UNIVERSIDAD DE SAN CARLOS DE GUATEMALA
FACULTAD DE INGENIERÍA
ESCUELA DE CIENCIAS
DEPARTAMENTO DE MATEMÁTICA
AUX. BRAYAN HAMLLELO PRADO MARROQUIN

MC2-P

Proyecto Árbol de Huffman

Nombres: Danny Josué González Lémus 202405828

Erick Vinicio Carbonell Esquite 202405365

Catedrático: Carlos Garrido

Fecha: 22/04/2025

Introducción

El presente documento expone el desarrollo y los resultados del proyecto del Árbol de Huffman. El objetivo fue diseñar una aplicación web funcional capaz de realizar la encriptación y desencriptación de mensajes mediante el algoritmo de Huffman, utilizando estructuras de árboles binarios para representar visualmente el proceso de codificación.

La aplicación fue implementada bajo una arquitectura de dos capas, utilizando Python en el backend y React en el frontend. Entre sus funcionalidades se incluyó el ingreso de una frase, su encriptación en binario, la visualización del árbol de Huffman generado con Graphviz, y la exportación del mismo en formato PDF. Además, se presentó una tabla con los caracteres únicos y sus correspondientes códigos binarios.

Objetivos

Objetivo General

Desarrollar una aplicación web que permita encriptar y desencriptar mensajes utilizando el algoritmo de Huffman, integrando tecnologías frontend y backend, y generando la visualización del árbol binario correspondiente en formato PDF.

Objetivos Específicos

1. Implementar un algoritmo propio de codificación de Huffman que permita calcular las frecuencias de los caracteres de una frase y generar el árbol binario correspondiente.
2. Diseñar una interfaz web intuitiva en React que facilite al usuario ingresar frases, ejecutar la codificación, visualizar el árbol generado y obtener la frase encriptada en binario.
3. Integrar herramientas de visualización y exportación, utilizando Graphviz para graficar el árbol de Huffman y generar su respectivo archivo en formato PDF para descarga.

Marco Teórico

1. Teoría de la Información y Compresión de Datos

Concepto de compresión sin pérdida

La compresión sin pérdida se basa en reducir el tamaño de los datos originales de tal forma que, una vez descomprimidos, recuperen exactamente la secuencia inicial de bits. Para ello, se aprovechan redundancias estadísticamente predecibles en el contenido, eliminando información repetida o codificándola de forma más eficiente sin sacrificar ni un solo bit de la información original.

Importancia de la codificación eficiente

Una codificación eficiente reduce el ancho de banda requerido para transmitir datos y optimiza el uso de espacio en sistemas de almacenamiento. En entornos donde el tiempo de respuesta y el costo de transferencia son críticos (por ejemplo, streaming de video o transferencia de grandes volúmenes de datos en la nube), una buena compresión sin pérdida mejora significativamente el rendimiento y reduce los costes operativos.

Aplicaciones reales de la compresión

- Archivos y documentos: formatos ZIP o PNG que usan algoritmos sin pérdida para garantizar la integridad de los datos.
- Imágenes médicas: DICOM con compresión sin pérdida para preservar detalles diagnósticos.
- Transmisiones en tiempo real: protocolos RTP/RTCP que emplean compresión de audio sin pérdida para conversaciones de alta fidelidad.

2. Algoritmo de Huffman

Fundamentos de la codificación por frecuencia

Huffman asigna códigos binarios de longitud variable a cada símbolo según su frecuencia de aparición: los símbolos más frecuentes obtienen códigos más cortos, y los menos frecuentes, códigos más largos. Este principio minimiza la longitud media del mensaje codificado.

Construcción y recorrido del árbol

- Inicialización: cada símbolo se convierte en un nodo con peso igual a su frecuencia.

- Fusión de nodos: se extraen los dos nodos de menor peso, se crea un nuevo nodo padre con peso igual a la suma de ambos y se reinserta en la lista.
- Generación de códigos: una vez sólo queda un nodo (la raíz), se recorre el árbol asignando “0” al enlace izquierdo y “1” al derecho; la concatenación de estos bits desde la raíz a cada hoja forma el código de cada símbolo.

3. Estructuras de Datos: Árboles Binarios

Definición y características generales

Un árbol binario es una estructura jerárquica en la que cada nodo tiene, como máximo, dos hijos: izquierdo y derecho. Esta característica logra un equilibrio entre la simplicidad de navegación y la flexibilidad en la representación de datos.

Componentes básicos

- Nodo: unidad fundamental que contiene un valor o dato.
- Hoja: nodo sin hijos; representa el fin de una rama.
- Raíz: nodo inicial desde el cual se despliegan todas las demás ramas.

Utilidad en estructuras jerárquicas

Los árboles binarios permiten modelar relaciones de dependencia y priorización, facilitan búsquedas binarias eficientes (O (log n) en árboles balanceados) y sirven de base para estructuras avanzadas como AVL, montículos binarios y árboles de decisión.

4. Lenguajes y Herramientas Tecnológicas

Python: lógica de codificación y generación de PDF

Python, con su sintaxis clara y rico ecosistema de librerías (por ejemplo, heapq para manejo de colas de prioridad y reportlab o FPDF para generación de PDF), facilita el procesamiento de cadenas, cálculo de frecuencias y producción de documentación en formatos portables.

React: creación de la interfaz web

React permite construir componentes reutilizables para la interacción con el usuario (cajas de texto, botones), actualizar dinámicamente la vista al cambiar el estado y gestionar eventos como el clic en “Codificar” de manera declarativa y eficiente.

Graphviz: visualización gráfica del árbol

Graphviz genera diagramas a partir de descripciones textuales en formato DOT. Al traducir la estructura del árbol de Huffman a un grafo, facilita la inspección visual de la jerarquía de códigos y su verificación.

Arquitectura de dos capas

Separar la lógica de negocio (cálculo de frecuencias, construcción del árbol, generación de códigos) de la presentación (interfaz React, exportación PDF) mejora la mantenibilidad, permite pruebas unitarias independientes y optimiza la escalabilidad del proyecto.

5. Exportación de Contenidos Digitales

La generación de archivos PDF se emplea como medio estándar para presentar resultados y documentación del proyecto. Un PDF garantiza preservación de la tipografía, formato fijo y compatibilidad cross-platform, lo que facilita la entrega y lectura por parte de terceros sin depender del entorno de desarrollo.

Algoritmo de la Solución

A[Inicio]

A --> B[Usuario accede a la web]

B --> C[Ingresa frase en input]

C --> D{¿Frase válida?}

D -- No --> E[Mostrar mensaje de error]

D -- Sí --> F[Llamada al backend API]

F --> G[Limpiar y normalizar frase]

G --> H[Calcular frecuencias de caracteres]

H --> I[Generar árbol de Huffman]

I --> J[Asignar códigos binarios a caracteres]

J --> K[Codificar frase original]

K --> L[Generar imagen del árbol con Graphviz]

L --> M[Guardar imagen como PDF]

K --> N[Enviar frase codificada al frontend]

M --> O[Habilitar botón de descarga PDF]

O --> P{¿Usuario presiona botón PDF?}

P -- Sí --> Q[Descargar archivo PDF con el árbol]

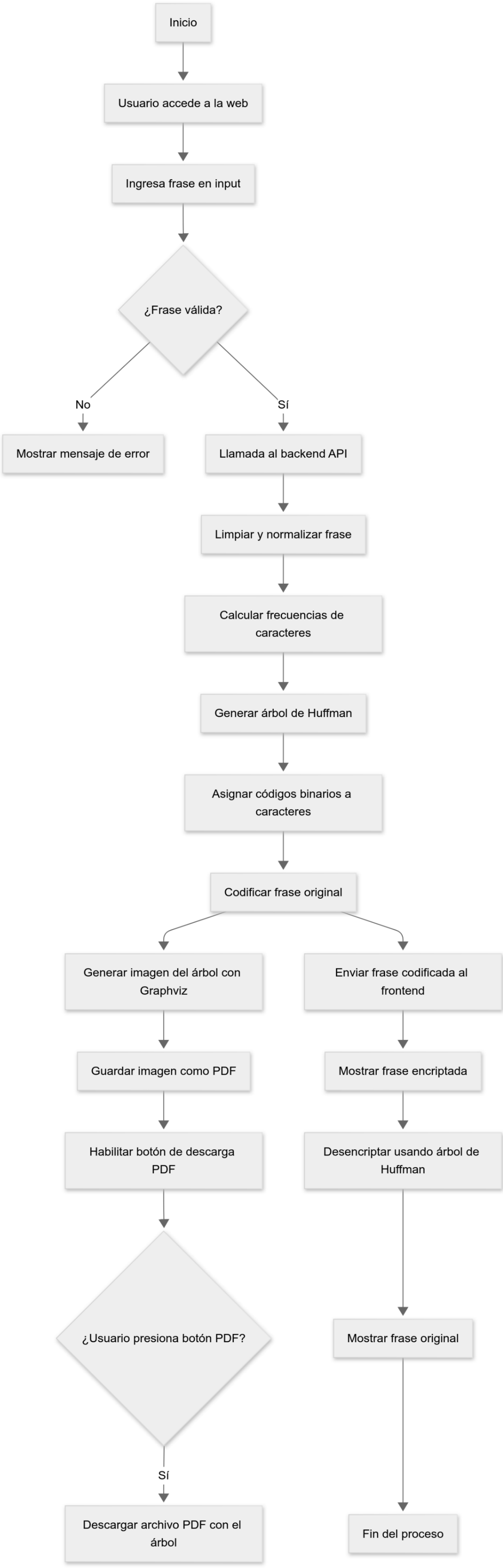
N --> R[Mostrar frase encriptada]

R --> S[Desencriptar usando árbol de Huffman]

S --> T[Mostrar frase original]

T --> U[Fin del proceso]

Diagrama de flujo



Implementación

1. Clase NodoArbolito

Define la estructura básica de un nodo en el árbol de Huffman, con su carácter (o None si es nodo interno), su peso (frecuencia) y punteros a hijos izquierdo/derecho.

```
1   class NodoArbolito:
2       def __init__(self, letra, peso):
3           self.letra = letra
4           self.peso = peso
5           self.LadoIzq = None
6           self.LadoDer = None
```

2. Fusión de nodos en hacerElArbolito

Ordena la lista por peso ascendente (burbuja), extrae los dos nodos de menor frecuencia, los combina en un nuevo nodo padre, y repite hasta quedar solo la raíz.

```
18      for vuelta in range(len(listaNodos)):
19          for indice in range(len(listaNodos)-1):
20              if listaNodos[indice].peso > listaNodos[indice+1].peso:
21                  temp = listaNodos[indice]
22                  listaNodos[indice] = listaNodos[indice+1]
23                  listaNodos[indice+1] = temp
24
25      nodoPequeño1 = listaNodos.pop(0)
26      nodoPequeño2 = listaNodos.pop(0)
27
28      nuevoContador = nodoPequeño1.peso + nodoPequeño2.peso
29      nodoNuevo = NodoArbolito(None, nuevoContador)
30      nodoNuevo.LadoIzq = nodoPequeño1
31      nodoNuevo.LadoDer = nodoPequeño2
32
33      listaNodos.append(nodoNuevo)
```

3. Recorrido en obtenerLosCodigos

Camina el árbol recursivamente, acumulando un string de bits: "0" para cada paso a la izquierda, "1" a la derecha; al llegar a hoja, asocia ese código a la letra.

```python
def explorarNodos(nodoActual, codigoTemporal):
    if nodoActual is None:
        return None;

    if nodoActual.letra != None:
        codigos[nodoActual.letra] = codigoTemporal
        return True;

    explorarNodos(nodoActual.LadoIzq, codigoTemporal + "0")
    explorarNodos(nodoActual.LadoDer, codigoTemporal + "1")
```

4. Conteo de frecuencias en contarLetras

Recorre cada carácter de la frase; si ya existe en listaLetras incrementa su contador, en caso contrario lo añade con cuenta 1. Devuelve una lista de tuplas (letra, frecuencia).

```python
    listaLetras = []
    listaCuentas = []

    for letra in frase:
        encontrado = False
        for i in range(len(listaLetras)):
            if listaLetras[i] == letra:
                listaCuentas[i] += 1
                encontrado = True
                break;
        if not encontrado:
            listaLetras.append(letra)
            listaCuentas.append(1);



    resultadoFinal = []
    for j in range(len(listaLetras)):
        resultadoFinal.append( (listaLetras[j], listaCuentas[j]) )



    return list(zip(listaLetras, listaCuentas))
```

5. Quicksort in-place

Usa el primer elemento como pivote, separa en sublistas menores y mayores, reescribe la sección correspondiente y aplica recursión en ambos lados.

```python
def quicksort(lista, inicio, final):
    if inicio >= final:
        return

    piv = lista[inicio][1]
    menores = []
    mayores = []

    for elemento in lista[inicio+1:final+1]:
        if elemento[1] <= piv:
            menores.append(elemento)
        else:
            mayores.append(elemento)

    nuevaLista = menores + [lista[inicio]] + mayores
    for k in range(len(nuevaLista)):
        lista[inicio + k] = nuevaLista[k]

    posicionPivote = inicio + len(menores)
    quicksort(lista, inicio, posicionPivote-1)
    quicksort(lista, posicionPivote+1, final)
```

6. Recorrido para graficar con Graphviz en dibujarArbolito

Utiliza una pila en lugar de recursión: por cada nodo crea un vértice con etiqueta (letra y peso o solo peso) y añade aristas etiquetadas "0"/"1" hacia los hijos izquierdo y derecho.

```
    while pila:
        nodoActual, nombre = pila.pop();

        if nodoActual is None:
            continue;

        if nodoActual.letra:
            etiqueta = f"{nodoActual.letra} ({nodoActual.peso})"

        else:
            etiqueta = f"{nodoActual.peso}"

        dibujo.node(name=nombre, label=etiqueta, shape='circle', style='filled', fillcolor='lightblue');

        if nodoActual.LadoIzq:
            nombreIzquierdo = nombre + "0";
            dibujo.edge(nombre, nombreIzquierdo, label='0');
            pila.append( (nodoActual.LadoIzq, nombreIzquierdo) );

        if nodoActual.LadoDer:
            nombreDerecho = nombre + "1";
            dibujo.edge(nombre, nombreDerecho, label='1');
            pila.append( (nodoActual.LadoDer, nombreDerecho) );


    return dibujo;
```

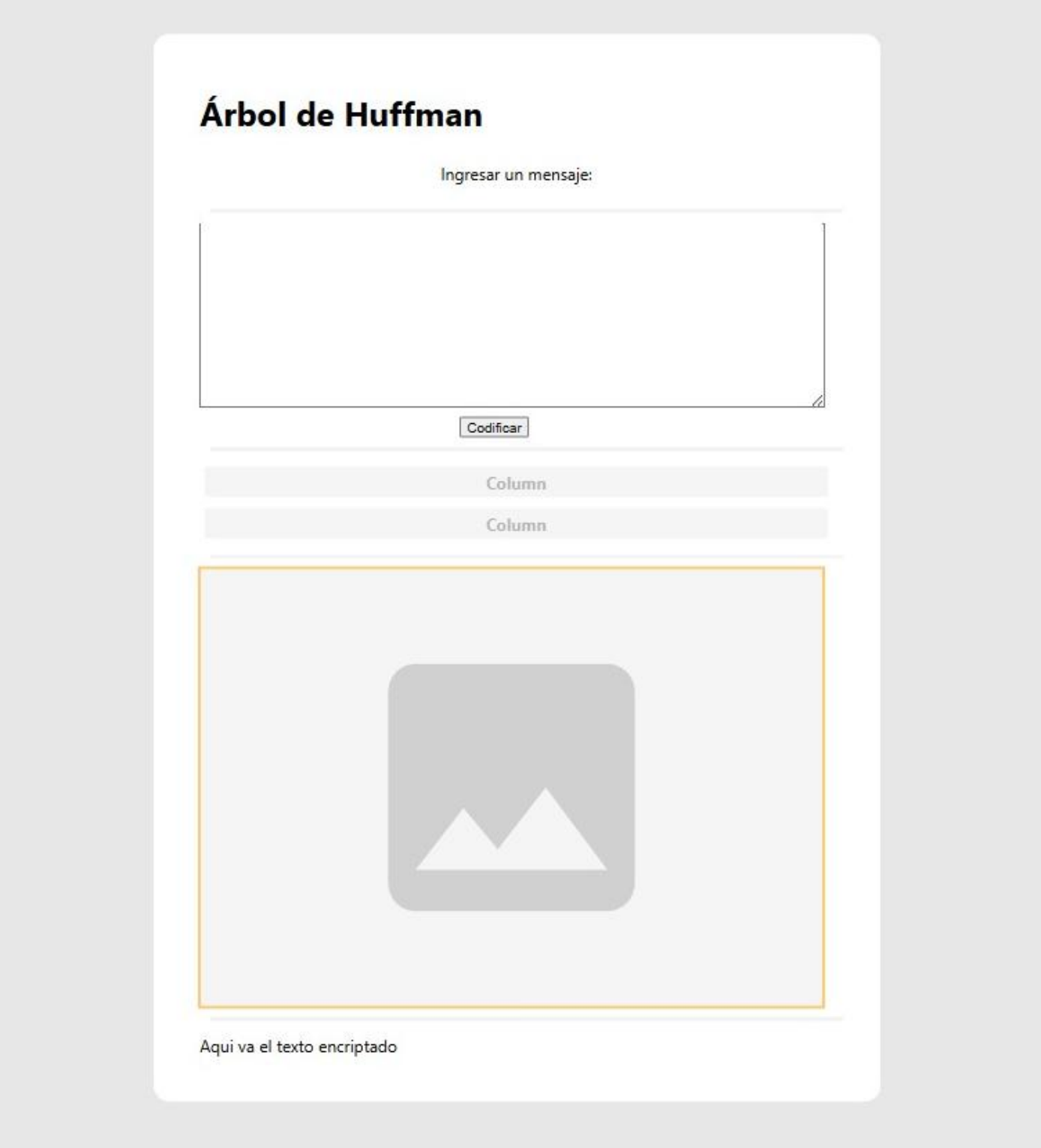
7. Endpoint /encode en Flask

Recibe un JSON con "palabra", valida su existencia, genera el árbol y los códigos, renderiza la imagen PNG con nombre único y devuelve un JSON con la palabra, el diccionario de códigos y la URL de la imagen.

```python
@app.route('/encode', methods=['POST'])
def procesarPalabra():
    try:

        datos = request.get_json()

        if 'palabra' not in datos:
            return jsonify({"error": "Falta el campo 'palabra'"}), 400

        palabra = datos['palabra']

        if not palabra:
            return jsonify({"error": "Debes enviar una palabra"}), 400


        arbol = getArbolito(palabra)
        codigos = obtenerLosCodigos(arbol)


        nombreArchivo = f"arbol_{uuid.uuid4().hex}"


        dibujarArbolito(arbol).render(rutaImagen, format='png', cleanup=True)

        return jsonify({
            "palabra": palabra,
            "codigos": codigos,
            "imagen": f"http://localhost:5000/imagen/{nombreArchivo}.png"
        })
```

Mocukps del Software

Primer Mocukp:



Conclusiones

- Se logró construir un algoritmo propio capaz de analizar una frase ingresada, calcular la frecuencia de sus caracteres y generar el árbol binario correspondiente. Este algoritmo permitió obtener códigos binarios únicos y sin ambigüedad para cada carácter, asegurando una codificación eficiente y sin pérdida de información.

- Se desarrolló una interfaz en React que facilita la interacción del usuario con la herramienta. La interfaz permite ingresar una frase, encriptarla con un solo clic y visualizar tanto el resultado en binario como el árbol de Huffman de forma clara e intuitiva, mejorando la experiencia de usuario.

- Se integró exitosamente Graphviz para generar la representación visual del árbol de Huffman, y se implementó la funcionalidad de exportación en formato PDF. Esto permitió documentar gráficamente el proceso de codificación y cumplir con los requisitos técnicos del proyecto.

Referencias

1. Sznajdleder, P. A. (s.f.). *Compresión de archivos mediante el algoritmo de Huffman*. En *Algoritmos a fondo*. Alfaomega.
2. Universidad Veracruzana. (2021). *Clase 8: Árboles*.
3. Universidad Autónoma Metropolitana. (2024). *Árboles binarios de búsqueda*.
4. Cuantum Technologies. (2024). *Algoritmos y estructuras de datos con Python: Una experiencia de aprendizaje interactiva*.
5. Altadill Izura, P. X. (2020). *Desarrollo web con React*. Marcombo.
6. Ediciones ENI. (2025). *React: Desarrolle el front-end de sus aplicaciones web y móviles con JavaScript*.
7. Gansner, E. R., Koutsofios, E., & North, S. C. (2015). *Drawing graphs with dot*.