

**Pontificia Universidad Javeriana**

Entrega #3 proyecto de Estructuras de Datos



Angel Eduardo Morales Abril  
Tomas Alejandro Silva Correal  
Juan Guillermo Pabón Vargas  
Gabriel Jaramillo Cuberos  
Salomon Avila

Estructuras de datos

Profesor: John Jairo Corredor Franco

Abril 24, 2025

# Índice

<b>Índice.....</b>	<b>2</b>
<b>Introducción.....</b>	<b>2</b>
<b>Acta de evaluación.....</b>	<b>3</b>
<b>Planteamiento de problemas.....</b>	<b>3</b>
<b>Propuesta de solución.....</b>	<b>4</b>
Componente 1: Proyección de imágenes.....	4
Componente 2: Codificación de imágenes.....	5
Componente 3: Componentes conectados en imágenes.....	5
<b>Tipos Abstractos de datos.....</b>	<b>6</b>
TAD Nodo.....	6
TAD Arbol:.....	7
TAD ManejadorCodificacion:.....	7
TAD Imagen:.....	8
TAD VolumenImágenes.....	9
TAD Grafo.....	9
<b>Diagrama de clases.....</b>	<b>11</b>
<b>Plan de pruebas.....</b>	<b>11</b>
<b>Conclusiones.....</b>	<b>14</b>
<b>Resumen.....</b>	<b>16</b>

## Introducción

Dentro del presente documento se evidenciará un reporte completo del desarrollo del proyecto de estructuras de datos, teniendo en cuenta el proceso de aprendizaje completado dentro de cada una de las entregas. Se demostrará cómo ha evolucionado el proyecto a lo largo del semestre para mejorar los comentarios hechos por el profesor mediante una acta de evaluación. Además, dentro de la entrega se completará el último requerimiento dado dentro del enunciado del proyecto.

Para el desarrollo de la tercera entrega se propone el siguiente problema: “Segmentar una imagen a partir de unas semillas dadas por el usuario, utilizando representaciones basadas en grafos”. Se propone como solución al problema la adaptación del sistema anterior para cumplir con el requerimiento, de esta manera, generando código y documentación de este. Dentro de la documentación se presentarán Tipos Abstractos de Datos para las estructuras de datos utilizadas dentro de la solución elegida y un plan de pruebas completo para cada funcionalidad nueva agregada al sistema.

# Acta de evaluación

Los comentarios de las anteriores entregas fueron en su mayoría relacionados con el desarrollo del documento descriptivo del proyecto, el cual debía detallar su funcionalidad. Teniendo esto en cuenta, desde el segundo componente en adelante se mejoró significativamente la organización del contenido y se ha documentado detalladamente cada cambio y dato relevante involucrado en la realización del proyecto.

Sin embargo, dentro de la segunda entrega del proyecto se recibieron comentarios en relación al desarrollo del plan de pruebas, teniendo esto en cuenta se decidió crear un plan de pruebas más específico y técnico enfocado a la funcionalidad del código en sí, conteniendo pruebas del funcionamiento del código desde el punto de vista del usuario.

## Planteamiento de problemas

El desarrollo de procesamiento de imágenes en escala de grises en formato PGM, con tres componentes (proyección de imágenes, codificación de imágenes y segmentación de imágenes), presenta una serie de desafíos técnicos y conceptuales que requieren un diseño robusto y eficiente de estructuras de datos acorde a su complejidad.

El primer componente, proyección de imágenes, enfrenta el reto de gestionar un volumen 3D compuesto por hasta 99 imágenes en formato PGM, lo que implica diseñar una estructura lineal para cargar y almacenar estas imágenes en memoria, validando la consistencia de dimensiones y la nomenclatura de archivos (`nombre_basexx.pgm`). Además, calcular proyecciones 2D en direcciones x, y o z según criterios como mínimo, máximo, promedio o mediana requiere iterar eficientemente sobre grandes volúmenes de datos, minimizando el costo computacional, y generar un archivo PGM válido, manejando errores como archivos inexistentes o formatos incorrectos. La restricción de trabajar con un único volumen por sesión añade la necesidad de gestionar la sobrescritura de datos sin comprometer la integridad.

El segundo componente, codificación de imágenes, plantea desafíos en la implementación del algoritmo de Huffman para comprimir y descomprimir imágenes. Esto incluye construir un árbol de Huffman óptimo que asigne códigos de longitud variable a hasta 256 intensidades de píxeles según sus frecuencias, lo que requiere un Tipo Abstracto de Datos (TAD) eficiente para el árbol

binario y el cálculo de frecuencias en imágenes de gran tamaño. La generación de un archivo binario con la estructura especificada (ancho, alto, intensidad máxima, frecuencias y bits comprimidos) y la posterior decodificación para reconstruir la imagen original demandan una gestión precisa de la escritura y lectura de datos, junto con la validación de formatos y el manejo de errores como datos corruptos o frecuencias inconsistentes.

Por su parte, el componente de segmentación de imágenes enfrenta retos en modelar una imagen como un grafo donde cada píxel es un nodo conectado a sus vecinos, con aristas cuyos costos son las diferencias de intensidad. Diseñar un TAD para el grafo que sea eficiente en memoria y tiempo, especialmente para imágenes grandes, es un desafío clave. Además, ejecutar el algoritmo de Dijkstra para cada una de hasta cinco semillas proporcionadas por el usuario, asignando etiquetas a los píxeles según la instancia que los alcance primero, requiere resolver conflictos de asignación y validar las semillas (coordenadas y etiquetas) para evitar errores como posiciones fuera de límites. La generación de la imagen segmentada en formato PGM y el manejo de errores, como la ausencia de una imagen cargada o semillas mal definidas, completan los desafíos de este componente.

## Propuesta de solución

El sistema de procesamiento de imágenes desarrollado aborda tres componentes fundamentales: proyección de imágenes, codificación de imágenes y segmentación de imágenes. Cada componente utiliza estructuras de datos específicas para cumplir con los requisitos establecidos, posteriormente integrándose en una consola interactiva. A continuación se presentan los detalles de las soluciones para cada componente.

### Componente 1: Proyección de imágenes

La implementación utiliza la clase “VolumenImágenes”, que gestiona un volumen 3D mediante un vector de Imágenes. El método “cargarDesdeArchivos” carga las imágenes desde archivos con nombres en el formato “nombre\_basexx.pgm”, validando la consistencia de dimensiones y el número de imágenes. El método “generarProyeccion2D”, implementado en VolumenImágenes.cpp, procesa el volumen según la dirección y criterio especificados:

- **Dirección x:** Colapsa las columnas, generando una imagen con “numeroTotalImágenes” filas y “dimX” columnas.
- **Dirección y:** Colapsa las filas, generando una imagen con “dimY” filas y “numeroTotalImágenes” columnas.

- **Dirección z:** Colapsa la profundidad, generando una imagen con “dimX” filas y “dimY” columnas.
- **Criterios:** Calcula el valor de cada píxel proyectado mediante promedio (media aritmética), mínimo, máximo o mediana (ordenando valores y seleccionando el central).

La proyección resultante se guarda en un archivo PGM con formato "P2" usando el método “guardarComoPGM”.

## Componente 2: Codificación de imágenes

La implementación se basa en las clases “ManejadorCodificacion”, “Arbol” y “Nodo”. El proceso de codificación, gestionado por el método codificar, incluye:

- Cálculo de frecuencias de intensidades de píxeles mediante “imagen.codificacion”.
- Construcción del árbol de Huffman con “arbol.crearArbol”, utilizando un vector de pares (intensidad, frecuencia).
- Generación de códigos binarios con “arbol.completarValores”, almacenados en un mapa (map<int, vector<bool>>).
- Escritura del archivo binario (.huf) con dimensiones, máximo de intensidad, frecuencias y la secuencia de bits comprimida, empaquetada en bytes.

Para la decodificación, el método “cargarDesdeHUF” lee el archivo, reconstruye el árbol Huffman a partir de sus frecuencias, y decodifica la secuencia de bits para generar la matriz de píxeles, que luego guarda como un archivo PGM.

## Componente 3: Componentes conectados en imágenes.

La clase “Grafo” implementa la segmentación. El método “segmentarImagen” realiza los siguientes pasos:

- **Inicialización:** Construye un grafo de adyacencia (vector<vector<pair<int,int>>>) donde los nodos son píxeles y las aristas tienen costos basados en la diferencia de intensidad entre píxeles vecinos, calculada en “inicializarGrafo”.
- **Dijkstra por Semilla:** Para cada semilla (coordenadas x, y y etiqueta), ejecuta el algoritmo de Dijkstra usando una cola de prioridad para propagar la etiqueta. La matriz de distancias (distancias) almacena la distancia más corta y la etiqueta asignada a cada píxel.
- **Segmentación:** Asigna a cada píxel la etiqueta de la semilla que lo alcanza primero, resolviendo conflictos mediante la distancia más corta.
- **Salida:** Genera una imagen PGM con formato "P2", donde cada píxel tiene la etiqueta correspondiente, usando “guardarComoPGM”.

Esta solución asegura una segmentación precisa y eficiente, adecuada para imágenes grandes, al optimizar el uso de la cola de prioridad para un tiempo de ejecución de  $O((V + E) \log V)$ , donde  $V$  es el número de píxeles y  $E$  el número de aristas.

En general, estas soluciones integran estructuras de datos lineales y no lineales, como árboles y grafos, para manejar eficientemente las operaciones de proyección, compresión y segmentación de imágenes, de esta manera cumpliendo con todos los objetivos propuestos dentro del proyecto.

## Tipos Abstractos de datos

Los tipos abstractos de datos se dividen en tres partes, cada una representando cada componente pero aun así, estos se relacionan entre sí, de manera que dentro de esta sección se describirán las estructuras de datos utilizadas a fondo, describiendo cuál fue el caso de uso de cada dato y cada operación. Al final se presentará un diagrama de clases el cual espera poder presentar de una manera más sencilla, visual y holística el sistema.

### TAD Nodo

Datos mínimos:

- frecuencia, entero, representa la frecuencia del símbolo o la suma de frecuencias de los nodos hijos.
- valor, entero, representa el símbolo asociado al nodo.
- esHoja, valor booleano, indica si el nodo es una hoja.
- nodoIzquierda, nodo, apuntador al nodo hijo izquierdo.
- nodoDerecha, nodo, apuntador al nodo hijo derecho.

Operaciones:

- `Nodo()`, constructor que inicializa un nodo vacío.
- `~Nodo()`, destructor que libera los recursos del nodo.
- `getFrecuencia()`, devuelve la frecuencia del nodo.
- `setFrecuencia(frec)`, establece la frecuencia del nodo.
- `getEsHoja()`: devuelve el estado de si el nodo es hoja.
- `setEsHoja(estado)`, establece si el nodo es hoja.
- `getValor()`, devuelve el valor asociado al nodo.
- `setValor(valor)`, establece el valor del nodo.
- `getNodeIzquierda()`, devuelve el puntero al nodo hijo izquierdo.
- `setNodeIzquierda(nodoIzq)`, establece el nodo hijo izquierdo.
- `getNodeDerecha()`, devuelve el puntero al nodo hijo derecho.

- setNodoDerecha(nodoDer), establece el nodo hijo derecho.

## TAD Arbol:

Datos minimos:

- codificacion, vector de valores booleanos, almacena la codificación actual de un recorrido o código de Huffman.
- raíz, apuntador al nodo raíz del árbol de Huffman.
- valor, mapa, asocia valores enteros (caracteres o símbolos) con sus respectivas codificaciones en forma de vectores de booleanos.

Operaciones:

- Arbol(): Constructor que inicializa un árbol vacío.
- getCodificacion(): Devuelve el vector de codificación actual.
- setCodificacion(bool): Establece el vector de codificación.
- getRaiz(): Devuelve el puntero al nodo raíz.
- setRaiz(Nodo): Establece el nodo raíz del árbol.
- getValores(): Devuelve el mapa de valores con sus codificaciones.
- setValores(valor): Establece el mapa de valores con sus codificaciones.
- eliminar(nodo): Elimina un nodo dado del árbol.
- crearArbol(frecuencias): Construye el árbol de Huffman a partir de un vector de pares (valor, frecuencia).
- completarValores(código, raíz): Genera las codificaciones de los valores recorriendo el árbol desde un nodo dado.

## TAD ManejadorCodificacion:

Datos minimos:

- Imagen, objeto de la clase Imagen, contiene la información de la imagen a procesar.
- Codificación, vector de valores booleanos, almacena la codificación resultante del proceso de Huffman.
- frecuencias, vector de pares, representa las frecuencias de los valores en la imagen.
- Árbol, objeto de la clase Árbol, representa el árbol de Huffman construido para la codificación.

Operaciones:

- Manejador(): Constructor que inicializa un manejador vacío.
- getImagen(): Devuelve el objeto imagen.
- setImagen(Imagen): Establece el objeto imagen.
- getCodificacion(): Devuelve el vector de codificación.

- setCodificacion(codificacion): Establece el vector de codificación.
- getFrecuencias(): Devuelve el vector de frecuencias.
- setFrecuencias(frec): Establece el vector de frecuencias.
- getArbol(): Devuelve el objeto árbol de Huffman.
- setArbol(arbol): Establece el objeto árbol de Huffman.
- codificar(): Realiza el proceso de codificación de la imagen utilizando el árbol de Huffman.
- cargarDesdePGM(ruta): Carga una imagen desde un archivo en formato PGM y prepara los datos para la codificación.
- cargarDesdeHUF(rutaHUF, rutaPGM): Carga una codificación desde un archivo en formato HUF para su procesamiento.
- numeroAbinario(): Convierte un número a su representación binaria

## TAD Imagen:

Datos minimos:

- vecImagen, vector de vectores de enteros, almacena los valores de los píxeles de la imagen en escala de grises.
- maxClaro, entero, representa el valor máximo de claridad (brillo) permitido en la imagen (generalmente en el rango 0-255).
- dimensionX, entero, indica el ancho de la imagen en píxeles.
- dimensionY, entero, indica el alto de la imagen en píxeles.
- formato, cadena de caracteres, especifica el formato de la imagen (por ejemplo, "P2" para PGM ASCII).
- nombre, cadena de caracteres, almacena el nombre del archivo de la imagen.

Operaciones:

- Imagen(): Constructor que inicializa una imagen con valores por defecto.
- getVecImagen(): retorna el vector de vectores de vecImagen.
- setVecImagen(vecImg): asigna a vecImagen un vector de vectores de enteros.
- getMaxClaro(): retorna el valor máximo de claridad (brillo) permitido en la imagen.
- setMaxClaro(maximo): asigna al valor de "maxClaro" un valor entero que entra de parámetro.
- getDimensionX(): retorna el entero de dimensionX.
- setDimensionX(dimension): asigna al valor de "dimensionX" un valor entero que entra de parámetro.
- getDimensionY(): retorna el entero de DimensionY.
- setDimensionY(dimension): asigna al valor de "dimensionY" un valor entero que entra de parámetro.
- getFormato(): retorna el string de Formato.



- setFormato(form): asigna al valor de “Formato” un string que entra de parámetro.
- getNombre(): retorna el string de Nombre.
- setNombre(nombre2): asigna al valor de “Nombre” un string que entra de parámetro.
- cargarDesdePGM(ruta): carga imagen .pmg usando una ruta de archivo.
- guardarComoPGM(ruta): guarda imagen como .pmg a una ruta específica.
- mostrarInfo(): muestra la información relacionada a la Imagen.

## TAD VolumenImágenes

Datos mínimos:

- cantidadImágenes: entero, define la cantidad de imágenes que se van a cargar
- imágenes: vector de imágenes, contiene las imágenes ordenadas según el orden de la imagen en el archivo
- nombreBase: cadena de caracteres, nombre base de la colección de imágenes

Operaciones:

- getCantidadImágenes(): retorna el número de cantidad de imágenes
- getImágenes(): retorna el vector de imágenes
- setCantidadImágenes(cantidad): asigna al valor de “cantidadImágenes” un valor entero que entra de parámetro.
- setImágenes(imagen): asigna al vector de vectores “Imágenes” un vector de mismo tipo que entra como parámetro de entrada.
- setNombre(nom\_base): asigna un nombre a la colección de imágenes.
- getNombre(): retorna el nombre de la colección de imágenes.
- cargarDesdeArchivos(base, numImágenes): Carga las imágenes desde la ruta e indica la cantidad de imágenes que se cargaron.
- mostrarInfo(): Muestra la información relacionada a la base y el número de imágenes cargadas.
- GenerarProyeccion2D(dirección, criterio, rutaFinal): Se genera una proyección 2D del conjunto de imágenes dado la dirección de vista y el criterio especificado.

## TAD Grafo

Datos mínimos:

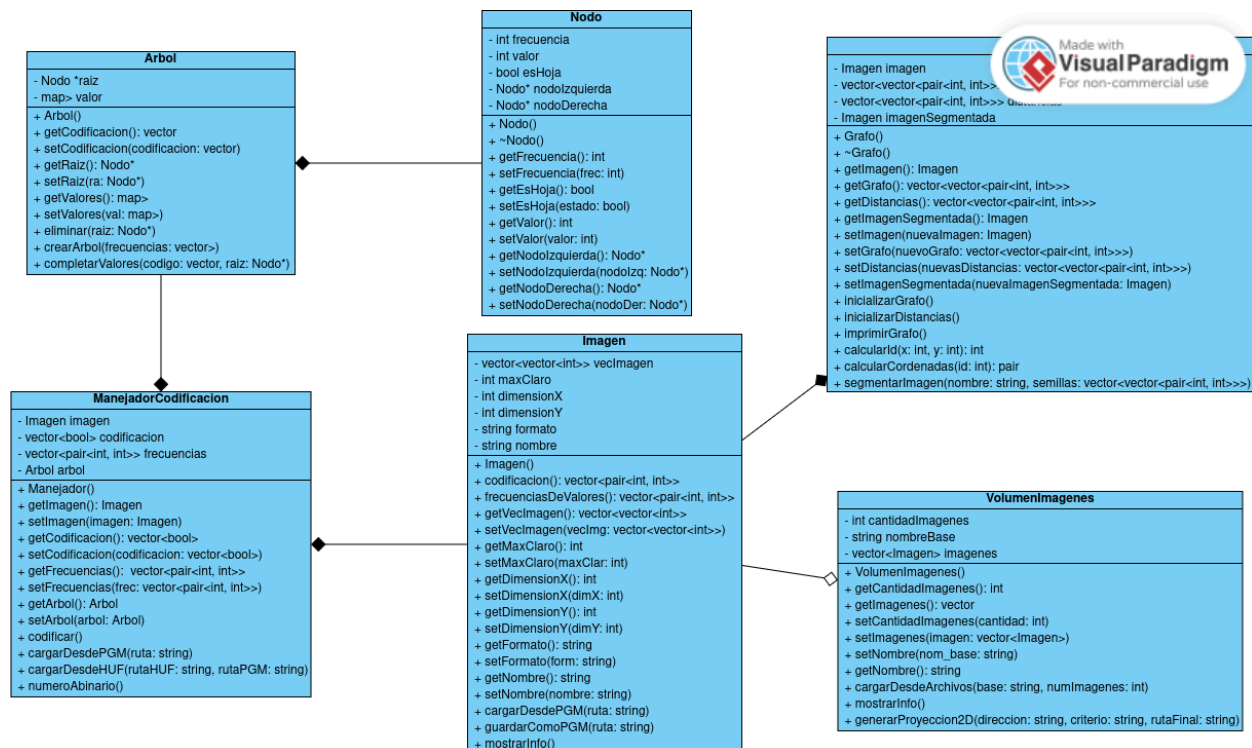
- imagen: Objeto de la clase Imagen, representa la imagen en escala de grises en formato PGM a segmentar.
- grafo: Vector de vectores de pares de enteros (vector<vector<pair<int,int>>>), representa la lista de adyacencia del grafo, donde cada par contiene el índice del nodo vecino y el costo de la arista (diferencia de intensidad entre píxeles).

- **distancias:** Vector de vectores de pares de enteros (`vector<vector<pair<int,int>>>`), almacena las distancias más cortas desde las semillas a cada píxel, junto con la etiqueta asignada durante el proceso de segmentación.
- **imagenSegmentada:** Objeto de la clase `Imagen`, representa la imagen resultante con las etiquetas asignadas tras el proceso de segmentación.

#### Operaciones:

- **Grafo():** Constructor que inicializa un grafo vacío con una imagen vacía, listas de adyacencia y distancias vacías, y una imagen segmentada vacía.
- **~Grafo():** Destructor que libera los recursos asociados al grafo, incluyendo las estructuras de datos dinámicas.
- **getImagen():** Devuelve el objeto `Imagen` que contiene la imagen original a segmentar.
- **setImagen(nuevaImagen):** Asigna un nuevo objeto `Imagen` al atributo `imagen`, actualizando la imagen a segmentar.
- **getGrafo():** Devuelve la lista de adyacencia (`vector<vector<pair<int,int>>>`) que representa el grafo.
- **setGrafo(nuevoGrafo):** Asigna una nueva lista de adyacencia al atributo `grafo`.
- **getDistancias():** Devuelve la matriz de distancias y etiquetas (`vector<vector<pair<int,int>>>`) generada durante la segmentación.
- **setDistancias(nuevasDistancias):** Asigna una nueva matriz de distancias y etiquetas al atributo `distancias`.
- **getImagenSegmentada():** Devuelve el objeto `Imagen` que contiene la imagen segmentada.
- **setImagenSegmentada(nuevaImagenSegmentada):** Asigna un nuevo objeto `Imagen` al atributo `imagenSegmentada`.
- **inicializarGrafo():** Inicializa la lista de adyacencia (`grafo`) según las dimensiones de la imagen, creando nodos para cada píxel y aristas a sus vecinos (arriba, abajo, izquierda, derecha) con costos basados en las diferencias de intensidad entre píxeles adyacentes.
- **inicializarDistancias():** Inicializa la matriz de distancias (`distancias`) con valores iniciales (distancias infinitas y etiquetas nulas) para todos los píxeles de la imagen, preparándose para el algoritmo de Dijkstra.
- **imprimirGrafo():** Muestra en consola la estructura del grafo (lista de adyacencia), útil para depuración y verificación de las conexiones entre píxeles.
- **calcularId(x, y):** Convierte las coordenadas (x, y) de un píxel en un índice único para identificar el nodo correspondiente en el grafo, basado en las dimensiones de la imagen.
- **calcularCordenadas(id):** Convierte un índice único de nodo en las coordenadas (x, y) del píxel correspondiente en la imagen.
- **segmentarImagen(nombre, semillas):** Ejecuta el proceso de segmentación de la imagen, utilizando el algoritmo de Dijkstra para propagar etiquetas desde un vector de semillas (definidas como tuplas `<int,int,int>` con coordenadas x, y y etiqueta). Genera la imagen segmentada y la guarda en un archivo PGM con el nombre especificado.

## Diagrama de clases




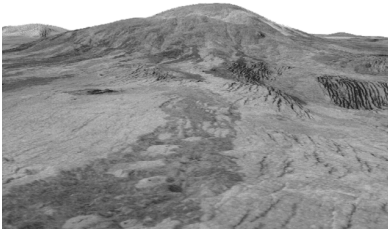
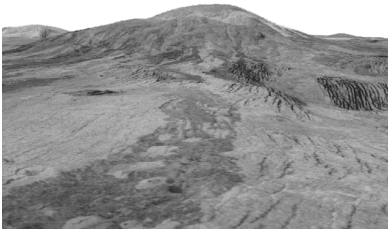
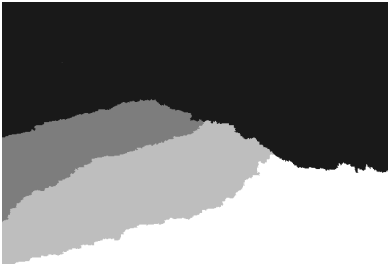
## Plan de pruebas

A continuación se presenta el plan de pruebas general como recopilado de las tres entregas, dividido entre funcionalidades.

Prueba	Resultado esperado	Resultado obtenido
Carga de imágenes desde archivos PMG		
cargar_imagen nombre_imagen.pgm	La imagen imagen.pgm ha sido cargada.	Se notifica la carga de la imagen en consola
cargar_imagen nombre_imagen.png (no existente)	La imagen imagen.pgm no ha podido ser cargada.	Se notifica fallo en carga de imágenes
cargar_imagen	Falta de argumentos.	Se notifica la falta de argumentos
cargar_imagen imagen_inexistente.pgm	La imagen imagen.pgm no ha podido ser cargada.	Se notificó por consola que el archivo no existe
cargar_imagen nombre_imagen.pgm (después de haber cargado otra imagen)	La imagen imagen.pgm ha sido cargada. (Sobreescribe la anterior)	La imagen fue cargada y reemplazó a la imagen anterior
Carga de volúmenes de imágenes desde PMG.		
cargar_volumen volumen 10	El volumen “volumen” ha sido cargado.	Se notifica la carga correcta del volumen
cargar_volumen volumen -5	La cantidad de imágenes no es válida.	Se notifica que la cantidad de imágenes es inválida
cargar_volumen volumen	Falta de argumentos.	Se notifica la falta de argumentos
cargar_volumen volumen 200	La cantidad de imágenes supera el límite permitido.	Se notifica que la cantidad de imágenes supera el límite establecido
cargar_volumen volumen 10 (con una imagen faltante)	El volumen “volumen” no ha podido ser cargado.	Indica por consola que el volumen no fue cargado
cargar_volumen volumen 10 (después de haber cargado otro volumen)	El volumen “volumen” ha sido cargado. (Sobreescribe el anterior)	Se sobreescribe el volumen correspondiente y se notifica sobre dicha carga
Almacenamiento y visualización de información de imágenes.		
Info_imagen (con imagen cargada)	Imagen cargada en memoria: imagen.pgm, ancho: W, alto:	Se indica que la imagen fue cargada y sus parámetros

	H.	correspondientes
Info_imagen (sin imagen cargada)	No hay una imagen cargada	Se indica la falta de imagen por cargar
Almacenamiento y visualización de información de volúmenes de imágenes.		
Info_volumen (con volumen cargado)	Volumen cargado en memoria: volumen, tamaño: n_im, ancho: W, alto: H.	Se indica carga de volumen con sus parámetros establecidos
Info_volumen (sin volumen cargado)	No hay un volumen cargado en memoria.	Se indica la falta de volumen por cargar
Proyección de imágenes 3D en un plano 2D.		
proyeccion2D x máximo salida.pgm (con volumen cargado)	La proyección 2D del volumen en memoria ha sido generada y almacenada en el archivo salida.pgm.	Se crea el archivo salida.pgm con la proyección correspondiente, se notifica al usuario dicha creación
proyeccion2D a máximo salida.pgm	El volumen aún no ha sido cargado en memoria.	Se indica la falta de volumen a cargar
proyeccion2D z promedio salida.pgm (sin volumen cargado)	Dirección no válida. Las opciones son: x, y, z.	Se indica que la dirección elegida fue inválida
proyeccion2D x suma salida.pgm	Criterio no válido. Las opciones son: mínimo, máximo, promedio, mediana.	Se indica que la operación no fue válida
proyeccion2D x máximo	Falta de argumentos.	Se indica falta de argumentos
Carga exitosa y decodificación de archivo HUF, junto a la creación de archivo pgm resultado.		
Cargar una imagen y decodificarla	La imagen fue cargada en memoria y se ha decodificado en un archivo .huf que se almacena en el disco	Se crea archivo .huf y se indica dicha creación al usuario
Prueba de decodificación (similitud entre imágenes)	La imagen resultado de la decodificación es igual a la imagen original que se	Imagen original:

	codificó	 <p>Imagen decodificada:</p> 
Prueba de decodificación (tamaño entre imágenes)	El peso de la imagen generada tras la decodificación es similar a la imagen original (al menos contiene poca variación)	<p>Imagen original: 665KB</p> <p>Imagen decodificada: 657KB</p>
Carga fallida de archivo huf, por diversos motivos		
Nombre de archivo erróneo o no existencia del archivo	Se envía una notificación sobre la falla en el nombre	Se indica el nombre erróneo del archivo
Formato erroneo	Se envía una notificación sobre la falla en el formato para codificación	Se indica al usuario que el formato no es correcto
Pruebas de uso y codificación del grafo		
Confirmar carga de imagen	Se notifica a través del panel de comandos la carga exitosa	Se indica al usuario la carga exitosa del archivo

	del archivo .pgm	
Fallo en carga de imagen	Se notifica al usuario la carga fallida y el motivo (imagen no encontrada)	Se indica falla en carga de imagen
Prueba con una sola semilla	Se produce una imagen llena del tono de dicha semilla (blanco en el caso de la prueba)	<p>Entrada:</p>  <p>Salida:</p> <p>(La imagen está completamente en blanco)</p>
Confirmar parecido de imagen con varias semillas	Se prueban 4 semillas, cada una con intensidad distinta, se espera recibir una imagen con algo de parecido a la imagen original, aunque reducida a dichas intensidades (considerando donde se coloca la semilla)	<p>Entrada:</p>  <p>Salida:</p> 

# Conclusiones

El desarrollo del sistema de procesamiento de imágenes en escala de grises en formato PGM, llevado a cabo como parte de curso de Estructuras de Datos, he presentado un ejercicio integral que ha permitido consolidar y aplicar conceptos fundamentales de estructuras de datos lineales y no lineales, así como técnicas de desarrollo avanzadas y modularidad. A lo largo de tres entregas, el proyecto ha evolucionado de la mano con el desarrollo de las clases, este grupo ha abordado cada problema y ofrecido soluciones para cada uno teniendo en cuenta comentarios hechos por el profesor, y cumpliendo con los objetivos establecidos en el enunciado.

El primer componente, proyección de imágenes, logró implementar con éxito la carga y procesamiento de volúmenes 3D compuestos por hasta 99 imágenes en formato PGM, generando proyecciones 2D que simulan el proceso de captura de radiografías. La utilización de una estructura lineal, implementada mediante la clase VolumenImagenes con un vector de objetos Imagen, permitió un manejo eficiente de grandes volúmenes de datos, validando dimensiones y nomenclaturas de archivo de manera robusta. La implementación de los criterios de proyección (mínimo, máximo, promedio, mediana) en las direcciones x, y, z demostró la capacidad de iterar eficientemente sobre estructuras tridimensionales, minimizando el costo computacional. Los desafíos iniciales relacionados con la gestión de memoria y la validación de archivos fueron superados mediante una planificación cuidadosa y pruebas exhaustivas, asegurando que el sistema manejara errores como archivos inexistentes o formatos incorrectos de manera adecuada. Este componente destacó la importancia de las estructuras lineales para organizar y procesar datos secuenciales, así como la necesidad de una interfaz clara para la interacción con el usuario.

El segundo componente, codificación de imágenes, implementó con éxito el algoritmo de Huffman para la compresión y descompresión de imágenes, optimizando el almacenamiento mediante códigos de longitud variable basados en las frecuencias de las intensidades de píxeles. La integración de los TADs Nodo, Arbol, y ManejadorCodificacion permitió una construcción modular del árbol de Huffman, facilitando tanto la generación de códigos binarios como la reconstrucción de la imagen original. Los retos relacionados con la escritura y lectura precisa de archivos binarios (.huf) y la gestión de hasta 256 intensidades posibles fueron abordados mediante una estructura de datos eficiente (mapas y vectores) y un manejo riguroso de errores, como frecuencias inconsistentes o datos corruptos. Este componente evidenció la relevancia de los árboles binarios en aplicaciones de compresión de datos y reforzó la importancia de una documentación detallada para garantizar la reproducibilidad y mantenibilidad del código.



El tercer componente, componentes conectados en imágenes, representó el desafío más complejo del proyecto al requerir la modelación de una imagen como un grafo y la aplicación del algoritmo de Dijkstra para segmentar regiones basadas en semillas proporcionadas por el usuario. La clase Grafo, con su lista de adyacencia y matriz de distancias, permitió una representación eficiente de los píxeles como nodos y las diferencias de intensidad como costos de aristas. La ejecución de Dijkstra para hasta cinco semillas, con resolución de conflictos basada en distancias mínimas, resultó en segmentaciones precisas, incluso para imágenes de gran tamaño. Los desafíos relacionados con la eficiencia computacional (tiempo  $O((V + E) \log V)$ ) y la validación de semillas fueron superados mediante el uso de colas de prioridad y una inicialización cuidadosa del grafo. Este componente destacó la potencia de las estructuras de datos no lineales, como los grafos, para resolver problemas complejos de procesamiento de imágenes, y subrayó la importancia de un diseño modular que facilite la integración con otros componentes.

En términos de aprendizaje, el proyecto consolida conocimientos clave sobre estructuras de datos, como vectores, árboles binarios, y grafos, y su aplicación práctica en problemas del mundo real, como el procesamiento de imágenes. También resaltó la importancia de la planificación, la validación de entradas, y el manejo de errores para garantizar la robustez del sistema. Los desafíos técnicos, como la optimización de algoritmos y la gestión de memoria, fueron oportunidades para aplicar conceptos teóricos en un contexto práctico, fortaleciendo las habilidades de programación en C++ y el uso de compiladores como gnu-g++.

## Resumen

El presente documento detalla la arquitectura y diseño de un sistema integral de procesamiento de imágenes en escala de grises en formato PGM. El sistema se divide en tres componentes fundamentales, cada uno enfocado a resolver aspectos distintos del procesamiento de imágenes, utilizando conceptos clave de estructuras de datos: proyección de imágenes, codificación de imágenes y segmentación de imágenes.

El primer componente, “Proyección de imágenes”, tiene como objetivo generar proyecciones 2D a partir de un volumen 3D compuesto por una serie ordenada de imágenes, teniendo como aplicación de la vida real la simulación de proceso de captura de radiografías. El segundo componente, “Codificación de imágenes”, presenta la compresión y descompresión de una imagen en formato PGM, mediante el algoritmo basado en árboles de huffman. El tercer componente, “Componentes conectados en imágenes”, se enfoca en segmentar una imagen en regiones etiquetadas a partir de semillas proporcionadas por el usuario, modelando la imagen

como un grafo y utilizando el algoritmo de Dijkstra para asignar etiquetas basadas en las distancias entre colores de píxeles.

Dentro de este documento se describe de forma detallada la planeacion e implementacion del codigo de cada componente, de la misma manera incluyendo los Tipos Abstractos de Datos para cada uno, Además se incluyen diagramas que ilustren el sistema de forma completa, de esta manera asegurando el cumplimiento de los requisitos de cada componente.