UNIVERSIDAD DE GUADALAJARA

Red Universitaria e Institución Benemérita de Jalisco

CENTRO UNIVERSITARIO DE CIENCIAS EXACTAS E IGENIERÍAS

ANALISIS DE ALGORITMOS

EQUIPO TR3S

GARCIA SALDIVAR HUGO GABRIEL

MACIEL VARGAS OSWALDO DANIEL

D06 09/11/2025

Act. 5: Técnica Voraz Huffman

Entregable para el final del día

Guía de Usuario

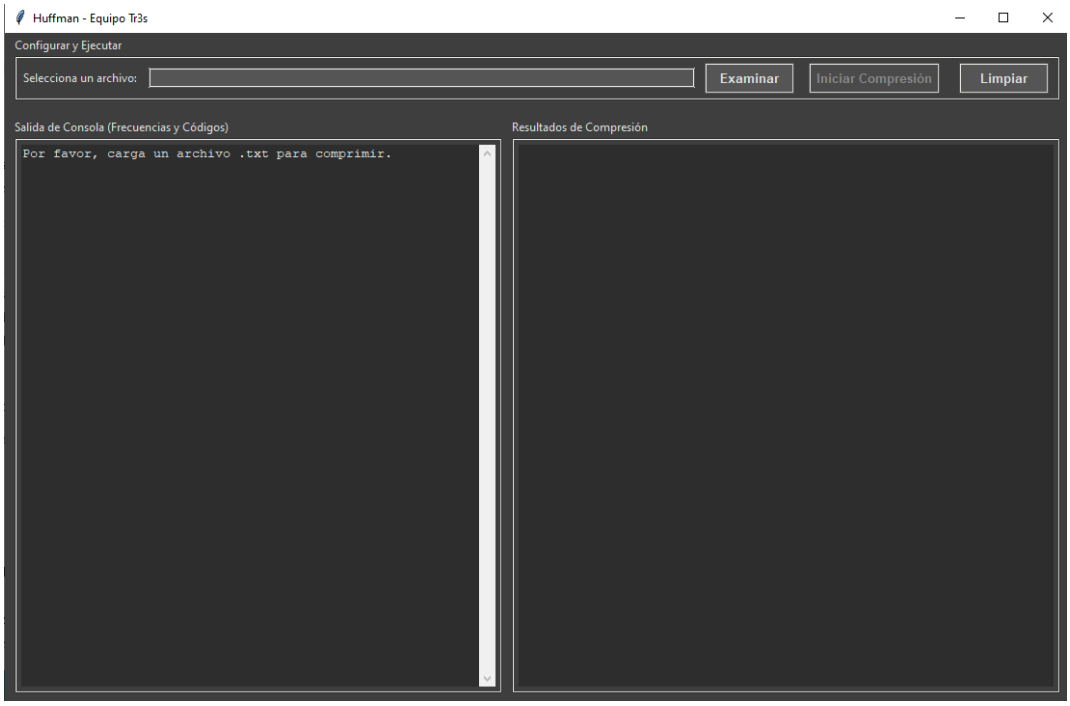
I. Como usar el código

Explicación general

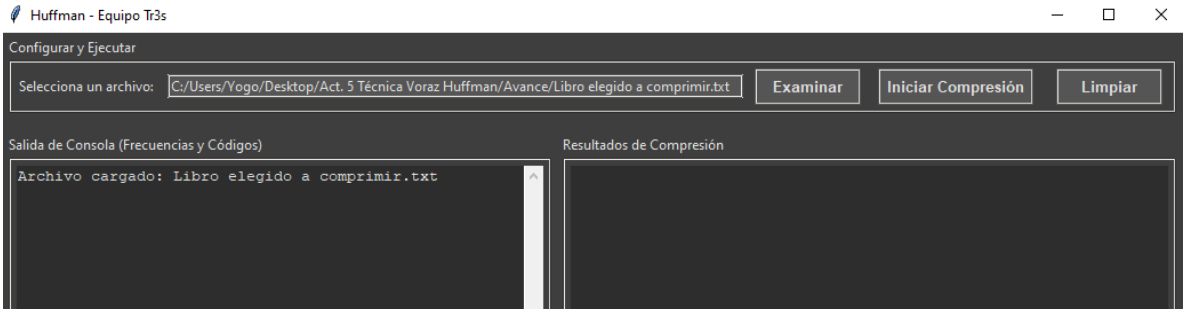
El funcionamiento y objetivo de este código es ver y entender la comprensión de datos (en este caso de texto) utilizando una técnica voraz, en este caso del algoritmo de Huffman. Lo primero que se debe hacer en este programa es cargar un archivo con extensión txt, una vez que el programa lo procese; se podrá iniciar la comprensión de este archivo entregando un archivo en binario. A continuación, se mostrará un ejemplo con el la novela “La Metamorfosis” de Franz Kafka.

Prueba del programa

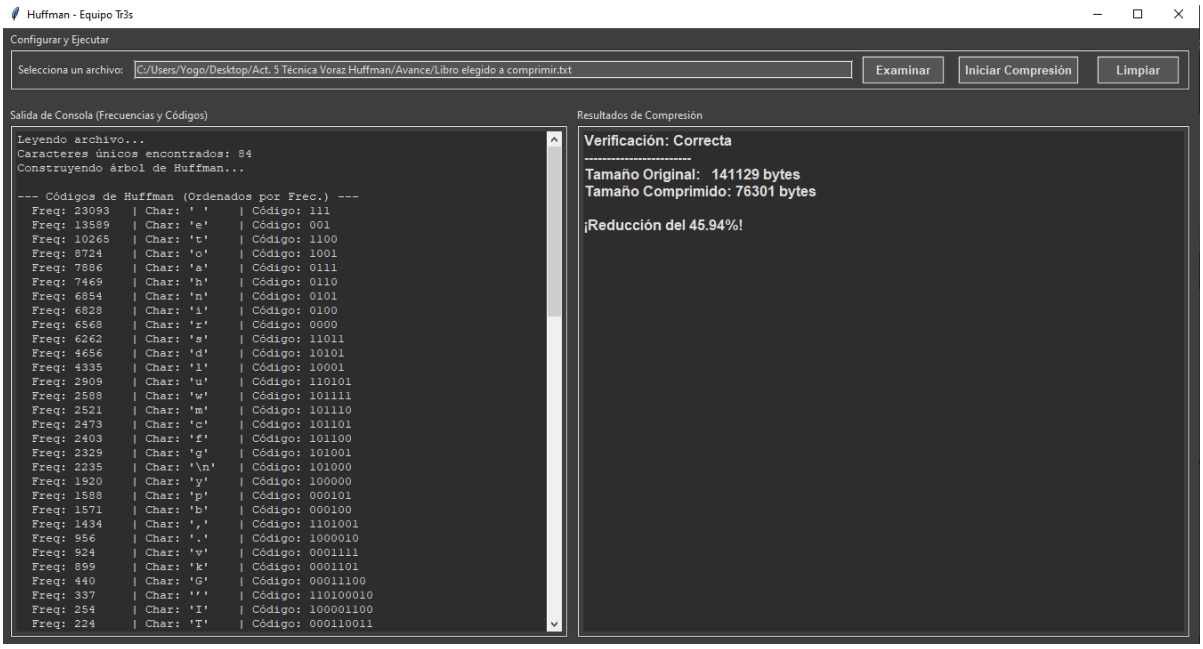
Al iniciar el programa nos encontraremos con la siguiente ventana:



Lo que se debe hacer a continuación es cargar un archivo de texto ya sea escribiendo la ruta absoluta del archivo o haciendo clic en el botón “Examinar”, como mencionamos anteriormente cargaremos la novela “La Metamorfosis” que tiene un peso de 141 KB, cuando carguemos este archivo el mismo programa nos dirá que esta listo para la comprensión.



Si nos equivocamos de archivo podemos hacer clic en el botón “Limpiar” para seleccionar otro archivo, ahora para continuar con la comprensión haremos clic en el botón “Iniciar Comprensión” (es importante añadir que la ruta donde se encuentra el programa debe tener permisos para escribir, porque se creará el archivo binario).



Resultados obtenidos

Al finalizar la comprensión y descomprensión (para verificar que los archivos sean iguales y efectivamente la comprensión sea correcta) el programa mostrará en 2 ventanas los resultados obtenidos, en la ventana de la izquierda se mostrará los estados del procedimiento, los caracteres encontradas, la frecuencia, sus códigos binarios y el estado de los archivos.

```
Leyendo archivo...
Caracteres únicos encontrados: 84
Construyendo árbol de Huffman...

--- Códigos de Huffman (Ordenados por Frec.) ---
  Freq: 23093   | Char: ' '    | Código: 111
  Freq: 13589   | Char: 'e'    | Código: 001
  Freq: 10265   | Char: 't'    | Código: 1100
  Freq: 8724    | Char: 'o'    | Código: 1001
  Freq: 7886    | Char: 'a'    | Código: 0111
  Freq: 7469    | Char: 'h'    | Código: 0110
  Freq: 6854    | Char: 'n'    | Código: 0101
  Freq: 6828    | Char: 'i'    | Código: 0100
  Freq: 6568    | Char: 'r'    | Código: 0000
  Freq: 6262    | Char: 's'    | Código: 11011
  Freq: 4656    | Char: 'd'    | Código: 10101
  Freq: 4335    | Char: 'l'    | Código: 10001
```

En la ventana de la derecha se mostrarán la comparación y conclusión de la comprensión señalando tanto si es correcta como su eficiencia.

```
Verificación: Correcta
----------------------
Tamaño Original:   141129 bytes
Tamaño Comprimido: 76301 bytes

¡Reducción del 45.94%!
```

II. Descripción del código

Explicación teórica

Lo que tenemos que entender primero es que programa logra la comprensión mediante una técnica de programación voraz, más específicamente utilizando el algoritmo de Huffman para la comprensión de datos (en este caso, de texto).

El algoritmo de Huffman, en cambio, implementa una codificación de longitud variable, la idea es asignar códigos binarios cortos a los caracteres más frecuentes y códigos binarios más largos a los caracteres menos frecuentes. De esta forma, el tamaño total del archivo codificado es significativamente menor que el original siempre y cuando se guarde en un archivo que almacene estos códigos binarios (es decir, no tendrá ningún efecto si estos datos se guardan en un archivo de texto).

La idea es crear un árbol con estos datos para que, durante la descomprensión, el programa tome el archivo binario que se generado y siga los códigos binarios dentro del árbol para encontrar el carácter al que hace referencia.

Explicación de funciones

La clase *NodoHuffman* se utiliza para construir el árbol donde se almacena los caracteres, la frecuencia de los mismos, los hijos del nodo entre otras funciones.

```
class NodoHuffman:
    def __init__(self, char, freq, izq=None, der=None):
        self.char = char
        self.freq = freq
        self.izq = izq
        self.der = der
    def __lt__(self, otro):
        return self.freq < otro.freq
```

La función *calcular_frecuencias* hace precisamente esto, lee el texto completo y obtiene las veces que aparece cada carácter, esta función devuelve un diccionario que contiene los caracteres y la frecuencia de aparición.

```
def calcular_frecuencias(texto):
    return Counter(texto)
```

Ahora, la función *construir_arbol_huffman* es la función voraz principal, esta se encarga de construir en el árbol según el carácter y frecuencia obtenidas de la función anterior. Ahora mete las hojas en una cola de prioridad (heapq), mientras haya mas de un nodo en la cola, vorazmente saca los dos nodos con la frecuencia mas baja. A su vez crea un nuevo nodo (padre) que une a estos dos nodos, la frecuencia de este padre es la suma de sus hijos.

```python
def construir_arbol_huffman(frecuencias):
    cola = [NodoHuffman(c, f) for c, f in frecuencias.items()]
    heapq.heapify(cola)
    while len(cola) > 1:
        izq = heapq.heappop(cola)
        der = heapq.heappop(cola)
        padre = NodoHuffman(None, izq.freq + der.freq, izq, der)
        heapq.heappush(cola, padre)
    return cola[0] if cola else None
```

Por otro, lado la función *generar_codigos_huffman* recorre el árbol desde la raíz para asignar los códigos binarios. Si va a la izquierda añade un “0” al código, pero si va a la derecha añade un “1”. Al final cuando llega a la hoja guarda el código binario acumulado en un diccionario.

```python
def generar_codigos_huffman(arbol):
    codigos = {}
    def recorrer(nodo, codigo):
        if nodo is None: return
        if nodo.char is not None:
            codigos[nodo.char] = codigo or "0"
        recorrer(nodo.izq, codigo + "0")
        recorrer(nodo.der, codigo + "1")
    recorrer(arbol, "")
    return codigos
```

La función *codificar_texto* vuelve a leer el texto original, carácter por carácter, y usa el mapa de códigos para reemplazar cada carácter por su nuevo código binario.

```python
def codificar_texto(texto, mapa_codigos):
    return "".join(mapa_codigos[c] for c in texto)
```

Por otro lado, la función *decodificar_texto* se utiliza para verificar que la codificación haya sido la correcta y el texto codificado coincida con el original, esta empieza desde la raíz del árbol y sigue el camino de cada código binario hasta llegar a la hoja (el carácter) repite este proceso hasta obtener el texto original decodificado gracias a la codificación.

```python
def decodificar_texto(codificado, arbol):
    if not arbol: return ""
    if arbol.char: return arbol.char * len(codificado)
    nodo, resultado = arbol, ""
    for bit in codificado:
        nodo = nodo.izq if bit == "0" else nodo.der
        if nodo.char:
            resultado += nodo.char
            nodo = arbol
    return resultado
```

Finalmente, la función *guardar_comprimido* toma el string binario (texto_codificado) y lo convierte en un archivo .bin real.

```python
def guardar_comprimido(ruta, texto_codificado):
    salida = os.path.splitext(ruta)[0] + "_comprimido.bin"
    bits_padding = (8 - len(texto_codificado) % 8) % 8
    info_padding = "{:08b}".format(bits_padding)
    texto_codificado_con_padding = info_padding + texto_codificado + ("0" * bits_padding)

    array_bytes = bytearray(
        int(texto_codificado_con_padding[i:i+8], 2)
        for i in range(0, len(texto_codificado_con_padding), 8)
    )

    with open(salida, "wb") as f:
        f.write(array_bytes)
    return salida
```