**INFORME DE LABORATORIO 2: SIMULACIÓN DE UN PROGRAMA PARA REALIZAR UN TRATAMIENTO DE IMÁGENES SIMPLIFICADO**



Nombre: Aracely Castro V.

Profesor: Roberto Gonzales I.

Asignatura: Paradigmas de Programación (2/2022)

Índice

[1. INTRODUCCIÓN 3](#_Toc118122682)

[**1.1** **DESCRIPCIÓN DEL PROBLEMA** 3](#_Toc118122683)

[**1.2** **DESCRIPCIÓN DEL PARADIGMA** 3](#_Toc118122684)

[2. DESARROLLO 4](#_Toc118122685)

[**2.1** **ANALISIS DEL PROBLEMA** 4](#_Toc118122686)

[**2.2** **DISEÑO DE LA SOLUCIÓN** 5](#_Toc118122687)

[**2.3** **ASPECTOS DE IMPLEMENTACIÓN** 6](#_Toc118122688)

[**2.3.1** **EJEMPLOS DE USO** 7](#_Toc118122689)

[**2.3.2** **RESULTADOS ESPERADOS** 7](#_Toc118122690)

[**2.3.3 POSIBLES ERRORES** 7](#_Toc118122691)

[**2.4** **Resultados y autoevaluación** 7](#_Toc118122692)

[**2.4.1** **RESULTADOS** 7](#_Toc118122693)

[**2.4.2 AUTOEVALUACIÓN** 7](#_Toc118122694)

[3. CONCLUSIÓN 7](#_Toc118122695)

[4. BIBLIOGRAFÍA Y REFERENCIAS 8](#_Toc118122696)

[5. ANEXOS 8](#_Toc118122697)

# **INTRODUCCIÓN**

Con el objetivo de aplicar conceptos del paradigma de programación lógico usando el lenguaje de programación Prolog en la resolución de un tratamiento de imágenes simple, a continuación, se dará detalle sobre el proceso de desarrollo de la solución al problema a través del compilador SWI-Prolog versión 8.4.3. El informe constará de una breve introducción de cómo surgió el problema, una descripción del paradigma utilizado, el análisis del problema y como fue el diseño de las soluciones para algunas funciones, sus aspectos de implementación, las instrucciones necesarias para compilar el archivo con el script de pruebas junto con ejemplos de algunas funciones, los resultados y la autoevaluación. Finalmente, se dará una conclusión respecto a todo lo anterior.

## **DESCRIPCIÓN DEL PROBLEMA**

Al igual que en el laboratorio uno se busca desarrollar la simulación de un programa para el tratamiento de imágenes de forma simplificada como GIMP y Adobe Photoshop. Este simulador permitiría crear imágenes con distintos formatos de pixeles sobre los cuales se pueden aplicar distintas operaciones como, por ejemplo, recortar una imagen, invertir una imagen, obtener histograma de la imagen, entre otros. Para implementar esto, se deben tener en cuenta que:

Una imagen Pixmap-d es una imagen donde cada uno de sus pixeles contiene información de espacio de colores y profundidad (R)ed, (G)reen, (B)lue y (D)epth. Cada color cubre valores entre 0 y 255 formando en conjunto un color del espectro RGB. La profundidad en cambio ofrece información más detallada de la imagen como espacio tridimensional.

Una imagen Bitmaps-d es una imagen donde sus pixeles tiene asociado su color por un bit, negro como 0 y blanco como 1. Cada píxel también tendría una profundidad asociada.

Una imagen Hexmap-d es similar a una imagen Pixmap-d con la diferencia que los valores RGB son representados en hexadecimal.

## **DESCRIPCIÓN DEL PARADIGMA**

El paradigma lógico forma parte de la familia de los paradigmas declarativos, funciona declarando una base de conocimientos con hechos y reglas en donde se pueden hacer consultas sobre estas. El paradigma lógico cuenta con tres mecanismos básicos que son:

* Unificación: Proceso que consiste en encontrar una asignación de variables que haga idénticas a las fórmulas que se desea unificar.
* Backtracking automático: Estrategia de búsqueda de soluciones en estructuras de árboles, utilizada para encontrar soluciones a una consulta.
* Estructuras de datos basadas en árboles:

Además de contar con otros conceptos como son:

Atomo: son aquellas cosas sobre las que basa el conocimiento que queremos expresar (Se escriben en minúsculas).

Predicado: Los predicados son las cosas que queremos decir. Los resultados o variables van en Mayúsculas.

Cláusulas de Horn o hechos: Tipo de clausuras que

Siendo los hechos un tipo de clausura siempre verdaderas donde se definen 1 o más relaciones entre términos. Una regla similar a un hecho con la excepción de que su veracidad depende de la conjunción de objetivos (empleando el operador lógico AND) y una consulta una pregunta sobre la base de conocimiento que puede entregar true, false o un elemento que satisfaga una consulta para que sea verdadera, en el caso de que se haya introducido una variable.

Cláusulas de Horn o hechos: Cada una de las sentencias se “iguala a” unidades de información de una base de conocimiento. Los hechos y reglas deben terminar con un punto. Además, se debe evitar el Problema de mundo cerrado que se puede generar al negar un predicado o un hecho y a su vez que el resultado que se espere no se haya definido por lo que no es nada de la base de conocimiento, que en teoría podría ser cualquier cosa. Cabe destacar que cuando Prolog da falso a una respuesta, no significa que sea un falso absoluto, si no que cuando pasa esto, significa que se dio el caso en que Prolog no fue capaz de encontrar un hecho y/o resultado que satisfaga la pregunta y por lo tanto, retorna falso.

Para la base de conocimientos, no es necesaria que esta contenga todo un conjunto de información, si no que contenga solo la información necesaria para satisfacer una respuesta a un problema. Muchos problemas computacionales pueden ser expresados en términos de Cláusulas de Horn y resueltos a través del Paradigma Lógico

# **DESARROLLO**

## **ANALISIS DEL PROBLEMA**

Se identifica que para hacer un tratamiento de imágenes simples en Prolog hay que implementar los siguientes elementos:

* **image:** Una imagen como una lista con ancho, largo (int) y lista de pixeles.
* **pixrgb:** Un píxel pixmap-d como una lista con dos coordenadas x e y, colores RGB y profundidad (int).
* **pixrgb\_comprimido:** Un píxel comprimido pixmap-d como una lista con dos coordenadas x e y (int), colores RGB (string) y profundidad (int).
* **pixbit:** Un píxel bitmap-d como una lista con dos coordenadas x e y, bit y profundidad (int).
* **pixbit\_comprimido:** Un píxel comprimido bitmap-d como una lista con dos coordenadas x e y (int), una lista con -1 y bit (int), y la profundidad (int).
* **pixhex:** Un píxel hexmap-d como una lista con dos coordenadas x e y (int), color hexadecimal (string) y la profundidad (int)
* **pixhex\_comprimido:** Un píxel comprimido hexmap-d como una lista con dos coordenadas x e y (int), color hexadecimal (list int) y profundidad (int).

Además de las siguientes operaciones pertenecientes al TDA image:

* **image** (Constructor): Crea una imagen bitmap, hexmap o pixmap.
* **imageIsBitmap, imageIsHexmap, imageIsPixmap, imageIsCompress** (Pertenencia): Verifica si la imagen es bitmap, hexmap, pixmap o si fue comprimida respectivamente.
* **imageFlipH, imageFlipV** (Modificador): invierte los pixeles de una imagen horizontal o verticalmente respectivamente.
* **imageCrop** (Modificador): Recorta una imagen a partir de un cuadrante definido por cuatro puntos y lo convierte en una nueva imagen.
* **imageRGBToHex** (Modificador): Convierte una imagen pixmap a hexmap.
* **imageToHistogram** (Otras funciones): Muestra un histograma de una imagen.
* **imageRotate90** (Modificador): Rota los pixeles de una imagen 90° a la derecha.
* **imageCompress** (Modificador): Comprime el color más frecuente de una imagen.
* **imageChangePixel** (Modificador): Reemplaza un píxel de una imagen por otro nuevo.
* **imageToString** (Otras funciones): Entrega una cadena string de la imagen.
* **imageDepthLayers** (Otras funciones): Entrega una lista de imágenes separados por profundidad.
* **imageToDecompress** (Modificador): Descomprime una imagen comprimida.

Se pide que cada TDA (Tipo de Dato Abstracto) desarrollado se implemente con lo necesario para cumplir con los requerimientos funcionales y que en el archivo principal se contenga las funciones con un script de pruebas para probar los predicados.

## **DISEÑO DE LA SOLUCIÓN**

Por otro lado, se distinguen algunos casos particulares para soluciones de algunos requerimientos funcionales utilizando compilador SWI-Prolog como lo son:

**ImageToHistogram**: Para poder crear el histograma se desarrollaron predicados auxiliares. El primero extraía el color del píxel de la cabeza de la lista de pixeles, el segundo contaba las veces que ese color se repetía en una lista de pixeles, el siguiente modificaba los pixeles para eliminar el color extraído en un principio de la lista y luego ocurría la llamada recursiva. Con base a lo anterior se fue formando la lista con la cantidad y el color, parando el ciclo cuando la lista de pixeles este vacía. Ver la Figura N°X, Figura N°XX y Figura N°XXX en ANEXOS para poder ver la salida de histogram con imágenes Pixmap-d, Bitmap-d y Hexmap-d. P.X

**imageCompress:** Para poder comprimir se pensó en una forma de cambiar la información del píxel más repetido con otro dato distinto, de modo que se pueda descomprimir con la función decompress más tarde. Para el caso de una imagen Bitmap-d se eligió reemplazar el valor del bit más repetido por una lista con -1 y su bit, así para volver a la forma original bastó con comprobar la lista y reemplazar a la forma original en consecuencia. En el caso de una imagen Hexmap-d se eligió reemplazar el píxel repetido por una lista con tres números, cada número representando en valor color rojo, verde y azul del string, con esto para volver a la imagen original se podría recuperar la lista y con base a ella crear el string original para reemplazarlo donde corresponde. Finalmente, para comprimir una imagen Pixmap-d se optó por reemplazar el valor entero del espectro RGB por su equivalente a string hexadecimal y así, para volver a la forma original, bastaría con leer el string de cada color y transformarlo a número antes de colocarlo donde se necesite. Ver la Figura N°X, Figura N°XX y Figura N°XXX en ANEXOS para poder ver la salida de compress con imágenes Pixmap-d, Bitmap-d y Hexmap-d. P.XX

**imageCrop:** Para recortar la imagen se utilizo el predicado auxiliar rangoXY que verificaba si el pixel estaba dentro del intervalo [X1 – X2] y [Y1 – Y2] definido y el predicado crop\_filtro que transformaba la lista de pixeles de forma que el pixel ahora sea una sublista de un valor entero y el pixel, siendo 0 si el valor NO pertenece al rango y 1 en caso de serlo, esto con el objetivo de reutilizar el predicado eliminarElemento utilizado en otros predicados para eliminar los pixeles de la lista de pixeles. Luego la lista resultante entro al predicado crop\_formato que modifica las coordenadas x e y del píxel de modo que comiencen desde el (0,0) hasta el (X2-X1, Y2-Y1) antes de ser ingresados en la nueva imagen. Ver la Figura N°X en ANEXOS para poder la transformación en imageCrop de una imagen 2x2. P.X

## **ASPECTOS DE IMPLEMENTACIÓN**

El compilador utilizado fue Prolog versión 8.4.3, no se utilizaron bibliotecas externas de ningún tipo, esto para pensar las soluciones a los requerimientos funcionales usando solamente el paradigma lógico y los predicados básicos de Prolog. Mientras se desarrollaban los requerimientos funcionales, se crearon los siguientes TDAs cuya estructura respecto el orden de representación, constructor, pertenencia, selectores, modificadores y otras funciones:

* **TDA image:** Corresponde a una imagen. Representación: (int x int x list). Ver Tabla N°1 en ANEXOS para ver su implementación, P.XX.
* **TDA pixbit:** Corresponde a un píxel bitmap. Representación: (int x int x bit ([0|1]) x int). Ver Tabla N°2 en ANEXOS para ver su implementación, P.XX.
* **TDA pixbit\_comprimido:** Corresponde a un píxel bitmap comprimido. Representación: (int X int X (-1, bit([0|1])) x int). Ver tabla N°3 en ANEXOS para ver su implementación, P.XX.
* **TDA pixhex:** Corresponde a un píxel hexmap. Representación: (int x int x string x int). Ver Tabla N°4 en ANEXOS para ver su implementación, P.XX.
* **TDA pixhex\_comprimido:** Corresponde a un píxel hexmap comprimido. Representación: (int x int x (int X int X int) x int). Ver Tabla N°5 en ANEXOS para ver su implementación, P.XX.
* **TDA pixrgb:** Corresponde a un píxel pixmap. Representación: (int x int x int x int x int x int). Ver Tabla N°6 en ANEXOS para ver su implementación, P.XX.
* **TDA pixrgb\_comprimido:** Corresponde a un píxel pixmap. Representación: (int x int x string x string x string x int). Ver Tabla N°7 en ANEXOS para ver su implementación, P.XX.

Ver la Figura N°X en ANEXOS para poder ver en que archivos se importan otros archivos. P.XX.

### **EJEMPLOS DE USO**

Para comenzar, se debe verificar que se tengan todos los archivos TDAs en una misma carpeta, de lo contrario el archivo pruebas\_21090869\_CastroVenegas no se ejecutará al compilar el programa con “Run” tras compilar. Luego de ejecutado se podrán ver los ejemplos de cada una de las operaciones en el archivo como script de pruebas. Se recomienda antes de consultar utilizar el comando “set\_prolog\_flag(answer\_write\_options,[max\_depth(0)]).” Para poder ver todos los pixeles de la imagen y no su forma medio visible. Unas observaciones son que cuando utilices una consulta del script hay que verificar que los píxeles en la imagen sean correctos y que las coordenadas comiencen dese (0,0) hasta (n,m), todo esto para que no surja un error al aplicar un predicado a la imagen. También hay que asegurarse que los argumentos de los predicados estén bien ingresados. Por ejemplo, se podrá aplicar la función imageInvertColorRGB a un píxel pixmap, pero si se desea aplicar a una imagen puede modificar un píxel de está utilizando el predicado imageChangePixel, pero para ello necesita ingresar como entrada la imagen original, el píxel a modificar y una variable en donde colocar la imagen resultante (en ese orden).

Ver Figura X en ANEXOS para ver ejemplos claros de ejecución de algunas funciones. P.XX

### **RESULTADOS ESPERADOS**

Se espera que se haya hecho un simulador de tratamiento de imágenes simple donde cada predicado no cause errores o arroje false cuando no debería y que compile el archivo con el script de pruebas correctamente.

### **2.3.3 POSIBLES ERRORES**

## **Resultados y autoevaluación**

### **RESULTADOS**

Tras consultar en consola, se confirmo que los predicados funcionan correctamente y se obtienen los resultados esperados. Se logró crear cada uno de los predicados obligatorios y opcionales verificando distintos casos.

### **2.4.2 AUTOEVALUACIÓN**

La Autoevaluación se realiza de la siguiente forma: 0: No realizado – 0.25: Funciona 25% de las veces – 0.5: Funciona 50% de las veces 0.75: Funciona 75% de las veces – 1: Funciona 100% de las veces. Para ver la tabla de Autoevaluación, ver la Tabla XX del anexo XX

Tras mejorar la implementación de los requerimientos funcionales no se encontró errores, por lo que se considera de que funcionan el 100% de las veces.

# **CONCLUSIÓN**

Tras haber trabajado en implementar los requerimientos funcionales para un simulador de tratamiento de imágenes simple, ser puede decir que se cumplió el objetivo de aplicar conceptos del paradigma lógico usando el lenguaje de programación Prolog. Las complicaciones que hubo durante el desarrollo de las funciones fueron el tener que mover las funciones hechas en SWISH (Prolog online) SWIP-Prolog en un comienzo, también resulto desafiante lograr filtrar elementos de una lista correctamente sin crear variables anónimas y lograr que un predicado unificará como se esperaba. Por otra parte, no hubo complicaciones en cuanto fue el uso de GitHup y Prolog a la hora de desarrollar el trabajo, señalando un logró en comprender conceptos como hechos, reglas y recursividad, etc. Finalmente, en comparación con el paradigma funcional desarrollado en el laboratorio uno, se observaron menos líneas de código y menos problemas a la hora de pensar como desarrollar una idea. A diferencia de Scheme en Prolog podrías utilizar un predicado de más de una manera y no solo una, ejemplo de esto sería la posibilidad de utilizar image como constructor, selector y modificador a la vez, algo que no se hizo finalmente porque se optó por separarlos, también como se podrían utilizar variables fue más fácil a la hora de recuperar datos para un predicado. Finalmente, se puede decir que la experiencia de este segundo laboratorio junto con el primero pueda servir de apoyo para próximo.

# **BIBLIOGRAFÍA Y REFERENCIAS**

# **ANEXOS**