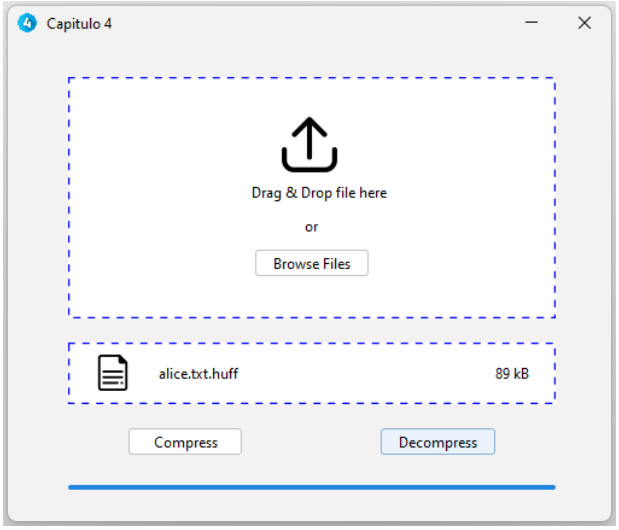

Ilustración 18

Una vez realizada la descompresión, aparecerá en el mismo directorio del archivo comprimido el nuevo archivo descomprimido. Para distinguirlo del original se ha optado por concatenarle al nombre "_dec".

Name	Status	Date modified	Type	Size
alice	✓	5/8/2022 5:07 PM	Text Document	149 KB
alice.txt.huff	✓	5/8/2022 5:35 PM	HUFF File	88 KB
alice_dec	✓	5/8/2022 5:44 PM	Text Document	149 KB

Como se puede observar, el archivo comprimido ocupa menos espacio que el original. Además, después de la descompresión, el nuevo archivo ocupa el mismo tamaño que el original, lo cual es un buen indicativo de que ha ido bien.

Finalmente, para comprobar que el proceso ha sido el correcto. Se deberán comprobar que ambos archivos son idénticos. No se han puesto imágenes de la comparación porque ocuparían un tamaño excesivo. Dicha demostración quedará ilustrada en el vídeo de la práctica.

VIII. Suposiciones y posibles mejoras

A. *Elección dinámica del tamaño de cada símbolo*

Una posible mejora seria cambiar el tamaño de cada símbolo al hacer el árbol de Huffman. Con esto se quiere decir que, en lugar de codificar un byte, elegir dinámicamente un grupo de bytes dependiendo del archivo. Por ejemplo, si el archivo es un archivo de texto codificado con UNICODE, podríamos hacer la compresión con 2 bytes, lo que ocupa cada carácter. Para imágenes con 3 canales de color, coger grupos de 3 bytes, que representaría un píxel de la imagen, etc.

B. *Escritura de los recorridos Inorden y Preorden para guardar el Árbol de Huffman.*

Actualmente, usamos la interfaz Serializable de Java para escribir el Objeto árbol en memoria al archivo. Esto ocupa unos cuantos kilobytes, y nos podríamos ahorrar un 90-95% de esos datos usando la siguiente técnica:
Escribir el recorrido Inorden y el recorrido Preorden del árbol. Solo escribiríamos el valor del nodo, es decir, el valor del byte. Con esto, tendríamos 2*N bytes más que leer, donde N sería como máximo 256. Necesitaríamos un byte extra para indicar la longitud de las dos listas, así que necesitaríamos como máximo 513 bytes.
Obviamente deberíamos implementar luego el algoritmo que reconstruye el árbol a partir de estos dos recorridos, como se puede observar en esta página: https://www.geeksforgeeks.org/construct-tree-from-given-inorder-and-preorder-traversal/

IX. Conclusiones

La codificación Huffman supone una técnica relativamente fácil de implementar y de gran utilidad. Esto ha permitido que aun a día de hoy siga estando vigente, ya que es además un algoritmo que asegura una no pérdida de información. Esto es algo que dependiendo de la información que se quiera tratar es muy conveniente.

Sí que es verdad que en comparación a otros algoritmos utilizados para la compresión de imágenes deja de ser tan útil. Ya que estos sacrifican un poco de la información con tal de aumentar en gran medida las velocidades de compresión y descompresión.

Ahora bien, aunque es una técnica con muchas ventajas, no siempre será una buena opción utilizarlo. Es por esta razón que es conveniente conocer estos casos donde sí supone una ventaja.

Por un lado, están los archivos de gran tamaño (no ha nivel de gigas), lo suficientemente grandes para que el hecho de escribir el árbol de Huffman no suponga que este ya supere el tamaño del archivo original. Archivos excesivamente grandes tampoco son recomendables porque ya se ha visto que existen alternativas mejores.

Otro punto también muy importante es la distribución que tengan los datos. En general, una distribución de datos no uniforme suele ser lo ideal. Cuanto más se repitan los símbolos mejor. Esto es debido a que la construcción del árbol de Huffman se beneficia mucho de esto, asignando a los símbolos más comunes las secuencias más cortas. Es por esta razón que con archivos con información muy uniforme, la técnica puede ser hasta contraproducente.

X. Bibliografía

Greedy Algorithm (programiz.com)
Greedy Algorithms - GeeksforGeeks
Prefix codes (explained simply) (github.com)
huffman.pdf (csulb.edu)