**Participación.  Fecha de entrega: 7/11/2025**

# GUÍA DE USUARIO

Análisis de Algoritmos

Integrantes del equipo:
Gutierrez Vazquez Axel
Quintero Arreola Laura Vanessa

Profesor:
Lopez Arce Delgado Jorge Ernesto

Tema: Técnica Voraz Huffman

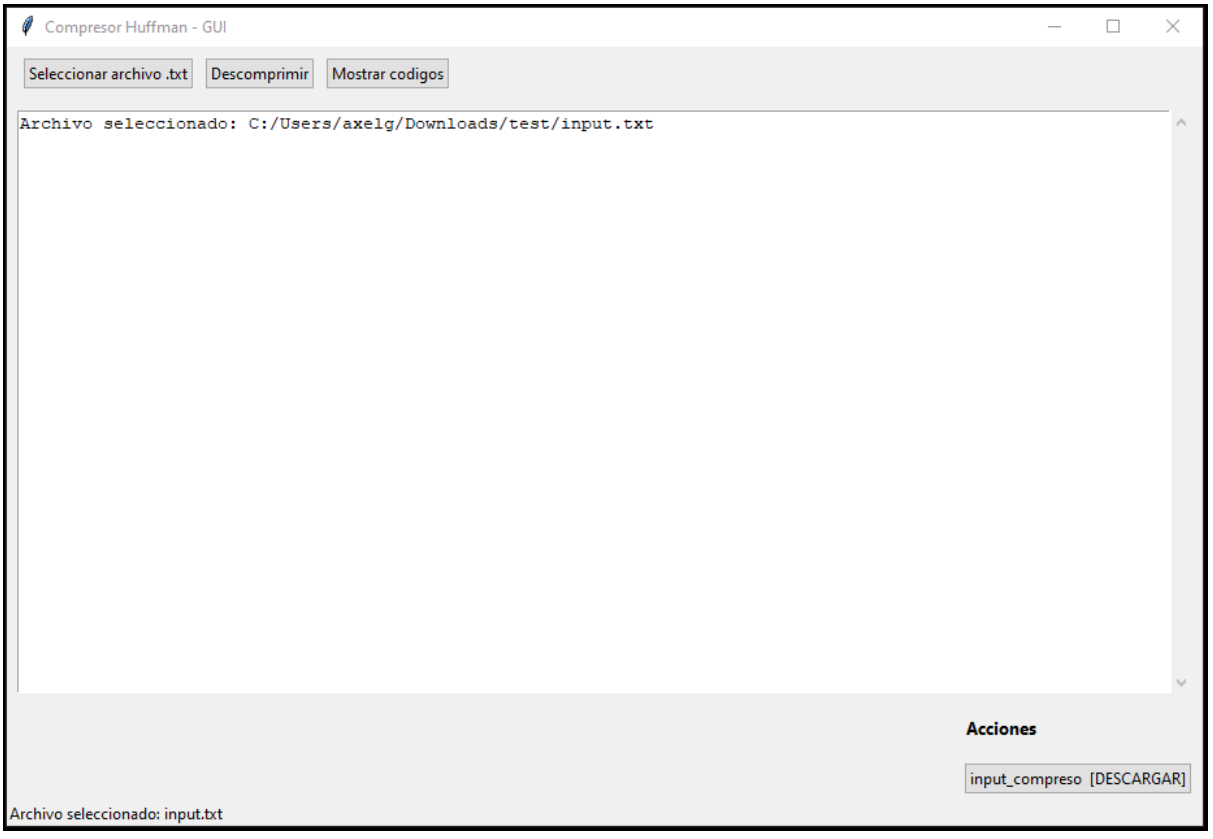
## Introducción

El presente proyecto tiene como finalidad implementar un sistema de compresión y descompresión de archivos de texto utilizando el algoritmo de Huffman, uno de los métodos más eficientes para la codificación sin pérdida de información.
La aplicación fue desarrollada en Python e integra una interfaz gráfica (GUI) que permite al usuario seleccionar, comprimir y descomprimir archivos de manera sencilla, además de visualizar los códigos binarios generados para cada carácter. Este enfoque facilita la comprensión del funcionamiento del algoritmo Huffman y ofrece una herramienta práctica e intuitiva.

## Objetivos

- Implementar el algoritmo de Huffman para la compresión y descompresión de archivos de texto.
- Desarrollar una interfaz gráfica interactiva que permita al usuario seleccionar, comprimir y descomprimir archivos sin utilizar la consola.
- Generar y mostrar los códigos binarios asociados a cada carácter, para comprender la asignación de bits según la frecuencia de aparición.
- Permitir la descarga directa de los archivos comprimidos y descomprimidos mediante botones dinámicos.

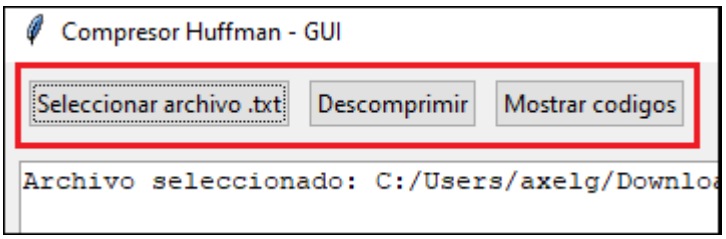
## Guia de usuario



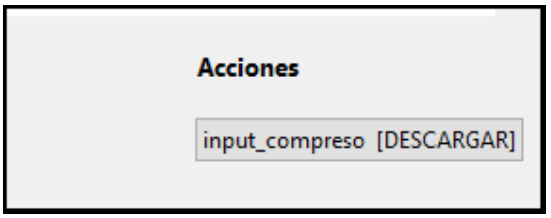

### Instrucciones de uso

Al ejecutar el programa, se mostrará una ventana con los siguientes botones:

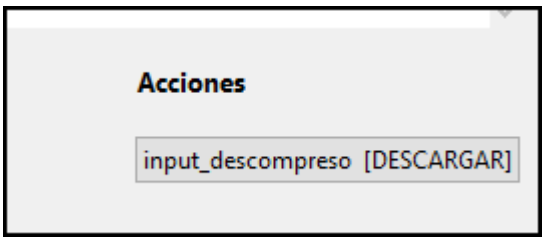
- **[Seleccionar archivo]**
- **[Descomprimir]**
- **[Mostrar códigos]**




Para comprimir un archivo, presiona el botón **[Seleccionar archivo]**.
Se abrirá el explorador de archivos; selecciona el archivo .txt que desees comprimir.
Una vez elegido, en la sección “Acciones” aparecerá un botón con el formato “nombredelarchivo_compreso [DESCARGAR]”.
Presiónalo para descargar el archivo comprimido.

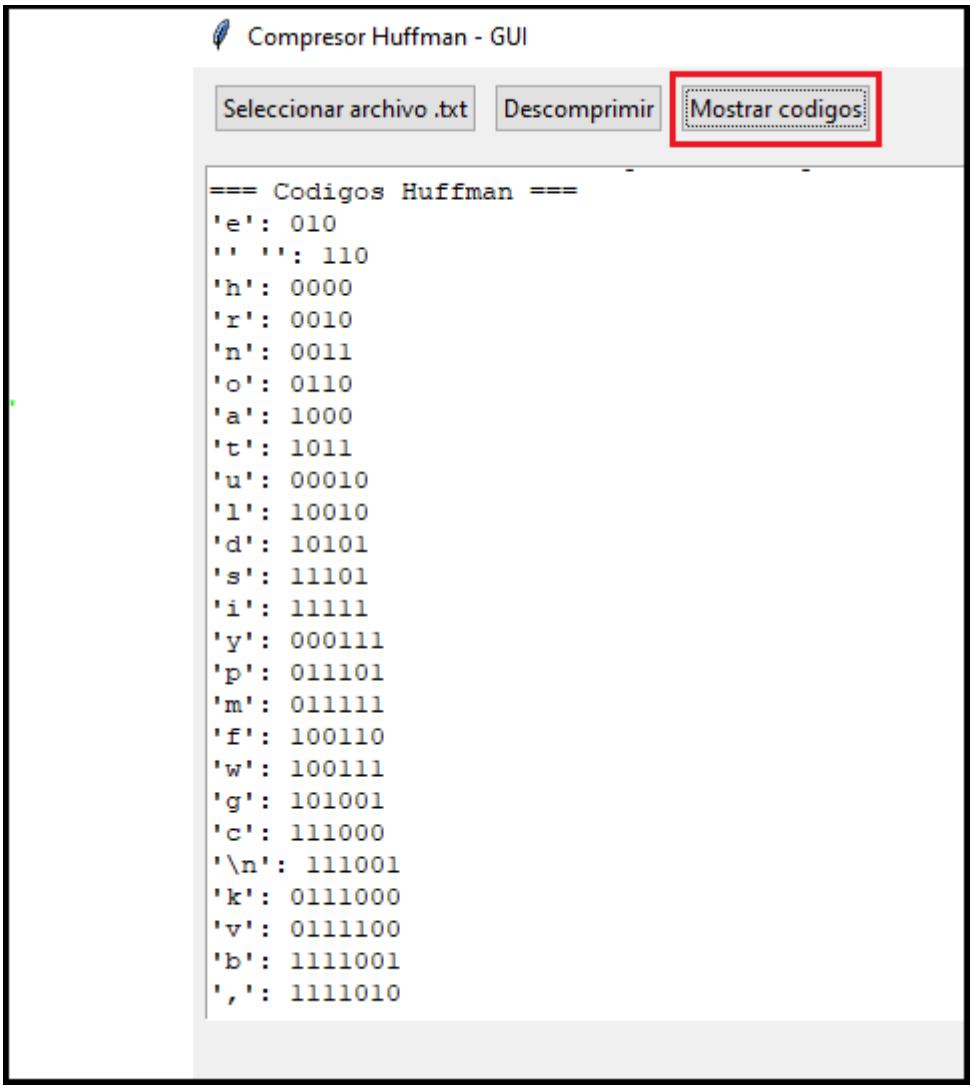


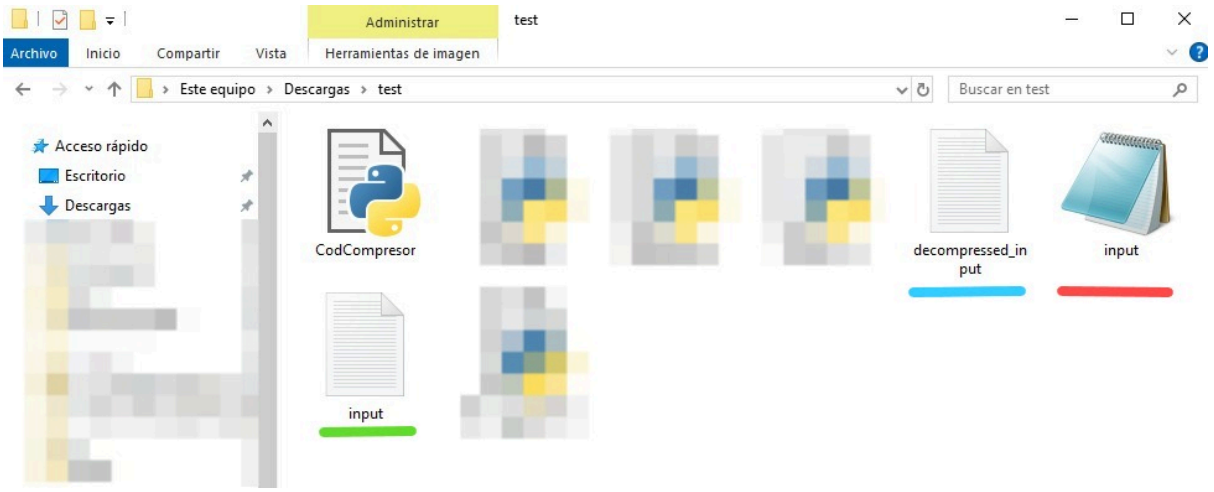

Para descomprimir un archivo, utiliza el botón **[Descomprimir]**.
Selecciona el archivo .bin que deseas descomprimir.
Después de seleccionarlo, aparecerá un nuevo botón con el formato “nombredelarchivo_descompreso [DESCARGAR]”.
Haz clic en él para guardar el archivo descomprimido.




Finalmente, si deseas visualizar los códigos binarios asignados a cada carácter, presiona el botón **[Mostrar códigos]** y se mostrarán todos los códigos binarios generados por el

algoritmo Huffman.




Archivo a comprimir
Archivo descompreso
Archivo comprimido

## Explicación por bloques

```python
import heapq
from collections import Counter
import tkinter as tk
from tkinter import filedialog, messagebox, ttk
import json
import os
import shutil
import sys
import subprocess

# Creamos la clase nodo, para los nodos del arbol
class NodoArbol:
    def __init__(self, caracter, frecuencia):
        self.caracter = caracter
        self.frecuencia = frecuencia
        self.izquierda = None
        self.derecha = None

    # Comparamos la prioridad segun la frecuencia
    def __lt__(self, otro):
        return self.frecuencia < otro.frecuencia

# Creamos el arbol
def arbol_binario(nodo_raiz):
    resultado = {}

    # Recorremos el arbol para saber donde colocar el nodo
    def recorrer_arbol(nodo_actual, codigo_actual):
        if nodo_actual is None:
            return

        # Si es una hoja
        if nodo_actual.caracter is not None:
            # En caso de que el arbol tenga solo un nodo
            if codigo_actual == "":
                resultado[nodo_actual.caracter] = "0"
            else:
                resultado[nodo_actual.caracter] = codigo_actual
            return

        # Si es un nodo interno, seguimos recorriendo
        recorrer_arbol(nodo_actual.izquierda, codigo_actual + "0")
        recorrer_arbol(nodo_actual.derecha, codigo_actual + "1")

    recorrer_arbol(nodo_raiz, "")
    return resultado
```

Este bloque de código establece las bases para el algoritmo de Huffman y la aplicación. Primero, importa todas las librerías necesarias, como heapq para la cola de prioridad (esencial para construir el árbol), tkinter para la interfaz gráfica, y json, os, etc., para el manejo de archivos. Luego, define la clase NodoArbol, que es la estructura fundamental para cada nodo del árbol, almacenando el caracter, su frecuencia, y sus hijos (izquierda, derecha). Es crucial el método __lt__, que permite a heapq ordenar los nodos por la frecuencia más baja. Finalmente, la función arbol_binario toma la raíz de un árbol de Huffman ya construido y genera el diccionario de códigos binarios; lo hace mediante una función recursiva interna (recorrer_arbol) que navega por el árbol, añadiendo "0" al ir a la izquierda y "1" a la derecha, hasta que llega a una hoja (un carácter), momento en el que guarda el código completo en el diccionario resultado.

---

```python
def codificar_texto(texto, codigo):
    texto_codificado = []
    for caracter in texto:
        texto_codificado.append(codigo[caracter])

    # Se juntan los resultados de cada codificacion
    return "".join(texto_codificado)

# Decodificamos el texto usando el arbol
def decodificar_texto(texto_codificado, nodo_raiz):
    # En caso de que el arbol tenga solo un nodo
    if nodo_raiz.izquierda is None and nodo_raiz.derecha is None:
        return nodo_raiz.caracter * len(texto_codificado)

    texto_decodificado = []
    nodo_actual = nodo_raiz

    # Recorremos bit por bit el texto codificado
    for bit in texto_codificado:

        # Si es '0', vamos a la izquierda
        if bit == '0':
            nodo_actual = nodo_actual.izquierda
        # Si es '1', vamos a la derecha
        else: # bit == '1'
            nodo_actual = nodo_actual.derecha

        # Verificamos si llegamos a una HOJA
        if nodo_actual.caracter is not None:
            texto_decodificado.append(nodo_actual.caracter)
            nodo_actual = nodo_raiz

    # Se juntan los caracteres
    return "".join(texto_decodificado)


# Nombres de archivos
archivo_entrada = "input.txt"
archivo_codificado = "codificado.txt"
archivo_decodificado = "decodificado.txt"
```

Este fragmento de código maneja la codificación y decodificación del texto. La función codificar_texto recibe el texto original y el diccionario de códigos; recorre cada carácter del texto, busca su código binario correspondiente en el diccionario y los une todos en una larga cadena de bits. La función decodificar_texto hace lo opuesto: recibe la cadena de bits y el árbol de Huffman (nodo_raiz). Recorre bit por bit la cadena, navegando por el árbol (izquierda para '0', derecha para '1') desde la raíz. Cada vez que llega a un nodo hoja (uno que tiene un caracter), añade ese carácter al resultado y reinicia la navegación desde la nodo_raiz para encontrar el siguiente carácter. Finalmente, las últimas líneas definen nombres de archivo por defecto que el programa podría usar.

---

```python
def pad_encoded_text(encoded):
    extra_padding = (8 - len(encoded) % 8) % 8
    encoded_padded = encoded + "0"*extra_padding
    return encoded_padded, extra_padding

def get_byte_array(padded_encoded):
    if len(padded_encoded) % 8 != 0:
        raise ValueError("Encoded text length not divisible by 8.")
    b_arr = bytearray()
    for i in range(0, len(padded_encoded), 8):
        byte = padded_encoded[i:i+8]
        b_arr.append(int(byte, 2))
    return bytes(b_arr)

def guardar_comprimido(ruta_salida, codes, padded_bytes, extra_padding, original_name):
    header = {
        "codes": codes,
        "padding": extra_padding,
        "original_name": original_name
    }
    header_json = json.dumps(header, ensure_ascii=False).encode('utf-8')
    header_len = len(header_json)
    with open(ruta_salida, "wb") as f:
        f.write(header_len.to_bytes(4, byteorder='big'))
        f.write(header_json)
        f.write(padded_bytes)

def leer_comprimido(ruta):
    with open(ruta, "rb") as f:
        header_len_bytes = f.read(4)
        if len(header_len_bytes) < 4:
            raise ValueError("Archivo comprimido corrupto o incompleto.")
        header_len = int.from_bytes(header_len_bytes, byteorder='big')
        header_json = f.read(header_len)
        header = json.loads(header_json.decode('utf-8'))
        compressed_bytes = f.read()
    return header, compressed_bytes

def bytes_a_bitstring(bts):
    bits = []
    for byte in bts:
        bits.append(f"{byte:08b}")
    return "".join(bits)
```

Este bloque de código maneja la serialización y deserialización del archivo comprimido, es decir, cómo se guarda y se lee en disco. La función pad_encoded_text añade ceros de relleno al final de la cadena de bits para asegurar que la longitud total sea un múltiplo de 8, permitiendo que get_byte_array la convierta en una secuencia de bytes reales. La función guardar_comprimido es vital: crea un header (encabezado) en formato JSON que almacena los códigos de Huffman y la cantidad de relleno, luego escribe en el archivo .bin la longitud del header, el header mismo, y finalmente los bytes comprimidos. leer_comprimido hace lo opuesto, lee el header basándose en su longitud y luego los datos comprimidos, devolviendo ambos. Finalmente, bytes_a_bitstring se usa en la descompresión para reconvertir los bytes leídos del archivo en la cadena de bits original.

---

```python
def construir_arbol_desde_codigos(codes):
    root = NodoArbol(None, 0)
    for ch, code in codes.items():
        node = root
        for bit in code:
            if bit == '0':
                if node.izquierda is None:
                    node.izquierda = NodoArbol(None, 0)
                node = node.izquierda
            else:
                if node.derecha is None:
                    node.derecha = NodoArbol(None, 0)
                node = node.derecha
        node.caracter = ch
    return root

# crear GUI con botones comprimir y descomprimir y botones dinamicos de descarga
class HuffmanGUI:
    def __init__(self, root):
        self.root = root
        root.title("Compresor Huffman - GUI")
        root.geometry("920x600")
        self.file_path = None
        self.compressed_path = None
        self.decompressed_path = None
        self.last_freq = None
        self.last_codes = None
        self.btn_dynamic_comp = None
        self.btn_dynamic_desc = None
        self.dynamic_comp_mode = None  # 'compress' o 'download'
        self.dynamic_desc_mode = None  # 'decompress' o 'download'
        self.selected_bin_for_desc = None

        top = ttk.Frame(root, padding=8)
        top.pack(side=tk.TOP, fill=tk.X)

        # boton para seleccionar input (opcional, pero necesario si input.txt no esta en la carpeta)
        self.btn_select = ttk.Button(top, text="Seleccionar archivo .txt", command=self.seleccionar_archivo)
        self.btn_select.pack(side=tk.LEFT, padx=4)

        # boton descomprimir (queda visible siempre): ahora solo selecciona .bin
        self.btn_decompress = ttk.Button(top, text="Descomprimir", command=self.seleccionar_bin_para_descomprimir)
        self.btn_decompress.pack(side=tk.LEFT, padx=4)
```

Este código es crucial para la descompresión y para iniciar la interfaz gráfica. La función construir_arbol_desde_codigos es la contraparte de arbol_binario; reconstruye el árbol de Huffman no a partir de las frecuencias, sino del diccionario de códigos que se guardó en el archivo .bin. Recorre el código binario de cada carácter (ej. '011') y crea el camino correspondiente en el árbol (izquierda, derecha, derecha) para poder decodificar. Inmediatamente después, comienza la clase HuffmanGUI, que define la aplicación visual con Tkinter. El constructor __init__ configura la ventana principal (título, tamaño) e inicializa variables de estado esenciales (como file_path, compressed_path y los modos de los botones dinámicos). Finalmente, crea el panel superior (top) y añade los botones estáticos: "Seleccionar archivo .txt" y "Descomprimir".

---

```python

        mid = ttk.Frame(root, padding=8)
        mid.pack(fill=tk.BOTH, expand=True)

        # cuadro de texto para informacion
        self.text_info = tk.Text(mid, wrap=tk.NONE)
        self.text_info.pack(side=tk.LEFT, fill=tk.BOTH, expand=True)

        yscroll = ttk.Scrollbar(mid, orient=tk.VERTICAL, command=self.text_info.yview)
        yscroll.pack(side=tk.LEFT, fill=tk.Y)
        self.text_info['yscrollcommand'] = yscroll.set

        # panel derecho donde apareceran los botones dinamicos (reemplaza la tabla)
        right = ttk.Frame(root, padding=8)
        right.pack(side=tk.RIGHT, fill=tk.Y)

        lbl = ttk.Label(right, text="Acciones", font=("Segoe UI", 10, "bold"))
        lbl.pack(anchor=tk.NW)

        # panel donde apareceran los botones dinamicos
        self.buttons_panel = ttk.Frame(right)
        self.buttons_panel.pack(fill=tk.Y, pady=10)

        self.status = ttk.Label(root, text="Listo")
        self.status.pack(side=tk.BOTTOM, fill=tk.X)

    # funcion para escribir info en el cuadro de texto
    def escribir_info(self, texto):
        self.text_info.insert(tk.END, texto + "\n")
        self.text_info.see(tk.END)

```

Este bloque de código construye la estructura principal de la interfaz gráfica (GUI). Primero, crea un panel central (mid) que se expande para rellenar el espacio, y dentro de él coloca el text_info (un cuadro de texto grande) y su yscroll (barra de desplazamiento vertical). Luego, crea el panel right (derecho), que servirá como contenedor para los botones de acción dinámicos (como descargar). Finalmente, añade una barra de status en la parte inferior de la ventana para mostrar mensajes cortos. También define la función escribir_info, un método muy importante que permite añadir texto al cuadro text_info y hacer scroll automáticamente hasta el final, mostrando siempre la información más reciente.

---

```python
    def seleccionar_archivo(self):
        path = filedialog.askopenfilename(filetypes=[("Text files","*.txt")])
        if not path:
            return
        self.file_path = path
        self.escribir_info(f"Archivo seleccionado: {path}")
        self.status.config(text=f"Archivo seleccionado: {os.path.basename(path)}")
        # quitar botones dinamicos si existian
        self._quitar_btns_dinamicos()
        # crear el boton dinamico de compresion (aparece al seleccionar .txt)
        basename = os.path.splitext(os.path.basename(path))[0]
        display_text = f"{basename}_compreso  [DESCARGAR]"
        self._crear_btn_dinamico_comp(display_text, path)

    # accion comprimir: lee input.txt (o archivo seleccionado si existe) y guarda .bin
    # nota: el boton dinamico realizara la compresion al primer click y luego permitira descarga
    def _crear_btn_dinamico_comp(self, display_text, entrada_path):
        # crear boton en modo 'compress' inicialmente
        self.dynamic_comp_mode = 'compress'
        self.btn_dynamic_comp = ttk.Button(self.buttons_panel, text=display_text, command=lambda: self._accion_dinamica_comp(entrada_path))
        self.btn_dynamic_comp.pack(pady=6, anchor=tk.N)
```

Este bloque de código define la lógica para seleccionar un archivo de texto e iniciar el proceso de compresión. La función seleccionar_archivo se activa con el botón "Seleccionar archivo .txt"; esta abre un diálogo (filedialog) para que el usuario elija un .txt. Una vez seleccionado, actualiza la GUI (el cuadro de texto y la barra de estado), borra cualquier botón dinámico anterior y, lo más importante, llama a _crear_btn_dinamico_comp. Esta segunda función crea un nuevo botón dinámico en el panel derecho, configurado inicialmente en modo 'compress'. Este nuevo botón, al ser presionado por primera vez, ejecutará la función _accion_dinamica_comp para comenzar la compresión del archivo seleccionado.

---

```python
    def _accion_dinamica_comp(self, entrada_path):
        # modo compress: crear .bin automaticamente en misma carpeta
        if self.dynamic_comp_mode == 'compress':
            entrada = entrada_path
            if not os.path.exists(entrada):
                messagebox.showwarning("Atencion", f"No se encontro el archivo de entrada: {entrada}")
                return
            try:
                with open(entrada, "r", encoding="utf-8") as f:
                    texto = f.read()
            except Exception as e:
                messagebox.showerror("Error", f"No se pudo leer el archivo: {e}")
                return
            if texto == "":
                messagebox.showwarning("Atencion", "El archivo esta vacio.")
                return

            # contar frecuencias y construir arbol
            conteo = Counter(texto)
            cola = []
            for caracter, frecuencia in conteo.items():
                nodo = NodoArbol(caracter, frecuencia)
                heapq.heappush(cola, nodo)
            if len(cola) == 0:
                messagebox.showwarning("Atencion", "No hay caracteres para procesar.")
                return
            while len(cola) > 1:
                n1 = heapq.heappop(cola)
                n2 = heapq.heappop(cola)
                freq_sum = n1.frecuencia + n2.frecuencia
                padre = NodoArbol(None, freq_sum)
                padre.izquierda = n1
                padre.derecha = n2
                heapq.heappush(cola, padre)
            raiz = heapq.heappop(cola)

            # generar codigos y codificar
            codigos = arbol_binario(raiz)
            texto_cod = codificar_texto(texto, codigos)
            padded, extra = pad_encoded_text(texto_cod)
            bts = get_byte_array(padded)

            # guardar automaticamente en misma carpeta del archivo de entrada
            default_name = os.path.splitext(os.path.basename(entrada))[0] + ".bin"
            carpeta_salida = os.path.dirname(entrada) if os.path.dirname(entrada) else os.getcwd()
            ruta_salida = os.path.join(carpeta_salida, default_name)
```

Este bloque de código es el corazón de la compresión y se ejecuta la primera vez que se presiona el botón dinámico de compresión (cuando está en modo 'compress'). Primero, realiza validaciones cruciales: comprueba que el archivo de entrada exista y que no esté vacío. Luego, implementa el algoritmo de Huffman: usa Counter para contar la frecuencia de cada carácter, después utiliza un heapq (cola de prioridad) para construir el árbol. El bucle while saca los dos nodos con menor frecuencia, los combina bajo un nodo padre, y vuelve a insertar al padre en el heap, repitiendo esto hasta que solo queda el nodo raiz. Finalmente, con el árbol ya construido, genera los códigos binarios, codifica el texto, lo convierte en un byte_array (añadiendo el padding necesario) y define la ruta de salida, que será el mismo nombre del archivo original pero con extensión .bin y en la misma carpeta.

---

```
284             try:
285                 guardar_comprimido(ruta_salida, codigos, bts, extra, os.path.basename(entrada))
286             except Exception as e:
287                 messagebox.showerror("Error", f"No se pudo guardar el archivo comprimido: {e}")
288                 return
289 
290             # actualizar estado
291             self.compressed_path = ruta_salida
292             self.last_freq = dict(conteo)
293             self.last_codes = codigos
294 
295             orig_size = os.path.getsize(entrada)
296             comp_size = os.path.getsize(ruta_salida)
297             bits_original = len(texto) * 8
298             bits_compressed = len(texto_cod)
299             ratio = comp_size / orig_size if orig_size>0 else 0
300 
301             self.escribir_info("=== Resultado de compresion ===")
302             self.escribir_info(f"Archivo original: {entrada} ({orig_size} bytes)")
303             self.escribir_info(f"Archivo comprimido (guardado automaticamente): {ruta_salida} ({comp_size} bytes)")
304             self.escribir_info(f"Bits totales original: {bits_original} bits")
305             self.escribir_info(f"Bits totales comprimido (sin header): {bits_compressed} bits")
306             self.escribir_info(f"Padding agregado: {extra} bits")
307             self.escribir_info(f"Relacion de compresion: {ratio:.3f}")
308             avg_bits_symbol = bits_compressed / len(texto) if len(texto)>0 else 0
309             self.escribir_info(f"Bits promedio por simbolo: {avg_bits_symbol:.3f}")
310             self.status.config(text=f"Comprimido: {os.path.basename(ruta_salida)}")
311 
312             # cambiar modo del boton para que ahora sirva para 'download'
313             self.dynamic_comp_mode = 'download'
314             # boton queda con el mismo texto; su siguiente pulsacion abrira dialogo para guardar copia
315         else:
316             # modo download: pedir ruta donde guardar la copia
317             if not self.compressed_path or not os.path.exists(self.compressed_path):
318                 messagebox.showwarning("Atencion", "No se encontro el archivo comprimido original.")
319                 return
320             default_name = os.path.basename(self.compressed_path)
321             destino = filedialog.asksaveasfilename(defaultextension=".bin", initialfile=default_name, filetypes=[("Huffman bin","*.bin")])
322             if not destino:
323                 return
```

Este bloque de código concluye la operación de compresión. Primero, intenta guardar físicamente el archivo .bin (que contiene el header JSON y los datos) usando guardar_comprimido. Si tiene éxito, actualiza el estado de la GUI, guardando la ruta del nuevo archivo (self.compressed_path) y los códigos generados (self.last_codes). Luego, calcula y muestra en el cuadro de texto de la interfaz un informe detallado con estadísticas clave, como el tamaño original, el tamaño comprimido, el ratio de compresión y los bits promedio por símbolo. Finalmente, y de manera crucial, cambia el modo del botón de 'compress' a 'download', de modo que el bloque else (líneas 316 en adelante) se ejecutará en los siguientes clics. Este modo "download" simplemente abre un diálogo para que el usuario pueda guardar una copia del archivo .bin recién creado en cualquier ubicación.

---

```
            try:
                shutil.copyfile(self.compressed_path, destino)
                messagebox.showinfo("Descarga completa", f"Archivo guardado en:\n{destino}")
                carpeta = os.path.dirname(destino)
                try:
                    if sys.platform.startswith("win"):
                        os.startfile(carpeta)
                    elif sys.platform == "darwin":
                        subprocess.run(["open", carpeta])
                    else:
                        subprocess.run(["xdg-open", carpeta])
                except Exception:
                    pass
            except Exception as e:
                messagebox.showerror("Error", f"No se pudo guardar la copia: {e}")

    # seleccionar .bin para descomprimir (solo selecciona, no procesa)
    def seleccionar_bin_para_descomprimir(self):
        path = filedialog.askopenfilename(filetypes=[("Huffman bin","*.bin")])
        if not path:
            return
        # guardar seleccion y crear boton dinamico para descompresion/descarga
        self.selected_bin_for_desc = path
        self.escribir_info(f"Archivo .bin seleccionado para descompresion: {path}")
        self.status.config(text=f"Bin seleccionado: {os.path.basename(path)}")
        # quitar botones dinamicos previos y crear nuevo boton
        self._quitar_btns_dinamicos()
        basename = os.path.splitext(os.path.basename(path))[0]
        display_text = f"{basename}_descompreso  [DESCARGAR]"
        # crear boton en modo 'decompress' inicialmente
        self.dynamic_desc_mode = 'decompress'
        self.btn_dynamic_desc = ttk.Button(self.buttons_panel, text=display_text, command=self._accion_dinamica_desc)
        self.btn_dynamic_desc.pack(pady=6, anchor=tk.N)

    # funcion dinamica para el boton descompreso: primero descomprime (al primer click), luego pasa a modo descarga
    def _accion_dinamica_desc(self):
        if self.dynamic_desc_mode == 'decompress':
            ruta = self.selected_bin_for_desc
            if not ruta or not os.path.exists(ruta):
                messagebox.showwarning("Atencion", "No se selecciono un archivo .bin valido.")
                return
```

Este bloque de código maneja dos flujos: primero, concluye la acción de "descarga" (copia) del archivo comprimido. Utiliza shutil.copyfile para guardar la copia en el destino elegido por el usuario y luego intenta abrir la carpeta contenedora en el explorador de archivos del sistema operativo (Windows, macOS o Linux). La segunda parte inicia el flujo de descompresión: la función seleccionar_bin_para_descomprimir se activa con el botón "Descomprimir", abre un diálogo para seleccionar un archivo .bin y, al igual que en la compresión, crea un botón dinámico. Este nuevo botón llamará a _accion_dinamica_desc, que, como se ve al final, verificará que el modo sea 'decompress' y que el archivo seleccionado sea válido antes de proceder.

---

```
            try:
                header, compressed = leer_comprimido(ruta)
                codes = header["codes"]
                padding = header.get("padding", 0)
                bits = bytes_a_bitstring(compressed)
                if padding:
                    bits = bits[:-padding]
                raiz = construir_arbol_desde_codigos(codes)
                texto_dec = decodificar_texto(bits, raiz)
                default_name = header.get("original_name", "descomprimido.txt")
                # guardar automaticamente el .txt descomprimido en la misma carpeta del .bin
                carpeta_bin = os.path.dirname(ruta) if os.path.dirname(ruta) else os.getcwd()
                ruta_salida = os.path.join(carpeta_bin, "decompressed_" + default_name)
                with open(ruta_salida, "w", encoding="utf-8") as f:
                    f.write(texto_dec)
                self.decompressed_path = ruta_salida
                self.escribir_info("=== Descompresion completada ===")
                self.escribir_info(f"Archivo leido: {ruta}")
                self.escribir_info(f"Archivo descomprimido (guardado automaticamente): {ruta_salida} ({len(texto_dec)} caracteres)")
                self.status.config(text=f"Descomprimido: {os.path.basename(ruta_salida)}")

                # cambiar modo del boton para que ahora permita descargar la copia
                self.dynamic_desc_mode = 'download'
                # boton mantiene el mismo texto; siguientes clicks abriran dialogo para guardar copia
            except Exception as e:
                messagebox.showerror("Error", f"No se pudo descomprimir: {e}")
        else:
            # modo download: preguntar donde guardar copia del .txt descomprimido
            if not self.decompressed_path or not os.path.exists(self.decompressed_path):
                messagebox.showwarning("Atencion", "No se encontro el archivo descomprimido original.")
                return
            default_name = os.path.basename(self.decompressed_path)
            destino = filedialog.asksaveasfilename(defaultextension=".txt", initialfile=default_name, filetypes=[("Text file","*.txt")])
            if not destino:
                return
```

Este bloque de código es la lógica central de la descompresión, ejecutándose la primera vez que se presiona el botón dinámico de descompresión (modo 'decompress'). Primero, lee el archivo .bin para obtener el header (que contiene los códigos y el padding) y los datos compressed. Luego, convierte los bytes comprimidos de nuevo en una cadena de bits, elimina los bits de relleno (padding), y usa construir_arbol_desde_codigos para recrear el árbol de Huffman. Con el árbol listo, llama a decodificar_texto para recuperar el texto original, lo guarda automáticamente como un nuevo archivo .txt en la misma carpeta y actualiza la interfaz. Finalmente, cambia el modo del botón a 'download', para que los clics siguientes (el bloque else) permitan al usuario guardar copias adicionales del archivo ya descomprimido.

---

```
            try:
                shutil.copyfile(self.decompressed_path, destino)
                messagebox.showinfo("Descarga completa", f"Archivo guardado en:\n{destino}")
                carpeta = os.path.dirname(destino)
                try:
                    if sys.platform.startswith("win"):
                        os.startfile(carpeta)
                    elif sys.platform == "darwin":
                        subprocess.run(["open", carpeta])
                    else:
                        subprocess.run(["xdg-open", carpeta])
                except Exception:
                    pass
            except Exception as e:
                messagebox.showerror("Error", f"No se pudo guardar la copia: {e}")

    # mostrar codigos en el cuadro de texto
    def mostrar_codigos(self):
        if not self.last_codes:
            messagebox.showinfo("Info", "No hay codigos generados aun. Comprime primero un archivo.")
            return
        self.escribir_info("=== Codigos Huffman ===")
        for ch, code in sorted(self.last_codes.items(), key=lambda x: (len(x[1]), x[1])):
            disp = ch
            if ch == "\n":
                disp = "\\n"
            elif ch == "\t":
                disp = "\\t"
            elif ch == " ":
                disp = "' '"
            self.escribir_info(f"'{disp}': {code}")

```

Este bloque de código finaliza la acción de "descargar" (copiar) el archivo descomprimido. Primero, usa shutil.copyfile para guardar la copia en el destino seleccionado por el usuario. Inmediatamente después, intenta abrir la carpeta que contiene el archivo guardado, usando el comando apropiado para el sistema operativo (Windows, macOS o Linux). A continuación, se define la función mostrar_codigos, que se activa con el botón "Mostrar códigos". Esta función verifica si se ha realizado una compresión (self.last_codes); si es así, recorre el diccionario de códigos de Huffman, los ordena por longitud, y los imprime de forma legible (manejando caracteres especiales como \n o \t) en el cuadro de texto de la interfaz.

---

```
433     def _quitar_btns_dinamicos(self):
434         if self.btn_dynamic_comp is not None:
435             try:
436                 self.btn_dynamic_comp.destroy()
437             except:
438                 pass
439             self.btn_dynamic_comp = None
440             self.dynamic_comp_mode = None
441         if self.btn_dynamic_desc is not None:
442             try:
443                 self.btn_dynamic_desc.destroy()
444             except:
445                 pass
446             self.btn_dynamic_desc = None
447             self.dynamic_desc_mode = None
448             self.selected_bin_for_desc = None
449 
450 def main():
451     root = tk.Tk()
452     app = HuffmanGUI(root)
453     root.mainloop()
454 
455 if __name__ == "__main__":
456     main()
457 
```

Este bloque de código final es el que gestiona la limpieza y el arranque de la aplicación. La función _quitar_btns_dinamicos es una utilidad interna crucial: se encarga de eliminar de la interfaz cualquier botón dinámico (tanto el de compresión como el de descompresión) que esté visible. Utiliza try...except para destruir el widget del botón de forma segura y luego limpia las variables de estado (dynamic_comp_mode, etc.) para evitar comportamientos inesperados. Por último, la función main y el bloque if __name__ == "__main__": son el punto de entrada estándar del programa: crean la ventana raíz de Tkinter, instancian la clase principal HuffmanGUI, y ejecutan root.mainloop() para iniciar el bucle de eventos y mostrar la aplicación al usuario.

## Conclusiones

Quintero Arreola Laura Vanessa
El desarrollo de este proyecto permitió comprobar la eficiencia del algoritmo de Huffman como un método de compresión sin pérdida de información. Mediante la asignación de códigos binarios más cortos a los caracteres más frecuentes, se logra reducir el tamaño total de los archivos sin alterar su contenido original. Este proceso optimiza el uso del almacenamiento y mejora la velocidad de transmisión de datos, garantizando una reconstrucción exacta del texto al descomprimirlo.

Gutierrez Vazquez Axel
La implementación de una interfaz gráfica con Tkinter facilitó la interacción del usuario con el algoritmo, permitiendo realizar procesos de compresión y descompresión de forma intuitiva y accesible. Además, las funciones adicionales, como la visualización de los códigos binarios y la descarga directa de los archivos resultantes, enriquecen la experiencia del usuario al mostrar de manera práctica cómo se aplica el algoritmo de Huffman.