R&R

Participación. Fecha de entrega 7/11/2025

Análisis de Algoritmos

Integrantes del equipo:
Gutierrez Vazquez Axel
Quintero Arreola Laura Vanessa

Profesor:
Lopez Arce Delgado Jorge Ernesto

Tema: Técnica Voraz Huffman

Asignación de actividades

Gutierrez Vazquez Axel

- ➢ Crear la clase nodo para crear los árboles en el código.
- ➢ Hacer la función para descomprimir el archivo.
- ➢ Hacer una GUI para el código.
- ➢ Seleccionar el libro para probarlo en nuestro código.

Quintero Arreola Laura Vanessa

- ➢ Crear la función para hacer el árbol binario usando la técnica voraz Huffman.
- ➢ Hacer la función para comprimir el archivo.
- ➢ Hacer el diagrama de flujo del código base (sin GUI).
- ➢ Hacer el archivo del equipo (nombre de equipo e integrantes).

*Nota: La guía de usuario se realizará entre ambos miembros del equipo tomando como base los resultados finales y la información de este documento.

Código Base

```
import heapq
from collections import Counter

# 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

# Creamos la codificacion huffman
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"

try:
# Leemos el archivo de entrada
with open(archivo_entrada, 'r', encoding='utf-8') as f:
texto_usuario = f.read()

if not texto_usuario:
      print(f"El archivo '{archivo_entrada}' está vacío.")
else:
# Contar frecuencias
conteo = Counter(texto_usuario)
```

```
# Construir la cola de prioridad
cola_prioridad = []
for caracter, frecuencia in conteo.items():
      nodo = NodoArbol(caracter, frecuencia)
      heapq.heappush(cola_prioridad, nodo)

# Construir el arbol
while len(cola_prioridad) > 1:
nodo_izquierdo = heapq.heappop(cola_prioridad)
nodo_derecho = heapq.heappop(cola_prioridad)
frecuencia_sumada = nodo_izquierdo.frecuencia + nodo_derecho.frecuencia
nodo_padre = NodoArbol(None, frecuencia_sumada)
nodo_padre.izquierda = nodo_izquierdo
nodo_padre.derecha = nodo_derecho
heapq.heappush(cola_prioridad, nodo_padre)

# Obtener el nodo raíz
nodo_raiz = heapq.heappop(cola_prioridad)

# Generar los códigos
codigos_finales = arbol_binario(nodo_raiz)

texto_codificado = codificar_texto(texto_usuario, codigos_finales)
texto_decodificado = decodificar_texto(texto_codificado, nodo_raiz)

# Escribimos los resultados en archivos
with open(archivo_codificado, 'w', encoding='utf-8') as f_cod:
f_cod.write(texto_codificado)
with open(archivo_decodificado, 'w', encoding='utf-8') as f_decod:
f_decod.write(texto_decodificado)
print(f"Proceso completado:")
print(f"{archivo_codificado}")
print(f"{archivo_decodificado}")

except FileNotFoundError:
      print(f"Error: No se encontro el archivo '{archivo_entrada}'.")
      print("Por favor, crea ese archivo con algun texto dentro.")

except Exception as e:
      print(f"Ocurrio un error inesperado: {e}")
```

Diagramas de flujo

Diagrama 1. Flujo Principal del Código

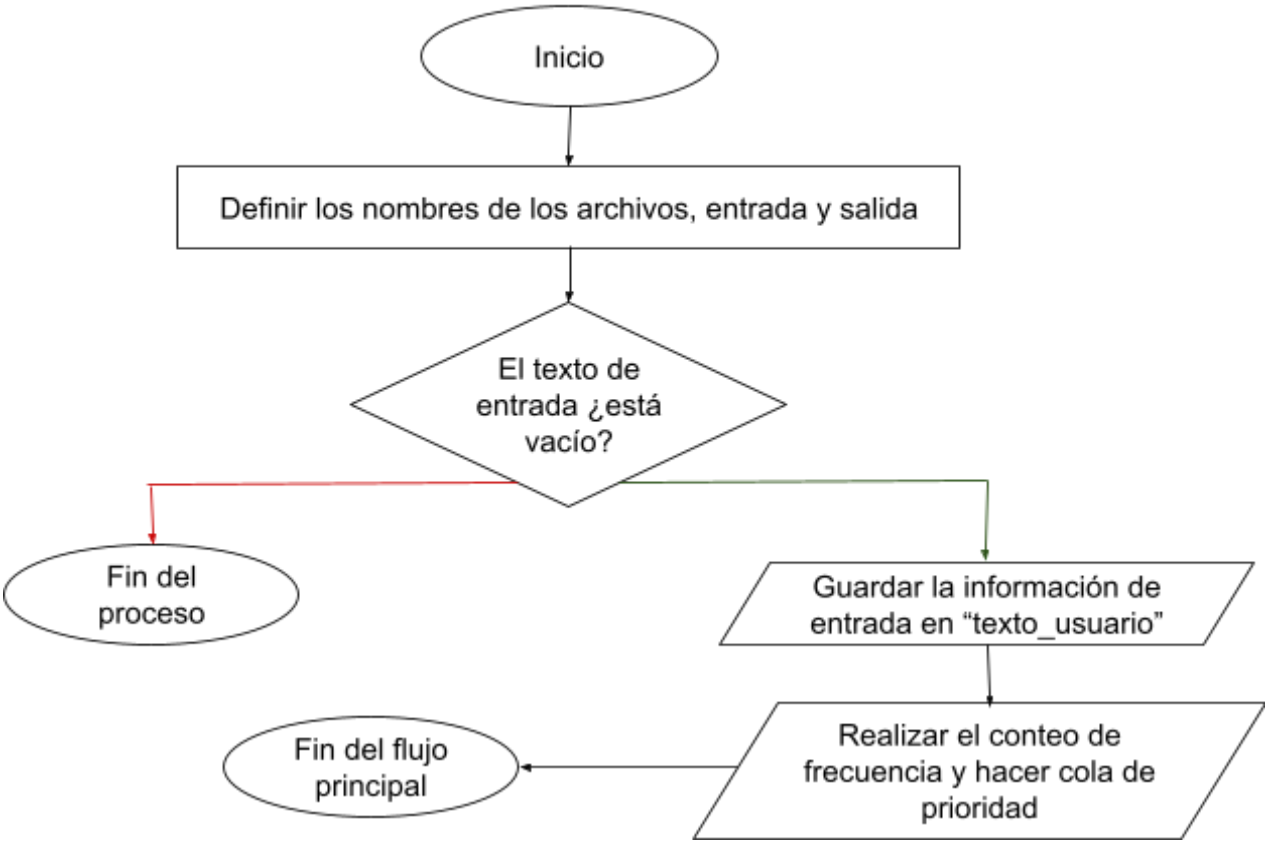
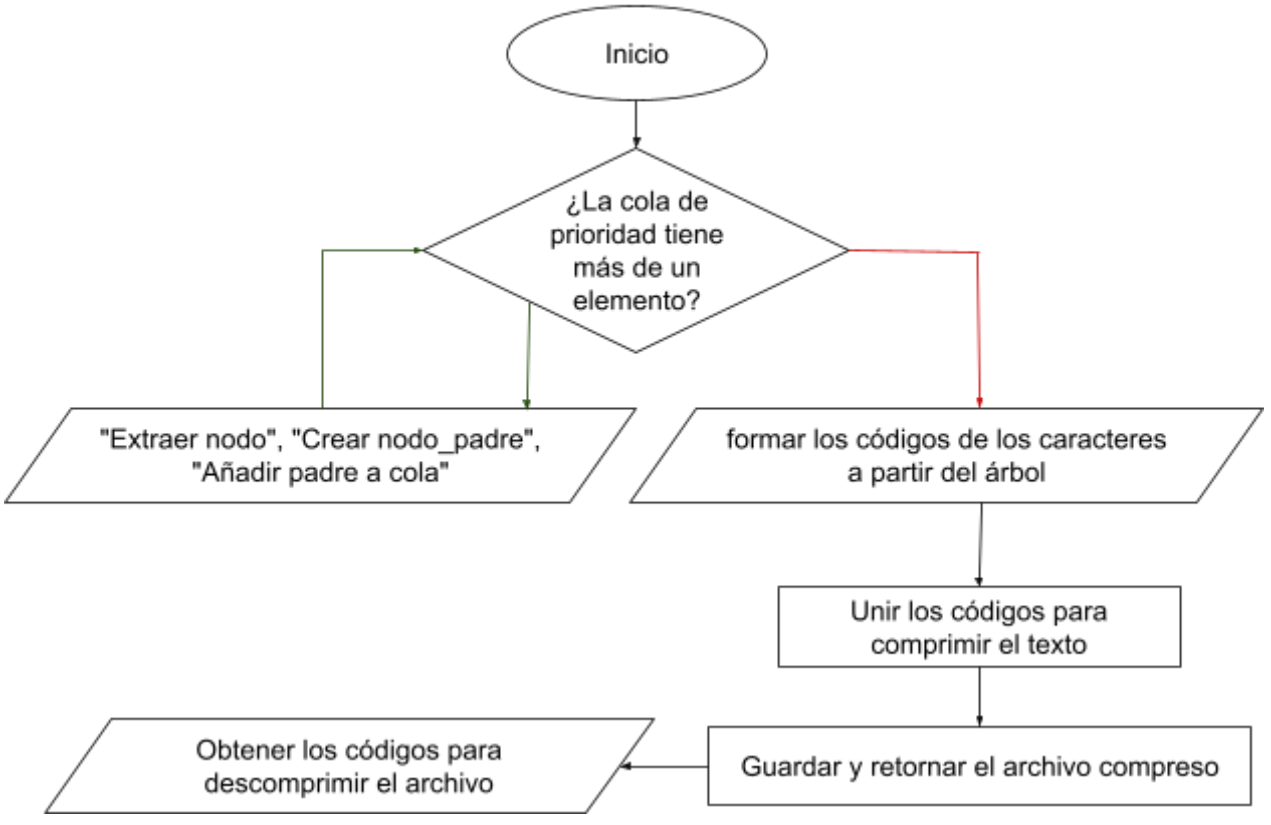
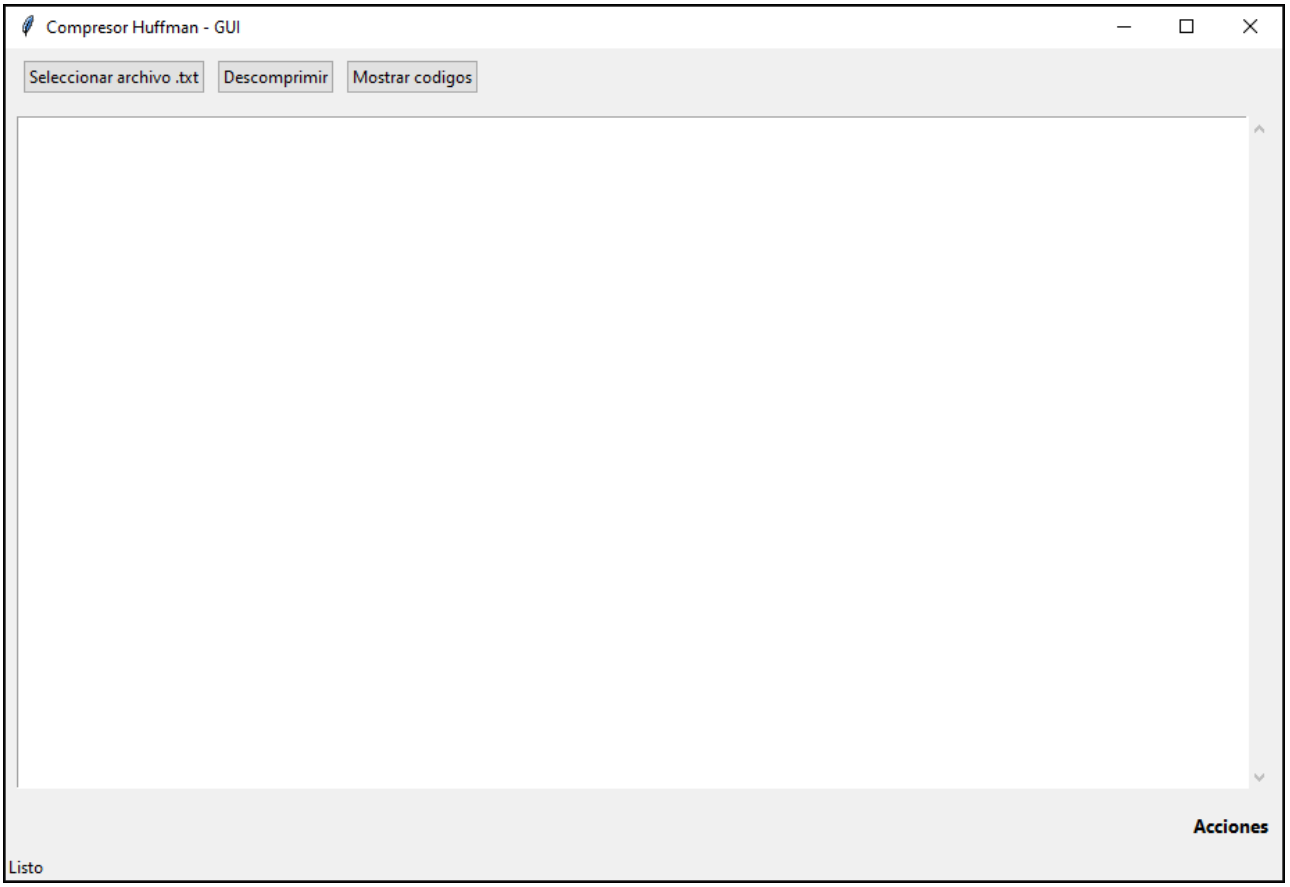


Diagrama 2. Flujo para comprimir y descomprimir



Front End



Libro elegido

Elegimos el libro de “The girl at Silver Thistle” descargado como txt de la página Project Gutenberg