Universidad de Santiago de Chile
Facultad de Ingeniería
Departamento de Ingeniería Informática

Proyecto de Laboratorio 2022 - Paradigma Lógico

Paradigmas de Programación

Lucas Mesias Soza
Sección: 13310-0-A-1
Profesor: Roberto Gonzalez Ibañez

Universidad de Santiago de Chile
Facultad de Ingeniería
Departamento de Ingeniería Informática

Índice

Universidad de Santiago de Chile
Facultad de Ingeniería
Departamento de Ingeniería Informática

1. Introducción

Un programa de edición de imágenes permite modificar o alterar imágenes existentes y crear los elementos visuales con los que interactuamos día a día, como una simple fotografía con filtros simples o ajustes de color, afiches, eliminación de fondos, o directamente el programa se usa como herramienta de dibujo digital, programas como Adobe Photoshop, GIMP, Photopea, entre otros, son las herramientas con las que se trabaja para crear este mundo digital al que estamos acostumbrados.

2. Problema

Se requiere crear un software de manipulación de imágenes simplificado en el lenguaje Prolog (*Swi-prolog versión 8.4 o superior*), bajo el paradigma lógico, es decir, con predicados y cláusulas, interactuando con el usuario a través de la consola de comandos, y trabajar teniendo en cuenta las diferentes características del paradigma lógico, entre ellas la incapacidad de usar control de flujo, para esto, se trabaja de forma declarativa, se hace uso extendido de la recursión y predicados de la librería incorporada.

Los pixeles serán la base del trabajo de manipulación, estos deben ser representados en un TDA que permita guardar la siguiente información: posición, valor de color y profundidad. Las imágenes se deben representar en un TDA adecuado, que permita trabajar pixel a pixel, además habrán 3 tipos distintos de pixeles, diferenciandose en el valor de color que almacenan, estos serán bit (0 o 1), RGB y hexadecimal, además entre las operaciones que debe tener el software, debe ser capaz de comprimir y descomprimir imágenes, por lo que la implementación de la imágen debe soportar estas características.

3. Solución

Los TDAs implementados para los pixeles fueron estructurados en un TDA principal llamado “pixel”, el cual almacena el tipo de pixel, las coordenadas X, Y, el valor de color, y la profundidad en una lista. Luego, se crearon 3 sub-TDAs, uno por cada tipo de valor de color, “pixbit” para almacenar el número que representa el bit, “pixrgb” para almacenar 3 números enteros con los valores (R G B) y “pixhex” para almacenar el valor RGB en formato hexadecimal “#RRGGBB”, cada uno de estos es un pixel que guarda información diferente, de esta forma, se pueden implementar predicados básicos para pixeles genéricos, simplificando la implementación de predicados que no dependen del tipo de pixel.

El TDA imagen guarda las dimensiones de la imágen, un valor de compresión, una lista de pixeles (los cuales no necesitan estar en un orden particular) y la imagen misma, los predicados implementados no dependen del orden de esta lista, pero todos los pixeles de la imagen deben estar presentes o habrá inconsistencias a la hora de la descompresión.

Para los predicados que identifican el tipo de imagen se revisa que el primer pixel de la lista almacenada sea del tipo adecuado, ya que el usuario debe introducir estos de manera homogénea.

Histograma: Se extraen todos los valores de color presentes en la imagen y se almacenan en una lista, la cual se ordena sin eliminar duplicados usando *msort*, luego, con el predicado *clumped* se

genera una nueva lista que presenta el valor y la cantidad de repeticiones, finalmente, esta se transforma de formato (valor-cantidad) a (valor, cantidad).

Compresión: Usando la información entregada por Histograma, se elimina de la imagen los pixeles con el color más común recorriendo recursivamente la lista de pixeles de la imagen, luego se guarda en el valor de compresión el color eliminado.

Recorrer la imagen: Para implementar la descompresión y transformación de imagen a string, fue necesario recorrer una imagen pixel a pixel, para esto se crearon las predicados *findPix* que encuentra un pixel dadas unas coordenadas X e Y, y *sortImage*, una predicado recursivo que simula un ciclo *for* anidado, de esta forma se simplifican las operaciones que requieren recorrer la imágen completa.

Descompresión: Al ingresar una imagen comprimida, se rellenan los pixeles que faltan en la imagen con pixeles nuevos, con el valor de compresión haciendo uso de un predicado recursivo *decompPixs*, el cual es muy similar a *findPix*. Cabe destacar que con esta implementación se pierde totalmente la información de profundidad en los pixeles comprimidos.

Transformación de imagen a string: Se recorre la imagen de salida en orden, buscando el pixel que va en cada posición y se transforma su valor de color a string, debido a esto, el orden en el que los pixeles están en la lista no importa, ya que se ordena al realizar este proceso.

Capas de Profundidad: Primero se extrae de la imagen todas las profundidades de los pixeles sin repetición y se guardan, luego se genera una imagen por cada profundidad, y se recorren los pixeles, cuando se encuentra un pixel de la profundidad deseada, se almacena sin modificaciones, pero cuando ocurre el caso contrario, se cambia por un pixel blanco. Para esto, fueron utilizadas 3 funciones idénticas, para cada uno de los 3 tipos de pixeles, se detecta cuál usar al inicio del proceso, para evitar revisar el tipo de pixel necesario en cada pixel revisado.

El proyecto se organizó en 2 archivos, tda_pixel.pl e tda_image.pl, los cuales incluyen las respectivas funciones propias de los TDA. No se usó ninguna librería externa, y se usó SWI-Prolog para correr el programa.

4. Instrucciones de uso

Se debe tener el programa principal *tda_image.pl* en la misma carpeta que *tda_pixel.pl*.
Se ejecuta el programa SWI-Prolog *(8.4 o superior)* y se consulta el archivo tda_image_21266659_MesiasSoza.pl, el programa cargará el archivo y el usuario podrá realizar todas las consultas que desee, siguiendo el formato adecuado. Si SWI-Prolog acorta las respuestas de la consulta, se puede presionar la tecla “W” para que la respuesta se muestre completamente.

Uso	Resultado
pixbit(0, 0, 1, 10, PA), pixbit(0, 1, 0, 20, PB), pixbit(1, 0, 0, 30, PC), pixbit(1, 1, 1, 4, PD), image(2, 2, [PA, PB, PC, PD], I).	I = [2, 2, -1, [[0, 0, 1, 10], [0, 1, 0, 20], [1, 0, 0, 30], [1, 1, 1, 4]]]
pixbit(0, 0, 1, 10, PA), pixbit(0, 1, 0, 20, PB), pixbit(1, 0, 0, 30, PC), pixbit(1, 1, 1, 4, PD), image(2, 2, [PA, PB, PC, PD], I), imageIsPixmap(I).	false.
pixbit(0, 0, 1, 10, PA), pixbit(0, 1, 0, 20, PB),	I = [2, 2, -1, [[0, 0, 1, 10], [0, 1, 0, 20], [1, 0, 0,

pixbit(1, 0, 0, 30, PC), pixbit(1, 1, 1, 4, PD), image(2, 2, [PA, PB, PC, PD], I), imageIsBitmap(I).	30], [1, 1, 1, 4]]]. (En vez de entregar *verdadero*, Prolog construye la imagen)

Se adjuntan más ejemplos de uso en el anexo, adicionalmente es incluido un archivo de pruebas "pruebas_21266659_MesiasSoza.pl", el que contiene 4 imagenes de prueba y comandos para copiar y pegar en la consola de prolog, consultando el mismo archivo de pruebas.

5. Resultados

Se lograron implementar todos los predicados tanto obligatorios como opcionales, todos funcionan si la entrada es correcta, aunque se pudo haber hecho un esfuerzo mayor y revisar que las entradas sean correctas antes de procesar, en particular, se pudo implementar la revisión de todos y cada uno de los pixeles de la imagen para comprobar la homogeneidad de esta, pero este aspecto no fue una prioridad en el desarrollo del proyecto.

6. Autoevaluación

La autoevaluación del proyecto con respecto a las funciones implementadas es de puntaje completo, las únicas pruebas donde el código falla es cuando la entrada es incorrecta, por lo tanto, las funciones cumplen si son usadas correctamente.

7. Conclusiones

El trabajo realizado en el lenguaje lógico prolog fue más manejable y fácil de aprender, ya que es el segundo lenguaje declarativo con el que se trabaja en el laboratorio, y los retos que se deben superar son similares, además se puede reutilizar la lógica o las bases matemáticas de las operaciones implementadas del laboratorio anterior, lo cual es una diferencia sustancial a empezar desde 0.

8. Anexo

Puntajes Autoevaluación

Requerimientos funcionales	Autoevaluación
TDAs	1
image	1
imageIsBitmap	1
imageIsPixmap	1

imageIsHexmap	1
imageIsCompressed	1
imageFlipH	1
imageFlipV	1
imageCrop	1
imageRGBToHex	1
imageToHistogram	1
imageRotate90	1
imageCompress	1
imageChangePixel	1
imageInvertColorRGB	1
imageToString	1
imageDepthLayers	1
imageDecompress	1

Ejemplos de uso:

Uso	Resultado
pixbit(0, 0, 1, 10, PA), pixbit(0, 1, 0, 20, PB), pixbit(1, 0, 0, 30, PC), pixbit(1, 1, 1, 4, PD), image(2, 2, [PA, PB, PC, PD], I).	I = [2, 2, -1, [[0, 0, 1, 10], [0, 1, 0, 20], [1, 0, 0, 30], [1, 1, 1, 4]]].
pixbit(0, 0, 1, 10, PA), pixbit(0, 1, 0, 20, PB), pixbit(1, 0, 0, 30, PC), pixbit(1, 1, 1, 4, PD), image(2, 2, [PA, PB, PC, PD], I), imageIsPixmap(I).	false.
pixbit(0, 0, 1, 10, PA), pixbit(0, 1, 0, 20, PB), pixbit(1, 0, 0, 30, PC), pixbit(1, 1, 1, 4, PD), image(2, 2, [PA, PB, PC, PD], I), imageIsBitmap(I).	I = [2, 2, -1, [[0, 0, 1, 10], [0, 1, 0, 20], [1, 0, 0, 30], [1, 1, 1, 4]]]. (En vez de entregar verdadero, Prolog construye la imagen)
pixbit(0, 0, 1, 10, PA), pixbit(0, 1, 0, 20, PB), pixbit(1, 0, 0, 30, PC), pixbit(1, 1, 1, 4, PD), image(2, 2, [PA, PB, PC, PD], I), imageIsCompressed(I).	false.

pixbit(0, 0, 1, 10, PA), pixbit(0, 1, 0, 20, PB), pixbit(1, 0, 0, 30, PC), pixbit(1, 1, 1, 4, PD), image(2, 2, [PA, PB, PC, PD], I), imageFlipH(I , I2).	I2 = [2, 2, -1, [[1, 0, 1, 10], [1, 1, 0, 20], [0, 0, 0, 30], [0, 1, 1, 4]]] .
pixbit(0, 0, 1, 10, PA), pixbit(0, 1, 0, 20, PB), pixbit(1, 0, 0, 30, PC), pixbit(1, 1, 1, 4, PD), image(2, 2, [PA, PB, PC, PD], I), imageFlipV(I, I2).	I2 = [2, 2, -1, [[0, 1, 1, 10], [0, 0, 0, 20], [1, 1, 0, 30], [1, 0, 1, 4]]] .
pixbit(0, 0, 1, 10, PA), pixbit(0, 1, 0, 20, PB), pixbit(1, 0, 0, 30, PC), pixbit(1, 1, 1, 4, PD), image(2, 2, [PA, PB, PC, PD], I), imageCrop(I, 0, 0, 1, 0, Img2).	Img2 = [2, 1, -1, [[0, 0, 1, 10], [1, 0, 0, 30]]] .
pixrgb(0, 0, 10, 10, 10, 10, P1), pixrgb(0, 1, 20, 20, 20, 20, P2), pixrgb(1, 0, 30, 30, 30, 30, P3), pixrgb(1, 1, 40, 40, 40, 40, P4), image(2, 2,[P1, P2, P3, P4], I1), imageRGBToHex(I1, I2).	I2 = [2, 2, -1, [[0, 0, "#0a0a0a", 10], [0, 1, "#141414", 20], [1, 0, "#1e1e1e", 30], [1, 1, "#282828", 40]]].
pixbit(0, 0, 1, 10, PA), pixbit(0, 1, 0, 20, PB), pixbit(1, 0, 0, 30, PC), pixbit(1, 1, 1, 4, PD), image(2, 2, [PA, PB, PC, PD], I), imageToHistogram(I, Histograma).	Histograma = [[0, 2], [1, 2]].
pixbit(0, 0, 1, 10, PA), pixbit(0, 1, 0, 20, PB), pixbit(1, 0, 0, 30, PC), pixbit(1, 1, 1, 4, PD), image(2, 2, [PA, PB, PC, PD], I), imageRotate90(I, I2).	I2 = [2, 2, -1, [[1, 0, 1, 10], [0, 0, 0, 20], [1, 1, 0, 30], [0, 1, 1, 4]]] .
pixbit(0, 0, 1, 10, PA), pixbit(0, 1, 0, 20, PB), pixbit(1, 0, 0, 30, PC), pixbit(1, 1, 1, 4, PD), image(2, 2, [PA, PB, PC, PD], I), imageCompress(I, I2).	I2 = [2, 2, 0, [[0, 0, 1, 10], [1, 1, 1, 4]]]
pixrgb(0, 0, 10, 10, 10, 10, P1), pixrgb(0, 1, 20, 20, 20, 20, P2), pixrgb(1, 0, 30, 30, 30, 30, P3), pixrgb(1, 1, 40, 40, 40, 40, P4), image(2, 2, [P1, P2, P3, P4], I1), pixrgb(0, 1, 54, 54, 54, 20, P2_modificado), imageChangePixel(I1, P2_modificado, I2).	P2_modificado = [0, 1, [54, 54, 54], 20], I2 = [2, 2, -1, [[0, 0, [10, 10, 10], 10], [0, 1, [54, 54, 54], 20], [1, 0, [30, 30, 30], 30], [1, 1, [40, 40, 40], 40]]] .
pixrgb(0, 0, 10, 10, 10, 10, P1), pixrgb(0, 1, 20, 20, 20, 20, P2), pixrgb(1, 0, 30, 30, 30, 30, P3), pixrgb(1, 1, 40, 40, 40, 40, P4), image(2, 2, [P1, P2, P3, P4], I1), imageInvertColorRGB(P2, P2_modificado), imageChangePixel(I1, P2_modificado, I2).	P2_modificado = [0, 1, [235, 235, 235], 20], I2 = [2, 2, -1, [[0, 0, [10, 10, 10], 10], [0, 1, [235, 235, 235], 20], [1, 0, [30, 30, 30], 30], [1, 1, [40, 40, 40], 40]]] .
pixbit(0, 0, 1, 10, PA), pixbit(0, 1, 0, 20, PB), pixbit(1, 0, 0, 30, PC), pixbit(1, 1, 1, 4, PD), image(2, 2, [PA, PB, PC, PD], I),	I = [2, 2, -1, [[0, 0, 1, 10], [0, 1, 0, 20], [1, 0, 0, 30], [1, 1, 1, 4]]], Str = "1\t0\n0\t1\n" .

imageToString(I, Str).	
pixbit(0, 0, 1, 10, PA), pixbit(0, 1, 0, 20, PB), pixbit(1, 0, 0, 30, PC), pixbit(1, 1, 1, 10, PD), image(2, 2, [PA, PB, PC, PD], I), imageDepthLayers (I, LI).	I = [2, 2, -1, [[0, 0, 1, 10], [0, 1, 0, 20], [1, 0, 0, 30], [1, 1, 1, 10]]], LI = [[2, 2, -1, [[0, 0, 1, 10], [0, 1, 1, 20], [1, 0, 1, 30], [1, 1, 1, 10]]], [2, 2, -1, [[0, 0, 1, 10], [0, 1, 0, 20], [1, 0, 1, 30], [1, 1, 1, 10]]], [2, 2, -1, [[0, 0, 1, 10], [0, 1, 1, 20], [1, 0, 0, 30], [1, 1, 1, 10]]]] .
pixbit(0, 0, 1, 10, PA), pixbit(0, 1, 0, 20, PB), pixbit(1, 0, 0, 30, PC), pixbit(1, 1, 1, 4, PD), image(2, 2, [PA, PB, PC, PD], I), imageCompress(I, I2), imageDecompress(I2, I3).	I = [2, 2, -1, [[0, 0, 1, 10], [0, 1, 0, 20], [1, 0, 0, 30], [1, 1, 1, 4]]], I2 = [2, 2, 0, [[0, 0, 1, 10], [1, 1, 1, 4]]], I3 = [2, 2, -1, [[0, 0, 1, 10], [1, 0, 0, 0], [0, 1, 0, 0], [1, 1, 1, 4]]] . I =/= I3, ya que se pierde la información de profundidad de los pixeles comprimidos