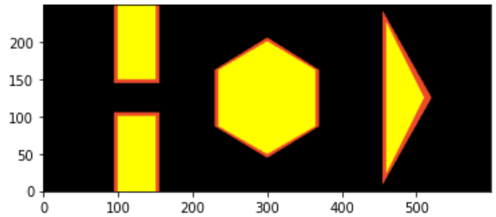
```python

# 119135891
# Ishaan_Parikh

# Github link :- https://github.com/Ishaan1810/ENPM-661-PROJECT-2.git

# Importing necessary libraries and modules
import cv2
import numpy as np
import timeit
from queue import PriorityQueue
import matplotlib.pyplot as plt

#Defining all the required values
#Defining the width and height of the map
width = 600
height = 250
#Assuming the robot to be a point robot and taking the clearance to be 5 mm
cl  = 5
#defining the cost required to perform movements
cost_straight=1
cost_slant=1.4

#Defining a class Node to store the information of each node in the search tree
class Node():
  # Defining Methods to return the state, move, parent, parent state, and cost to come of the node
    def __init__(self, state, parent, move, CostToCome):
        self.state, self.parent, self.move, self.CostToCome = state, parent, move, CostToCome
    def State_Return(self):
        return self.state
    def Move_return(self):
        return self.move
    def Parent_return(self):
        return self.parent
    def Parentstate_return(self):
        return self.parent.State_Return() if self.Parent_return() else None
    def Cost_return(self):
        return self.CostToCome
    def __lt__(self, other):
        return self.CostToCome < other.CostToCome

   # Defining the Method to return the path of the node
    def Path_return(self):
        moves, path = [], []
        node = self
        while node.Move_return():
            moves.append(node.Move_return())
            path.append(node)
            node = node.Parent_return()
        path.append(node)
        return moves[::-1], path[::-1]
#Defining all the functions required in the Dijkstra Algorithm
#Defining a function to generate the map area with obstacles
#Finding the Mathematical Representation of Free Space using line equations
def generate_obstacle_map(width, height):
  # Initializing an array of zeros with the given dimensions
    obstacle_map = np.zeros((height, width))
    for y in range(height):
        for x in range(width):
            #Rectangle 1
            obs_rect_1, obs_rect_2, obs_rect_3, obs_rect_4 = (x+5) - 100, (y-5) - 100, (x-5) - 150, y - 0
            #Rectangle 2
            obs_rect_1_u, obs_rect_2_u, obs_rect_3_u, obs_rect_4_u = (x+5) - 100, y - 250, (x-5) - 150, (y+5) - 150
            # Hexagon
            obs_hex_1, obs_hex_2, obs_hex_3, obs_hex_4, obs_hex_5, obs_hex_6 = (x+6.5) - 235.04, y - 0.58*x - 26.8, y + 0.58*x - 373.2, x
            # Triangle
            obs_tri_1, obs_tri_2, obs_tri_3 = x+5-460, y+2*x-1145, y-2*x+895
        # Checking if the current coordinates lie inside any of the obstacles
            if (obs_hex_6>0 and obs_hex_5>0 and obs_hex_4<0 and obs_hex_3<0 and obs_hex_2<0 and obs_hex_1>0) or(obs_rect_1>0 and obs_rect_
                obstacle_map[y, x] = 1
    return obstacle_map

#defining a function to take in the start state of the node individually taking x and y coordinates
def input_start_state():
    print("Enter the X coordinate of the start node: ")
    x = int(input())
```

```
    print("Enter the Y coordinate of the start node: ")
    y = int(input())
    Init_State = [x, y]
    return Init_State
#defining a function to take in the goal state of the node individually taking x and y coordinates
def input_goal_state():
    print("Enter the X coordinate of the Goal node: ")
    x = int(input())
    print("Enter the Y coordinate of the Goal node: ")
    y = int(input())
    Goal_State = [x, y]
    return Goal_State
#Defining a function to check the validity of the start and goal coordinates
def check_input_validity(start_state, goal_state, obstacle_map):
    if bound_check(start_state[0], start_state[1]):
        print("START STATE IS OUT OF BOUNDS")
        return False
    elif bound_check(goal_state[0], goal_state[1]):
        print("GOAL STATE IS OUT OF BOUNDS")
        return False
    elif check_location_validity(start_state[0], start_state[1], obstacle_map):
        print("START STATE IS IN AN OBSTACLE")
        return False
    elif check_location_validity(goal_state[0], goal_state[1], obstacle_map):
        print("GOAL STATE IS IN AN OBSTACLE")
        return False
    elif start_state == goal_state:
        print("INITIAL STATE AND GOAL STATE ARE SAME")
        return False
    else:
        return True

#Defining a function to check if the node location lies at a valid point not within the obstacles
def check_location_validity(x, y, obstacle_map):
    size = obstacle_map.shape
    return (x >= size[1] or x < 0 or y >= size[0] or y < 0
            or obstacle_map[y, x] in {1, 2})

#Defining a function to check if the exploration is within the Map taking into consideration the clearance
def bound_check(x_coordinate, y_coordinate):
    Bound_X = 600-5
    Bound_Y = 250-5
    if (x_coordinate > Bound_X or int(x_coordinate)<1 or int(y_coordinate)<1 or y_coordinate>Bound_Y):
        return 1
    return 0
#####################################################################################################################################
#Defining 8 functions for 8 directional movements
def Move_up(x, y, obstacle_map, parent_state):
    moves_list = []
    move_x, move_y = x, y+1
    if not (check_location_validity(move_x, move_y, obstacle_map) or bound_check(move_x, move_y) or parent_state == [move_x, move_y]):
        moves_list.append('Move_up')
    return moves_list

def Move_down(x, y, obstacle_map, parent_state):
    moves_list = []
    move_x, move_y = x, y-1
    if not (check_location_validity(move_x, move_y, obstacle_map) or bound_check(move_x, move_y) or parent_state == [move_x, move_y]):
        moves_list.append('Move_down')
    return moves_list

def Move_right(x, y, obstacle_map, parent_state):
    moves_list = []
    move_x, move_y = x+1, y
    if not (check_location_validity(move_x, move_y, obstacle_map) or bound_check(move_x, move_y) or parent_state == [move_x, move_y]):
        moves_list.append('Move_right')
    return moves_list

def Move_left(x, y, obstacle_map, parent_state):
    moves_list = []
    move_x, move_y = x-1, y
    if not (check_location_validity(move_x, move_y, obstacle_map) or bound_check(move_x, move_y) or parent_state == [move_x, move_y]):
        moves_list.append('Move_left')
    return moves_list

def Move_Up_right(x, y, obstacle_map, parent_state):
    moves_list = []
```

```
    move_x, move_y = x+1, y+1
    if not (check_location_validity(move_x, move_y, obstacle_map) or bound_check(move_x, move_y) or parent_state == [move_x, move_y]):
        moves_list.append('Move_Up_right')
    return moves_list

def Move_down_right(x, y, obstacle_map, parent_state):
    moves_list = []
    move_x, move_y = x+1, y-1
    if not (check_location_validity(move_x, move_y, obstacle_map) or bound_check(move_x, move_y) or parent_state == [move_x, move_y]):
        moves_list.append('Move_down_right')
    return moves_list

def Move_down_left(x, y, obstacle_map, parent_state):
    moves_list = []
    move_x, move_y = x-1, y-1
    if not (check_location_validity(move_x, move_y, obstacle_map) or bound_check(move_x, move_y) or parent_state == [move_x, move_y]):
        moves_list.append('Move_down_left')
    return moves_list

def Move_Up_left(x, y, obstacle_map, parent_state):
    moves_list = []
    move_x, move_y = x-1, y+1
    if not (check_location_validity(move_x, move_y, obstacle_map) or bound_check(move_x, move_y) or parent_state == [move_x, move_y]):
        moves_list.append('Move_Up_left')
    return moves_list
####################################################################################################################################
#Defining a function to create a list of possible move options from the parent node
def move_options(current_node):
    x, y = current_node.State_Return()
    parent_state = current_node.Parentstate_return()
    moves_list = []

    moves_list += Move_up(x, y, obstacle_map, parent_state)

    moves_list += Move_Up_right(x, y, obstacle_map, parent_state)

    moves_list += Move_right(x, y, obstacle_map, parent_state)

    moves_list += Move_down_right(x, y, obstacle_map, parent_state)

    moves_list += Move_down(x, y, obstacle_map, parent_state)

    moves_list += Move_down_left(x, y, obstacle_map, parent_state)

    moves_list += Move_left(x, y, obstacle_map, parent_state)

    moves_list += Move_Up_left(x, y, obstacle_map, parent_state)

    return moves_list
#Defining a function to shade the map area
def shade_map_area(map_area, Location, Color):
    x,_,_ = map_area.shape
    translation_y = Location[0]
    translation_x = x - Location[1] - 1
    map_area[translation_x,translation_y,:] = Color
    return map_area

#Defining a funcction to check if the current node has reached the goal or not
def check_current_goal(current_node, GoalNode):
    if np.array_equal(current_node, GoalNode) or current_node == GoalNode:
        return True
    else:
        return False

obstacle_map = generate_obstacle_map(width, height)
#Known Dimensions for the hexagon
hexagon_centre_x = 300
hexagon_centre_y = 125
#Obstacles
#Bottom Rectangle
B_rect = np.array([[100,100],[100,0],[150,0],[150,100]])
#Top Rectangle
T_rect = np.array([[100,250],[100,150], [150,150],[150,250]])
#Middle Hexagon
hex= np.array([[300,50],[364,87],[364,162],[300,200],[235,162],[235,87]])
#Triangle
tri = np.array([[460,25],[460,225],[510,125]])
#Final obstacles including the  clearance sapce
```

```
#Bottom Rectangle
B_Rect_cl = np.array([[100 - cl,100 + cl],[100 - cl, 0],[150 + cl, 0],[150 + cl, 100 + cl]])
#Top Rectangle
T_Rect_cl = np.array([[100 - cl, 250],[100-cl,150-cl], [150+cl,150-cl],[150+cl,250]])
#Middle Hexagon
hex_cl = np.array([[300,45],[369,87],[369,162],[300,205],[230,162],[230,87]])
#Triangle
tri_cl= np.array([[460-cl,25-3*cl],[460-cl,225+3*cl],[510 + 2*cl, 125]])

#Initializing the video file to be saved later in the workspace
vid_write = cv2.VideoWriter_fourcc(*"mp4v")
video = cv2.VideoWriter("Dijkstra_Ishaan_Parikh.mp4", vid_write, 100, (width, height))
map_area = np.zeros((height, width,3), dtype = np.uint8)
map_area[:,:] = (0,0,0)

#Using cv2.pillpoly to create a visual reprentation of the Map
Rectangle_lower_clearance = cv2.fillPoly(map_area, [B_Rect_cl], [238, 75, 43])
Rectangle_upper_clearance = cv2.fillPoly(map_area, [T_Rect_cl], [238, 75, 43])
Triangle_clearance= cv2.fillPoly(map_area, [tri_cl], [238, 75, 43])
Hexagon_clearance = cv2.fillPoly(map_area, [hex_cl], [238, 75, 43])
#Filling the inner shapes later so they show up above the clearance area
Rectangle_lower= cv2.fillPoly(map_area, [B_rect], [255,255,0])
Rectangle_upper = cv2.fillPoly(map_area, [T_rect], [255,255,0])
Triangle= cv2.fillPoly(map_area, [tri], [255,255,0])
Hexagon = cv2.fillPoly(map_area, [hex], [255,255,0])
#Showing the Map for a refrence to select start and goal state
plt.imshow(map_area, origin='lower')
plt.show()

# Starting the Dijkstra Algorithm
Closed_List= []
relative_space = np.array([[Node([i,j], None, None, float('inf')) for j in range(height)] for i in range(width)])

bool1 = True
while(bool1 == True):

    StartState = input_start_state()
    GoalState = input_goal_state()

    if check_input_validity(StartState, GoalState, obstacle_map):
        Open_List = PriorityQueue()
        Closed_List= []


        starting_node = Node(StartState, None, None, 0)
        Open_List.put((starting_node.Cost_return(), starting_node))

        GoalReach = False

        Points = shade_map_area(map_area, StartState, [255,0,0])
        Points = shade_map_area(map_area, GoalState, [255,0,0])

        relative_space = np.array([[Node([i,j], None, None, float('inf')) for j in range(height)] for i in range(width)])

        start = timeit.default_timer()

        print("Starting the Dijkstra search algorithm.")

        while not (Open_List.empty() and GoalReach):
          #Using the node with the lowest cost value
            current_node = Open_List.get()[1]
            i, j = current_node.State_Return()
            #appending the node to the closed list
            Closed_List.append([i,j])
            Points = shade_map_area(Points, current_node.State_Return(), [0,0,255])
            video.write(cv2.cvtColor(Points, cv2.COLOR_RGB2BGR))

            #creating 3 dictionaries for moves in x and y and cost
            move_x = {'Move_up':i ,
                      'Move_Up_right':i+1,
                      'Move_right':i+1,
                      'Move_down_right':i+1,
                      'Move_down':i,
                      'Move_down_left':i-1,
                      'Move_left':i-1,
                      'Move_Up_left':i-1}

```

```
        move_y = {'Move_up':j+1 ,
                  'Move_Up_right':j+1,
                  'Move_right':j,
                  'Move_down_right':j-1,
                  'Move_down':j-1,
                  'Move_down_left':j-1,
                  'Move_left':j,
                  'Move_Up_left':j+1}

        Moves_CostToCome = {'Move_up':1 ,
                            'Move_Up_right':1.4,
                            'Move_right':1,
                            'Move_down_right':1.4,
                            'Move_down':1,
                            'Move_down_left':1.4,
                            'Move_left':1,
                            'Move_Up_left':1.4}


        check_if_complete = check_current_goal(current_node.State_Return(), GoalState)

        if check_if_complete:
            print("Goal Reached!")
            MovesPath, Path = current_node.Path_return()

            for nodes in Path:
                Position = nodes.State_Return()
                Points = shade_map_area(Points, Position, [255,0,0])
                video.write(cv2.cvtColor(Points, cv2.COLOR_RGB2BGR))
            break

        else:
            node_new = move_options(current_node)
            Parent_Cost = current_node.Cost_return()
            # Check if the new nodes have already been visited (i.e., are in the Closed_List).
            if node_new not in Closed_List:
               # If the new node has not been visited, iterate over each possible move.
                for move in node_new:
                    # Get the position of the child node based on the move.
                    Child_Position = [move_x.get(move), move_y.get(move)]
                    CostToCome = Parent_Cost + Moves_CostToCome.get(move)
                    #Main algorithm to compare the cost of the moves and deciding the next move
                    if(CostToCome < relative_space[Child_Position[0], Child_Position[1]].Cost_return()):
                        child_node_new = Node(Child_Position, current_node, move, CostToCome)
                        relative_space[Child_Position[0], Child_Position[1]] = child_node_new
                        Open_List.put((child_node_new.Cost_return(), child_node_new))
        # Check if the goal state has been reached.
        if check_if_complete:
           # If the goal state has been reached, exit the loop.
            break

    end = timeit.default_timer()
    video.release()
    print("Total Runtime:-",end-start)
    print("The video has been saved in the workspace")

    bool1 = False
else:
    print("TRY DIFFERENT VALUES")
```




```
Enter the X coordinate of the start node:
50
Enter the Y coordinate of the start node:
50
Enter the X coordinate of the Goal node:
100
Enter the Y coordinate of the Goal node:
120
Starting the Dijkstra search algorithm.
Goal Reached!
Total Runtime:- 15.210486214999037
```