UNIVERSIDAD DE GUADALAJARA
CENTRO UNIVERSITARIO DE CIENCIAS EXACTAS E INGENIERÍAS

Análisis de algoritmos

Mtro. Jorge Ernesto Lopez Arce Delgado
Act. 5: Técnica Voraz Huffman

Integrantes:
Braulio Hurtado Escoto 220426225
Jorge Daniel Hernández Reyes 220027797
Juan Pablo Solis Regin 220468416

Explicación del código:

El código que desarrollamos para esta actividad se basa en la técnica voraz de Huffman, el cual se basa en la selección de la opción más eficiente (la menor frecuencia) para construir un árbol de prefijos óptimos, lo que nos da como resultado una codificación concisa donde los caracteres más comunes tienen los códigos binarios más cortos y los menos comunes los más largos.
Para la comprensión de nuestro programa decidimos hacer una guía para que cada usuario que desee probar el código, lo pueda hacer sin problemas.

Nuestro código utiliza la librería de custom tkinter para la interfaz gráfica, por lo que debes asegurarte de tenerla instalada, además de la librería estándar `tkinter` (que viene incluida con Python). Tambien se necesitan estos archivos los cuales serán necesarios para usar el programa:

1. Gui_huffman.py y Algoritmo_huffman.py, esto es para poder arrancar el programa, donde en la GUI se encuentra el apartado grafico y en el archivo del algoritmo, el programa base.
2. El archivo txt para probar el programa, en este caso decidimos utilizar “The Great Gatsby.txt”

Uso de la interfaz gráfica:
La aplicación tiene una interfaz muy sencilla con tres botones principales:

Botón	**Función**
Cargar archivo	Selecciona un archivo de texto (.txt) del sistema. Carga su contenido en memoria para la compresión.
Codificar y guardar .nogpu	Ejecuta el algoritmo Huffman sobre el texto cargado, genera el código binario comprimido y guarda el resultado en un archivo binario con extensión **.nogpu** (junto con los códigos Huffman para la decodificación).
Decodificar desde .nogpu	Lee el último archivo **.nogpu** generado, reconstruye el árbol de códigos y decodifica el texto binario, guardando el resultado en un nuevo archivo de texto (_decodificado.txt).

Pasos para comprimir y descomprimir

1. **Cargar el Archivo:** Haz clic en el botón **"Cargar archivo"** y selecciona tu archivo de texto (.txt).
- Verás el nombre del archivo, su contenido y el tamaño original en caracteres en el área de texto y en la etiqueta de información.
2. **Codificar (Comprimir):** Haz clic en el botón **"Codificar y guardar .nogpu"**.

- La aplicación calculará las frecuencias, construirá el Árbol de Huffman, generará los códigos binarios y comprimirá el texto.
- El archivo comprimido se guardará en la misma ubicación que el original, con el nombre [nombre_archivo]_comprimido.nogpu.
- El área de texto mostrará los **Códigos Huffman** generados para cada carácter.
- La etiqueta de información mostrará el **tamaño original en bytes**, el **tamaño comprimido en bytes** y la **Eficiencia teórica**.

3. **Decodificar (Descomprimir):** Una vez que has codificado, haz clic en el botón **"Decodificar desde .nogpu"**.

- La aplicación leerá el archivo .nogpu, reconstruirá la estructura de códigos y revertirá el proceso de compresión.
- El texto decodificado se guardará en un nuevo archivo llamado [nombre_archivo]_decodificado.txt.
- El área de texto mostrará el **Texto Decodificado**.

El código principal (Gui_huffman.py) gestiona la interfaz gráfica y coordina las llamadas a las funciones del algoritmo Huffman (que se encuentran en el archivo Algoritmo_huffman).

Clase HuffmanApp

Es la clase principal de la aplicación, que hereda de ctk.CTk y configura la ventana y sus elementos.

Variable de Clase	**Descripción**
self.texto_original	Almacena el contenido del archivo .txt cargado.
self.codificado	Almacena el texto codificado como una cadena de bits (ej: '1011001...').
self.raiz	La raíz del Árbol de Huffman construido.
self.codigos	Diccionario que mapea cada carácter a su código binario (ej: {'a': '10', 'b': '0', ...}).
self.ruta_archivo	Ruta completa del archivo .txt original.
self.archivo_comprimido	Ruta completa del archivo binario comprimido (.nogpu).

Los métodos principales que implementamos en este programa son los siguientes:

1. cargar_archivo()

- Utiliza filedialog.askopenfilename para permitir al usuario seleccionar un archivo de texto (*.txt).
- Lee el contenido y lo almacena en self.texto_original.
- Actualiza el **cuadro de texto** (self.resultado) para mostrar el contenido cargado y la **etiqueta de información** (self.etiqueta_info) con el tamaño en caracteres.

2. codificar()

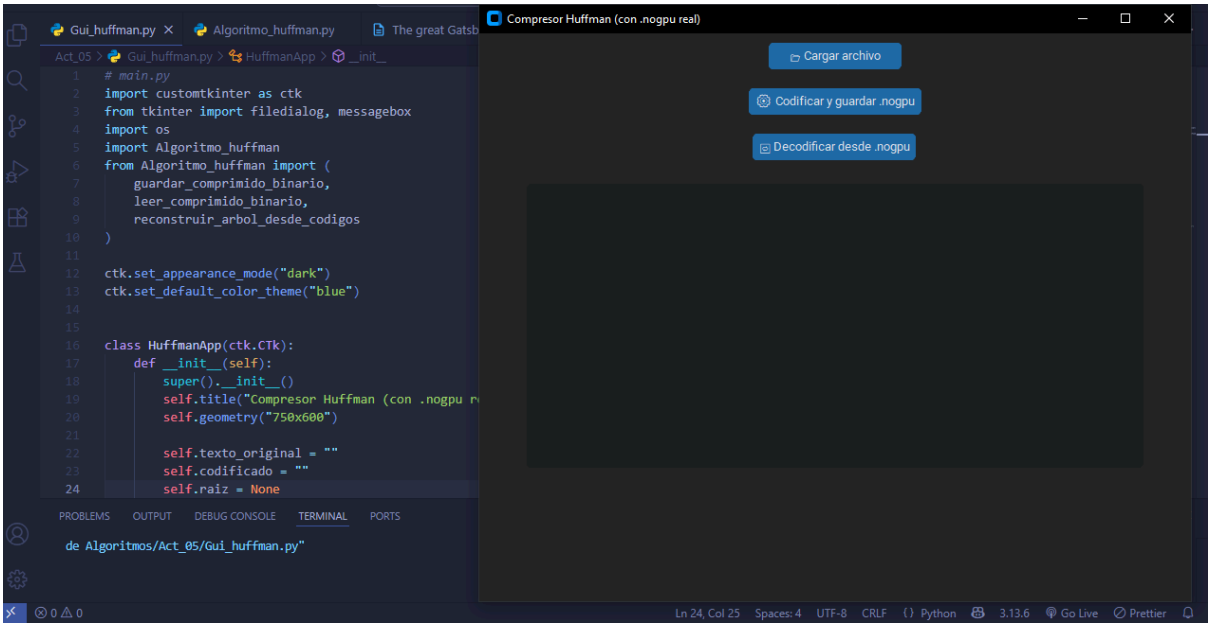
- **Valida** que se haya cargado un archivo.
- Llama a las funciones de Algoritmo_huffman para realizar la compresión:
    - calcular_frecuencias(): Obtiene la frecuencia de cada carácter.
    - construir_arbol(): Crea el árbol binario de Huffman.
    - generar_codigos(): Recorre el árbol para obtener los códigos binarios.
    - codificar_texto(): Aplica los códigos al texto original.
- Llama a **guardar_comprimido_binario(nombre_salida, self.codificado, self.codigos)**: Esta es la parte crucial que guarda el texto codificado (bits) y los códigos Huffman (necesarios para la decodificación) en un archivo binario .nogpu.
- Calcula y muestra la **eficiencia teórica** y la **comparación de tamaños en bytes** en la interfaz.

3. decodificar()

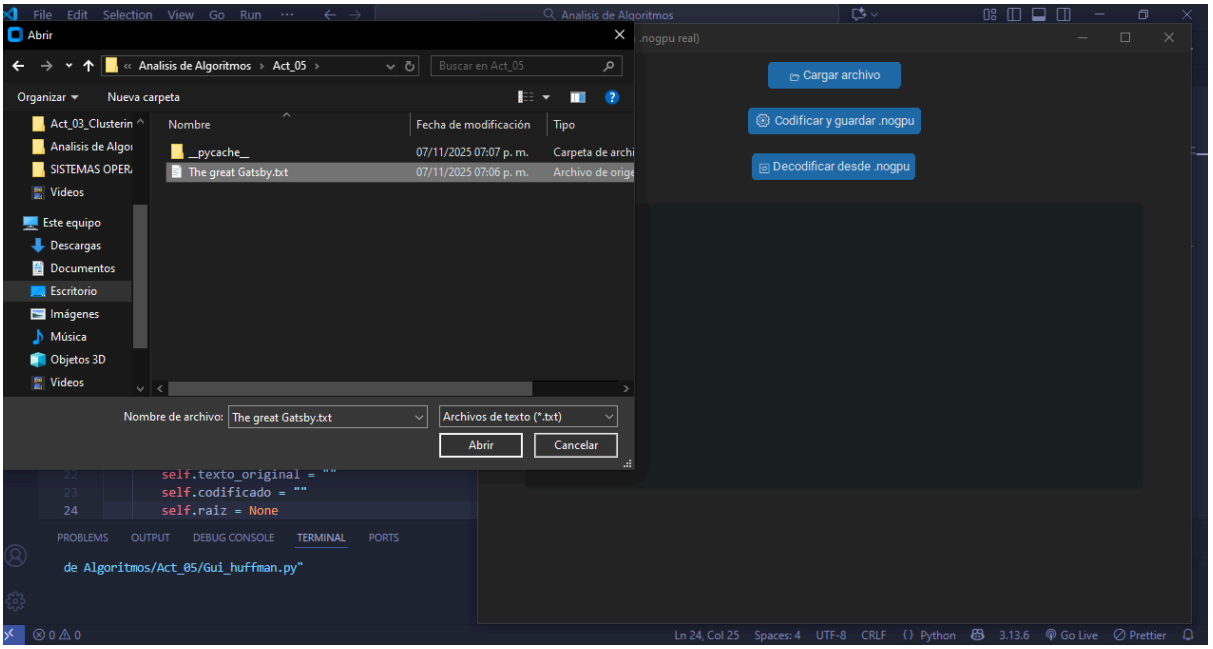
- **Valida** que exista un archivo .nogpu generado previamente.
- Llama a **leer_comprimido_binario(self.archivo_comprimido)**: Lee el archivo .nogpu para obtener la cadena de bits comprimida y los códigos Huffman.
- Llama a **reconstruir_arbol_desde_codigos(cods)**: Usa los códigos guardados para recrear el Árbol de Huffman.
- Llama a decodificar_texto(bits, raiz) para obtener el texto original a partir de la cadena de bits y el árbol.
- Guarda el resultado en un nuevo archivo de texto (_decodificado.txt) y lo muestra en la interfaz.

Capturas de pantalla del código:

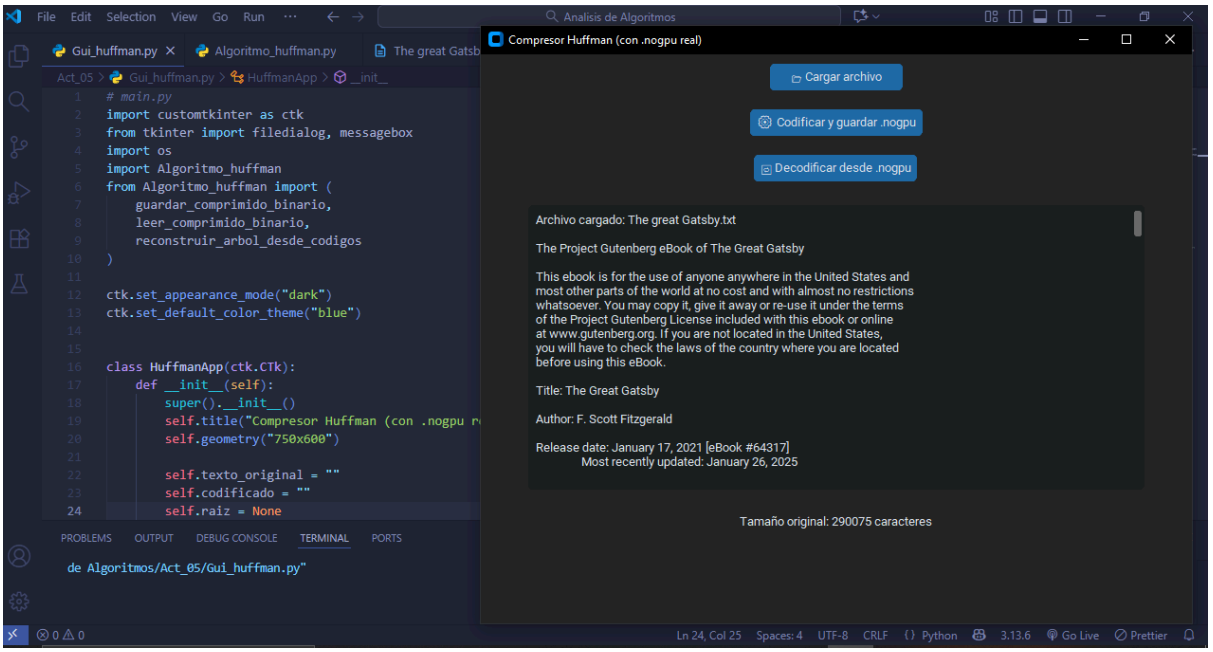
Código junto con la GUI

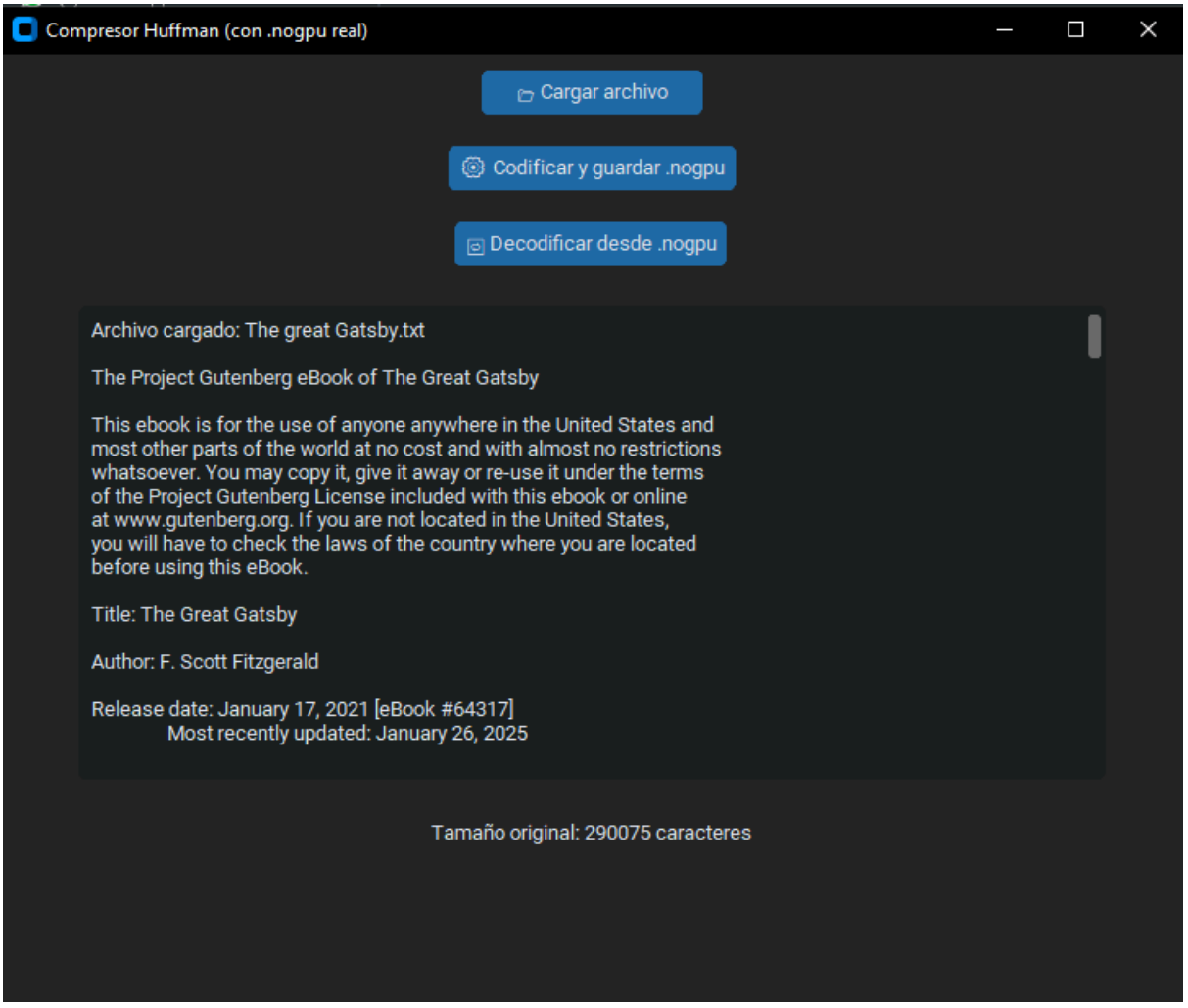


Se carga el archivo de texto que se quiere comprimir (en este caso fue el de “The great Gatsby.txt”)

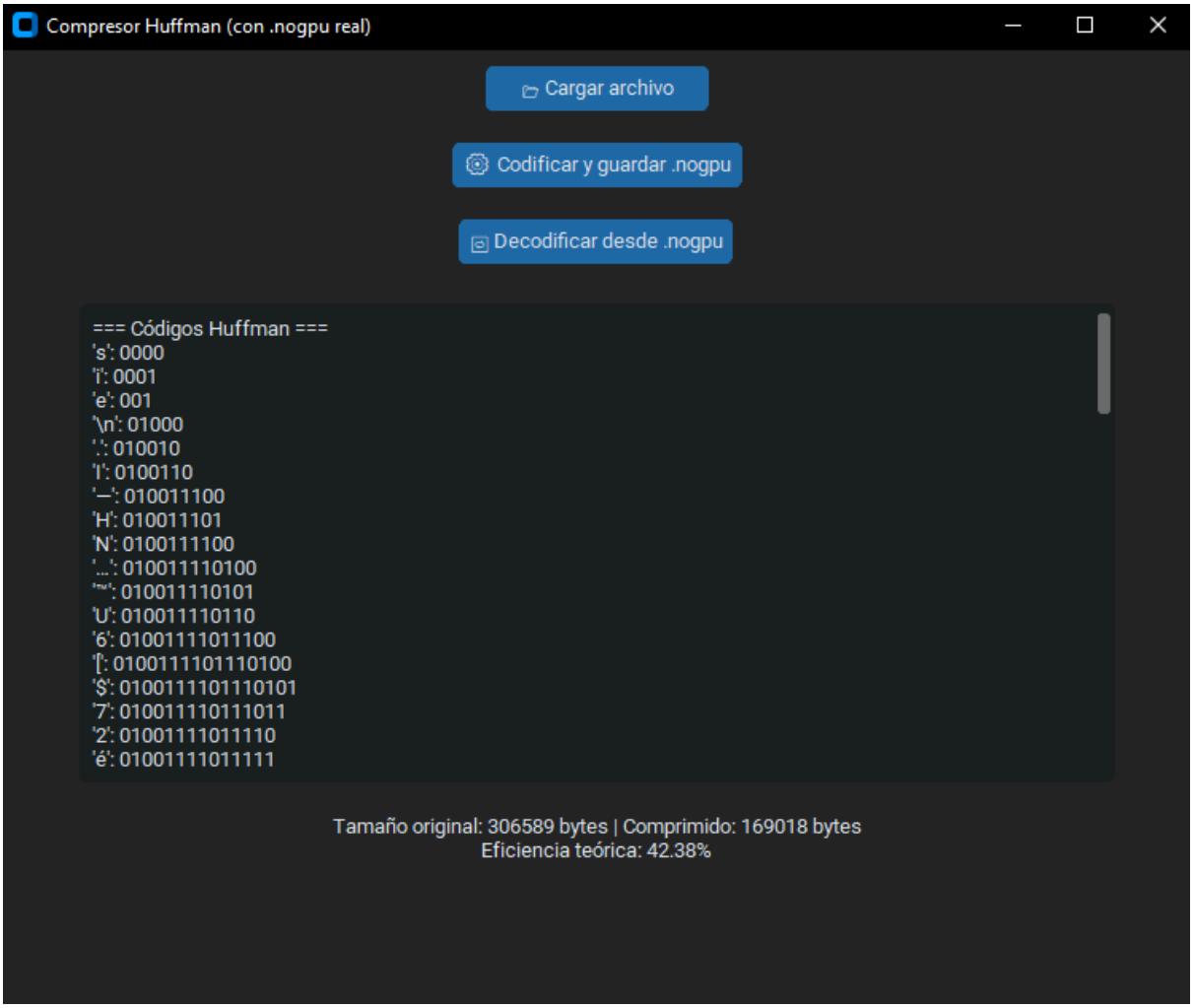


Una vez cargado el archivo, se puede apreciar el contenido del archivo, donde nos señala un tamaño original de 290075 caracteres, el siguiente paso será codificar el texto.

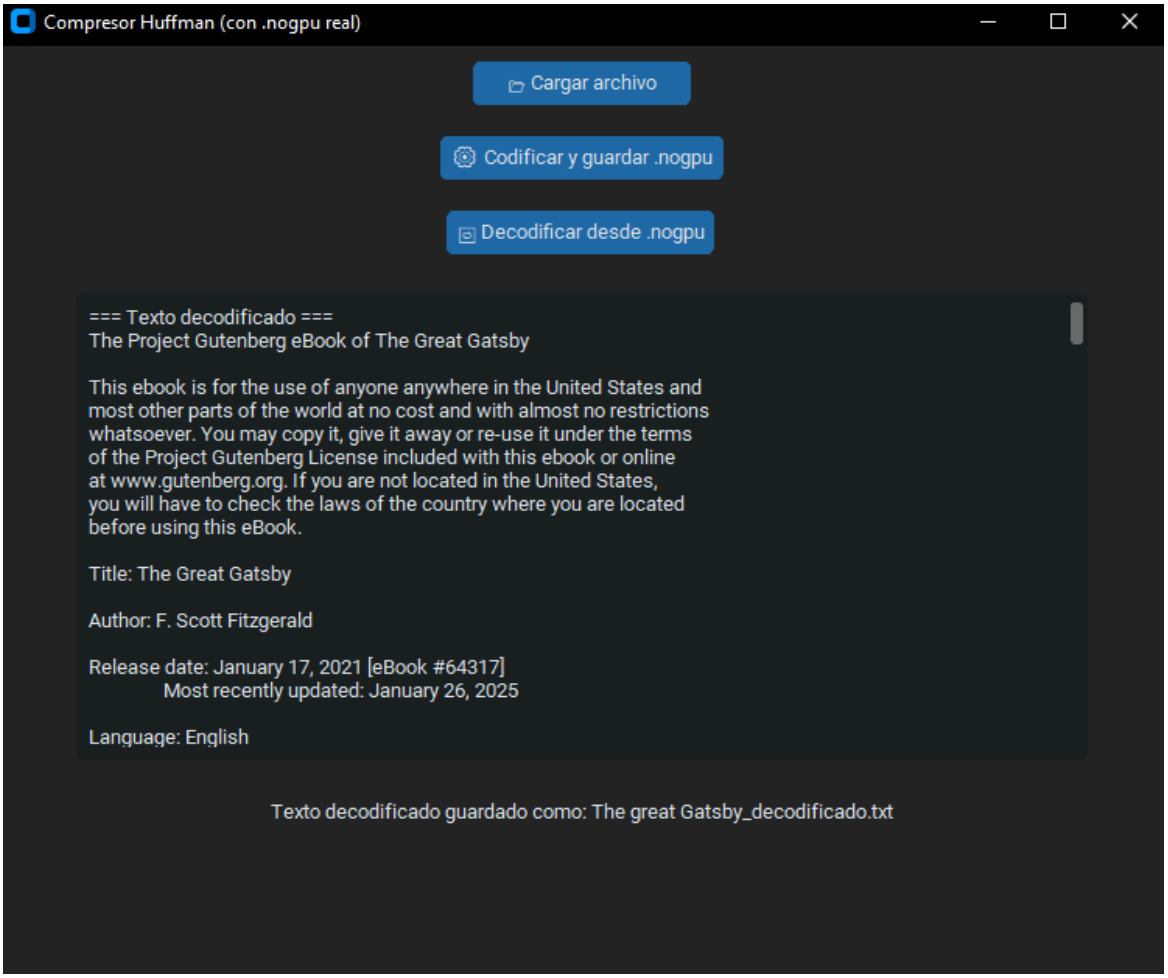




Una vez al seleccionar el boton de codificar cuando ya tenemos cargado nuestro archivo de texto, logramos apreciar la codificación de todos los caracteres, donde se señala el tamaño original del archivo y el tamaño actual del archivo una vez ya comprimido, donde se refleja una eficiencia del 42.38%



Con el botón decodificar desde .nogpu podemos tener el archivo txt directamente para poderlo comprimir, donde busca el archivo de texto que tenemos guardado.



Una vez terminado todo el proceso, se nos crea una carpeta llamada _pycache_ con los archivos binarios que se generaron en el proceso, al igual que se nos guarda las versiones comprimidas y decodificada de nuestro archivo txt