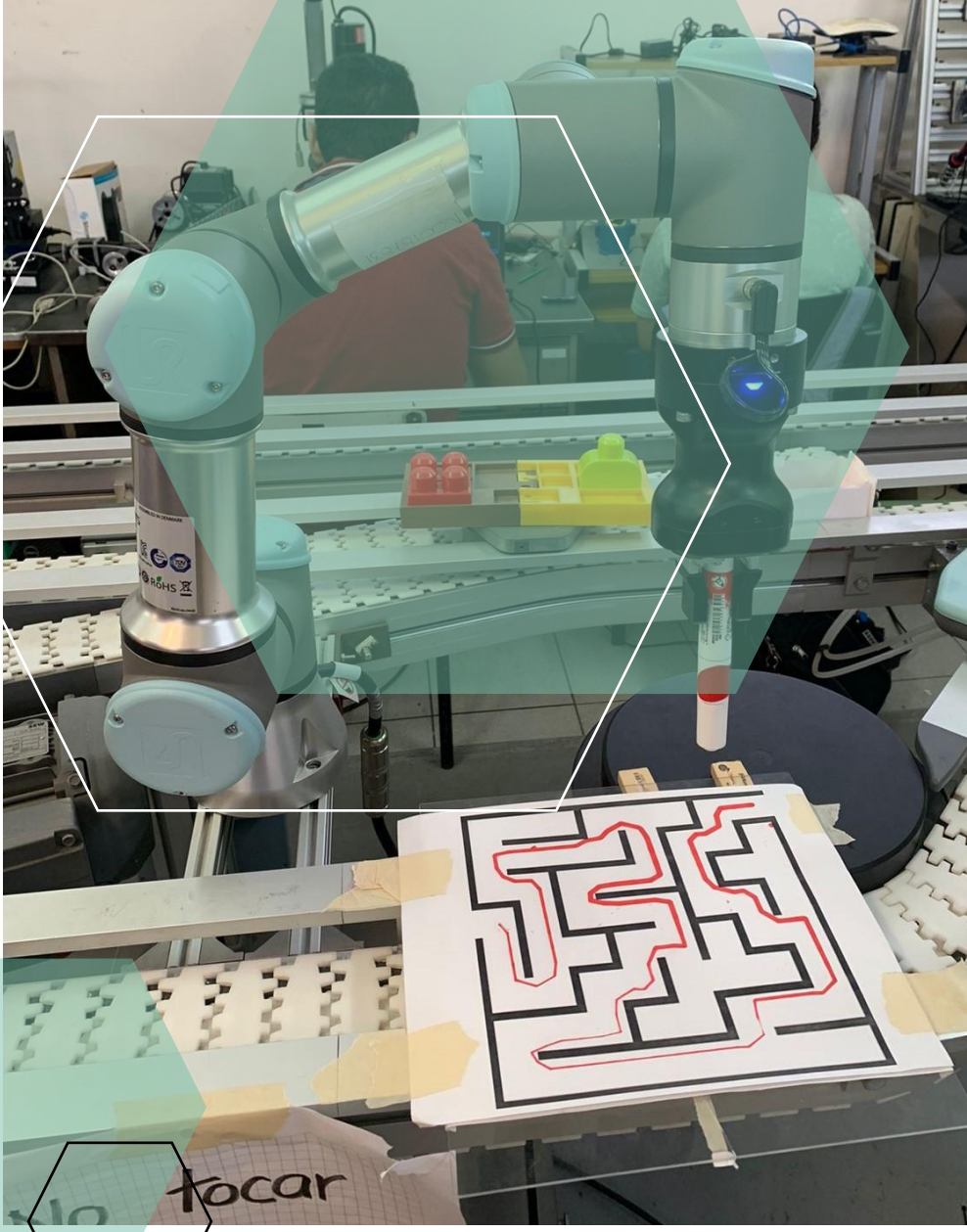
DOCUMENTO DESCRIPTIVO

SOLUCIONADOR DE LABERINTOS



INTRODUCCIÓN A LA INTELIGENCIA ARTIFICIAL

Catedrático

Dr. Luis Felipe Marín Urias.

Universidad Veracruzana

RESOLUCIÓN DE LABERINTOS POR MEDIO DE AGENTES INTELIGENTES Y UN ROBOT MANIPULADOR

INTEGRANTES:

* Cerecero Amador María Cristina.
* Salgado Gómez Kevin.
* Sosa Guzmán Mariana.
* Ruíz Ríos Eduardo.

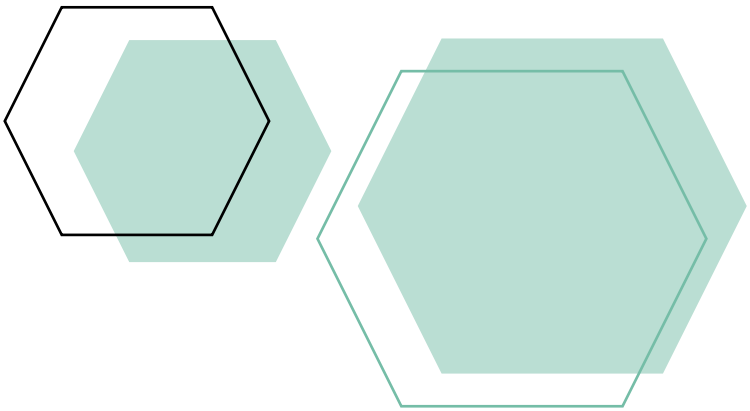
CONTENIDO

INTRODUCCIÓN 3

CÓDIGO FUENTE..... 3

COMPARACIÓN DE RESULTADOS 19

DIAGRAMA DE BLOQUES DEL PROYECTO 20



INTRODUCCIÓN

Este proyecto es un conjunto de algoritmos que integran un solucionador de laberintos a nivel software y hardware, debido a la implementación física del mismo con el uso de un robot manipulador UR3e de 6 grados de libertad. Se implementaron dos algoritmos de búsqueda **a estrella (a*)** y **greedy**, así como también se incluyo un algoritmo de tratamiento de imágenes.

CÓDIGO FUENTE

THICKER BORDERS

Se creó el archivo **"thicker_borders.py"** para procesamiento de imágenes, para usar como máscara con el fin de engrosar los bordes de la imagen del laberinto puesto que el algoritmo busca el camino más corto y lo traza pegado a los bordes, por el tamaño de los pixeles da la impresión de estar tocando los bordes, por lo que para solucionar esta problemática se ensancharon los bordes del laberinto para que el camino propuesto esté lo más centrado posible.

Se hizo uso de las librerías:

- **numpy** puesto que a través de matrices manipulamos la imagen.
- **Opencv** cuya función en este caso es desplegar la imagen ya engrosada.
- **PIL**, Python Image Library nos auxilia para hacer la manipulación de la imagen.

```
   # ALGORITMO QUE GENERA LA IMAGEN CON GROSOR

import numpy as np
import cv2
from PIL import Image
```

Agregamos al path_ (camino) la ubicación de la imagen del laberinto, leemos la imagen y cambiamos su formato a RGB (red, green, blue). La función que se lleva a cabo dentro del ciclo for es binarizar la imagen con números de 255 o 0, dependiendo si el píxel es blanco o negro respectivamente, convirtiendo la escala de grises a binario.

```
path_ = 'img/borderless.png'
img_pil3 = Image.open(path_)

# Imagen binarizada formato RGB
picture = Image.new('RGB', img_pil3.size)
for i in range(img_pil3.size[0]):
    for j in range(img_pil3.size[1]):
        color = img_pil3.getpixel((i, j))
        if color < (100, 100, 100):
            picture.putpixel((i, j), (0, 0, 0))
        else:
            picture.putpixel((i, j), (255, 255, 255))
```

Cada pixel está binarizado en RGB es decir que contiene una tupla de 3 valores, por lo tanto, en este ciclo for convertimos esa tupla a un solo valor entero, entonces [0 0 0]= 0 (negro) y [255 255 255]=1 (blanco). Sacamos una copia para no alterar la imagen original.

```
# Imagen binarizada 0-1
img_np_binary = np.zeros(picture.size)
for i in range(picture.size[0]):
    for j in range(picture.size[1]):
        color = picture.getpixel((i, j))
        if color == (0, 0, 0):
            img_np_binary[i][j] = 0
        if color == (255, 255, 255):
            img_np_binary[i][j] = 1

img_np_binary_copy = img_np_binary[:]
```

Se crea la lista “black_points” y en ella agregamos todas las posiciones de pixeles negros.

```
black_points = []
for i in range(img_np_binary_copy.shape[0]):
    for j in range(img_np_binary_copy.shape[1]):
        if img_np_binary_copy[i][j] == 0:
            black_points.append((i, j))
```

Thick es el grosor de las paredes negras, le asignamos el número de pixeles de grosor en este caso es de 17 pixeles. Ahora evaluamos y buscamos si es posible ensanchar los bordes y a que dirección, por ejemplo, si junto a un pixel negro hay un pixel blanco y del otro lado hay otro píxel negro, en el lado del pixel blanco se agregarían 17 pixeles negros, en otras palabras, se agregaría el ensanchamiento de bordes de acuerdo al grosor definido en Thick.

```
THICK = 17 # pixeles de grosor
for bp in black_points:
    x, y = bp
    # si el de la izquierda es negro pero el de la derecha no
    if (x-1, y) in black_points and (x+1, y) not in black_points:
        # poner los siguientes THICK a la derecha de negro
        img_np_binary_copy[x: x+THICK+1, y] = 0
        # for i in range(x, x+THICK+1):
        #     img_np_binary_copy[i, y] = 0
    # si el de la derecha es negro px-THICK: x, yero el de la izquierda no
    elif (x+1, y) in black_points and (x-1, y) not in black_points:
        # poner los sigueintes THICK a la izquierda de blanco
        img_np_binary_copy[x-THICK: x, y] = 0
        # for i in range(x-THICK, x+1):
        #     img_np_binary_copy[i, y] = 0
    # elif (x+1, y) not in black_points and (x-1, y) not in black_points:
    #     img_np_binary_copy[x-THICK: x+THICK+1, y] = 0
    # elif el de arriba es negro pero el de abajo no
    elif (x, y+1) in black_points and (x, y-1) not in black_points:
        # poner el de abajo como negro
        img_np_binary_copy[x, y: y+THICK+1] = 0
```

```
    # elif el de abajo es negro pero el de arriba no
    elif (x, y-1) in black_points and (x, y+1) not in black_points:
        img_np_binary_copy[x, y-THICK: y] = 0
```

Por último, guardamos la imagen binarizada con números enteros y bordes ya ensanchados.

```
GUARDAR = True
if GUARDAR:
    im = Image.new('1', img_np_binary_copy.shape)
    pixels = im.load()
    for i in range(im.size[0]):
        for j in range(im.size[1]):
            if img_np_binary_copy[i][j].astype(np.int64) == 1:
                pixels[i, j] = 1
            else:
                pixels[i, j] = 0
    im.save('thicker_img.png')
else:
    cv2.imshow('thicker', img_np_binary_copy)
    cv2.waitKey(0)
```

MAIN A STAR (PROGRAMA PRINCIPAL)

Se implemento el algoritmo **"main.py"** este es el programa principal para la solución del laberinto. En este proyecto se implementaron 2 algoritmos de búsqueda **a estrella (a*)** y **greedy,** el proceso para calcular sus heurísticas es bastante similar. Comenzaremos explicando el "main_a_star.py" programa principal por el método de búsqueda de **a estrella (a*).**

Importamos la librerías y recursos necesarios, incluido el programa "image_process.py" el cual carga la imagen a utilizar.

```
from cell import Cell
from a_star import a_star_search, print_labyrinth, labyrinth
import numpy as np
import image_process as img
```

IMAGE PROCESS

Image_process.py

```
import cv2 as cv
import numpy as np

def load_image(fileName: str):
    """
    Read and load an image in grayscale
    """
    # read image in grayscale
    gray_img = cv.imread(fileName, cv.IMREAD_GRAYSCALE)

    return gray_img

def binarize_image(img, threshold: int=100, maxVal: int=255):
    """
    Perform the binary thresholding process to each pixel of the image.

    Parameters
    ----------
    * `img`: a mat as returned by `cv2.imread()`
    """
    _, bin_img = cv.threshold(img, threshold, maxVal, cv.THRESH_BINARY)

    return bin_img

def display(windowName: str, img):
    cv.imshow(windowName, img)
    cv.waitKey(0)
```

Se inicializa el archivo en el que se guardarán todas las coordenadas de los puntos de paso del laberinto en pixeles. Se crea la función main en la que se declaran el punto inicial y final, y se establecen sus coordenadas en (y,x).

```
OFFSET = 10
file = open('coordenadas.txt', 'w')

def main():
    """Main Function"""

    start_cell = Cell(position=(165, 20)) # 63, 12

    goal_cell = Cell(position=(25, 360)) # 8, 134

    print()
```

Posteriormente, se manda a llamar al programa **"a_star.py"** el cual contiene todo el algoritmo de búsqueda, **a* utiliza la heurística desde donde se encuentra el nodo cell actual hasta el punto de inicio (g(n)) y de igual forma desde el nodo cell actual hasta el punto final (h(n)) por lo tanto, f(n)=g(n)+h(n).**

Importamos la librerías y recursos necesarios. Empezamos cargando la imagen original y la imagen modificada con los bordes ensanchados, para obtener la nueva imagen binarizada con valores de 0 y 255.

```
from cell import Cell
from collections import deque
import image_process as img

# inicialización
# carga la imagen
original_labyrith = img.load_image("img/borderless.png")
thick_labyrinth = img.load_image("img/thicker_borders.png")

# aquí obtenemos la imagen del laberinto binarizada (0/255)
labyrinth = img.binarize_image(original_labyrith)
```

De acuerdo con las dimensiones de la imagen, obtenemos sus filas y columnas.

```
ROWS = thick_labyrinth.shape[0] # la shape es (filas, columnas)
COLUMNS = thick_labyrinth.shape[1]

B = 0 # bloqueado
F = 255 # libre/desbloqueado
```

Como un auxiliar durante la creación del código imprimimos el estado de la matriz para ver su comportamiento, a través de la función “print_labyrinth”.

```
def print_labyrinth(lab):
    for row in lab:
        for column in row:
            print(column, end=" ")
        print()
```

El algoritmo a* empieza su funcionamiento, para esto se declara el punto de inicio y el punto de fin en el programa main (como se mostró antes y como se muestra a continuación).

```
  path = a_star_search(start_cell, goal_cell)

    print()
```

Regresando al programa “a_star.py”, se realiza una serie de validaciones o filtros en la función “a_star_search”. Primero corroboramos que el punto de inicio este dentro del tamaño de la imagen, si lo está entonces el punto de inicio es válido, se realiza la misma validación para el punto final de destino. Posteriormente, verificamos si ambas celdas están bloqueadas (ósea que sean 0), si lo están entonces se trata de un borde. Después evaluamos si la celda inicial y final son la misma, si es así significa que llegamos al final.

```
def a_star_search(start: Cell, dst: Cell):
    # validaciones
    if not is_valid(start.row, start.column):
        print("Start cell is invalid")
        return []

    if not is_valid(dst.row, dst.column):
        print("Destionation cell is invalid")
        return []

    if is_blocked(start.row, start.column) or is_blocked(dst.row, dst.column):
        print("Start or destionation cell are blocked!")
        return []

    if is_destination(start.row, start.column, dst):
        print("We are in goal cell")
        return start.get_path()
```

Sí ninguna de estas condiciones se cumplió entonces empezamos con la búsqueda por a* como tal.

La **heurística** es la métrica de que tan acertada es la búsqueda que se está realizando, no todas las heurísticas están relacionadas a la búsqueda de distancias en este caso se trata del mapa de un laberinto y el objetivo del agente es encontrar el camino más corto y eficiente que nos lleve del punto inicial al punto final, es por esto que se utiliza la Manhattan distance y puesto que el laberinto está limitado a moverse en 4 direcciones (arriba, abajo, izquierda y derecha).

Se crean dos listas, la open_list contendrá a los nodos hijos que son candidatos a ser visitados y la closed_list contendrá a los ya visitados. Al mismo tiempo tendremos una matriz booleana del tamaño de la imagen, en la que se registra para cada celda TRUE si la celda ya fue visitada y FALSE para las no visitadas, de esta forma se clasifican las celdas en su lista correspondiente.

Se crea otra matriz del tamaño de la imagen, cada elemento es un objeto “cell” y se inicializa con su posición (fila,columna), como un árbol cada nodo tiene una referencia a otro nodo celda pues es como se construye el camino, cuando se llegue al punto final se va a regresar al punto inicial a través de dichos nodos de referencia y de esta forma obtendremos el camino. El nodo inicial es el primero en agregarse a la lista de nodos por visitar (open_list).

```
    open_list = deque() # lista doblemente ligada, su usará como cola (FIFO) para
almacenar las celdas

    # Matriz booleana para saber que celdas se han visitado
    # Se inicializa a False indicando que no se ha visitado ni uno
    closed_list = []
    for _ in range(ROWS):
        closed_list.append([False for _ in range(COLUMNS)])

    cell_details = []
    for row in range(ROWS):
        cell_details.append([Cell(position=(row, col)) for col in range(COLUMNS)])

    open_list.append(start) # agrega al final

    found_dst = False
```

La bandera “found_dst” es el indicador de haber encontrado el destino si esta es false significa que aún no ha sido encontrado y se continua con el algoritmo de busqueda.

Dentro del ciclo while corroboramos si los hijos están dentro de la imagen y si no son el destino. Si para un hijo dado se cumplen estas condiciones entonces podemos valorar si dicho nodo hijo ya se visitó y a la vez si no está bloqueado (ósea que se trate de un borde), si ambas condiciones son falsas entonces calculamos las variables **g(n), h(n) y f(n) (previamente inicializadas en 0)** de la heurística apoyándonos del “cell_details” que es la imagen representada en forma de greed con objetos tipo celda.

Después de haber pasado por los filtros de si el nodo hijo es válido, no esta bloqueado y no es el destino, entonces verificamos si la **f(n)** del hijo = 0, o si la f(n) del hijo menor que la del padre, después de haber pasado por estas 6 validaciones ese nodo hijo es considerado para ser visitado y se agrega a la open_list de nodos por visitar.

```python
while len(open_list) != 0:
        current_cell = open_list.popleft() # elimina por la izquierda en O(1)
        row, column = current_cell.row, current_cell.column
        closed_list[row][column] = True

        f_new = g_new = h_new = 0
        # generacion de sucesores
        if is_valid(row-1, column):
            # generar arriba
            if is_destination(row-1, column, dst):
                cell_details[row-1][column].parent = current_cell
                print("We are at destination")
                found_dst = True
                break
            elif closed_list[row-1][column] == False and not is_blocked(row-1, column):
                g_new = cell_details[row][column].g + 1
                h_new = current_cell.manhattan_distance(dst)
                f_new = g_new + h_new

                if cell_details[row-1][column].f == 0 or cell_details[row-1][column].f > f_new:
                    open_list.append(Cell(position=(row-1, column), parent=current_cell))
                    successor_cell = cell_details[row-1][column]
                    successor_cell.f = f_new
                    successor_cell.g = g_new
                    successor_cell.h = h_new
                    successor_cell.parent = current_cell
```

Esta última parte del Código se hizo para validar la generación de hijos hacia arriba, se utiliza la misma lógica para generar hijos en las otras 3 direcciones, solo se modifica por el hijo que se pregunta: **arriba** =(row-1, column), **abajo**=(row+1, column), **izquierda**=(row, column-1), **derecha**=(row, columna+1).

```python
if is_valid(row, column+1):
            # genera derecha
            if is_destination(row, column+1, dst):
                cell_details[row][column+1].parent = current_cell
                print("We are at destination")
                found_dst = True
                break
            elif closed_list[row][column+1] == False and not is_blocked(row, column+1):
                g_new = cell_details[row][column].g + 1
                h_new = current_cell.manhattan_distance(dst)
                f_new = g_new + h_new

                if cell_details[row][column+1].f == 0 or cell_details[row][column+1].f > f_new:
                    open_list.append(Cell(position=(row, column+1), parent=current_cell))
                    cell_details[row][column+1].f = f_new
                    cell_details[row][column+1].g = g_new
                    cell_details[row][column+1].h = h_new
                    cell_details[row][column+1].parent = current_cell

        if is_valid(row+1, column):
            # genera abajo
            if is_destination(row+1, column, dst):
                cell_details[row+1][column].parent = current_cell
                print("We are at destination")
                found_dst = True
                break
            elif closed_list[row+1][column] == False and not is_blocked(row+1, column):
                g_new = cell_details[row][column].g + 1
                h_new = current_cell.manhattan_distance(dst)
                f_new = g_new + h_new

                if cell_details[row+1][column].f == 0 or cell_details[row+1][column].f > f_new:
                    open_list.append(Cell(position=(row+1, column), parent=current_cell))
                    cell_details[row+1][column].f = f_new
                    cell_details[row+1][column].g = g_new
                    cell_details[row+1][column].h = h_new
                    cell_details[row+1][column].parent = current_cell

        if is_valid(row, column-1):
            # genera izquierda
            if is_destination(row, column-1, dst):
                cell_details[row][column-1].parent = current_cell
                print("We are at destination")
                found_dst = True
                break
            elif closed_list[row][column-1] == False and not is_blocked(row, column-1):
                g_new = cell_details[row][column].g + 1
                h_new = current_cell.manhattan_distance(dst)
                f_new = g_new + h_new

                if cell_details[row][column-1].f == 0 or cell_details[row][column-1].f > f_new:
                    open_list.append(Cell(position=(row, column-1), parent=current_cell))
                    cell_details[row][column-1].f = f_new
                    cell_details[row][column-1].g = g_new
                    cell_details[row][column-1].h = h_new
                    cell_details[row][column-1].parent = current_cell

```

Cuando termina entra a la validación de si es el punto final, ese hijo se vincula con su padre, con eso ya se tienen unidos los nodos que forman parte del camino.

En caso de no llegar al punto destino, el ciclo while se rompe, y found_dst es falso, de haber sido encontrado entonces sería verdadero. Finalmente, en cell_details vamos a acceder a la posición del hijo que es candidato porque ya pasó las validaciones y está dentro del open_list.

```
1
2       # limpieza
3       open_list.clear()
4       closed_list.clear()
5       cell_details.clear()
6
7       if not found_dst:
8           print("Failed to find path to destionation cell")
9           return current_cell.get_path()
10
11
12      return current_cell.get_path() #RETORNA
13
14  def is_destination(row: int, column: int, goal: Cell):
15      return row == goal.row and column == goal.column
16
17  def is_valid(row: int, column: int):
18      return (row >= 0 and row < ROWS) and (column >= 0 and column < COLUMNS)
19
20  def is_blocked(row: int, column: int):
21      return thick_labyrinth[row][column] == B
```

El programa **“Cell.py“** es el nodo que guarda el estado de la celda, su posición coordenada en pixeles, la heurística h(n), la g(n) y la suma de ambas f(n).

```
1   class Cell():
2       def __init__(self, *, position, parent=None):
3           self.row, self.column = position
4           self.parent = parent
5           self.successors = []
6           self.f = 0 # f(n)
7           self.g = 0 # g(n)
8           self.h = 0 # h(n)
9
10      def __repr__(self) -> str:
11          return f'{self.column}, {self.row}'
12
13      def manhattan_distance(self, goal):
14          h = abs(self.row - goal.row) + abs(self.column - goal.column)
15          return h
16
17      def get_path(self):
18          path = []
19          path.append(self)
20          father = self.parent
21          while father is not None:
22              path.append(father)
23              father = father.parent
24
25          return path[::-1] #Como empezamos en el ultimo nodo, lo que hará es darnoslo del inicio al fin
```

Para recuperar la solución del camino encontrado, creamos una función llamada “get_path” para que guarde los nodos hijos del camino y se retornen en una lista llamada “path=[]”.

En el código “main_a_star.py” se itera la lista que contiene el camino con nodos para tomar en cuenta un punto cada 20 con la finalidad de reducir la cantidad de datos, pues se obtienen aproximadamente 2 mil puntos (para este caso específico), sin embargo, el robot manipulador no es capaz de procesar esa cantidad de datos.

```
assert path is not None
    _labyrinth = labyrinth.copy().astype(np.uint8)
    for index in range(0, len(path), 20):
        _cell = path[index]
        #print(_cell)
        if _labyrinth[_cell.row - 1][_cell.column + 8] == 0 or _labyrinth[_cell.row - 1][_cell.column - 8] == 0:
            _cell.row += OFFSET
        if _labyrinth[_cell.row + 1][_cell.column + 8] == 0 or _labyrinth[_cell.row + 1][_cell.column - 8] == 0:
            _cell.row -= OFFSET
        #Se hicieron para despegar los puntos del camino de los bordes para que en fisico el plumon no toque el borde
        # es el de arriba un obstaculo?
        if _labyrinth[_cell.row - 1][_cell.column] == 0:
            _cell.row += OFFSET
        # es el de abajo un obstaculo?
        if _labyrinth[_cell.row + 1][_cell.column] == 0:
            _cell.row -= OFFSET
        # es el de la izquierda un obstaculo?
        if _labyrinth[_cell.row][_cell.column - 1] == 0:
            _cell.column += OFFSET
        # es el de la derecha un obstaculo?
        if _labyrinth[_cell.row][_cell.column + 1] == 0:
            _cell.column -= OFFSET
        _labyrinth[_cell.row][_cell.column] = 127

        print(_cell, file=file)
    #Para diferenciar cada nodo del camino se cambian a color gris oscuro
    if path != []:
        _labyrinth[goal_cell.row][goal_cell.column] = 127

    file.close()
    # print_labyrinth(labyrinth)
    img.display("Laberinto :D", _labyrinth)

if __name__ == "__main__":
    main()
```

GET CARTESIAN

Una vez obtenido las coordenadas del camino en pixeles, deberán ser transformadas a coordenadas cartesianas en centímetros respecto al origen del laberinto, esto se hace por medio del algoritmo “get_cartesian.py”.

```python
import image_process as img

original_labyrith = img.load_image("img/borderless.png")

WIDTH = 20.7
HIGH =20.7
PIXEL_WIDTH = original_labyrith.shape[0]
PIXEL_HIGH = original_labyrith.shape[1]

def pixel_to_cartesian(point:tuple):
    # para este caso la imagen es de 375 x 375 pixeles
    # 375 pixeles representan 20.7cm
    pixel_x, pixel_y = point
    cart_x = round(pixel_x * WIDTH / PIXEL_WIDTH, 2)
    cart_y = round(pixel_y * HIGH / PIXEL_HIGH, 2)
    return cart_x, cart_y

file = open('cartesian_path.txt', 'w')

with open('coordenadas.txt', 'r') as f:
    w = input('Ingresa el ancho: (default es 20.7): ')
    h = input('Ingresa el alto: (default es 20.7): ')

    try:
        w = float(w) # si la conversion es posible, usala
        h = float(h)
        WIDTH = w
        HIGH = h
    except:
        pass

    for line in f:
        x, y = [int(data) for data in line.split(', ')]
        cart_x, cart_y = pixel_to_cartesian((x, y))
        print('{}, {}'.format(cart_x, cart_y), file=file)

file.close()
```

Lo siguiente es transformar los puntos de paso cartesianos con respecto al laberinto obtenidos con “get_cartesian.py” a puntos con respecto a la base del robot UR3e, para ser usados por el robot y así generar la trayectoria. Convertimos de cm a metros y corroboramos que el robot se encuentre en la posición del pixel 0.

```python
UR3e usamos en Metros, colocamos el robot en la posición del pixel 0
file = open('ur3e_path.txt', 'w')
#Convertimos el origen del laberinto a metros
ORIGEN_X =  142.62 / 1000 # mm -> m
ORIGEN_Y = -153.30 / 1000 # mm -> m
#Creamos un arreglo puesto que el URScript trabaja con arreglos
array = []
#Abrimos el archivo de text que me generó get_cartesian.py para leer cada punto y convertirlo con referencia del cartesiano del laberinto al del robot
with open('cartesian_path.txt', 'r') as f:
    for line in f:
        x, y = [float(data) for data in line.split(', ')]
        cart_x = x / 100 + ORIGEN_X
        cart_y = - y / 100 + ORIGEN_Y

        point = [round(cart_x, 4), round(cart_y, 4)]
        array.append(point)
        print(point, file=file, end=", \n")
    #print(array, sep='\n', file=file)
    print(len(array))   #Lo guarda en el archivo ur3e_path.txt
file.close()
```

GREEDY

En caso de usar el **método de busqueda "Greedy"** sucede exactamente los mismo que con A*, con la excepción de su heurística, pues greedy utiliza unicamente la heuristica **h(n)** del nodo cell hasta el nodo final.

Main Greedy:

```python
from cell import Cell
from greedy import greedy, print_labyrinth, labyrinth
import numpy as np
import image_process as img

OFFSET = 10
file = open('coordenadas.txt', 'w')

def main():
    """Main Function"""

    start_cell = Cell(position=(165, 20)) # 63, 12

    goal_cell = Cell(position=(25, 360)) # 8, 134

    print()

    path = greedy(start_cell, goal_cell)

    print()

    assert path is not None
    _labyrinth = labyrinth.copy().astype(np.uint8)
    for index in range(0, len(path), 20):
        _cell = path[index]
        #print(_cell)
        # es el de arriba un obstaculo?
        if _labyrinth[_cell.row - 1][_cell.column] == 0:
            _cell.row += OFFSET
        # es el de abajo un obstaculo?
        if _labyrinth[_cell.row + 1][_cell.column] == 0:
            _cell.row -= OFFSET
        # es el de la izquierda un obstaculo?
        if _labyrinth[_cell.row][_cell.column - 1] == 0:
            _cell.column += OFFSET
        # es el de la derecha un obstaculo?
        if _labyrinth[_cell.row][_cell.column + 1] == 0:
            _cell.column -= OFFSET
        _labyrinth[_cell.row][_cell.column] = 127

        print(_cell, file=file) #Aqui guardamos las coordenadas en pixeles
    #Imprime laberinto
    if path != []:
        _labyrinth[goal_cell.row][goal_cell.column] = 127

    file.close()
    # print_labyrinth(labyrinth)
    img.display("Laberinto :D", _labyrinth)

if __name__ == "__main__":
    main()
```

Algoritmo Greedy:

```python
from cell import Cell
from collections import deque
import image_process as img

# inicialización
# carga la imagen
original_labyrith = img.load_image("img/borderless.png")
thick_labyrinth = img.load_image("img/thicker_borders.png")

# aquí obtenemos la imagen del laberinto binarizada (0/255)
labyrinth = img.binarize_image(original_labyrith)

ROWS = thick_labyrinth.shape[0] # la shape es (filas, columnas)
COLUMNS = thick_labyrinth.shape[1]

B = 0 # bloqueado
F = 255 # libre/desbloqueado

def print_labyrinth(lab):
    for row in lab:
        for column in row:
            print(column, end=" ")
        print()

def greedy(start: Cell, dst: Cell):
    # validaciones
    if not is_valid(start.row, start.column):
        print("Start cell is invalid")
        return []

    if not is_valid(dst.row, dst.column):
        print("Destionation cell is invalid")
        return []

    if is_blocked(start.row, start.column) or is_blocked(dst.row, dst.column):
        print("Start or destionation cell are blocked!")
        return []

    if is_destination(start.row, start.column, dst):
        print("We are in goal cell")
        return start.get_path()

    # lista doblemente ligada, su usará como cola (FIFO) para almacenar las celdas
    open_list = deque()

    # Matriz booleana para saber que celdas se han visitado
    # Se inicializa a False indicando que no se ha visitado ni uno
    closed_list = []
    for _ in range(ROWS):
        closed_list.append([False for _ in range(COLUMNS)])
```

```
51
52      cell_details = []
53      for row in range(ROWS):
54          cell_details.append([Cell(position=(row, col))
55                               for col in range(COLUMNS)])
56
57      open_list.append(start)  # agrega al final
58
59      found_dst = False
60
61      while len(open_list) != 0:
62          current_cell = open_list.popleft()  # elimina por la izquierda en O(1)
63          row, column = current_cell.row, current_cell.column
64          closed_list[row][column] = True
65
66          h_new = 0
67          # generacion de sucesores
68          if is_valid(row-1, column):
69              # generar arriba
70              if is_destination(row-1, column, dst):
71                  cell_details[row-1][column].parent = current_cell
72                  print("We are at destination")
73                  found_dst = True
74                  break
75              elif closed_list[row-1][column] == False and not is_blocked(row-1, column):
76                  h_new = current_cell.manhattan_distance(dst)
77
78                  if cell_details[row-1][column].h == 0 or cell_details[row-1][column].h > h_new:
79                      open_list.append(
80                          Cell(position=(row-1, column), parent=current_cell))
81                      successor_cell = cell_details[row-1][column]
82                      successor_cell.h = h_new
83                      successor_cell.parent = current_cell
84
85          if is_valid(row, column+1):
86              # genera derecha
87              if is_destination(row, column+1, dst):
88                  cell_details[row][column+1].parent = current_cell
89                  print("We are at destination")
90                  found_dst = True
91                  break
92              elif closed_list[row][column+1] == False and not is_blocked(row, column+1):
93                  h_new = current_cell.manhattan_distance(dst)
94
95                  if cell_details[row][column+1].h == 0 or cell_details[row][column+1].h > h_new:
96                      open_list.append(
97                          Cell(position=(row, column+1), parent=current_cell))
98                      cell_details[row][column+1].h = h_new
99                      cell_details[row][column+1].parent = current_cell
100
101         if is_valid(row+1, column):
102             # genera abajo
103             if is_destination(row+1, column, dst):
104                 cell_details[row+1][column].parent = current_cell
105                 print("We are at destination")
106                 found_dst = True
107                 break
108             elif closed_list[row+1][column] == False and not is_blocked(row+1, column):
109                 h_new = current_cell.manhattan_distance(dst)
110
111                 if cell_details[row+1][column].h == 0 or cell_details[row+1][column].h > h_new:
112                     open_list.append(
113                         Cell(position=(row+1, column), parent=current_cell))
114                     cell_details[row+1][column].h = h_new
115                     cell_details[row+1][column].parent = current_cell
```

```
116
117         if is_valid(row, column-1):
118             # genera izquierda
119             if is_destination(row, column-1, dst):
120                 cell_details[row][column-1].parent = current_cell
121                 print("We are at destination")
122                 found_dst = True
123                 break
124             elif closed_list[row][column-1] == False and not is_blocked(row, column-1):
125                 h_new = current_cell.manhattan_distance(dst)
126 
127                 if cell_details[row][column-1].h == 0 or cell_details[row][column-1].h > h_new:
128                     open_list.append(
129                         Cell(position=(row, column-1), parent=current_cell))
130                     cell_details[row][column-1].h = h_new
131                     cell_details[row][column-1].parent = current_cell
132 
133     # limpieza
134     open_list.clear()
135     closed_list.clear()
136     cell_details.clear()
137 
138     if not found_dst:
139         print("Failed to find path to destionation cell")
140         return current_cell.get_path()
141 
142     return current_cell.get_path()
143 
144 def is_destination(row: int, column: int, goal: Cell):
145     return row == goal.row and column == goal.column
146 
147 def is_valid(row: int, column: int):
148     return (row >= 0 and row < ROWS) and (column >= 0 and column < COLUMNS)
149 
150 def is_blocked(row: int, column: int):
151     return thick_labyrinth[row][column] == B
```

COMPARACIÓN DE RESULTADOS

Al ejecutar ambos algoritmos de búsqueda podemos comprobar que el método greedy es más rápido que el método A*.

PROBLEMS OUTPUT TERMINAL AZURE DEBUG CONSOLE

```
> Measure-Command { python .\code\main_a_star.py }

Days              : 0
Hours             : 0
Minutes           : 0
Seconds           : 1
Milliseconds      : 361
Ticks             : 13612234
TotalDays         : 1.5754900462963E-05
TotalHours        : 0.000378117611111111
TotalMinutes      : 0.0226870566666667
TotalSeconds      : 1.3612234
TotalMilliseconds : 1361.2234
```

```
> Measure-Command { python .\code\main_greedy.py }

Days              : 0
Hours             : 0
Minutes           : 0
Seconds           : 1
Milliseconds      : 337
Ticks             : 13378980
TotalDays         : 1.54849305555556E-05
TotalHours        : 0.000371638333333333
TotalMinutes      : 0.0222983
TotalSeconds      : 1.337898
TotalMilliseconds : 1337.898
```

DIAGRAMA DE BLOQUES

A grandes rasgos se definen las etapas del proyecto mediante el siguiente diagrama de bloques: