Documentación Ultima Entrega Proyecto

Pontificia Universidad Javeriana

Estructuras De Datos

Autores:

Santiago Hernandez

Juan Esteban Bello

Esteban Navas

John Corredor

27 de mayo 2025

Contenido

1. Acta de evaluación de entregas anteriores 3
2. Documento de diseño 4
2.1. Introducción 4
2.2. Descripción general del sistema 5
2.3. Arquitectura y módulos 7
2.4. Descripción de operaciones (comandos) 8
3. Definición de TADs 15
4. Compilación y ejecución 18
5. Plan de Pruebas (Comando Segmentar) 19
Conclusiones 25

1. Acta de evaluación de entregas anteriores

1.1. Entrega 1

Comentarios Textuales del Profesor:

1. “El plan de pruebas no es suficientemente claro ni específico. No se evidencia un análisis completo de funcionalidades ni la inclusión de pruebas de escritorio que muestren cómo se comporta el sistema.”
2. “Aunque el programa funciona correctamente, sería importante mejorar la interfaz de usuario. Desde el inicio del programa debería mostrarse un mensaje de bienvenida más claro. Asimismo, el comando de ayuda debería ser más intuitivo y explícito sobre cómo se deben utilizar los comandos.”
3. “Eviten el uso de términos en inglés como ‘exit’, ya que deben mantener coherencia idiomática en todo el sistema.”

Acciones realizadas para la corrección:

1. **Plan de pruebas**
    - Se reestructuró por completo, añadiendo casos de escritorio detallados con entradas y salidas esperadas que cubren escenarios normales, límite y de error.
    - Se incorporaron capturas de pantalla y comentarios explicativos para facilitar la replicación de cada prueba.
2. **Interfaz de usuario**
    - Al iniciar el sistema ahora aparece un mensaje de bienvenida claro y orientador.
    - El comando de “ayuda” fue rediseñado: muestra la lista completa de comandos, su sintaxis y ejemplos de uso.
    - Todos los mensajes se redactaron en español, eliminando términos en inglés como “exit” y reemplazándolos por “salir”.

1.2. Entrega 2

Comentarios Textuales del Profesor:
No se registraron observaciones formales tras la segunda entrega.

Revisión interna y puntos para reforzar:
Aunque no hubo retroalimentación directa, al revisar el documento de la segunda entrega hemos identificado áreas que se pueden pulir para la versión final:

- **Documentación de TADs:** enriquecer la se-cción del **TAD ÁrbolHuffman** con un diagrama esquemático del árbol y ejemplo de recorrido de bits.
- **Uniformidad de estilos:** unificar el formato de los bloques de código en toda la sección de compresión/descompresión (bash vs. cpp).
- **Plan de pruebas:** añadir casos de prueba para comandos codificar_imagen y decodificar_archivo con distintos tamaños de imagen y diferentes relaciones de compresión.
- **Manejo de errores:** documentar explícitamente los mensajes ante archivos .huf corruptos o rutas de archivo no existentes.
- **Ejemplos de uso:** incluir al menos un ejemplo real de comparación de tamaños pre- y post-compresión para ilustrar la efectividad de Huffman.

Estas mejoras se han incorporado en la presente tercera entrega para asegurar que la documentación sea completa, coherente y preparada para la sustentación.

2. Documento de diseño

2.1. Introducción

En este documento se presenta la tercera entrega del proyecto “Procesamiento y Segmentación de Imágenes PGM”, cuyo propósito es consolidar y completar las funcionalidades desarrolladas en las dos entregas previas, añadiendo un módulo de segmentación basado en etiquetas y un algoritmo de Dijkstra multi-fuente. El sistema está implementado en C++ y se ejecuta desde consola, garantizando portabilidad y eficiencia con el compilador GNU g++ (versión mínima 4.0.0).

Objetivo general

Diseñar, implementar y documentar un sistema modular de procesamiento de imágenes en formato PGM que permita:

1. Cargar y visualizar información de imágenes y volúmenes (Entrega 1).

2. Realizar operaciones de proyección 2D y compresión/descompresión mediante Huffman (Entrega 2).
3. Segmentar componentes conectados a partir de semillas de usuario, utilizando una construcción de grafo implícito y un algoritmo de Dijkstra multi-fuente (Entrega 3).

Objetivos específicos de la tercera entrega

1. Integrar y corregir todas las funcionalidades pendientes de las entregas anteriores, de acuerdo con los comentarios recibidos en cada evaluación.
2. Implementar el comando segmentar, que:
    - Valida parámetros y coordina la lectura de semillas.
    - Construye un grafo implícito donde cada píxel es un nodo.
    - Aplica Dijkstra multi-fuente para propagar etiquetas de región.
    - Genera un archivo PGM de salida con la segmentación.
3. Documentar exhaustivamente la arquitectura, los TADs utilizados, los diagramas de flujo y UML, y las operaciones de cada comando.
4. Elaborar un plan de pruebas sistemático para el comando segmentar, cubriendo casos de uso normales, límites y erróneos.

2.2. Descripción general del sistema

En esta sección presentamos una visión global de cómo interactúan los distintos módulos del proyecto y cuál es el flujo de trabajo desde que el usuario invoca un comando hasta que se genera el archivo de salida. A continuación:

1. **Resumen del flujo de trabajo**
    1. El usuario ejecuta el programa desde línea de comandos, indicando el subcomando deseado (cargar_imagen, proyeccion2D, codificar_imagen, segmentar, etc.) junto con sus parámetros.
    2. El **parser de comandos** valida la sintaxis y los rangos de los argumentos, mostrando un mensaje de ayuda si algo falta o es incorrecto.
    3. Según el subcomando, se invoca el **módulo correspondiente**:

- **Carga / Consulta**: lee un archivo PGM y/o muestra metadatos de imagen o volumen.
- **Proyección 2D**: genera vistas bidimensionales a partir de un volumen 3D.
- **Compresión Huffman**: codifica o decodifica datos siguiendo el algoritmo de Huffman.
- **Segmentación**: construye un grafo implícito sobre los píxeles de la imagen y aplica Dijkstra multi-fuente para propagar etiquetas de región.

4. Cada módulo procesa internamente su TAD (Imagen, Volumen, ÁrbolHuffman, Grafo, etc.) y produce un resultado en memoria.
5. Finalmente, el sistema **escribe el archivo de salida** (PGM segmentado, proyección, o flujo de bits codificados) y muestra un mensaje de confirmación o error por consola.

2. **Diagrama de bloques (esquema conceptual)**
   A nivel conceptual, la arquitectura puede resumirse en cinco bloques principales:

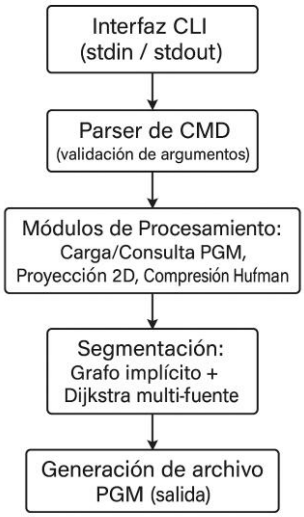

Imagen 1.

2.3. Arquitectura y módulos

En esta sección describimos cómo está organizado el proyecto en tres componentes principales, y presentamos un diagrama UML de paquetes que ilustra sus dependencias y clases más relevantes.

2.3.1 Lista de componentes

1. **Componente 1: Carga e información**
    - **Responsabilidad**: leer archivos PGM, almacenar en memoria la imagen o volumen, y mostrar metadatos.
    - **Clases / TADs clave**:
        - Imagen
        - Volumen
    - **Comandos**:
        - cargar_imagen / info_imagen
        - cargar_volumen / info_volumen
2. **Componente 2: Proyección y Huffman**
    - **Responsabilidad**:
        - Generar proyecciones 2D desde un volumen 3D.
        - Comprimir y descomprimir datos con Huffman.
    - **Clases / TADs clave**:
        - Proyeccion
        - NodoHuffman / ArbolHuffman
    - **Comandos**:
        - proyeccion2D
        - codificar_imagen / decodificar_archivo
3. **Componente 3: Segmentación**
    - **Responsabilidad**: segmentar la imagen PGM en regiones conectadas según semillas de usuario, usando un grafo implícito y Dijkstra multi-fuente.
    - **Clases / TADs clave**:

- Grafo
- State (tupla para priority_queue)

- **Comando**:
  - Segmentar

2.3.2 Diagrama UML de paquetes

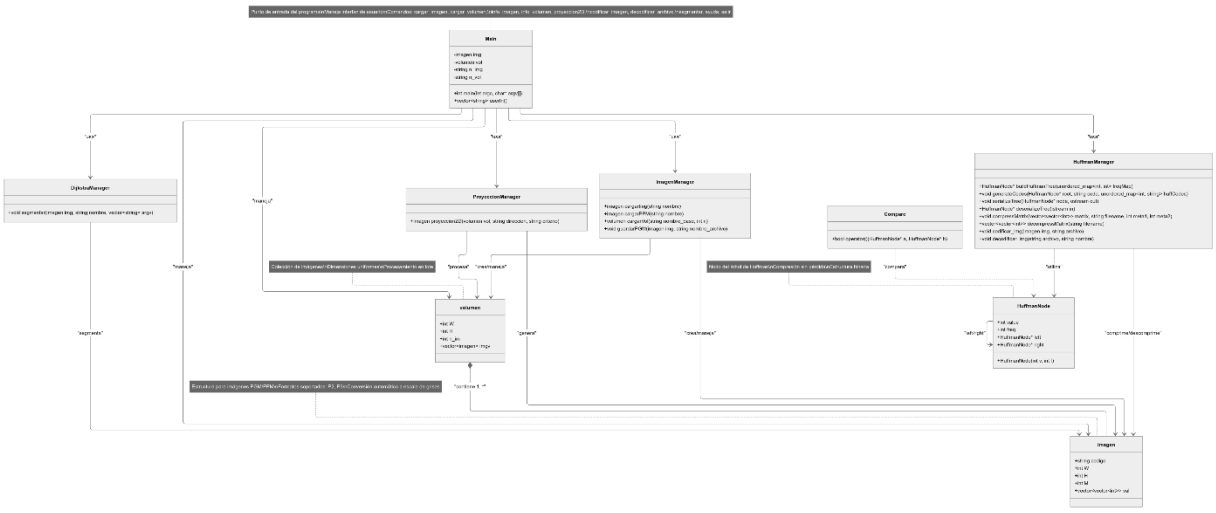

Imagen 2. Diagrama General UML

Descripción del diagrama

- El paquete Proyecto PGM agrupa todos los elementos.
- Main y ParserCMD están fuera de los subpaquetes y orquestan la selección de comando.
- Cada paquete de componente contiene sus clases/TADs y métodos principales.
- Las flechas muestran la relación “usa/depende de” entre el parser y cada comando, y entre los comandos y sus estructuras de datos.

2.4. Descripción de operaciones (comandos)

Los comandos que implementamos en este programa son indispensables para un correcto funcionamiento de este mismo, así, se muestran a continuación los comandos disponibles, su sintaxis, sus entradas y salidas, y el flujo interno de cada uno.

Cada comando sigue la estructura:

- **Descripción**: qué hace y para qué sirve.
- **Sintaxis**: forma de invocarlo en la consola.
- **Entradas**: parámetros y formatos esperados.
- **Salidas**: resultados (archivos generados o mensajes).
- **Precondiciones**: condiciones que deben cumplirse antes de la llamada.
- **Postcondiciones**: estado del sistema tras la ejecución.
- **Pseudocódigo** (o resumen de pasos).

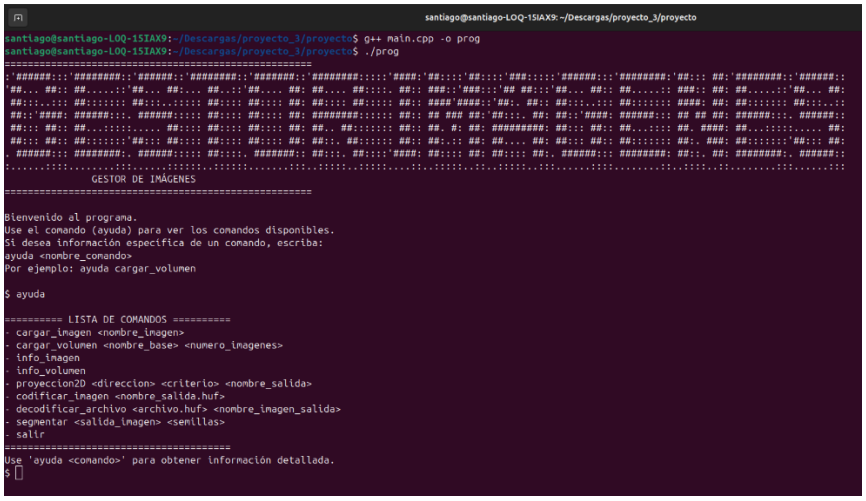

Imagen 3. Gestor de imágenes

2.4.1 cargar_imagen

- Descripción
  Lee un archivo PGM (formato P2 o P5) y almacena su contenido en un objeto Imagen, validando cabecera y valores de píxel.
- Sintaxis
  cargar_imagen <ruta_imagen.pgm>
- Entradas
  - <ruta_imagen.pgm>: nombre o ruta relativa del archivo PGM a cargar.
- Salidas
  - En memoria: objeto Imagen con la matriz de intensidades.
  - Por consola:
    - Mensaje de éxito:

  Se cargó correctamente <ruta_imagen.pgm>

        - En caso de error: descripción breve y salida con código distinto de cero.
- Precondiciones
    - El archivo existe y es accesible.
    - Tiene formato PGM válido.
- Postcondiciones
    - Imagen contiene ancho, alto y matriz de valores (0–255).
    - hayImagenCargada = true.
- Pseudocódigo

  1. Abrir archivo en modo lectura binaria/texto.
  2. Leer “magic number” (P2/P5), ancho, alto y valor máximo.
  3. Reservar matriz de tamaño alto × ancho.
  4. Leer cada valor de píxel en la matriz.
  5. Cerrar archivo y mostrar mensaje.

2.4.2 cargar_volumen

- **Descripción**

  Carga una secuencia de imágenes PGM numeradas como un volumen 3D en el TAD Volumen.
- **Sintaxis**

  cargar_volumen <nombre_base> <número_de_imágenes>
- **Entradas**
    - <nombre_base>: prefijo común de los archivos (p. ej. “slice_”).
    - <número_de_imágenes>: cantidad de capas que forman el volumen.
- **Salidas**
    - En memoria: objeto Volumen con número_de_imágenes imágenes cargadas.
    - Por consola: mensajes de éxito o error por cada capa.
- **Precondiciones**
    - Todas las imágenes existen con nombres <nombre_base>0.pgm, <nombre_base>1.pgm, …
    - Formato PGM válido en cada una.
- **Postcondiciones**
    - Volumen contiene lista de objetos Imagen.
    - hayVolumenCargado = true.
- **Pseudocódigo**

```
para i en 0 ... número_de_imágenes−1:
    ruta ← nombre_base + i + ".pgm"
    si !cargar_imagen(ruta):
        informar error y abortar
```

añadir imagen al Volumen
mostrar "Volumen cargado correctamente"

2.4.3 info_imagen / info_volumen

- **Descripción**
  Muestra metadatos de la imagen cargada o del volumen (dimensiones, número de capas, valor máximo).
- **Sintaxis**

  info_imagen

  info_volumen
- **Entradas**
  Ninguna.
- **Salidas**
  Tablas o líneas con:
  - Imagen: ancho × alto, valor máximo.
  - Volumen: cantidad de imágenes, dimensiones de cada capa.
- **Precondiciones**
  - Haber ejecutado cargar_imagen o cargar_volumen.
- **Postcondiciones**
  No modifican el estado en memoria.
- **Pseudocódigo**

  si hayImagenCargada:

  imprimir "Ancho: ... Alto: ... Max: ..."

  sino si hayVolumenCargado:

  imprimir "Capas: ... Ancho: ... Alto: ..."

  sino:

  imprimir "No hay datos cargados"

2.4.4 proyeccion2D

- **Descripción**
  Genera una imagen 2D a partir de un volumen 3D, aplicando un criterio (mínimo, máximo o promedio) sobre cada columna de píxeles.
- **Sintaxis**

  proyeccion2D <criterio> <archivo_salida.pgm>
- **Entradas**
  - <criterio>: min, max o avg.
  - <archivo_salida.pgm>: nombre del PGM de salida.
- **Salidas**
  - Archivo PGM con la proyección resultante.
  - Mensaje en consola de éxito o error.
- **Precondiciones**
  - hayVolumenCargado = true.
- **Postcondiciones**
  - Se crea y guarda <archivo_salida.pgm>.
- **Pseudocódigo**

  para cada columna (x, y):

  extraer todos los valores en z

  según criterio:

  elegir mínimo / máximo / calcular promedio

  asignar valor al píxel (x, y) de la imagen resultado

  guardar resultado en archivo_salida

2.4.5 codificar_imagen / decodificar_archivo

- **Descripción**
  - codificar_imagen: comprime la imagen cargada usando Huffman y genera un archivo .huf.
  - decodificar_archivo: reconstruye el PGM original desde un .huf.
- **Sintaxis**

codificar_imagen <archivo_salida.huf>

decodificar_archivo <archivo_entrada.huf> <imagen_salida.pgm>

- **Entradas**
  - codificar_imagen: nombre destino .huf.
  - decodificar_archivo: nombre origen .huf y destino .pgm.
- **Salidas**
  - Archivo de bits codificados (.huf).
  - Archivo PGM reconstruido.
  - Mensajes de avance y confirmación.
- **Precondiciones**
  - Para codificar_imagen: hayImagenCargada = true.
  - Para decodificar_archivo: el .huf existe y es válido.
- **Postcondiciones**
  - Se genera el archivo correspondiente en disco.
- **Pseudocódigo**
  Codificar

1. Contar frecuencias de cada valor.
2. Construir árbol de Huffman.
3. Generar tabla de códigos y escribir cabecera + datos.
4. Mostrar “Codificación completada”.

Decodificar

5. Leer árbol y cabecera.
6. Leer bits y traducir con el árbol.
7. Reconstruir matriz y escribir PGM.
8. Mostrar “Decodificación completada”.

2.4.6 segmentar

- **Descripción**
  Divide la imagen PGM en regiones conectadas, propagando etiquetas desde semillas dadas mediante un Dijkstra multi-fuente en un grafo implícito de píxeles.
- **Sintaxis**

segmentar <archivo_salida.pgm> sx1 sy1 sl1 [sx2 sy2 sl2 ...]

- **Entradas**
  - <archivo_salida.pgm>: nombre de la imagen segmentada.
  - Una o más triples (sx, sy, etiqueta): coordenada de semilla y etiqueta [1–255].
  - Mínimo 1 semilla, máximo 5.
- **Salidas**
  - Archivo PGM donde cada píxel vale la etiqueta de su región.
  - Mensaje de “Segmentación completada” o error.
- **Precondiciones**
  - hayImagenCargada = true.
  - Cada (sx, sy) dentro de la imagen y 1 ≤ etiqueta ≤ 255.
- **Postcondiciones**
  - Se guarda <archivo_salida.pgm> con la matriz de etiquetas.
- **Pseudocódigo**

1. Inicializar dist[][] = INF, label[][] = 0.
2. Para cada semilla (sx,sy,lab):
   - *dist[sy][sx]=0, label[sy][sx]=lab, encolar (0,sx,sy,lab).*
3. Mientras la cola no esté vacía:
   - Extraer (d,x,y,lab) de menor d.x|
   - Para cada vecino (nx,ny):

w = abs(I[y][x] - I[ny][nx])

if d + w < dist[ny][nx]:

dist[ny][nx] = d + w

label[ny][nx] = lab

encolar(dist[ny][nx], nx, ny, lab)

4. Escribir label[][] en <archivo_salida.pgm>.

5. Mostrar “Segmentación completada. Archivo guardado en <archivo_salida.pgm>”.

3. Definición de TADs

TAD: Imagen

Datos mínimos:

- codigo: cadena de caracteres
  Indica el formato del archivo de imagen, por ejemplo, “P2” para imágenes en escala de grises.
- ancho: número entero
  Número de columnas que tiene la imagen.
- alto: número entero
  Número de filas que tiene la imagen.
- maximo: número entero
  Valor máximo posible de intensidad de píxel en la imagen.
- pixeles: matriz de números enteros
  Representa los valores de intensidad en cada píxel de la imagen.

Operaciones:

- obtenerAncho(): devuelve la cantidad de columnas.
- obtenerAlto(): devuelve la cantidad de filas.
- obtenerPixel(): devuelve el valor de un píxel en una posición específica.
- fijarPixel(): modifica el valor de un píxel en una posición específica.
- cargarDesdeArchivo(): carga la imagen desde un archivo PGM o PPM.

- guardarEnArchivo(): guarda la imagen actual en un archivo PGM.

TAD: SegmentadorDijkstra

Datos mínimos:

- imagenOriginal: referencia a un TAD Imagen
  Imagen sobre la cual se realiza el proceso de segmentación.
- x: número entero
  Coordenada horizontal desde la cual comienza la propagación.
- y: número entero
  Coordenada vertical desde la cual comienza la propagación.
- etiqueta: número entero
  Valor entero que se asigna a todos los píxeles alcanzados desde esta semilla.
- distancias: matriz de números decimales
  Guarda la distancia mínima acumulada desde alguna semilla hasta cada píxel de la imagen.
- etiquetas: matriz de números enteros
  Contiene el resultado de la segmentación: la etiqueta asignada a cada píxel.

Operaciones:

- obtenerX(): devuelve la coordenada x.
- obtenerY(): devuelve la coordenada y.
- esValidaRespectoA(imagen): indica si la semilla se encuentra dentro de los límites de una imagen.
- agregarSemilla(): añade una nueva semilla de segmentación.
- inicializar(): prepara las matrices de distancias y etiquetas antes de ejecutar el algoritmo.
- ejecutar(): aplica el algoritmo de Dijkstra desde todas las semillas para propagar etiquetas.
- obtenerEtiqueta(): devuelve la etiqueta actual de un píxel.
- generarImagenSegmentada(): construye una imagen a partir de la matriz de etiquetas.

- guardarResultado(): guarda la imagen segmentada en un archivo.

TAD: ImagenSegmentada

Datos mínimos:

- ancho: número entero
  Número de columnas que tiene la imagen resultante.
- alto: número entero
  Número de filas que tiene la imagen resultante.
- etiquetas: matriz de números enteros
  Representa los valores de etiqueta para cada píxel en la imagen segmentada.

Operaciones:

- obtenerEtiqueta(): devuelve la etiqueta de un píxel específico.
- fijarEtiqueta(): modifica la etiqueta de un píxel.
- convertirAImagen(): transforma la matriz de etiquetas en una estructura compatible con el TAD Imagen.
- guardarComoPGM(): guarda la imagen generada en formato PGM.

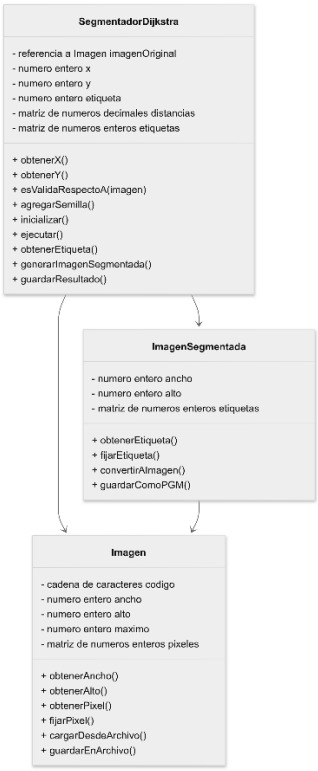

Imagen 4. Diagrama de relación TAD's

4. Compilación y ejecución

4.1. Requisitos

Compilador: g++ (versión 4.0.0 o superior)

Sistema operativo: Linux, Windows o MacOS (consola compatible)

4.2. Instrucciones de compilación

Pasos realizados:

- Compilacion de los archivos imagen.h, imagen.hxx, proyeccion.h, proyeccion.hxx, huffman.h, huffman.hxx, dijkstra.h, dijkstra.hxx main.cpp

- Comando utilizado para la compilación con información de depuración: G+ -std-c++11 -o prog imagen.h imagen.hxx proyeccion.h proyeccion.hxx huffman.h huffman.hxx dijkstra.h dijkstra.hxx main.cpp

4.3. Instrucciones de ejecución

Pasos realizados:

- Ejecucion del programa mediante ./prog en la consola compatible a utilizar.

- Cargar la imagen en memoria a segmentar mediante el comando cargar_imagen y el nombre de la imagen que se utilizara

- Comando ejemplo: cargar_imagen entrada_imagen.pgm

- Segmentar la imagen en memoria especificando el nombre de la imagen resultante y las semillas a utilizar para la segmentacion con la estructura x,y, 1-255 donde x , y son las cordenadas de un pixel dentro de la imagen y 1-255 son los valores que puede tomar

- Comando ejemplo: segmentar salida_imagen.pgm sx1 sy1 sl1 sx2 sy2 sl2 ...

4.4 Resultado esperado:

- Se espera que mediante esta secuencia de comandos la imagen cargada en memoria sea segmentada siguiendo las semillas dadas por el usuario y guardada en una imagen con nombre dado por el usuario, también al momento de finalizar la segmentacion de la imagen la consola imprima el mensaje de: “Segmentación completada. Archivo guardado en salida_imagen.pgm”

4.5. Manejo de errores y mensajes de ayuda

- Para la implementacion de el comando segmentar se necesita un comando de ayuda por si el usuario requiere conocer la sintaxis o para que sirve el comando segmentar.

- Se necesita un mensaje de error “No hay una imagen cargada en memoria” en caso de que en el sistema no se haya cargado una imagen.

- Se necesita un mensaje de error “Semilla inválida. (x,y) etiqueta (sl)” en caso de que una de las semillas dadas no será válida, especificando los valores de la semilla.

5. Plan de Pruebas (Comando Segmentar)

5.1. Objetivo

- Realizar un plan de pruebas para el comando segmentar del componente tres del proyecto, con el objetivo de verificar su correcta funcionalidad y corregir errores que no se hayan identificado al momento de implementarlo.

5.2. Criterios y metodología

Para realizar el plan de pruebas se tuvieron los siguientes criterios:

- Los mensajes de éxito y errores especificados en el enunciado del proyecto a desarrollar en el componente 3.

- La descripción de cuando deberian ejecturse el mensaje de éxito y los mensajes de error del comando segmentar de acuerdo con el enunciado del componente 3.

Para realizar el plan de pruebas se tuvo en cuenta la siguiente metodología:

- Garantizar la funcionalidad completa de los comandos implementados en la aplicacion, teniendo en cuenta el mensaje de exito asociado a cada uno y los mensajes de error necesarios para atrapar errores al momento de ejecturar algun comando con el objetivo de que el programa no muera directamente, si no que se pueda atrapar el error antes de que mate el programa y pueda seguir utilizándose, advirtiendo del error que se cometió.

5.3. Casos de prueba

Caso de Prueba	Comando Ejecutado	Descripcion	Resultado Obtenido	Paso a Paso
CP01	Segmentar salida.pgm sx1 sy1 sl1 ... sx5 sy5 sl5	Segmenta la imagen cargada en memoria basándose en las semillas proporcionadas y se almacena en un archivo salida.pgm	La imagen se segmenta correctamente y se muestra el mensaje de exito: “Segmentacion completada. Archivo guardado en salida.pgm”	1. Ejecutar cargar_imagen entrada.pgm 2. Ejecutar Segmentar salida.pgm con las semillas que desee 3. Verificar la creacion de salida.pgm en el directorio.
CP02	Segmentar salida.pgm sx1 sy1 sl1	Trata de segmentar una imagen cargada en memoria sin haber cargado una imagen en memoria previamente mostrando el mensaje de error sin matar al programa.	Se muestra el mensaje de error: “Error: No hay una imagen en memoria.” y el programa no muere.	1. Ejecutar Segmentar salida.pgm con las semillas que desee
CP03	Segmentar salida.pgm sx1 sy1 sl1	Trata de segmentar la imagen cargada en memoria basándose en las semillas, pero alguna de las semillas es invalida.	Se muestra el mensaje de error: “Semilla invalida (x,y) etiqueta “valor de etiqueta” donde la semilla invalida es una que se salia fuera de los valores especificados.	1. Ejecutar cargar_imagen entrada.pgm 2. Ejecutar Segmentar salida.pgm con las semillas que desee

Tabla 1: Casos de Prueba del comando segmentar

5.4. Prueba de escritorio

- Primera prueba (CP01): Caso estándar de uso de el comando segmentar, cargando una imagen en el sistema con el Comando cargar_imagen y segmentándola con el máximo número de semillas posibles (5) mediante el comando Segmentar, guardandolo en un archivo llamada salida02.pgm.

- Resultado esperado: Se muestra el mensaje de éxito “Segmentacion completada. Archivo guardado en salida.pgm” y el archivo resultante se crea en la carpeta donde se ejecuta el programa.

Imagen 5: Prueba de escritorio Caso de Prueba 1.

- Segunda prueba (CP02): Tratar de utilizar el comando segmentar sin haber cargado una imagen en el sistema.

- Resultado esperado: Se muestra el mensaje de error: “Error: No hay una imagen en memoria.” y el programa sigue ejecutandose.

Imagen 6: Prueba de escritorio Caso de Prueba 2.

- Tercera prueba (CP03): Utilizar el comando segmentar como en el caso estandar pero dandole unas semillas no validas.

- Resultado esperado: Se muestra el mensaje de error: “Semilla invalida (x,y) etiqueta “valor de etiqueta”.

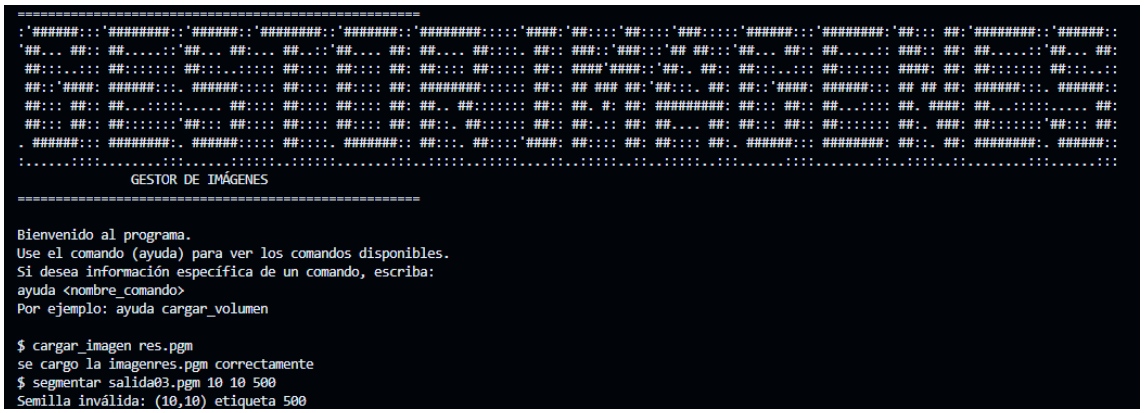

Imagen 7: Prueba de escritorio Caso de Prueba 3.

5.5. Análisis de resultados

Basándonos en los casos de uso y las pruebas de escritorio realizadas a estos pudimos analizar lo siguiente:

- Si se trataba de segmentar sin haber cargado una imagen en memoria anteriormente el comando intentaría ejecutarse sin establecer los valores de los vectores necesarios provocando un error en el sistema que mataba el programa, gracias a esto se corrigieron errores y se implementó el mensaje de error pertinente.

- Los tres casos de prueba planteados al momento de llevarlos a la prueba de escritorio siguiendo los pasos a seguir definidos para cado uno nos dio como resultado el resultado esperado ya definido para cada caso de prueba.

- El programa cumple las funciones ya implementadas de los componentes 1 y 2 junto a las implementadas en el componente 3 sin ningún error.

- Se verificó el correcto funcionamiento del comando Segmentar y los mensajes de error implementados para atrapar errores antes de que afecten la ejecución del programa, cumpliendo así el objetivo de este plan de pruebas.

- Se verifico que la imagen resultante del comando si fuera segmentada:

- Para verificar se utilizó la imagen res.pgm

Imagen antes de la segmentacion:

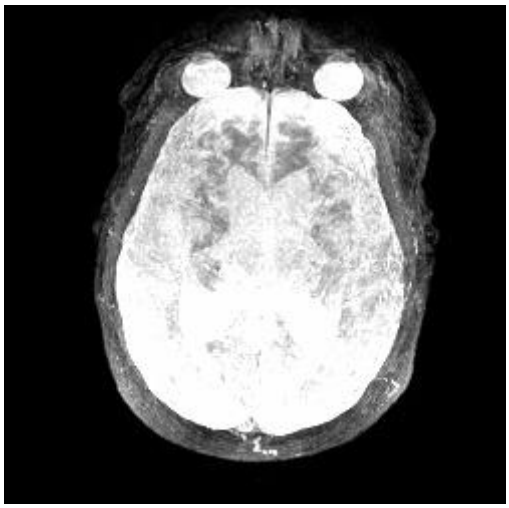

Imagen 8: Imagen antes de la segmentacion (res.pgm)

Imagen despues de la segmentacion:

Imagen 9: Imagen despues de la segmentacion (segm.pgm)

Conclusiones

En la tercera entrega del proyecto Procesamiento y Segmentación de Imágenes PGM se han unificado y afinado todas las capacidades desarrolladas en las fases anteriores. La fase inicial garantizó una carga fiable de archivos PGM y la exposición clara de sus metadatos; la segunda introdujo con éxito la generación de proyecciones 2D y la compresión por Huffman. Sobre esa base, esta entrega incorpora el módulo de segmentación, completando así un flujo de trabajo continuo desde la lectura de la imagen hasta la obtención de un resultado segmentado.

El comando segmentar constituye el núcleo de esta fase. Mediante la construcción de un grafo implícito que trata cada píxel como nodo y la aplicación de un algoritmo de Dijkstra multi-fuente, se propagan etiquetas a lo largo de regiones conexas partiendo de semillas definidas por el usuario. De esta forma, el sistema produce un mapa de etiquetas numéricas donde cada zona homogénea queda delimitada automáticamente, demostrando la efectividad de las estructuras de datos diseñadas: matrices de distancias, matrices de etiquetas, tuplas de estado y colas de prioridad.

La documentación asociada refleja el mismo nivel de detalle y claridad. Se han incluido diagramas de bloques y esquemas UML que ilustran la arquitectura global y las dependencias entre componentes. La definición de cada TAD —desde la representación de la imagen hasta el manejo del grafo de segmentación— sigue la plantilla establecida en clase, y el plan de pruebas dedicado al comando segmentar cubre escenarios normales, casos límite y situaciones de error. Esto asegura una lectura sencilla y una rápida comprensión de las decisiones técnicas adoptadas.

Con todos los módulos operativos y debidamente contrastados, el proyecto está listo para su presentación y sustentación. El trabajo demuestra no solo un dominio técnico de algoritmos y estructuras de datos, sino también un compromiso con las buenas prácticas de ingeniería: modularidad, documentación exhaustiva y pruebas rigurosas. Además, la estructura resultante ofrece un punto de partida sólido para futuras mejoras, como la incorporación de nuevos métodos de segmentación o la optimización del rendimiento en escenarios más complejos.