Universidad de santiago de Chile

Labor Lætitia nostra

Informe para el ramo Paradigmas de programación

Laboratorio 2: Paradigma Lógico
Editor de imágenes usando Prolog

Sergio Osvaldo Andres Espinoza Gonzalez

Facultad de ingeniería

Noviembre de 2022

Índice

Índice de cuadros

1. Introducción

En este informe se abordará el laboratorio número 2 de la asignatura Paradigmas de programación con el formato expuesto en el indice del mismo. A continuación, se darán breves descripciones de los dos grandes pilares a ver para encontrar la posterior resolución; se invita a continuación a conocer el problema a resolver:

1.1. Descripción del problema

Para empezar debemos conocer el problema a resolver, el cual es la creación de un software que permita la edición de imágenes de manera similar al software de código abierto GIMP o el de pago conocido como Photoshop; la finalidad del proyecto, es poder realizar las funcionalidades de un editor como los antes mencionados sin la necesidad de una interfaz gráfica.

1.2. Descripción del paradigma

Para la resolución del problema planteado anteriormente, se hará uso de un lenguaje de programación llamado Prolog el cual se encuentra enfocado en el paradigma de programación lógica, ya teniendo en cuenta esto podría surgirnos la pregunta "¿Que es el paradigma lógico?"

1.2.1. Paradigma de programación lógica

El paradigma lógico es aquel donde se hace uso de la lógica que subyace a un problema para su posterior resolución, esto a través de un conjunto de fórmulas lógicas que suelen verse, al llevar esto a la programación, como predicados o cláusulas de Horn(Taltavull y Antoni, 2015). Lo anterior convierte la resolución de un problema en la búsqueda de la lógica detrás del mismo, y no en una lista de pasos a seguir como acostumbramos en otros paradigmas; esto a través de una base de conocimientos con hechos y reglas, haciendo uso también de la unificación.

2. Análisis del problema

Ya entendiendo por encima el problema y el paradigma a utilizar, para solucionarlo profundizaremos en el mismo, ya que, conociendo bien el problema y sus requisitos específicos llegaremos a una mejor solución.

1. TDAs: El primer desafio será diseñar los tipos de datos abstractos que nos permitan alcanzar la solución de la manera más eficiente posible, por lo que este primer requisito es más que nada de diseño de la solución que estará en la sección 3.

2. Contructor de imágenes: Ya habiendo mencionado los TDAs el que principalmente se llevará nuestra atención será aquel que represente una imagen como tal, ya que, al implementarlo, debemos tener diferentes consideraciones como:

   - Tipo de imagen: En un principio se deberían tener 3 tipos de imagen correspondientes a bitmap, pixmap y hexmax y que el programa pueda reconocer de que tipo se trata; la implementación de lo antes mencionado se verá en la siguiente sección.
   - Debe ser posible también poder cambiar entre imágenes tipo pixmap y hexmap (RGB y hexadecimal).
   - ¿Está comprimida?: Es necesario saber si una imagen está o no comprimida
   - Comprimir y descomprimir: Como ya se dijo que una imagen puede estar o no comprimida entonces deben haber funciones que compriman y descompriman la imagen como tal, dependiendo de la implementación podría verse también como requisito una función que nos diga cuál es el color más usado en la imagen (Histograma).

3. Cambiar dirección o rotar: Otro requisito especifífico a abordar, es dar la posibilidad de invetir la imagen ya sea en el eje X o Y, además de poder rotar esta en 90° hacia la derecha.

4. Recortar: También se debe dar la posibilidad de recortar la imagen para obtener una nueva imagen de un subsector de la original.

5. Invertir colores: Como el nombre lo dice, se debe ser capaz de invertir los colores de imágenes RGB.

6. Imagen a string: Debe también cubrirse la posibilidad de que la imagen pueda ser representada como un conjunto de caracteres.

7. Separar en capas: La imagen debe tener profundidad, y para esto, se deben tener varias capas entre las que le es posible separarse.

3. Diseño de la solución

Comenzando ahora con el diseño de la solución para la problemática de este semestre, se realizará la definición y especificación de los distintos TDAs para luego ver como puede ser la posible implementación de los mismos y lo que se realizará para los problemas específicos.

3.1. TDAs

Empezando con los TDAs el enfoque será que una imagen esta compuesta de pixeles los cuales son de alguno de los siguientes 3 tipos, pixbit, pixrgb o pixhex; realizando los TDAs de estos 3 tipos de pixel junto con el TDA imagen se podría abordar la mayoría de los requisitos específicos sin problema, sin embargo para un manejo del requisito del histograma se definiría un último TDA llamado histogram.

3.1.1. Definición

Ya sabiendo cuales son los TDA a implementar se procederá con su definición:

- TDA image: Una imagen será aquel conjunto de pixeles con un ancho y un alto específicos que puede estar o no comprimida, por lo cual, se debe poder saber cual es el color más usado en la misma.
- TDA pixbit: Será aquel pixel con una posición específica en una imagén de 3 dimensiones con 2 posibles colores, blanco o negro.
- TDA pixrgb: Será aquel pixel con una posición específica en una imagén de 3 dimensiones con color de tipo rgb.
- TDA pixhex: Será aquel pixel con una posición específica en una imagén de 3 dimensiones con color de tipo hexadecimal.
- TDA histogram: Será el gráfico de cuantos pixeles de cada color hay contenidos en una imagén

3.1.2. Especificación

La especificación general se puede encontrar en el cuadro 1 de los anexos; si se desean tener más detalles se recomienda ver el código de cada TDA donde se específica de manera más detallada debido al poco espacio de este medio.

3.1.3. Implementación

La implementación de una imagen será con una lista que contendrá 1 bit de compresión, donde 0 será sin comprimir y 1 será comprimida, además de 2 enteros positivos que son el ancho y el alto de la misma, junto con una lista con los diferentes pixeles que tiene; por último tendrá un elemento con el color más usado en la imagen, cuyo tipo dependerá del tipo de imagen que sea (bitmap, pixmap o hexmap).

Los pixeles en sí estarán un poco más adelante, pero algo que los 3 tipos de estos tendrán en común, es que serán implementados con listas donde el primer y segundo elemento corresponden a la posición (x,y)

del pixel y el último elemento de la lista corresponderá a la profundidad del pixel, donde, estos 3 valores mencionados, son enteros positivos.

La gran diferencia se encontrará en los datos que hay entre medio de los valores (x,y) y depth (profundidad) los cuales corresponden al color del pixel; los TDA de pixeles quedarían de la siguiente forma:

- Pixbit-d: Este tipo de pixel tendrá un bit entre medio de tipo entero, el cual solo podrá tomar los valores 0 y 1. De esta forma un pixbit-d será una lista de la forma:

$$[int, int, int, int] = [x \geq 0, y \geq 0, bit(0|1), depth \geq 0]$$

- Pixrgb-d: Este tipo de pixel tendrá además una lista de 3 enteros entre 0 y 255 que representarán la cantidad de cada color que tiene el pixel siendo r = rojo (red), g = verde (green) y b = azul (blue) quedandonos una lista de la forma:

$$[int, int, [int, int, int], int] = [x \geq 0, y \geq 0, [0 \leq r \leq 255, 0 \leq g \leq 255, 0 \leq b \leq 255], depth \geq 0]$$

- Pixhex-d: Este tipo de pixel tendrá un string con formato #XXXXXX donde las X pueden tomar valores de un dígito hexadecimal (entre 0 y F(15)) donde las 2 primeras X es la cantidad de rojo, las 2 de al medio son de verde y las 2 últimas son de azul; la representación quedaría de la forma:

$$[int, int, string, int] = [x \geq 0, y \geq 0, hex, depth \geq 0]$$

Por otro lado el histograma será una lista de listas de 2 elementos, en estas listas de 2 la cabeza será un color y el segundo elemento la cantidad de veces que ese color se repite.

4. Consideraciones de implementación

El código del proyecto estará en la carpeta Código, donde se encontrará el main con la base de conocimientos que nos permitirá cumplir los requisitos principales que se exigen en el enunciado; en esta carpeta se encontrará también el archivo de pruebas y otra carpeta que contiene los TDAs mencionados en la sección anterior "TDAs". No se usará ningún módulo externo al proyecto y el interpretador usado será "swipl"versión 8.4.3.

5. Instrucciones de uso

En un principio se realizó todo pensando en que las imagenes deberían ser completamente ingresadas para que la compresión funcione bien, sin embargo no es necesario que se ingresen en orden los pixeles, ya que, el constructor de la imagen se encarga de eso; debido a esto no hay muchas instrucciones de uso como tal, más allá de que se cumpla el formato de los ejemplos del cuadro 2 e ingresar todos los pixeles, aunque claro, hay muchos más ejemplos en el archivo de pruebas.

Se debe tener en cuenta que (Se usaron pixrgb debido a que permiten usar todos los predicados):

- Pixeles:
  - P1: pixrgb(0,0,0,0,0,0,P1).
  - P2: pixrgb(0,1,255,255,255,10,P2).
  - P3: pixrgb(1,0,255,255,255,20,P3).
  - P4: pixrgb(1,1,0,122,255,30,P4).
  - MP3: invertColorRGB(P3,MP3).
- Imagenes:

- I1: image(2,1,[P1, P3],I1).
- I2: image(2,2,[P1,P2,P3,P4],I2).
- CI2: imageCompress(I1,CI1).

Las imagenes de ejemplo son pequeñas para que quepan en la tabla.

6. Resultados obtenidos

Los resultados de la implementación y sus requisitos especifícficos fueron resumidos en el cuadro 3 junto con cuales se completaron, el puntaje de funcionamiento según la pauta dada de autoevaluación (entre 0 y 1); las pruebas realizadas fueron consultas a la base de conocimientos en el archivo de pruebas que se subirá a la plataforma github y uvirtual.

Con respecto a la posible razón de los fallos en el requisito 18 (Descomprimir una imagen), se debe a que, si bien se recuperan los pixeles con el color más usados en la posición (x,y) respectiva, se da por hecho que la profundidad de estos es 0 por lo que funcionaría bien en imagenes bidimensionales pero no en las que se usaron; una idea para solucionar esto sería que, al comprimir la imagen, se almacene también una lista con las profundidades en orden, pero, la razón por la que no se quiso implementar esto, es que, si tenemos que almacenar una lista completa para comprimir una imagen, pierde un poco el sentido de hacerlo, ya que, aún se estaría ocupando un gran espacio.

7. Conclusión

Para concluir, cabe hablar acerca de como fue abordar el proyecto haciendo uso de este paradigma; si bien no se tuvieron las facilidades que dan las variables mutables del paradigma imperativo procedural, ni las funciones del paradigma funcional visto en el laboratorio pasado, los alcances de este paradigma se ven también como ilimitados, ya que, al no tener que pensar tanto en los pasos a seguir para resolver el problema, sino más bien pensar en el problema y sus caracteristicas en sí; dejar que el interprete lo resuelva haciendo uso de sus herramientas ,que no necesitamos conocer, simplifica mucho la resolución del problema; es más, bastó con tener un buen entendimiento del mismo.

Finalmente a modo de opinión personal, el paradigma lógico se me hizo muy interesante, es más, fue entretenida la parte de programar durante el laboratorio haciendo uso de este paradigma, incluso más que en el paradigma funcional, y espero poder seguir aprendiendo del mismo por mi cuenta.

Referencias

Taltavull, C., y Antoni, J. (2015). Programación lógica. , 10–12. Descargado de http://hdl.handle.net/2445/64643

Anexos

X	Image	Histogram	Pixbit-d	Pixrgb-d	Pixhex-d
Constructor	Sí	Sí	Sí	Sí	Sí
Pertenencia	Sí	No	Si	Sí	Sí
Selectores	Sí	No	Si	Sí	Sí
Modificadores	Sí	No	Sí	Sí	Sí
Otras funciones	Sí	No	No	No	No

Cuadro 1: Especificación de TDAs

Función	Uso	Resultado esperado
2. image	image(1,2,[P1,P2],I).	I=[0,1,2,[[0,0,[0,0,0],0],[0,1,[255,255,255],10]],[0,0,0]]
3. bitmap?	imageIsBitmap(I1).	false
4. pixmap?	imageIsPixmap(I1).	I1=[0,2,1,[[0,0,[0,0,0],0],[1,0,[255,255,255],20]],[0,0,0]]
5. hexmap?	imageIsPixmap(I1).	false
6. compressed?	imageIsCompressed(I1).	false
7. flipH	imageFlipH(I1,I3).	I3=[0,2,1,[[0,0,[255,255,255],20],[1,0,[0,0,0],0]],[0,0,0]]
8. flipV	imageFlipV(I1,I3).	I3=[0,2,1,[[0,0,[0,0,0],0],[1,0,[255,255,255],20]],[0,0,0]]
9. crop	imageCrop(I1,0,0,0,0,I2).	I3=[0,1,1,[[0,0,[0,0,0],0]],[0,0,0]]
10. imgRGB->imgHex	imageRGBToHex(I1,I3).	I3=[0,2,1,[[0,0,"#000000",0],[1,0,"#fffff",20]],"#000000"]
11. histogram	imageToHistogram(I2,H).	H=[[[0,0,0],1],[[255,255,255],2],[[0,122,255],1]]
12. rotate90	imageRotate90(I1,I3).	I3=[0,1,2,[[0,0,[0,0,0],0],[0,1,[255,255,255],20]],[0,0,0]]
13. compress	imageCompress(I2,I3).	I3=[1,2,2,[[0,0,[0,0,0],0],[1,1,[0,122,255],30]],[255,255,255]]
14. changePixel	imageChangePixel(I1,MP3,I3).	[0,2,1,[[0,0,[0,0,0],0],[1,0,[0,0,0],20]],[0,0,0]]
15. invertColorRGB	imageInvertColorRGB(P4,MP4)	MP4=[1,1,[255,133,0],30]
16. image->string	imageToString(I1,Str).	Str="[0,0,0]\t[255,255,255]\t\n"
17. depthLayers	imageDepthLayers(I1,IL).	IL=[[0,2,1,[0,0,[0,0,0],0],[0,0,0]],[0,2,1,[1,0,[255,255,255],20],[255,255,255]]]
18. decompress	imageDecompress(CI1,I3).	I3=[0,2,1,[[0,0,[0,0,0],0],[1,0,[255,255,255],20]],[0,0,0]]

Cuadro 2: Ejemplos de uso

Requisito	Completado?	Puntaje autoevaluación	Posible razón fallos
1. TDAs	Si	1	N/A
2. image	Si	1	N/A
3. bitmap?	Si	1	N/A
4. pixmap?	Si	1	N/A
5. hexmap?	Si	1	N/A
6. compressed?	Si	1	N/A
7. flipH	Si	1	N/A
8. flipV	Si	1	N/A
9. crop	Si	1	N/A
10. imgRGB->imgHex	Si	1	N/A
11. histogram	Si	1	N/A
12. rotate90	Si	1	N/A
13. compress	Si	1	N/A
14. changePixel	Si	1	N/A
15. invertColorRGB	Si	1	N/A
16. image->string	Si	1	N/A
17. depthLayers	Si	1	N/A
18. decompress	Si	0.5	En sección 6

Cuadro 3: Resultados obtenidos