CUCEI

CENTRO UNIVERSITARIO DE CIENCIAS EXACTAS E INGENIERÍAS

Análisis de Algoritmos

Proyecto Final: Compresor de canciones

Implementado con Huffman y Algoritmo de Prim

Integrantes del Equipo:
Gutierrez Vazquez Axel
Hernandez Macias Axxel Gael
Quintero Arreola Laura Vanessa

Profesor:
Lopez Arce Delgado Jorge Ernesto

Asignatura: Análisis de Algoritmos
Sección: D06 **Calendario:** 2025 B

Fecha de entrega: 27 de noviembre de 2025

Indice

Tabla de asignación de roles

En la siguiente tabla se escribe cuáles serán las actividades/tareas por desarrollar para cada integrante del equipo, así como cuál es el resultado esperado y para cuando se debe entregar la respectiva tarea, de esta forma se tendrá un control de que es lo que falta para terminar la actividad y que todos sean parte de la elaboración del reporte y el algoritmo.

Integrante del equipo	Tarea a realizar	Resultado	Fecha de entrega
Gutierrez Vazquez Axel	Investigar que es el algoritmo de Huffman	Escribir una explicación sobre el tema y poner las referencias	18/11
	Cómo funciona el procesamiento de audio y tipos de formato	Escribir una explicación sobre el tema y poner las referencias	18/11
	Implementar interfaz gráfica y reproductor del resultado	Desarrollar y escribir código en python y dar una explicación en el desarrollo	21/11
	Diagramas de flujo	Realizar los diagramas de flujo y agregarlos al reporte	22/11
Hernandez Macias Axxel Gael	Objetivos del proyecto	Definir los objetivos que se pretenden alcanzar con el proyecto	18/11
	Investigar que es el algoritmo de Prim	Escribir una explicación sobre el tema y poner las referencias	18/11
	Agregar el algoritmo de Prim al código base	Desarrollar y escribir código en python y dar una explicación en el desarrollo	20/11
	Análisis de complejidad computacional	Analizar el resultado final del programa y definir su expresión asintótica	23/11

Quintero Arreola Laura Vanessa	Introducción del reporte	Dar una introducción del contenido de lo que habrá en todo el reporte	24/11
	Justificación y definición	Definir la justificación del proyecto y decir porqué se escogió	18/11
	Investigar que son los algoritmos voraces	Escribir una explicación sobre el tema y poner las referencias	18/11
	Implementar algoritmo base del compresor con Huffman	Desarrollar y escribir código en python y dar una explicación en el desarrollo	19/11
	Pruebas y resultados del algoritmo	Hacer las pruebas y evaluar los resultados del proyecto	23/11

NOTA* Para el desarrollo de la presentación se usará la información más relevante del reporte, es decir, cada integrante tendrá que aportar al desarrollo de la presentación tomando como punto de partida sus tareas asignadas.

Introducción

El siguiente reporte aborda el desarrollo para la elaboración de un algoritmo capaz de comprimir canciones, el desarrollo comienza desde la investigación de conceptos básicos y técnicos hasta la implementación del algoritmo en python utilizando algoritmos aprendidos en clase como Huffman y Prim.

Si bien el proyecto tiene como objetivo solo comprimir canciones, consideramos que esta clase de algoritmos son buenos para comprender mejor el funcionamiento de un algoritmo voraz, así como de tener la capacidad de ver de manera indirecta que otros algoritmos se están implementando, es decir, que si hay algún algoritmo como fuerza bruta, divide y vencerás o programación dinámica que se apliquen dentro pero no de una forma tan literal y que requieran de análisis para identificarlos.

Este proyecto nos permite plasmar todos los conocimientos adquiridos durante el curso, buscando la forma más óptima de desarrollarlo en su totalidad.

Objetivo General

- Desarrollar un sistema de compresión y descompresión de archivos de audio utilizando codificación de Huffman, optimizado mediante la reorganización de los valores de byte con un árbol de expansión mínima construido mediante el algoritmo de Prim, integrando técnicas de fuerza bruta, divide y vencerás, algoritmos voraces y de grafos, con el fin de evaluar su eficiencia en términos de tamaño de salida y complejidad computacional, garantizando la preservación de la calidad del audio y facilitando la comprensión del proceso mediante diagramas de flujo.

Objetivos

- ➢ Investigación y comprensión de nuestros algoritmos voraces que son Prim y Huffman con la finalidad
- ➢ Implementar un algoritmo de compresión utilizando codificación de Huffman con el propósito de comprimir y descomprimir canciones.
- ➢ Integrar Algoritmo de Prim para construir un árbol de Expansión mínima entre los símbolos del archivo para reorganizar los valores de byte y mejorar la distribución previa a la codificación.
- ➢ Implementar las técnicas Fuerza bruta, Divide, Algoritmos Voraces y algoritmos de grafos adecuadamente en el contenido de nuestro código.
- ➢ Evaluar su comportamiento mediante tamaño de salida y complejidad computacional garantizando como "éxito" el poder comprimir y descomprimir nuestra canción sin perder calidad.
- ➢ Analizar la aportación de todos las técnicas en el rendimiento del sistema final.
- ➢ Desarrollar diagramas de flujo para explicar de forma gráfica y sencilla el funcionamiento de nuestro algoritmo.

Justificación del proyecto

Si bien durante gran parte del semestre estuvimos trabajando con nuestro antiguo algoritmo para encontrar la mejor opción de hardware para el usuario, quisimos que el proyecto escalará para poder implementar una compresión con el algoritmo de Huffman, no nos fue posible porque el siguiente paso para este algoritmo era usar una base de datos que nos permitiera consultar información más reciente y en tiempo real sobre las mejores opciones de hardware, pero después de buscar alguna api como la que usábamos de steam para obtener información acerca de los

videojuegos y darnos cuenta que no había ninguna api así, quisimos buscar bases de datos a de las cuales tomar información y también fracasamos al intentarlo y como realmente queríamos implementar algo que sirviera de forma correcta fue entonces que decidimos migrar hacia un proyecto diferente.

Mientras buscamos que opciones teníamos y que tuviera alguna utilidad en el algoritmo de Huffman fue que se nos ocurrió llevar a cabo un compresor de canciones (música) y nos llamó la atención hacerlo para música ya que solo habíamos visto al algoritmo de Huffman para comprimir texto o imágenes, entonces nos dio curiosidad varias cosas, como hariamos para interpretar y codificar la información en un archivo mp3, porque una vez con esa respuesta sabríamos que el resto de las cosas para el proyecto se irían dando de forma natural. Entonces nos pusimos manos a la obra y tomamos como algoritmo base usar Huffman para ver si este algoritmo verdaderamente podría cumplir con la tarea de reducir el tamaño de una canción, en caso de que se pudiera conseguir tomaríamos este algoritmo como proyecto final.

Los Algoritmos Voraces

Los algoritmos voraces o también conocidos como algoritmos greedy, son métodos que buscan resolver un problema tomando siempre la decisión que parece más conveniente en ese momento, es decir, funcionan como si recorrieras un laberinto eligiendo siempre el camino que se ve más fácil, sin tener una opción para retornar para revisar decisiones anteriores, esto hace que estos algoritmos sean muy rápidos y simples, aunque no siempre aseguran la mejor solución posible. Su funcionamiento se basa en seguir una serie de pasos: escoger la opción más prometedora, comprobar que esa elección cumple las reglas del problema, evaluar si se acerca al objetivo y, si todo va bien, continuar hasta obtener una solución final.

Estos algoritmos se usan mucho en áreas como la computación, por ejemplo en Dijkstra o Prim para encontrar rutas o árboles de menor costo, la optimización combinatoria y algunas técnicas de inteligencia artificial. Además sus principales ventajas son la rapidez y la simplicidad, pero su mayor limitación es que pueden quedarse con una solución que no es la óptima porque solo analizan lo que conviene a corto plazo, por esta razón es que los algoritmos voraces al igual que otros algoritmos solo son buenos en ciertos casos, depende mucho de lo que se tenga pensado desarrollar.

Algoritmo de Huffman

El algoritmo de Huffman es una método popular para la compresión de datos sin pérdida, Fue desarrollado por David Huffman en 1952 y se basa en la frecuencia de aparición de los caracteres en un archivo.

Asigna códigos binarios más cortos a los caracteres que aparecen con más frecuencia y códigos más largos que aparecen menos; Este algoritmo es fundamental porque permite reducir el tamaño de los archivos sin perder ni un solo bit de información original al descomprimirlo. Es la base de formatos como ZIP y juega un rol crucial en la etapa final de compresión de imágenes (JPEG) y audio (MP3).

Algoritmo de Prim

Prim es un algoritmo voraz. Es un método clásico de teoría de grafos que permite construir un árbol de expansión mínima (MST) a partir de un grafo conectado y ponderado. Sirve para conectar todos los puntos de una red con la longitud o el costo total más bajo posible, evitando bucles.

Prim está totalmente basado en la propiedad de cortes y la aplica en cada iteración : Entre todas las aristas que cruzan un corte que separa un árbol parcial y el resto del grafo, la arista de menor peso siempre pertenece a algún árbol de expansión mínima (MST)
En cada paso se toma la decisión localmente óptima: elegir la arista con menor peso que conecte a un nodo nuevo con el conjunto de nodos ya añadidos al árbol.

La descripción detallada de la lógica del algoritmo de prim es la siguiente:
-Se inicializa un vértice inicial.
-Inicializa un conjunto que guarda los Nodos ya incluidos.
-Colocar todas las aristas conectadas al Nodo inicial en una cola de prioridad(min-heap)
-Este proceso se repite:
    -Extraer la arista de menor peso de la cola.
    -Si conecta un nodo nuevo, agregarla al MST.
    -Insertar en la cola de todas las aristas de ese nodo hacia nodos no visitados.
-Terminar cuando se han añadido todos los vértices.

La utilidad del MST radica en su capacidad para resolver problemas de conectividad de la manera más eficiente y económica posible, lo que lo convierte en una buena herramienta en la optimización de recursos, distancias o costos.

Procesamiento de Audio

El procesamiento de audio es la manipulación de señales sonoras mediante sistemas digitales. Para que una computadora pueda trabajar con sonido, este debe pasar del mundo físico (analógico) al digital.

Tipos de procesamiento:

- Audio “crudo” (Sin compresión): Son archivos que mantienen la información exacta y completa de la grabación original, sin alteraciones. Al no recortar datos, ofrecen la máxima fidelidad posible, pero generan archivos muy pesados que ocupan mucho almacenamiento (WAV, AIFF).
- Compresión Fiel (LossLess): Estos formatos reducen el tamaño del archivo “empaquetando” los datos de forma inteligente (similar a un archivo .zip), pero sin eliminar la información. Al reproducirlos, el sonido se construye idéntico al original, garantizando calidad perfecta con un peso moderado (FLAC, ALAC).
- Compresión Eficiente (Lossy): Son los formatos más ligeros y populares para internet. Funcionan eliminando sonidos que el oído humano difícilmente percibe. Aunque técnicamente se pierde calidad, logran reducir drásticamente el tamaño del archivo, haciéndolos ideales para transmitir música rápidamente(MP3, AAC).

Desarrollo del Proyecto Final

Entonces bajo la premisa de la justificación de proyecto, partimos de que la problemática que encontramos fue la necesidad de comprimir la música sin que se pierda información de la misma, por ende, siga conservando calidad pero sea portable la información, y este problema es típico en muchos formatos en los que se comparte información como texto o imágenes, decidimos abordarlo por el lado del audio (canciones en concreto) porque teníamos curiosidad por la forma en la que se trabaja con audio a la hora de programar en relación al procesamiento del mismo. El proyecto lo dividimos en 3 etapas, la primera etapa consta del código base que utiliza el algoritmo de Huffman, la segunda etapa que agrega el algoritmo de Prim y la última etapa que hace una interfaz gráfica y un reproductor para probar los resultados dentro de la misma GUI.

Versión 1. *Etapa base, implementación del compresor de canciones con Huffman.*

Este bloque importa las librerías necesarias para el programa, por un lado heapq se utiliza para manejar una cola de prioridad basada en un min-heap, algo esencial para construir el árbol de Huffman porque permite obtener de manera eficiente los nodos con menor frecuencia, por otro lado os ayuda a manejar rutas y archivos dentro del sistema operativo, facilitando la localización y apertura del archivo MP3, finalmente, Counter permite contar de manera rápida y sencilla

cuántas veces aparece cada byte dentro del archivo, algo fundamental para asignar frecuencias a los nodos del árbol de Huffman.

```
import heapq
import os
from collections import Counter
```

Esta clase define la estructura básica de cada nodo del árbol de Huffman, cada nodo guarda un byte y la frecuencia con la que aparece en el archivo original, además puede tener dos hijos (izquierdo y derecho) que permiten construir la estructura binaria del árbol. La clase también sobreescribe el método especial __lt__, que sirve para indicar cómo se comparan dos nodos. En este caso, un nodo se considera menor que otro si su frecuencia es más baja, esto permite que la cola de prioridad de heapq pueda ordenar correctamente los nodos según su frecuencia.

```
# Clase nodo
class NodoHuffman:
    def __init__(self, byte, frecuencia):
        self.byte = byte
        self.frecuencia = frecuencia
        self.izquierda = None
        self.derecha = None

    def __lt__(self, otro):
        return self.frecuencia < otro.frecuencia
```

Esta función recibe los datos del archivo y construye con ellos el árbol de Huffman, para hacerlo primero calcula cuántas veces aparece cada byte usando Counter, luego cada byte y su frecuencia se convierten en nodos individuales que se insertan en una cola de prioridad, a partir de ahí, la función extrae constantemente los dos nodos con menor frecuencia, los une en un nodo padre cuya frecuencia es la suma de ambas, y vuelve a insertarlo en la cola.

```
# Funcion para contruir el arbol binario
def construir_arbol_huffman(datos):
    frecuencias = Counter(datos)
    cola = []

    for byte, frecuencia in frecuencias.items():
        heapq.heappush(cola, NodoHuffman(byte, frecuencia))

    while len(cola) > 1:
        izquierda = heapq.heappop(cola)
        derecha = heapq.heappop(cola)

        nodo_combinado = NodoHuffman(None, izquierda.frecuencia + derecha.frecuencia)
        nodo_combinado.izquierda = izquierda
        nodo_combinado.derecha = derecha

        heapq.heappush(cola, nodo_combinado)

    return heapq.heappop(cola)
```

Esta función recorre el árbol de Huffman para asignar un código binario único a cada byte., comienza desde la raíz y construye los códigos agregando un "0" al bajar por la izquierda y un "1" al bajar por la derecha, cuando llega a un nodo hoja, es decir, un nodo que contiene un byte real del archivo, asigna el código que se ha formado durante el recorrido.

```python
# Funcion para generar los codigos
def generar_codigos(nodo, codigo_actual="", codigos={}):
    if nodo is None:
        return

    if nodo.byte is not None:
        codigos[nodo.byte] = codigo_actual
        return

    generar_codigos(nodo.izquierda, codigo_actual + "0", codigos)
    generar_codigos(nodo.derecha, codigo_actual + "1", codigos)

    return codigos
```

Esta función se encarga de la compresión completa del archivo MP3, primero obtiene la ruta absoluta del archivo y lo lee en formato binario, luego construye el árbol de Huffman y genera los códigos para cada byte, a partir de ahí convierte todo el contenido del archivo en una larga cadena de bits sustituyendo cada byte por su código correspondiente.

```python
# Funcion para comprimir la cancion mp3
def comprimir_archivo(nombre_mp3):
    ruta_mp3 = os.path.join(os.getcwd(), nombre_mp3)

    # Leer archivo MP3 como bytes
    with open(ruta_mp3, "rb") as f:
        datos = f.read()

    # Construir Huffman
    arbol = construir_arbol_huffman(datos)
    codigos = generar_codigos(arbol)

    # Convertir a bits
    datos_codificados = "".join(codigos[byte] for byte in datos)

    # Asegurar multiplos de 8 bits
    relleno = 8 - len(datos_codificados) % 8
    datos_codificados += "0" * relleno

    # Convertir a bytes reales
    datos_como_bytes = bytearray()
    for i in range(0, len(datos_codificados), 8):
        byte = datos_codificados[i:i+8]
        datos_como_bytes.append(int(byte, 2))

    # Guardar comprimido
    archivo_bin = "archivo_comprimido.bin"
    with open(archivo_bin, "wb") as f:
        f.write(bytes([relleno]))
        f.write(datos_como_bytes)

    print(f"Archivo comprimido creado: {archivo_bin}")
    return arbol
```

Esta función toma la cadena completa de bits y reconstruye los bytes originales utilizando el árbol de Huffman, comienza en la raíz del árbol y avanza hacia la izquierda o hacia la derecha dependiendo del valor de cada bit, cuando llega a un nodo hoja, reconoce el byte asociado a ese nodo y lo agrega a la salida, después vuelve a la raíz para repetir el proceso con los siguientes bits.

```python
# Funcion para decodificar los bits
def decodificar_bits(bits, arbol):
    resultado = bytearray()
    nodo = arbol

    for bit in bits:
        nodo = nodo.izquierda if bit == "0" else nodo.derecha

        if nodo.byte is not None:
            resultado.append(nodo.byte)
            nodo = arbol

    return resultado
```

Esta función lee el archivo binario comprimido y lo descomprime utilizando el árbol de Huffman generado anteriormente, primero abre el archivo .bin, lee el número de bits de relleno y extrae el resto de los datos comprimidos, luego transforma todos los bytes del archivo en una cadena de bits y elimina los bits de relleno que fueron agregados durante la compresión, con esa secuencia limpia, llama a la función decodificadora para reconstruir los bytes originales, finalmente guarda los datos resultantes en un nuevo archivo llamado archivo_descomprimido.mp3.

```python
# Funcion para descomprimir el bin
def descomprimir_archivo(arbol):
    archivo_bin = "archivo_comprimido.bin"

    with open(archivo_bin, "rb") as f:
        relleno = f.read(1)[0]
        datos = f.read()

    bits = "".join(f"{byte:08b}" for byte in datos)
    bits = bits[:-relleno]

    datos_originales = decodificar_bits(bits, arbol)

    archivo_salida = "archivo_descomprimido.mp3"
    with open(archivo_salida, "wb") as f:
        f.write(datos_originales)

    print(f"Archivo descomprimido creado: {archivo_salida}")
```

La función principal solo toma el archivo mp3 de la canción de prueba y la utiliza llamando a las otras funciones para comprimir y descomprimir la canción.

```python
# Funcion principal
def main():
    archivo = input("Ingresa el nombre del archivo MP3: ")

    print("\n--- COMPRIMIENDO ---")
    arbol = comprimir_archivo(archivo)

    print("\n--- DESCOMPRIMIENDO ---")
    descomprimir_archivo(arbol)

if __name__ == "__main__":
    main()
```

Versión 2. *Segunda etapa, añadir el algoritmo de Prim al algoritmo base.*

Para mejorar la eficiencia de la compresión mediante Huffman, se integró una etapa previa al algoritmo de Huffman. Esta etapa es responsable de reorganizar los símbolos del archivo de entrada de acuerdo con su similitud.

El objetivo de esta función es convertir los Bytes del Mp3 en nodos de un grafo donde los símbolos parecidos tienen conexiones más baratas. Lo que hace es obtener los símbolos únicos del archivo y sobre ellos crea un grafo donde cada símbolo es un nodo conectando cada par ya sea a o b con una arista cuyo peso es: |a-b|. De esta forma el Grafo representa la relación entre cada byte, cuanto más parecido son dos bytes, menor es el costo de conectarlos.

```python
def construir_grafo(datos):
    frecuencias = Counter(datos)
    simbolos = list(frecuencias.keys())
    grafo = {s: [] for s in simbolos}
    for a, b in itertools.combinations(simbolos, 2):
        peso = abs(a - b)
        grafo[a].append((b, peso))
        grafo[b].append((a, peso))
    return grafo, simbolos
```

La función Prim construye un MST a partir del grafo de símbolos. Elige un símbolo inicial y lo marca como visitado, mete al heap todas sus aristas y sacara la arista más barata del heap y si conecta a un nuevo nodo (no visitado) lo agregara al MST insertando todas las nuevas aristas del nuevo nodo y repetirá hasta incluir todos los símbolos.
Utilizar heap es clave en el funcionamiento porque garantiza siempre tomar la arista más barata.

```python
def prim(grafo, simbolos):
    if not simbolos:
        return []
    inicio = simbolos[0]
    visitado = {inicio}
    aristas_mst = []
    heap = []
    for vecino, peso in grafo[inicio]:
        heapq.heappush(heap, (peso, inicio, vecino))
    while heap and len(visitado) < len(simbolos):
        peso, u, v = heapq.heappop(heap)
        if v in visitado:
            continue
        visitado.add(v)
        aristas_mst.append((u, v, peso))
        for vecino, peso2 in grafo[v]:
            if vecino not in visitado:
                heapq.heappush(heap, (peso2, v, vecino))
    return aristas_mst
```

Esta función convierte el árbol en una estructura de adyacencias y recorre el árbol con la función dfs que obtiene un orden lineal de los símbolos esto lo hacemos por que después de construir el árbol MST, se necesitábamos convertirlo en una secuencia lineal y eso se consigue recorriendolo con un dfs que toma el orden natural del árbol.

```python
def construir_orden_mst(mst, simbolos):
    if not simbolos:
        return []
    g = {s: [] for s in simbolos}
    for u, v, _ in mst:
        g[u].append(v)
        g[v].append(u)
    inicio = simbolos[0]
    visitado = set()
    orden = []
    def dfs(n):
        visitado.add(n)
        orden.append(n)
        for vecino in g[n]:
            if vecino not in visitado:
                dfs(vecino)
    dfs(inicio)
    return orden
```

Esta función realiza la transformación más importante del algoritmo de Prim pues cada byte del archivo se sustituye por la posición del símbolo en el orden MST. Básicamente el orden generado por Prim nos ayuda a representar de una nueva forma los bytes del archivo. Esta estructura reduce el desorden del sistema y ayuda a que Huffman pueda comprimir mejor.

```
def aplicar_reasignacion_mst(datos, orden_mst):
    posicion = {simbolo: i for i, simbolo in enumerate(orden_mst)}
    nuevos = bytearray(posicion[b] for b in datos)
    return bytes(nuevos), orden_mst
```

Esta función es la encargada de restaurar los valores originales después del proceso de descompresión. Toma los índices generados por el MST y los convierte de regreso a los valores originales usando orden_mst que guardamos en nuestro archivo comprimido, sin esta función la descompresión no podría reconstruir el archivo original

```
def revertir_reasignacion_mst(datos, orden_mst):
    originales = bytearray(orden_mst[i] for i in datos)
    return bytes(originales)
```

Versión 3. *Etapa final, agregar una interfaz gráfica junto con un reproductor.*

La Arquitectura de la Interfaz (GUI)

La interfaz gráfica se construyó bajo un diseño modular y reactivo utilizando la librería Tkinter. El componente central de la navegación es un contenedor denominado dynamic_frame. En lugar de establecer una disposición estática de controles, se implementó una lógica de "limpieza y reconstrucción": este panel elimina sus widgets internos y genera nuevos controles dinámicamente según la operación requerida (comprimir o reproducir), optimizando así el espacio visual y la gestión de estados.

Implementación de la limpieza dinámica: Para lograr este comportamiento, se diseñó la función _limpiar_dynamic_frame, la cual itera sobre los elementos hijos del contenedor y los destruye antes de renderizar la nueva vista, asegurando que no existan superposiciones de botones.

```
def _limpiar_dynamic_frame(self):
    for w in self.dynamic_frame.winfo_children():
        w.destroy()
    self.compressed_button = None
    self.decompressed_button = None
```

Asimismo, se integró un sistema de concurrencia para evitar el congelamiento de la ventana principal durante las tareas intensivas de compresión. Dado que Tkinter no es thread-safe (seguro para hilos), se implementó una arquitectura de comunicación asíncrona mediante una cola (queue). Los hilos de trabajo depositan los mensajes en esta cola, y la interfaz gráfica revisa periódicamente el buzón para actualizar la bitácora de manera segura.

```
def _poll_log_queue(self):
    try:
        while True:
            msg = self._log_queue.get_nowait()
            self.log.insert(END, msg + "\n")
            self.log.see(END)
    except queue.Empty:
        pass
    self.root.after(100, self._poll_log_queue)
```

Implementación del Reproductor de Audio

El reproductor de canciones se desarrolló integrando la librería pygame.mixer. Dado que el formato de archivo comprimido (.bin) utiliza un algoritmo personalizado (Huffman + Prim) no legible nativamente por los reproductores estándar, se diseñó un flujo de descompresión en tiempo real con almacenamiento temporal (caching).

Al solicitar la reproducción, el sistema no decodifica todo en memoria RAM para reproducirlo directamente como flujo de bytes (stream), sino que ejecuta la descompresión inversa y escribe el resultado en un archivo físico temporal (_temp_playback.mp3) en el disco. Esto permite que el motor de audio cargue un archivo estándar válido de manera transparente para el usuario.

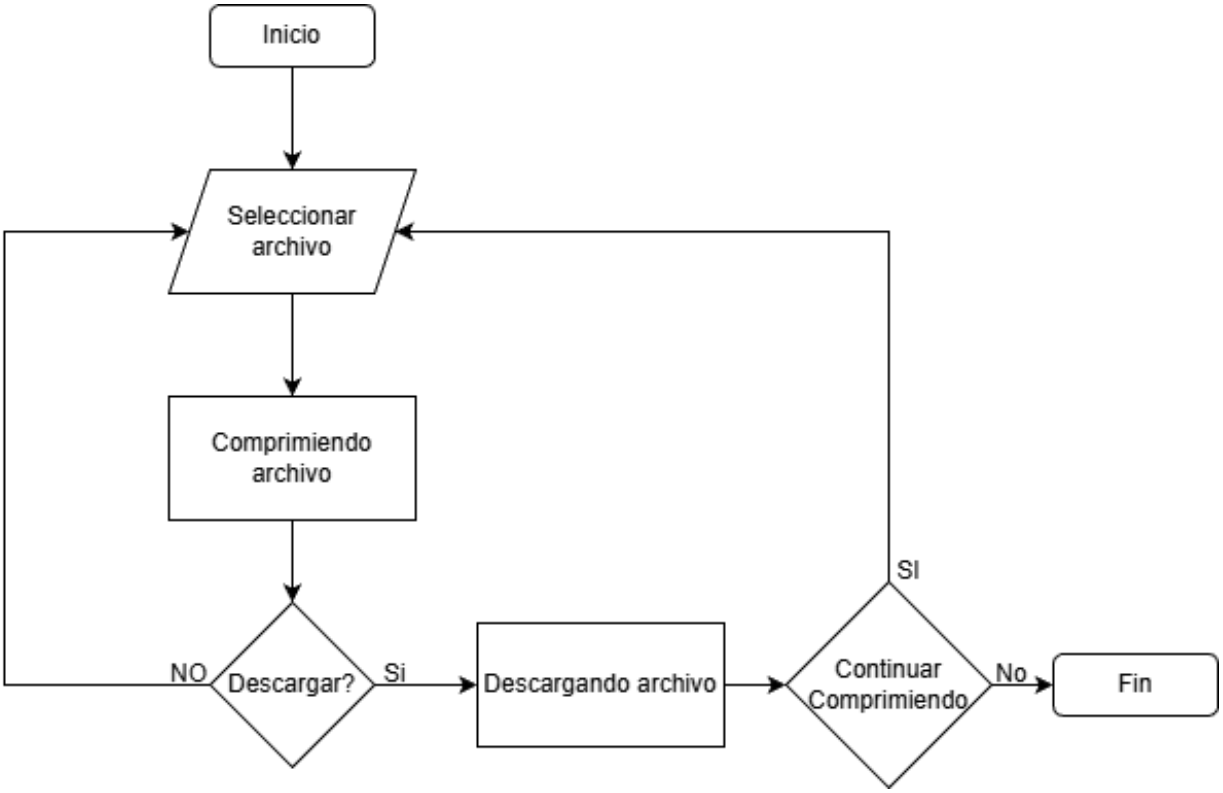
```
temp_filename = "_temp_playback.mp3"
with open(temp_filename, "wb") as f:
    f.write(datos_originales)

# Actualizar estado
self.temp_audio_path = temp_filename
```

Un aspecto crítico del desarrollo fue la gestión del ciclo de vida del archivo temporal. La función de detención (stop_music) fue programada no sólo para detener la reproducción, sino para liberar explícitamente el recurso de audio (unload) y eliminar el archivo temporal del sistema de archivos. Esto evita conflictos de "archivo en uso" y previene la acumulación de datos residuales en el disco duro.

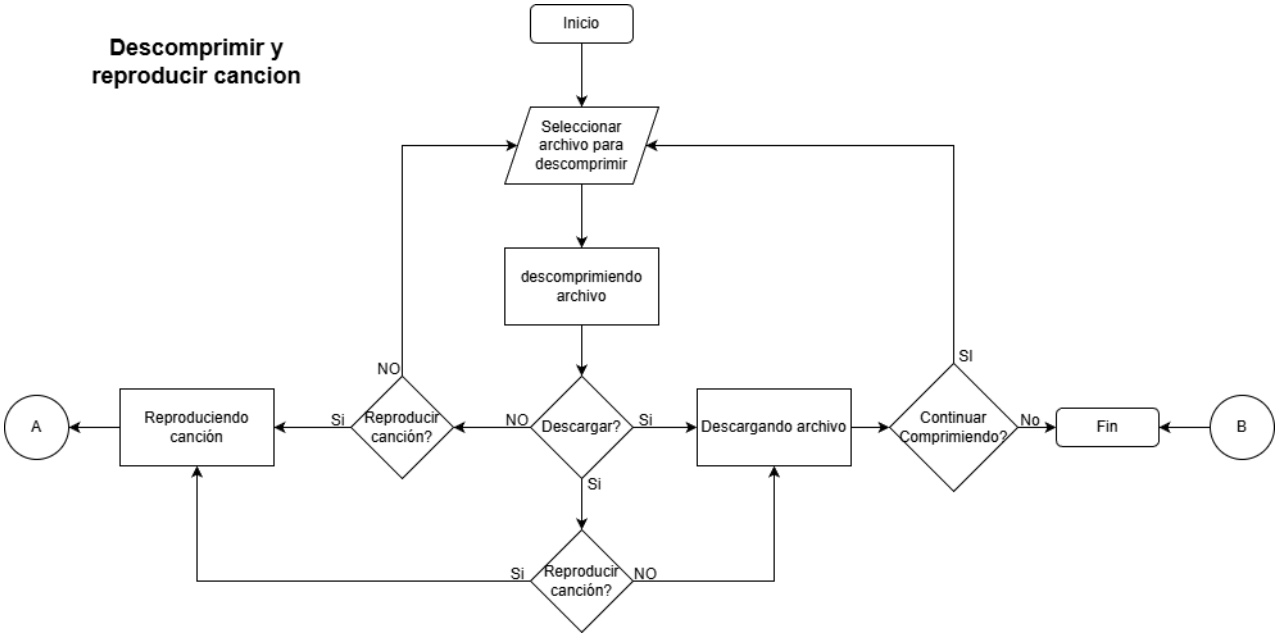
```
def stop_music(self):
    pygame.mixer.music.stop()
    try:
        pygame.mixer.music.unload()
    except AttributeError:
        pass

    # Borrar el archivo temporal inmediatamente
    if self.temp_audio_path and os.path.exists(self.temp_audio_path):
        try:
            os.remove(self.temp_audio_path)
```

Diagramas de flujo

El siguiente diagrama explica el funcionamiento al seleccionar un archivo y comprimirlo.



El diagrama a continuación muestra el proceso de seleccionar un archivo para descomprimir, para posteriormente reproducir la canción o descargar el archivo descomprimido.



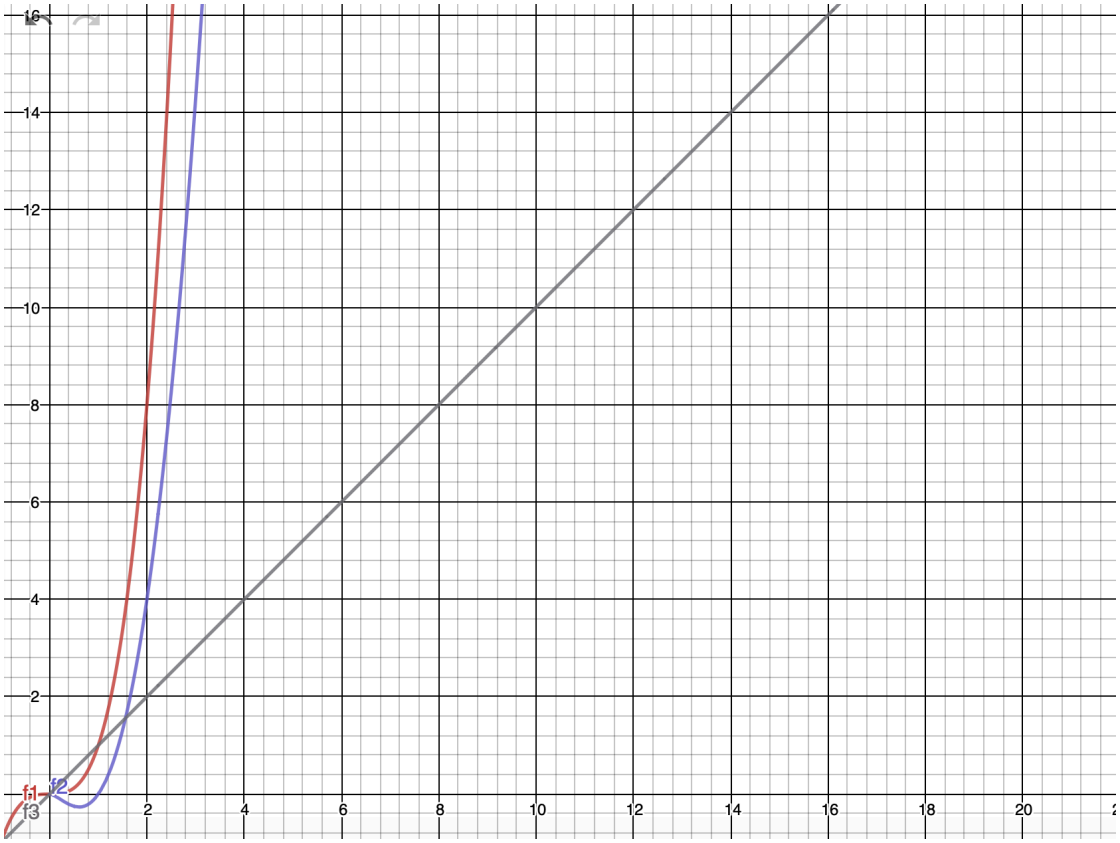
Comparativa con otras técnicas

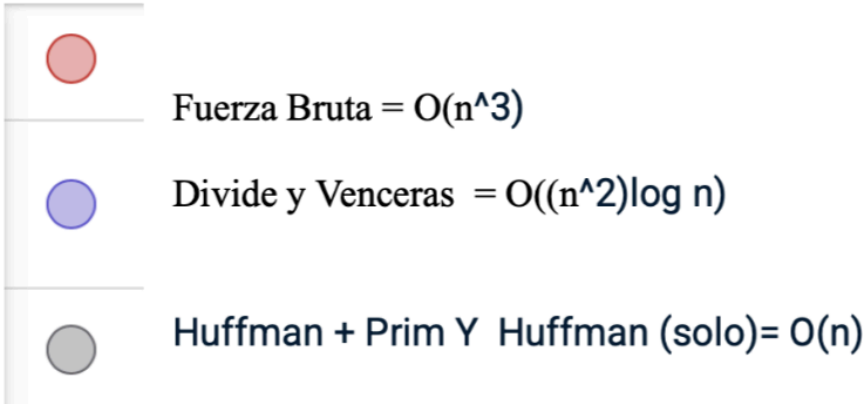
A continuación explicamos las diferencias que existen entre los dos algoritmos que implementamos a lo largo del semestre, es decir, el algoritmo para seleccionar la mejor opción de hardware y el compresor de canciones. Entonces compararemos las técnicas, fuerza bruta, divide y vencerás, algoritmo de Huffman y algoritmo de Prim.

	Voraz de Huffman (Aplicado en el compresor de canciones)	Divide y Venceras (Aplicado en el compresor de canciones)	Fuerza Bruta (Aplicado en el algoritmo de videojuegos)	P. Dinamica (Aplicado en el algoritmo de videojuegos)
Funcionamiento	Selecciona la mejor opción local en cada paso inmediato, sin reconsiderar decisiones pasadas o pensar en	Descompone el problema en subproblemas independientes, los resuelve y combina los resultados.	Examina exhaustivamente todas las combinaciones posibles del espacio de	Resuelve subproblemas superpuestos una sola vez y almacena los resultados para su reutilización.

	consecuencias futuras.		búsqueda.	
Eficiencia	Generalmente rápida y con bajo consumo de memoria.	El tiempo de ejecución crece exponencialmente o factorialmente con el tamaño de la entrada.	Suele reducir la complejidad a la logarítmica, Aunque utiliza recursión.	Mas rápida que la fuerza bruta, pero requiere mucha memoria para almacenar tablas de datos.
Optimizmo	No garantiza optimismo, aunque en algoritmos específicos como Huffman si puede resultar óptimo.	Si es óptimo, al revisar todas las opciones, asegura encontrar la mejor solución absoluta.	Su uso es muy óptimo, pues garantiza encontrar la solución correcta si el algoritmo de combinación está bien diseñado.	Asegura encontrar la solución global al explorar todas las sub-soluciones necesarias de manera óptima.

Gráfica comparativa de las técnicas





Al comparar nuestros distintos algoritmos se puede observar que Fuerza bruta tiene la mayor complejidad computacional lo que implica que su tiempo de ejecución crece rápidamente al aumentar el tamaño de datos siendo poco eficiente para el manejo de datos grandes.
La técnica Divide y vencerás ofrece una mejora significativa sobre fuerza bruta aunque todavía puede volverse costosa para conjuntos de datos grandes.

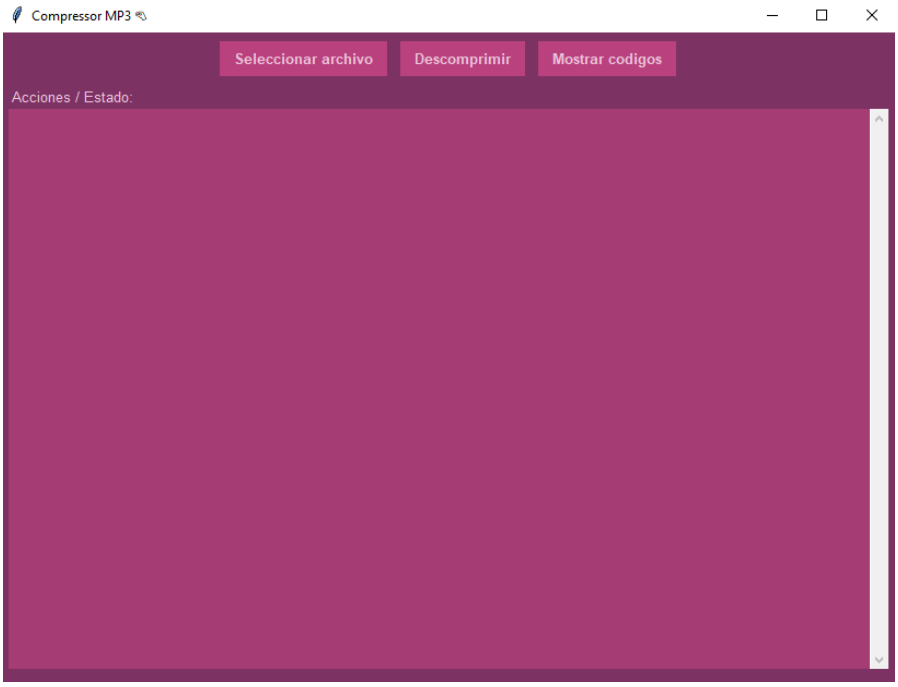
El combinar las técnicas de Huffman y Prim no mejora en su complejidad computacional directamente por que se mantiene, sin embargo mejora otros aspectos como la distribución de los bytes, reducción efectiva y la optimización de la práctica de la codificación, Es decir mejora la compresión efectiva y la calidad de la codificación, logrando archivos más compactos sin pérdida de información.

Finalmente los algoritmos voraces o lineales resultaron mucho más eficientes y escalables, permitiendo procesar grandes volúmenes de datos en tiempos razonables.

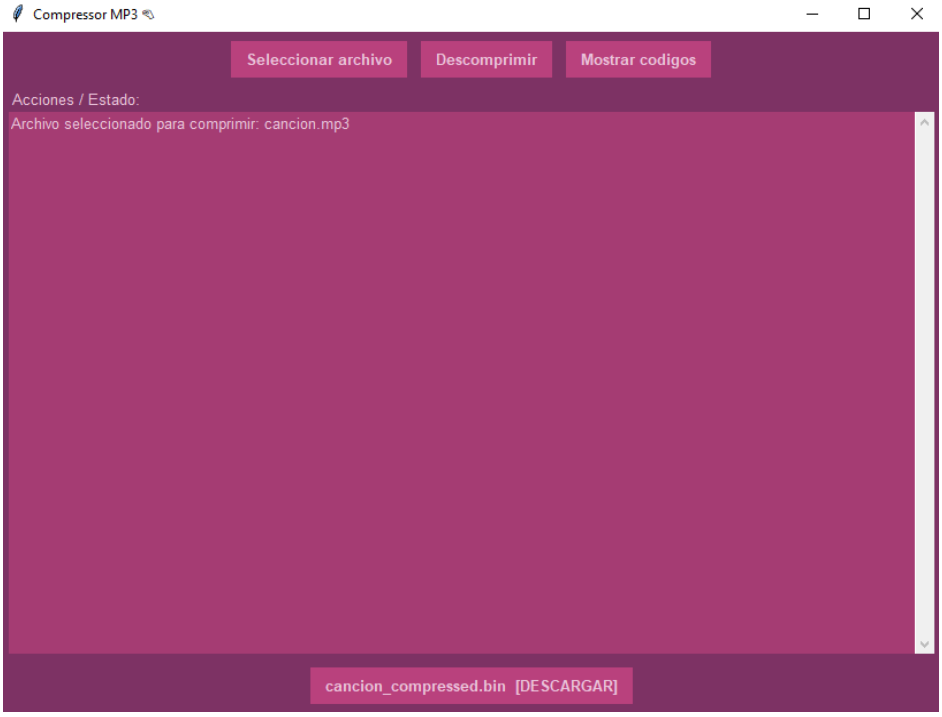
En términos prácticos, esto justifica el uso de algoritmos voraces como Huffman y Prim en sistemas de compresión, ya que se combinan eficiencia con resultados óptimos, mientras que fuerza bruta se limita a casos de validación o análisis pequeños de conjunto de datos.
Nuestro análisis comparativo evidencia la importancia de seleccionar la técnica adecuada según el tamaño y su naturaleza de procesar información.

Pruebas y Resultados del Compresor de Canciones

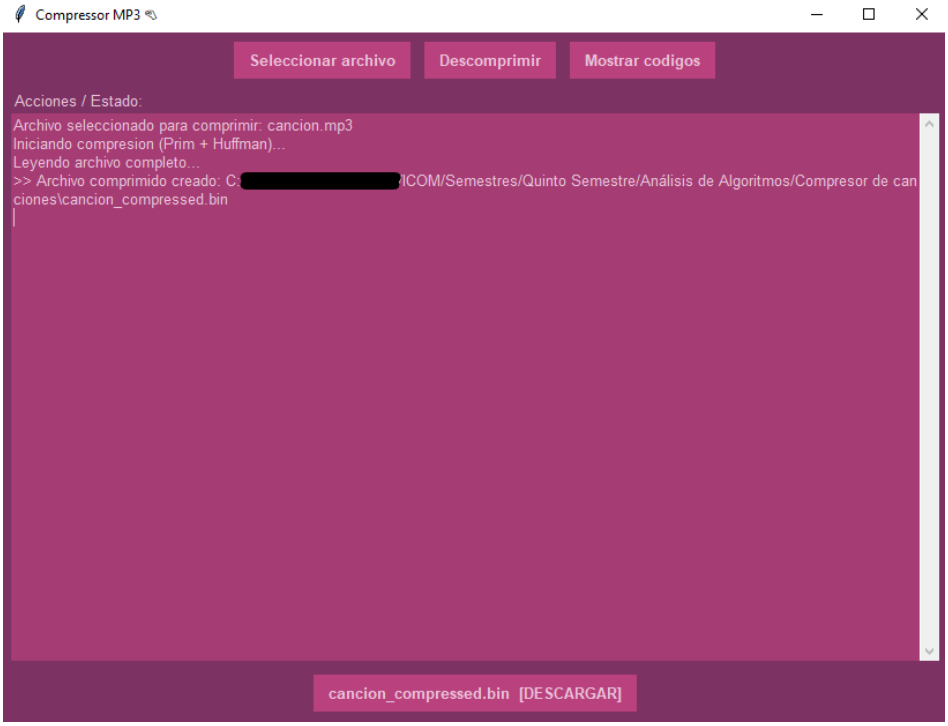
Una vez que elaboramos el proyecto, hicimos pruebas para verificar que todo funcionara de forma correcta, a continuación mostramos los resultados obtenidos del proyecto.

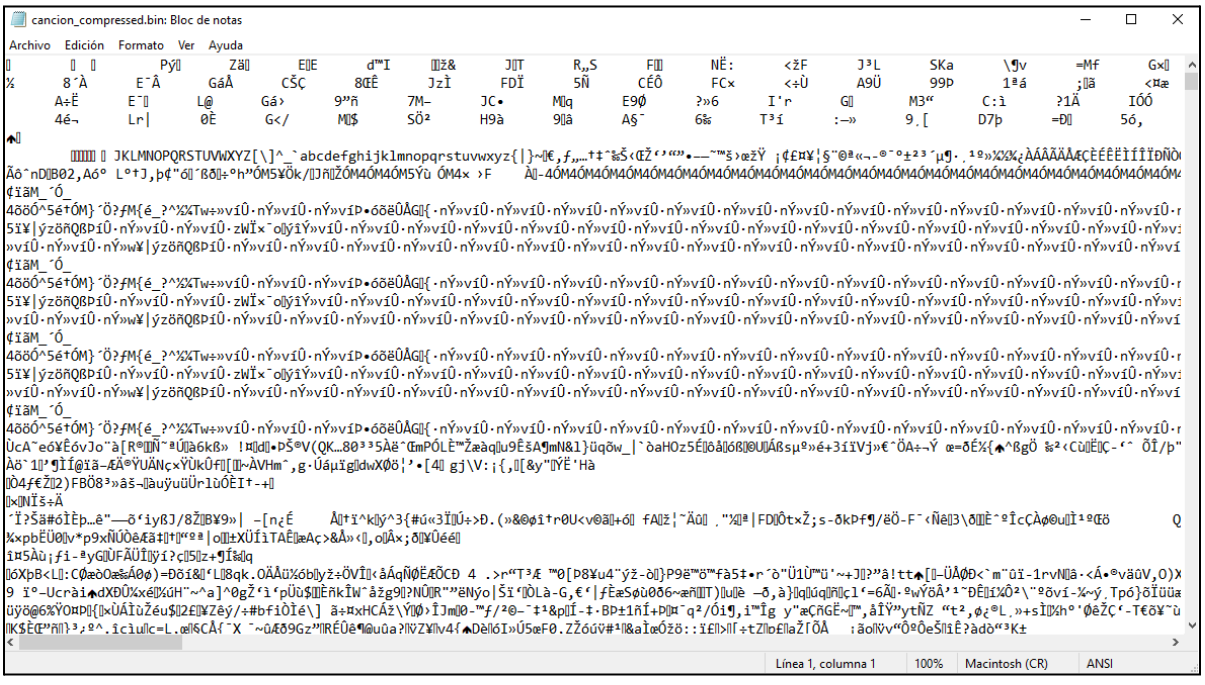


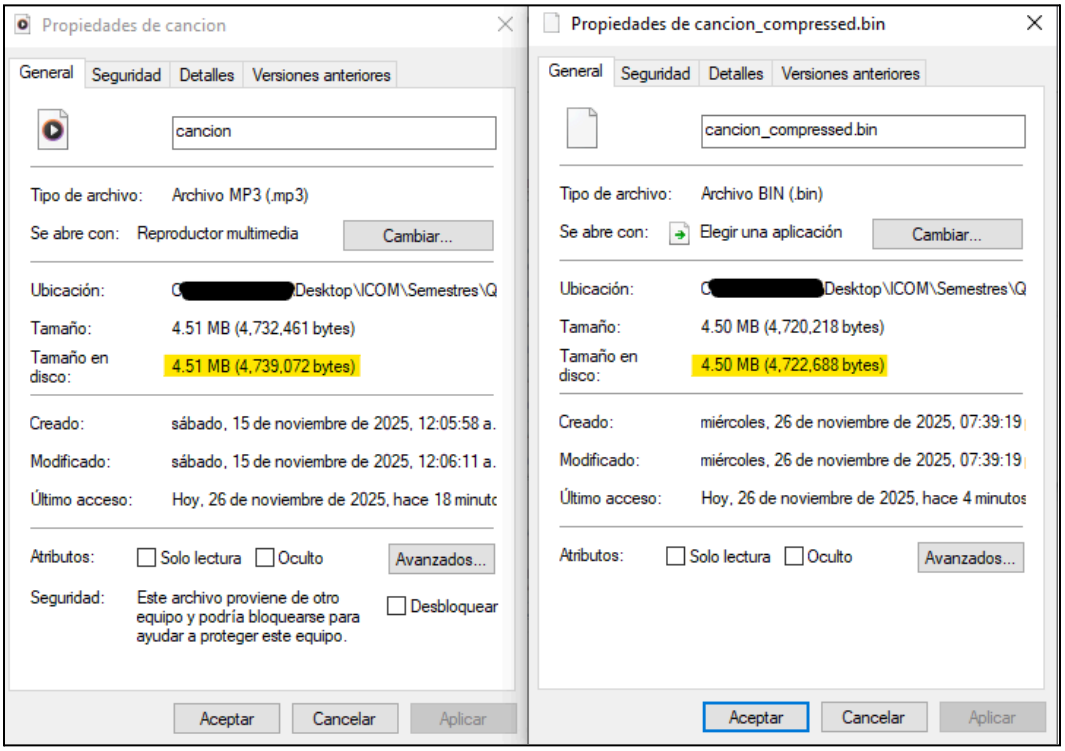
Al darle click a seleccionar archivo no permitirá seleccionar una canción que esté dentro de nuestro explorador de archivos (siempre y cuando sea mp3).



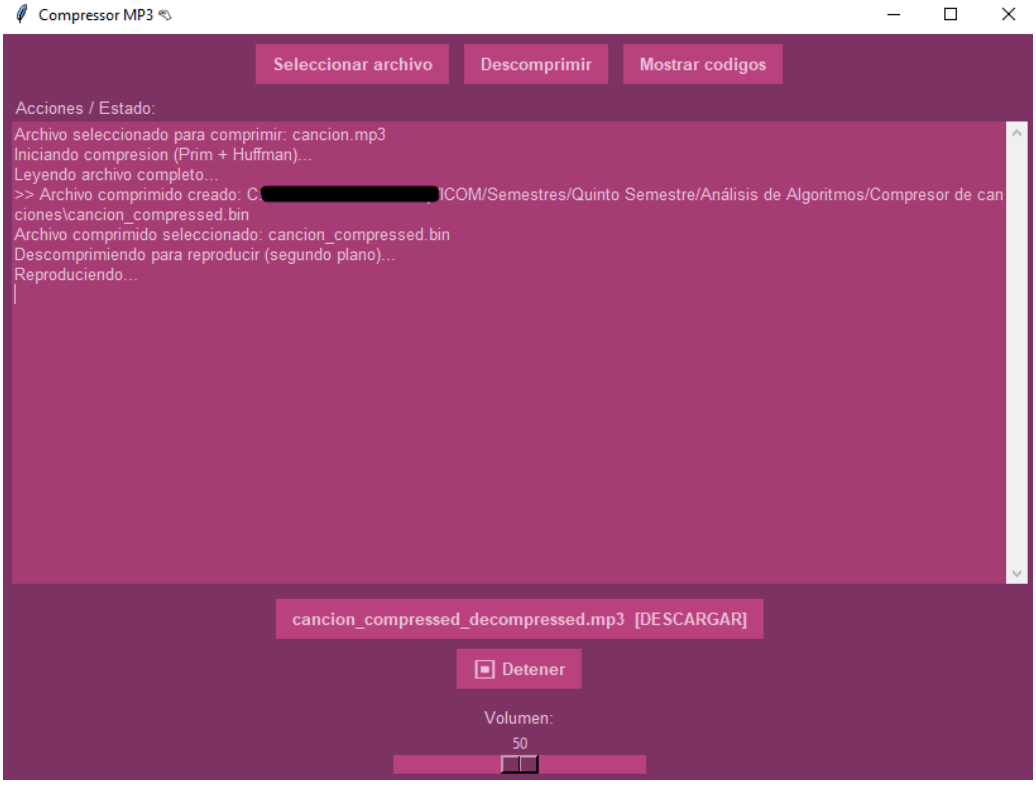
Si damos click en descargar podremos ver el contenido del bin.

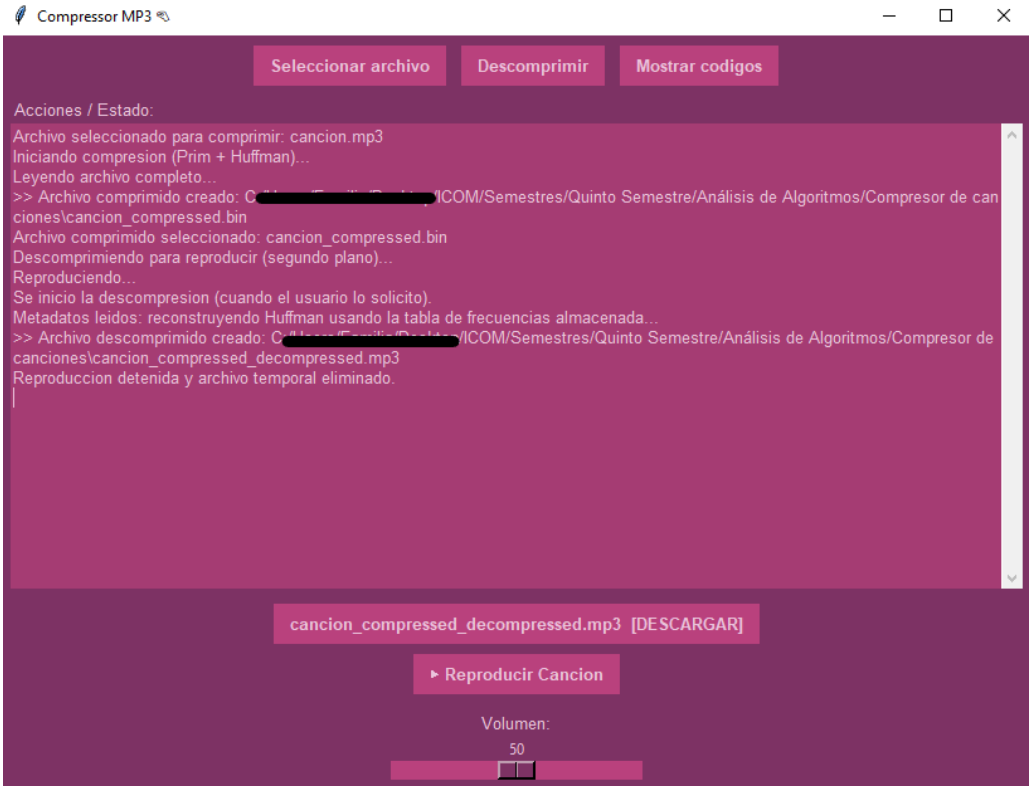

Compressor MP3
Seleccionar archivo
Descomprimir
Mostrar codigos
Acciones / Estado:
Archivo seleccionado para comprimir: cancion.mp3
Iniciando compresion (Prim + Huffman)...
Leyendo archivo completo...
>> Archivo comprimido creado: C:[redacted]ICOM/Semestres/Quinto Semestre/Análisis de Algoritmos/Compresor de canciones\cancion_compressed.bin
cancion_compressed.bin [DESCARGAR]

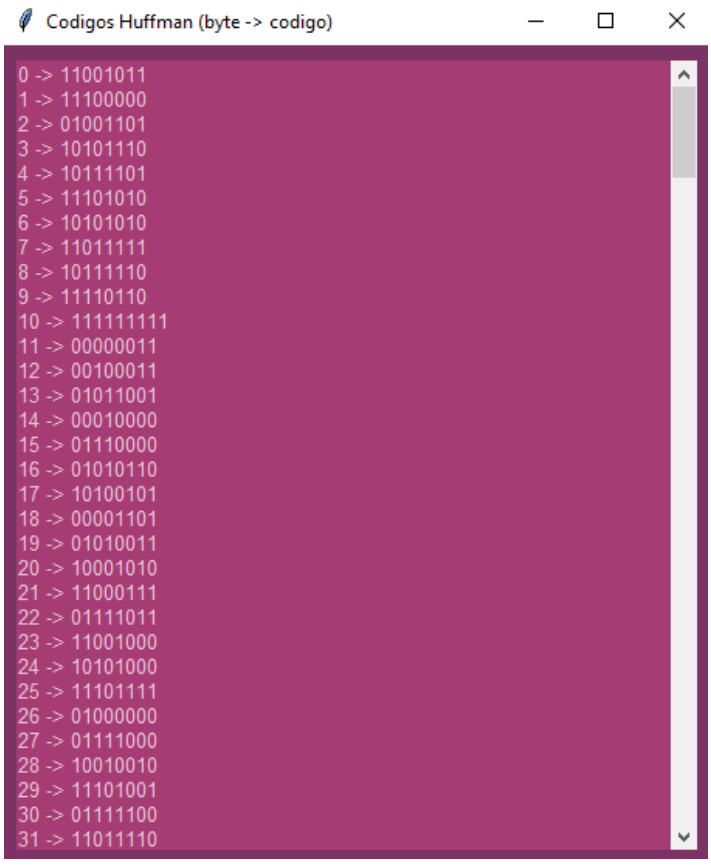

cancion_compressed.bin: Bloc de notas
Archivo Edición Formato Ver Ayuda
Línea 1, columna 1
100%
Macintosh (CR)
ANSI



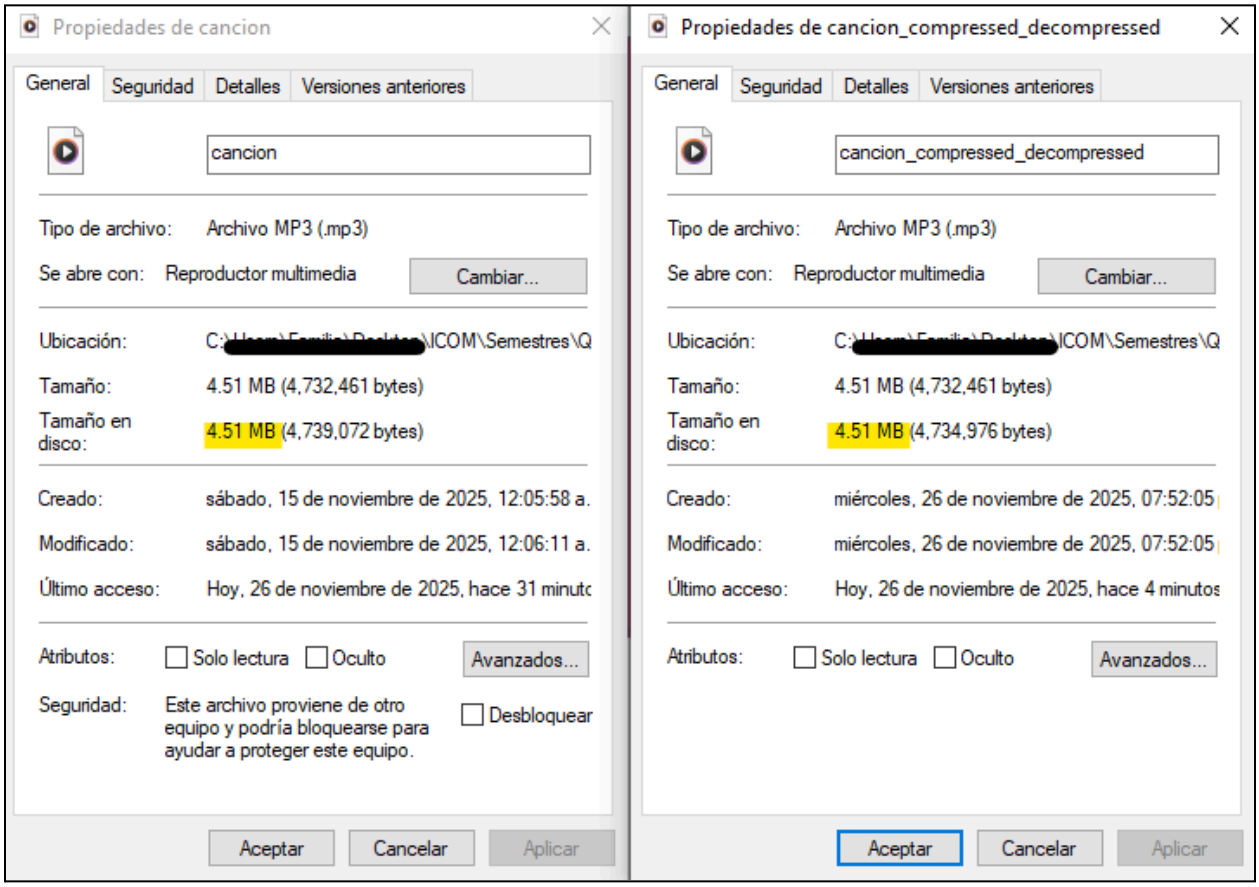
Al dar click en descomprimir de igual forma tendremos que seleccionar el archivo que nos dio previamente y así mismo no pondrá un reproductor para escuchar la canción descomprimida.




Compressor MP3
Seleccionar archivo
Descomprimir
Mostrar codigos
Acciones / Estado:
Archivo seleccionado para comprimir: cancion.mp3
Iniciando compresion (Prim + Huffman)...
Leyendo archivo completo...
>> Archivo comprimido creado: C[redacted]/ICOM/Semestres/Quinto Semestre/Análisis de Algoritmos/Compresor de canciones\cancion_compressed.bin
Archivo comprimido seleccionado: cancion_compressed.bin
Descomprimiendo para reproducir (segundo plano)...
Reproduciendo...
Se inicio la descompresion (cuando el usuario lo solicito).
Metadatos leidos: reconstruyendo Huffman usando la tabla de frecuencias almacenada...
>> Archivo descomprimido creado: C[redacted]/ICOM/Semestres/Quinto Semestre/Análisis de Algoritmos/Compresor de canciones\cancion_compressed_decompressed.mp3
Reproduccion detenida y archivo temporal eliminado.
cancion_compressed_decompressed.mp3 [DESCARGAR]
▸ Reproducir Cancion
Volumen:
50


Codigos Huffman (byte -> codigo)
0 -> 11001011
1 -> 11100000
2 -> 01001101
3 -> 10101110
4 -> 10111101
5 -> 11101010
6 -> 10101010
7 -> 11011111
8 -> 10111110
9 -> 11110110
10 -> 111111111
11 -> 00000011
12 -> 00100011
13 -> 01011001
14 -> 00010000
15 -> 01110000
16 -> 01010110
17 -> 10100101
18 -> 00001101
19 -> 01010011
20 -> 10001010
21 -> 11000111
22 -> 01111011
23 -> 11001000
24 -> 10101000
25 -> 11101111
26 -> 01000000
27 -> 01111000
28 -> 10010010
29 -> 11101001
30 -> 01111100
31 -> 11011110

Para esta canción se construyeron 256 códigos.

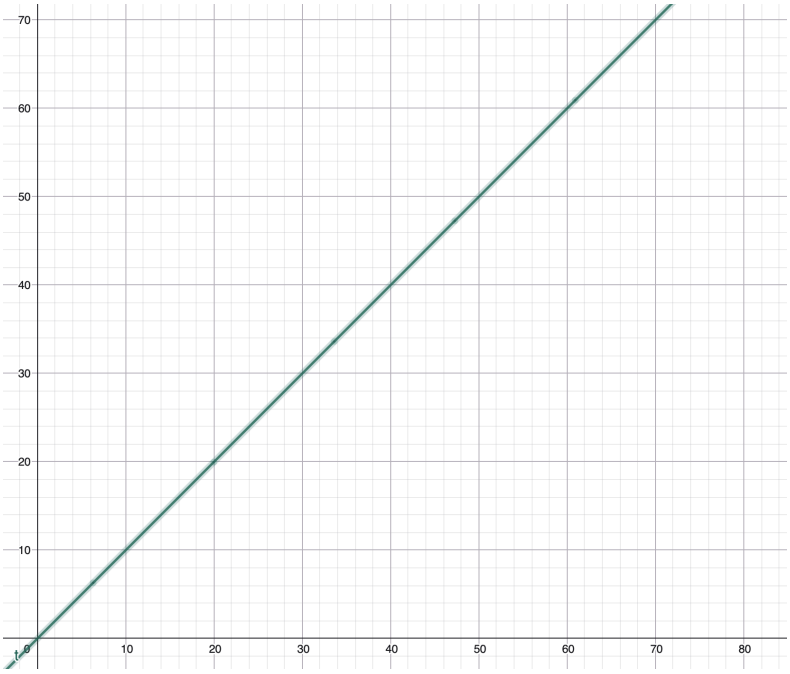


Análisis de Complejidad Computacional

Para analizar la complejidad computacional de nuestro algoritmo utilizaremos la notación asintótica Big O.
Nuestro algoritmo combina dos distintas etapas principales. Nuestro preposicionamiento en un MST y la compresión y descompresión mediante Huffman.
Analizando eso, tenemos Operaciones dependientes del número de símbolos (s), este conjunto de símbolos corresponde a los valores posibles de un Byte(0-255), es decir s<=256.
Es decir que operaciones como: Construcción del grafo de símbolos (O(s^2)), Ejecución del algoritmo de Prim (O(s^2 log s)) y Construcción del árbol de Huffman (O(s log s)) tendrán todas un coste máximo fijo, ya que s está acotado por una constante, en pocas palabras estas operaciones son consideradas O(1).

Por otra parte tenemos operaciones que son totalmente dependientes del tamaño del archivo(n).
Por ejemplo durante la compresión y descompresión es necesario procesar cada byte del archivo.

Estas Operaciones (conteo de frecuencias, Reasignación según el orden obtenido del MST, codificación de Huffman byte a byte, conversión de bits a formato lineal y codificación durante la descompresión) tengan un crecimiento proporcional al tamaño del archivo es decir: O(n).



En donde el eje de las X representa el Tamaño del archivo(n) y el eje de las Y su tiempo de ejecución

---

Conclusiones Personales

Gutierrez Vazquez Axel

La realización de este proyecto demostró la eficacia del algoritmo de Huffman para la compresión de audio sin pérdida, garantizando la integridad de los datos mediante la eliminación de redundancia estadística. La implementación de una arquitectura concurrente, junto con una interfaz gráfica funcional, permite gestionar operaciones complejas de manera fluida y transparente. En definitiva, el sistema valida la integración práctica entre la optimización algorítmica y el desarrollo de software para el manejo y almacenamiento eficiente de información multimedia.

Hernandez Macias Axxel Gael

Durante el desarrollo de este proyecto pude comprender de manera adecuada cómo se combinan algoritmos clásicos para resolver problemas reales. Trabajar con Prim y Huffman me permitió

visualizar directamente como implementar la teoría con la práctica en este algoritmo exactamente por qué este Algoritmo contiene la lógica real de la compresión de archivos. Este proyecto me ayudó a fortalecer mis conocimientos y mi capacidad de integrar distintos enfoques algoritmos para llegar a una solución óptima, organizada y eficiente.

Quintero Arreola Laura Vanessa
La verdad me gusto cambiar de algoritmo para la entrega final, porque como se mencionó en la justificación el código ya no pudimos evolucionarlo por falta de recursos para su mejoramiento. De cualquier forma aprendí mucho sobre los diferentes algoritmos y sobre cómo se deben usar y que realmente todos son muy útiles dependiendo de para que los uses y el contexto, porque si bien unos algoritmos te pueden ayudar a que un programa sea más eficiente también pueden causar lo contrario, y en cuanto al proyecto final me gusto desarrollar el compresor de canciones que en realidad no le veo una aplicación más allá de lo didáctico que puede ser o quizá pueda abrir pie a desarrollar formas de comprimir más una canción.

Bibliografía

- Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to algorithms (3.ª ed.). MIT Press.
- Huber, D. M., & Runstein, R. E. (2018). Modern recording techniques (9.ª ed.). Routledge.
- KeepCoding. (12 de Agosto de 2024). Algoritmos voraces: ¿Qué son y cómo funcionan? KeepCoding Bootcamps. https://keepcoding.io/blog/que-son-los-algoritmos-voraces/
- Martínez Sandoval, L. I. (22 de agosto de 2022). Árboles de peso mínimo: Algoritmos de Prim y Kruskal. MADI NekoMath. https://madi.nekomath.com/P5/ArbolPesoMin.html