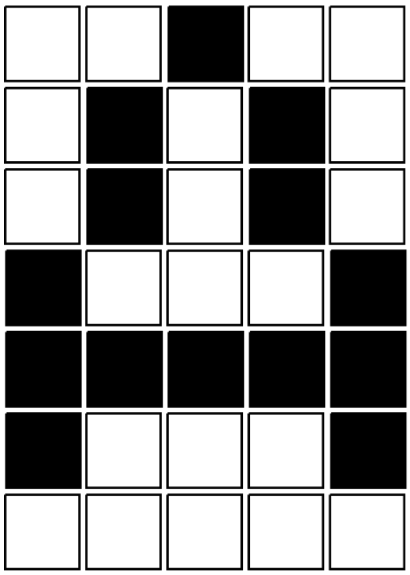
DEPARTAMENTO DE
INGENIERÍA
INFORMÁTICA
UNIVERSIDAD DE SANTIAGO DE CHILE

Las Imágenes y su manejo en una perspectiva lógica.



Laboratorio 2: Paradigma lógico

Nombre: Branco García Santana

Sección: A-1

Asignatura: Paradigmas de Programación

Índice

1. Introducción

En la antigüedad, para recrear algún objeto, persona, sentimiento o elemento visual, se usaban dibujos o pinturas, para representar imágenes. Luego la tecnología llegó y con ello aparecieron las imágenes digitales y softwares de edición de estas como GIMP, programa de trabajo de alta calidad para la manipulación de imágenes con secuencias de comandos y con compatibilidad con varios idiomas, como C, C++, Perl, Python, Scheme, entre otros.[1]

Por otro lado, en torno a los tipos de imágenes, se encuentran las que son tipo bitmap (de pixeles blanco y negro (binarios)), hexmap (de pixeles con colores en hexadecimal) y pixmap (pixeles RGB-D con los canales Red, Green y Blue en representación al color).

Una vez mencionado el concepto imagen, sus tipos y operaciones, se tiene en cuenta el **problema** de cómo querer operar (editar) una imagen y para ello se debe tener en cuenta el concepto Tipo de Dato Abstracto (**TDA**) que es un conjunto de datos u objetos al cual se le asocian operaciones.

Entonces, ¿Cómo resolver dicho problema?, para hacerlo este informe tiene de **objetivos** analizar dicho problema, diseñar una **solución**, en específico un programa representativo de edición de imágenes relacionado a un paradigma declarativo: el lógico, donde los lenguajes tienen una base de conocimiento, a la cual se le puede hacer consultas. Y en torno a la **estructura** del informe, se identificará el problema asociado al proyecto, su relación con el paradigma lógico, el análisis del problema, diseño de la solución, aspectos de implementación de esta, las instrucciones de uso y una conclusión de cierre al respecto.

---

2. Desarrollo

- **Problema**:

Cómo se mencionó brevemente, en torno al concepto imagen, surge el problema de cómo operar una bajo un concepto específico en la informática: el **paradigma lógico** (descrito abajo), entonces simulando algunas operaciones que hace un software de edición de imágenes, se busca ver si una imagen es un bitmap-d, un hexmap, un pixmap, si está comprimida; rotarla 90 grados, invertirla horizontal y verticalmente, recortarla, pasar una imagen de representación RGB a hexadecimal, obtener las frecuencias de colores, como comprimirla y descomprimirla, cambiar un píxel.

El principal desafío, es como representar a la imagen que contiene un ancho, largo y pixeles definidos, más los pixeles que debe contener esta, los que tienen una representación según 3 tipos de pixeles: pixbit que tienen de color un bit 0 o 1, pixrgb que tienen 3 canales de color “Rojo (R), Verde (G), azul (B); y pixhex que tienen un color en formato string hexadecimal.

Entonces, en resumen, el **problema** sería cómo hacer los 17 requerimientos mencionados, sumado a cómo representar el concepto de imagen (con su ancho, alto y pixeles), en el paradigma lógico, específicamente en el lenguaje Prolog.

Y las **limitaciones** de este problema sería trabajar directamente con una imagen (solo se trabaja con un TDA representativo), ya que este laboratorio no cuenta con el desarrollo de interfaz gráfica.

- **Paradigma Lógico:**

Por otro lado, la programación que entrega nuevas herramientas como GIMP para editar imágenes está relacionada a los lenguajes de programación que están ligados a diferentes paradigmas (modelos y clasificación), entre ellos se encuentran el paradigma funcional, el orientado a objetos y el relacionado a este laboratorio: el paradigma lógico, con el cual se tratará el problema.

El **paradigma lógico** está encuadrado dentro de los paradigmas **declarativos** (los cuales no detallan los pasos de un procedimiento a seguir para llegar a un resultado, sino solo importa llegar a este).

En programación lógica, es posible representar los datos disponibles mediante **hechos** y **reglas**, de la misma forma es posible representar la cadena de deducciones necesaria para obtener la solución. Esto permite automatizar total o parcialmente el proceso de encontrar la solución.[2]

En específico los **hechos** corresponden a una expresión atómica que verifica un predicado sobre diferentes "variables" y son utilizados para declarar verdades desde el principio de la ejecución, formando la base de conocimientos. Mientras que las **reglas** son un conjunto de proposiciones lógicas que permiten inferir el valor de verdad de nuevas proposiciones.[3]

Lenguajes asociados a este paradigma, como **PROLOG**, se basan en 3 mecanismos básicos: **unificación** (que permite la extracción de respuestas, donde las variables lógicas toman un valor), **backtracking** automático (técnica algorítmica para hallar soluciones a problemas que satisfacen restricciones) y estructuras de datos basadas en árboles.[4]

Por último, otro término importante en esta área es el mecanismo de **inferencia** correspondiente a la automatización sobre una computadora (cuando el programa actúa sobre el mismo).[5]

- **Análisis del Problema:**

Retomando el problema, para este se deberá primero establecer una representación (como TDA) de la imagen, la cual contiene el ancho, largo y pixeles, para poder trabajar con ella y elaborar los predicados que representen operaciones. También hay que hacer lo mismo con los pixeles, ya que estos se pueden clasificar en los tipos pixrgb, pixhex y pixbit, además en cada representación se debe tener la posición en x del píxel, la posición en y, el color y la profundidad asociada. Una vez se tengan las representaciones adecuadas para estos datos, se deben implementar diversos algoritmos para poder crear cada predicado y solucionar las 17 operaciones de imagen, a través del ya mencionado programa Prolog.

Entrando más a detalle, específicamente se necesitan los siguientes **requisitos** para acatar el problema:

- TDAs generales para la implementación, como selectores de elementos de la imagen o de pixeles, constructores, operadores de pertenencia, entre otros.
- Constructor de imagen ["image"], que permita construir una representación de la imagen y en donde se pueda almacenar sus dimensiones y pixeles (para operar más adelante).
- Predicados de pertenencia que determinen si una imagen es bitmap, hexmap y pixmap, en las cual se requiere analizar la estructura y representación de los pixeles ["imageIsBitmap", "imageIsPixmap", "imageIsHexmap"].
- Cláusula de pertenencia que determine si una imagen está comprimida ["imageIsCompressed"]
- Predicados modificadores que permitan mostrar el invertir de una imagen horizontal y verticalmente [imageFlipH, imageFlipV], cambiando las posiciones de los pixeles de esta.
- Cláusula para el recorte de una imagen, cambiándose las dimensiones de esta [imageCrop].

- Predicado para la transformación de una imagen de una representación RGB a una Hexadecimal, cambiando los tipos de pixeles en esta con la representación del color [imageRGBToHex]
- Cláusula para obtener frecuencia de colores (repeticiones) en una imagen [imageToHistogram].
- Predicado modificador para poder ver una imagen rotada a 90 grados a la derecha, invirtiéndose las dimensiones de la imagen y cambiando los pixeles de posición [imageRotate90]
- Cláusula para comprensión de imagen con pixeles de color más frecuente [imageCompress]
- Predicado para el reemplazo de píxel (según coordenadas) en una imagen [imageChangePixel]
- Cláusula para inversión de los valores de los canales RGB de un pixrgb [invertColorRGB]
- Predicado para obtención de representación modo string de una imagen [imageToString]
- Operación para separación de una imagen y observar creación de varias en base a la profundidad de los pixeles (sustituyendo en pixeles blancos en donde no coincidan) [depthLayers]
- Y un predicado para descomprensión de una imagen y poder seguir operándola [imageDecompress]

Respecto a los TDAs, para poder acceder a elementos de estos (como el ancho de una imagen) o para modificar elementos de ellos, hay que crear selectores y operadores asociados (primer requerimiento), esto facilitará el análisis del problema. Y para cada predicado, habrá que tener en cuenta el dominio del predicado, el cual es imagen y la imagen de salida (en general), además de establecer otras cláusulas y hechos para poder trabajar con partes específicas de la estructura (como los pixeles de una imagen).

Además, en el proceso se requiere autoevaluación para probar los requerimientos, el uso de lenguaje y versión (para que sea revisable), el no uso de bibliotecas externas, documentación para mantener orden en los predicados (con dominio, descripción, estrategia), la organización de archivos por tda para mantener las clausulas separadas, un historial para ir avanzando constantemente, un script de pruebas (para ver el uso de los predicados) y los prerrequisitos de cada uno.

- Diseño de la solución:

Para el diseño de problemas se realizaba (en general) la siguiente secuencia de pasos y técnicas:

o Implementar los TDAs y operadores asociados:

El primer paso, antes de analizar detalladamente cada requerimiento, se pensó que tipo de dato se requería construir (en este caso el TDA image, y el de los pixeles: TDA pixbit, pixhex, pixrgb), la información de una parte de los datos compuestos que se requiere obtener mediante un predicado (selectores) o evaluar si los datos cumplen una determinada característica (pertenencia). En general, tanto los constructores como selectores fueron los más usados (ya que se utilizaron en todos los predicados). Cabe destacar, que todos los constructores y datos principalmente trabajados como las imágenes y pixeles, tienen **representaciones** en listas.

o Analizar parámetros del predicado (dominio, incluyendo variable para salida):

En todos los casos de dominio se tiene una imagen, entonces por lo tanto se forzaba (en la mayoría de los casos) seguir desarrollando predicados más específicos para no complejizar el proceso y no trabajar con tanta información (recordando que los datos trabajados se representaron como listas, entonces algunos predicados solo requieren modificar pixeles, que es mucho menos información).

- **Desarrollo de predicados que llaman a otros y recursión:**

Como se mencionó, muchos predicados requieren operar con solo pixeles, por ello primero se desarrollaba un predicado que modificará un píxel (como en el caso de imageFlipH, imageFlipV, imgRGBtoimgHex, imageRotate90), y luego aplicar otro predicado recursivo que haga el mismo proceso con todos los pixeles. Y, por último, se usaba el constructor de imagen, conservando el ancho y largo (menos en imageRotate90 que en este se intercambiaban) y aplicando un predicado que modificara todos los pixeles.

En otros predicados como imageCrop se aplicó este procedimiento de analizar un píxel, pero en vez de modificarlo se preguntaba si se aplicaba en ese tipo de dato cierto criterio (pertenencia), en ese caso si pertenecía al intervalo x e y a recortar; luego se evaluaba cada píxel recursivamente con dicho predicado de pertenencia, y se volvía a usar el constructor con el ancho, largo (restas entre los parámetros a recortar) y los nuevos pixeles resultantes.

Hubo también predicados que requirieron de contar y juntar por elementos (como el imageToHistogram con los colores, o el depthLayers en el que se necesitó previamente almacenar los pixeles de diferente profundidad), para estos casos se contaba una profundidad o color específico, luego el resto, clasificando los pixeles. Y existieron predicados que requerían si o si la implementación de otros de los 17 requerimientos, como compress que requería el histogram, o decompress que requiere a compress para operar.

En general, la técnica y descomposición de problemas fue trabajar desde lo particular (como un píxel) a lo general (los pixeles y luego la imagen entera)

- **Aspectos de implementación:**

La **estructura del proyecto** se conformó de implementación de TDAs constructores, luego selectores, cláusulas de pertenencia y predicados que aplicaban aquellos, los que trabajaban (en la mayoría de los requerimientos, con un píxel, luego con todos los pixeles y luego construyen la imagen nuevamente, pero con los pixeles cambiados).

Como se mencionó, se utilizó el **lenguaje** lógico Swi-prolog 8.4.3, sin emplear alguna biblioteca externa, y para cada requerimiento se aplicaron predicados propios del lenguaje como “string_concat”, entre otros.

El **lenguaje usado** específicamente fue **Swi-prolog** (ya que era el directo para el desarrollo del laboratorio), sin embargo, se trabajó constantemente en la plataforma **SWISH** de navegador, ya que esta ofrecía una mejor visión de las estructuras a trabajar cuando se consultaba.

- **Instrucciones de uso:**

Generalmente, las instrucciones de uso para poder trabajar con los predicados es colocar en consola primero los pixeles, asociándoles variables con nombre, luego usar el predicado image ingresando el ancho, largo y cada píxel definido entre corchetes como lista [], además de una variable para la salida, siguiendo la siguiente estructura: **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).**

Una vez definida una imagen con un nombre, se puede aplicar cualquiera de los otros predicados (los cuales deben tener en su dominio el mismo nombre), de la siguiente forma: **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).**

Y aunque se vea una lista de esa forma, a ese resultado se le puede seguir componiendo con otro predicado, teniendo en cuenta el nombre que se le coloca a las variables, por ejemplo: **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), imageFlipH(I2,I3).**

Aunque hay restricciones respecto al usar predicados, como predicados que no tienen en el dominio de salida una imagen, como imageToHistogram e imageToString; donde las variables de salida no se pueden seguir operando con los otros requerimientos; y no poder operar con imágenes comprimidas (menos predicados como decompress), o predicados que trabajan con ciertos tipos de imágenes (ej: imageInvertColorRGB que solo trabaja con imágenes pixmap), retornándose falso: **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), imageInvertColorRGB(I,I2). % Entrega false**

- **Resultados y Autoevaluación:**

Para todos los predicados se aplicaron imágenes de distintas dimensiones para experimentar y de distinto tipo (pixmap, hexmap, bitmap)

Predicados	Grado de alcance
TDAs (constructores de pixeles, selectores, de pertenencia, modificadores, etc.)	Se destaca el uso de constructores “pixrgb”, “pixhex” y “pixbit”, selectores “getX”, “getY” “getAncho”, “getPixeles”, “getLargo”, “getColor”, “getD”
Image – constructor, imageIsBitmap, imageIsHexmap, imageIsPixmap, imageIsCompressed, imageFlipH, imageFlipV, imageCrop, imageRGBToHex, imageToHistogram, imageRotate90, imageCompress, imageChangePixel, imageInvertColorRGB, imageToString, imageDepthLayers, imageDecompress	Completo

Si hay alguna disconformidad en el código es sobre que algunas estructuras pudieron resumirse, pero por motivos de tiempo no se hizo (como crear un modificador set para imagen).

---

3. Conclusiones

En conclusión, se cumplieron los **objetivos** del informe al analizar el problema relacionado al querer operar una imagen, y el diseño de una solución bajo el paradigma lógico en Prolog.

De **limitaciones** se tuvo el tiempo, el intentar mantener un trabajo constante (subiendo avances en GitHub) y tratar que en cada predicado saliera una variable con resultado deseado; de dificultades se tuvo acostumbrarse a colocar la variable de salida en el dominio, usar recursión y casos bases (ya que se trabaja más cómodo con iteración). En relación de los **alcances**, hay un sentimiento de conformidad al poder completar todos los requerimientos, si bien algunos algoritmos y códigos suelen estar complejos y muy repletos, fueron varias veces modificados para ver si se podía encontrar un método de resolución más eficiente y menos costoso entorno al tiempo y complejidad.

Respecto a la **comparación** con la entrega del laboratorio pasado con el paradigma funcional, se considera que para este trabajo el lenguaje lógico pudo ofrecer un desarrollo más rápido para requerimientos como depthLayers o Histogram, además de ser muy útil definir hechos para los casos bases. También al tener los algoritmos desarrollados del laboratorio anterior, solo se tradujo e implementó en la forma de trabajo del paradigma lógico.

4. Referencias

[1] Gimp. (2022). GIMP. gimp.org. https://www.gimp.org/

[2] Peri, J. A. & Godoy, D. L. (2012, 29 noviembre). Utilización de acertijos lógicos como ejercicios motivadores para la enseñanza de la programación lógica. http://sedici.unlp.edu.ar/handle/10915/24833

[3] Universidad Nacional de Colombia. (s. f.). Programación lógica UNAL. https://ferestrepoca.github.io/paradigmas-de-programacion/proglogica/logica_teoria/proglogica.html

[4] Flores V. (2020). INTRODUCCIÓN AL PARADIGMA LÓGICO - Uvirtual (Universidad Santiago de Chile) https://uvirtual.usach.cl/moodle/mod/hvp/view.php?id=156617

[5] Salgado Matías, E. (2018, 6 diciembre). Programación lógica y su semántica en espacios métricos generalizados. BUAP. https://repositorioinstitucional.buap.mx/handle/20.500.12371/7271

- Material complementario (utilizados en general):

[6] Flores V. (2020). PARTES DE UN PROGRAMA LÓGICO EN PROLOG- Uvirtual (Universidad Santiago de Chile) https://uvirtual.usach.cl/moodle/mod/hvp/view.php?id=156622

[7] Flores V. (2020). HECHOS EN PROLOG - Uvirtual (Universidad Santiago de Chile) https://uvirtual.usach.cl/moodle/mod/hvp/view.php?id=156623

[8] Flores V. (2020). REGLAS EN PROLOG - Uvirtual (Universidad Santiago de Chile) https://uvirtual.usach.cl/moodle/mod/hvp/view.php?id=156624

[9] Flores V. (2020). UNIFICACIÓN Y BACKTRACKING - Uvirtual (Universidad Santiago de Chile) https://uvirtual.usach.cl/moodle/mod/hvp/view.php?id=156629

5. Anexos

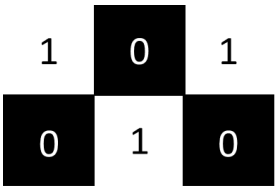
- Ilustraciones script de pruebas (a continuación, se muestran ejemplos de ciertos predicados)

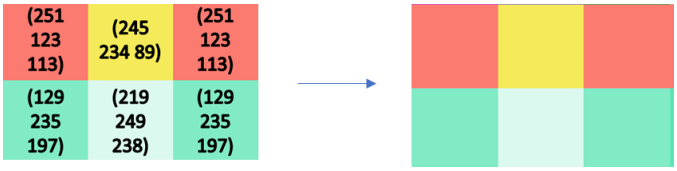
Se forman 3 imágenes: I1 será una imagen tipo bitmap, I2 tipo pixmap, I3tipo hexmap:

pixbit(0, 0, 1, 15, P1), pixbit(0, 1, 0, 75, P2), pixbit(0, 2, 1, 15, P3), pixbit(1, 0, 0, 15, P4), pixbit(1, 1, 1, 75, P5), pixbit(1, 2, 0, 15, P6), image(3, 2, [P1, P2, P3, P4, P5, P6], I1).

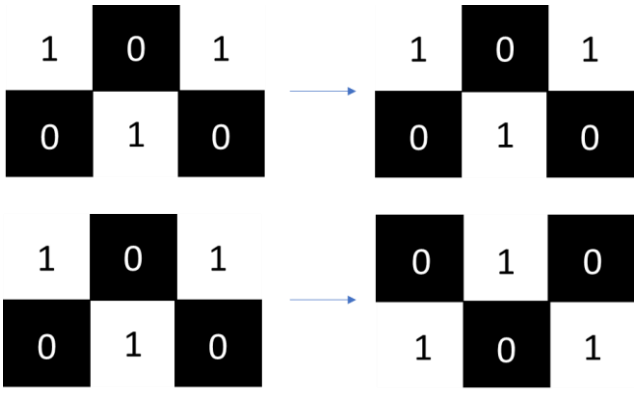
pixrgb(0, 0, 251, 123, 113, 35, P1), pixrgb(0, 1, 245, 234, 89, 80, P2), pixrgb(0, 2, 251, 123, 113, 35, P3), pixrgb(1, 0, 129, 235, 197, 80, P4), pixrgb(1, 1, 219, 249, 238, 80, P5), pixrgb(1, 2, 129, 235, 197, 80, P6), image(3, 2, [P1, P2, P3, P4, P5, P6], I2).

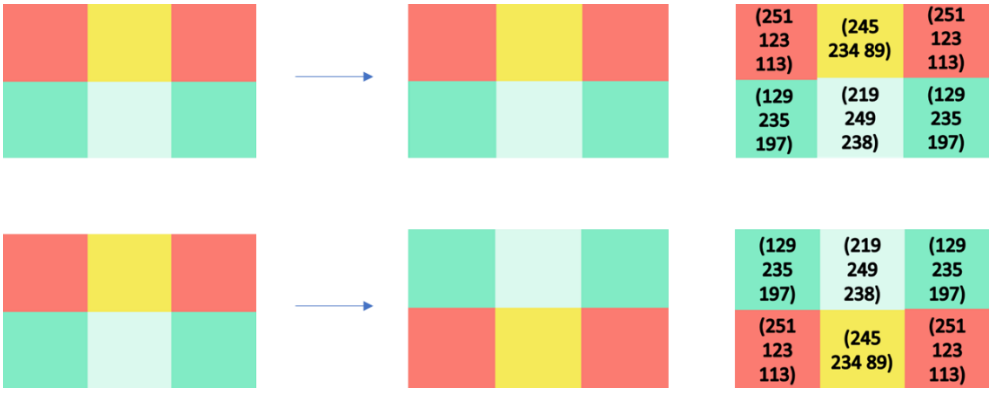
pixhex(0, 0, "#7DC8D5", 10, P1), pixhex(0, 1, "#0070C0", 20, P2), pixhex(0, 2, "#3B3838", 30, P3), pixhex(1, 0, "#E7F771", 10, P4), pixhex(1, 1, "#F4B183", 10, P5), pixhex(1, 2, "#F6B048", 20, P6), pixhex(2, 0, "#DEEBF7", 10, P7), pixhex(2, 1, "#3B3838", 10, P8), pixhex(2, 2, "#C643EF", 10, P9), image(3, 3, [P1, P2, P3, P4, P5, P6, P7, P8, P9], I3).

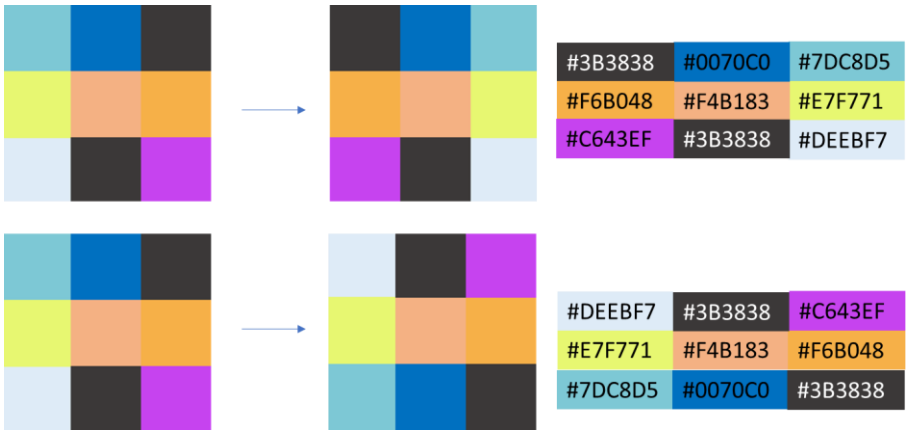




- imageFlipH y imageFlipV:

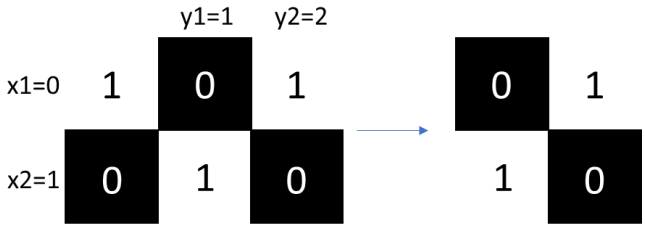




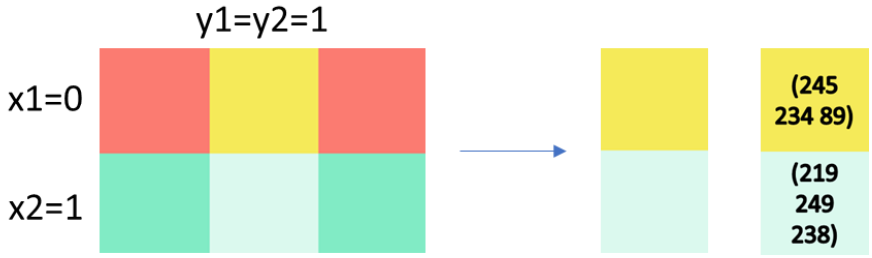


- imageCrop

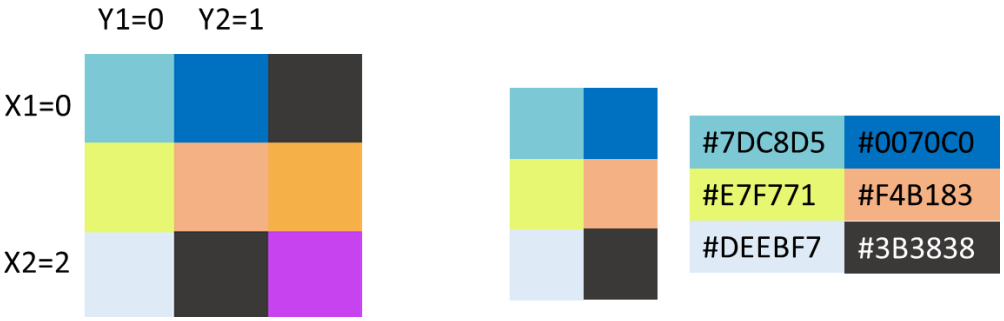
imageCrop(I1, 0, 1, 1, 2, I1crop).



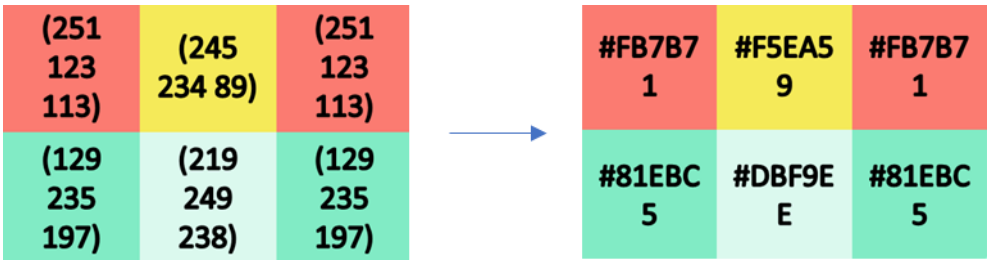
imageCrop(I2, 0, 1, 1, 1,I2crop).



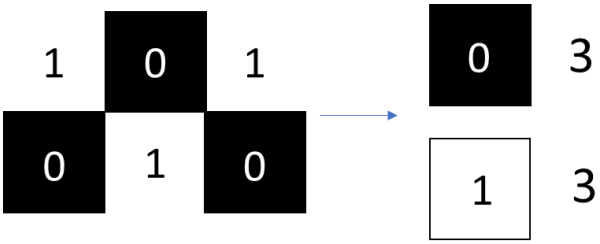
imageCrop(I3, 0, 0, 2, 1, I3crop).

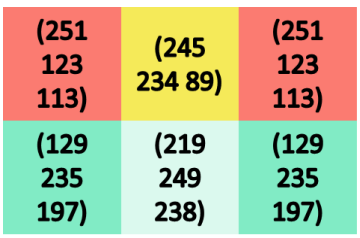


- imageRGBToHex



- imageToHistogram

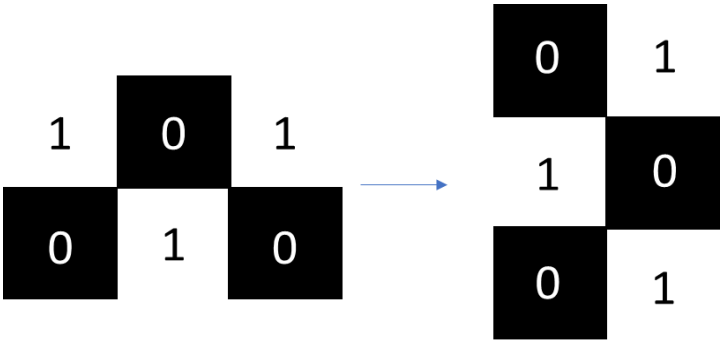


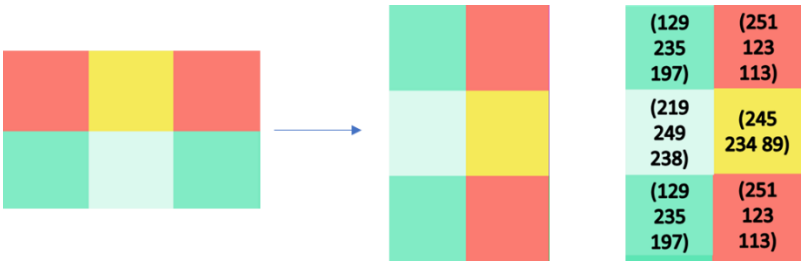


(251 123 113)	2	(129 235 197)	2		
(245 234 89)	1	(219 249 238)	1		

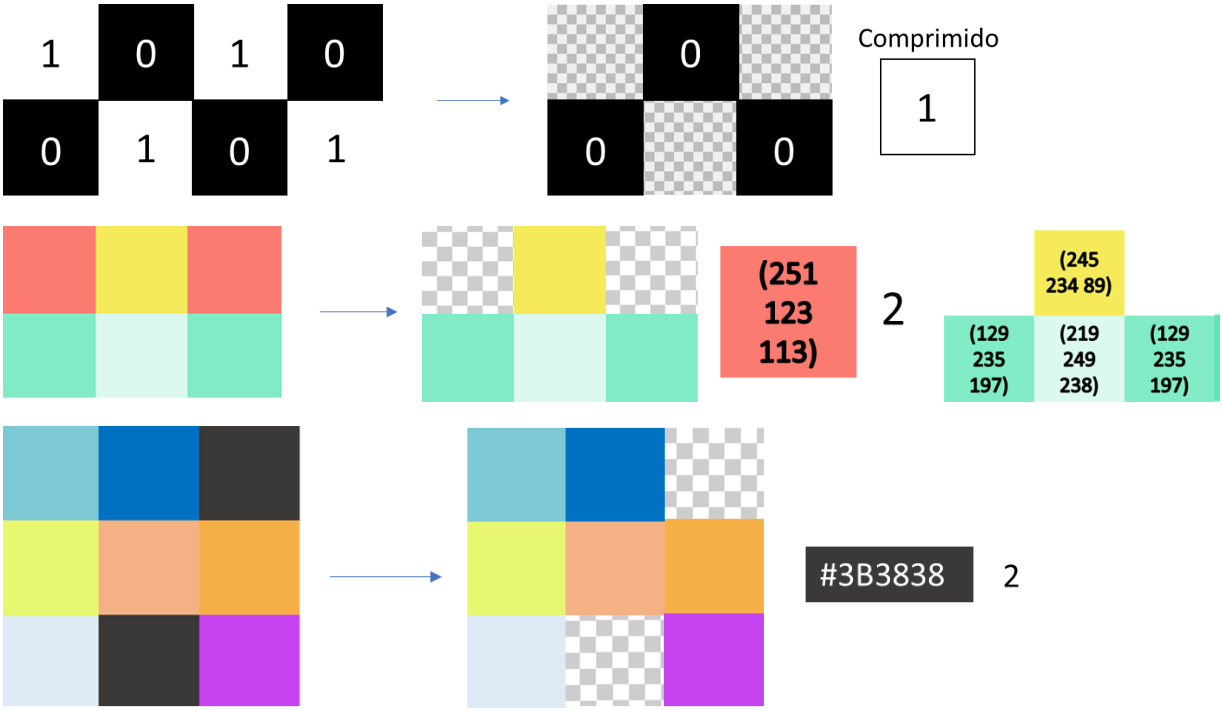
#7DC8D5	1	#0070C0	1	#3B3838	2
#E7F771	1	#F4B183	1	#F6B048	1
#DEEBF7	1			#C643EF	1

- imageRotate90



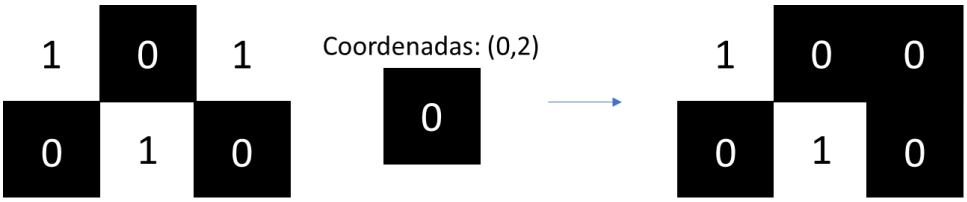


- imageCompress

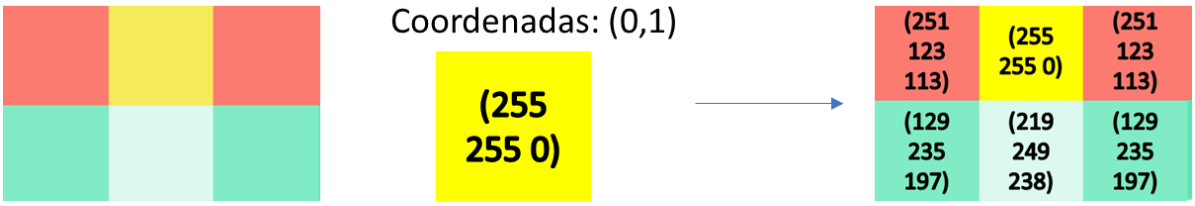


- imageChangePixel:

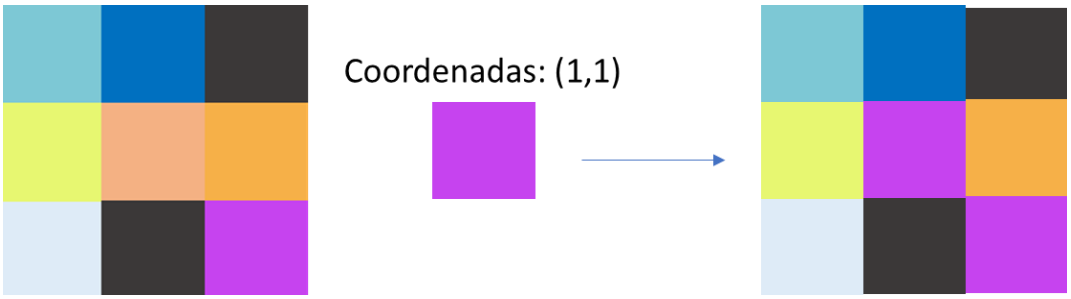
pixbit(0, 2, 0, 80, P3_new), imageChangePixel(I1, P3_new, I1_new).



pixrgb(0, 1, 255, 255, 0, 35, P2_new), imageChangePixel(I2,P2_new,I2_new).

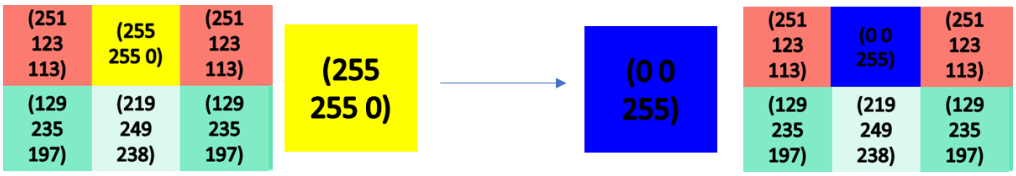


pixhex(1, 1, "#C643EF", 10, P5_new), imageChangePixel(I3,P5_new,I3_new).

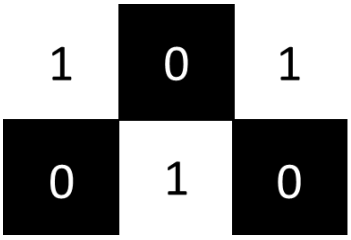


- imageInvertColorRGB:

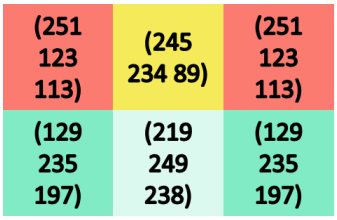
pixrgb(0, 1, 255, 255, 0, 35, P2_new), imageInvertColorRGB(P2_new, P2_new_invert), imageChangePixel(I2,P2_new_invert,I2_newA).



- imageToString



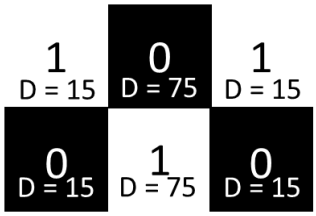
1 0 1
0 1 0

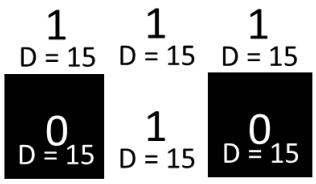


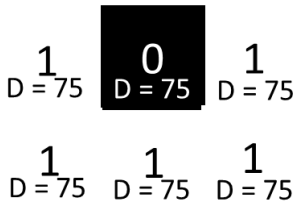
(251 123 113 35) (245 234 89 80) (251 123 113 35)
(129 235 197 80) (219 249 238 80) (129 235 197 80)

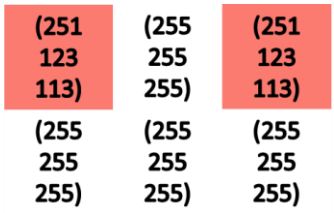
#7DC8D5 #0070C0 #3B3838
#E7F771 #F4B183 #F6B048
#DEEBF7 #3B3838 #C643EF

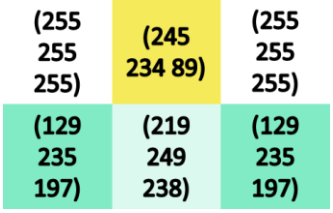
- imageDepthLayers



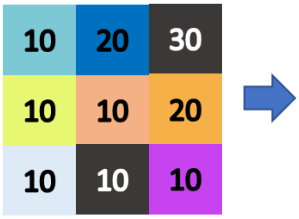


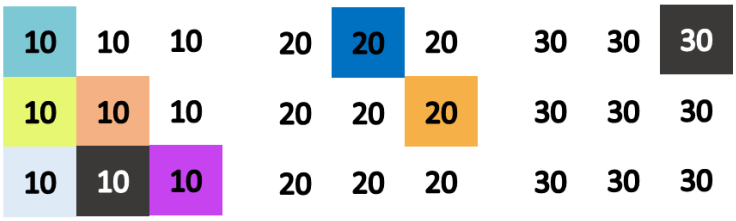






Profundidades:





- imageDecompress