```
#!/usr/bin/env python
# coding: utf-8


import numpy as np
import cv2
import
matplotlib.pyplot as plt
import heapq
import math
from matplotlib.animation import
FuncAnimation

def draw_hexagon(side_length, centroid):
    angle = 60
    angles_rad =
np.radians([(60 * i + 90) for i in range(6)])
    x = centroid[0] + side_length *
np.cos(angles_rad)
    y = centroid[1] + side_length * np.sin(angles_rad)
    points =
np.array([[int(x[i]), int(y[i])] for i in range(6)], np.int32)
    points =
points.reshape((-1, 1, 2))
    return points

'''
First Plotting the Bloated
Figure
'''
step_size = float((input("Enter the step size: ")))
radius =
int(input("Enter the radius of the robot: "))
cle = int(input("Enter the
clearence: "))
total_clearence = cle + radius

pts_hexagon = draw_hexagon(150 + cle ,
(650 , 250))

# Coordinates of the first polygon
x1_polygon1, x2_polygon1, y1_polygon1,
y2_polygon1 = 99 - cle , 174 + cle , 99 - cle , 499

# Coordinates of the second
polygon
x1_polygon2, x2_polygon2, y1_polygon2, y2_polygon2 = 274 - cle , 349 + cle , 0 , 399 +
cle

x1_polygon3, x2_polygon3, y1_polygon3, y2_polygon3 = 899 - cle , 1099 + cle , 49 - cle ,
124 + cle

x1_polygon4, x2_polygon4, y1_polygon4 , y2_polygon4 = 1019 - cle , 1099 + cle ,
124 - cle , 374 + cle

x1_polygon5, x2_polygon5, y1_polygon5 , y2_polygon5 = 899 - cle , 1099
+ cle , 374 - cle , 449 + cle


pts_polygon3 = np.array([[x1_polygon3, y1_polygon3],
[x2_polygon3, y1_polygon3], [x2_polygon3, y2_polygon3], [x1_polygon3, y2_polygon3]],
np.int32)
pts_polygon3 = pts_polygon3.reshape((-1, 1, 2))

pts_polygon4 =
np.array([[x1_polygon4, y1_polygon4], [x2_polygon4, y1_polygon4], [x2_polygon4, y2_polygon4],
[x1_polygon4, y2_polygon4]], np.int32)
pts_polygon4 = pts_polygon4.reshape((-1, 1,
2))

pts_polygon5 = np.array([[x1_polygon5, y1_polygon5], [x2_polygon5, y1_polygon5],
[x2_polygon5, y2_polygon5], [x1_polygon5, y2_polygon5]], np.int32)
pts_polygon5 =
pts_polygon5.reshape((-1, 1, 2))
```

```
# Create a blank image with size 1190x490
img_check =
np.zeros((500, 1200 , 3), dtype=np.uint8)

# Define the vertices of the first
polygon
pts_polygon1 = np.array([[x1_polygon1, y1_polygon1], [x2_polygon1, y1_polygon1],
[x2_polygon1, y2_polygon1], [x1_polygon1, y2_polygon1]], np.int32)
pts_polygon1 =
pts_polygon1.reshape((-1, 1, 2))

# Define the vertices of the second polygon
pts_polygon2 =
np.array([[x1_polygon2, y1_polygon2], [x2_polygon2, y1_polygon2], [x2_polygon2, y2_polygon2],
[x1_polygon2, y2_polygon2]], np.int32)
pts_polygon2 = pts_polygon2.reshape((-1, 1, 2))


#
Fill the first polygon with white color
cv2.fillPoly(img_check, [pts_polygon1], (255 , 255 ,
255))

# Fill the second polygon with white color
cv2.fillPoly(img_check, [pts_polygon2],
(255 , 255 , 255))


cv2.fillPoly(img_check, [pts_polygon3], (255 , 255 ,
255))

cv2.fillPoly(img_check, [pts_polygon4], (255 , 255 , 255))

cv2.fillPoly(img_check,
[pts_polygon5], (255 , 255 , 255))

cv2.fillPoly(img_check, [pts_hexagon], (255 , 255 ,
255))

'''
Now Plotting the Maze on top of the bloated figure
'''
pts_hexagon =
draw_hexagon(150 , (650 , 250))


# Coordinates of the first polygon
x1_polygon1,
x2_polygon1, y1_polygon1, y2_polygon1 = 99 , 174 , 99 , 499

# Coordinates of the second
polygon
x1_polygon2, x2_polygon2, y1_polygon2, y2_polygon2 = 274 , 349 , 0 ,
399

x1_polygon3, x2_polygon3, y1_polygon3, y2_polygon3 = 899 , 1099 , 49 ,
124

x1_polygon4, x2_polygon4, y1_polygon4 , y2_polygon4 = 1019 , 1099 , 124 ,
374

x1_polygon5, x2_polygon5, y1_polygon5 , y2_polygon5 = 899 , 1099 , 374 ,
449


pts_polygon3 = np.array([[x1_polygon3, y1_polygon3], [x2_polygon3, y1_polygon3],
[x2_polygon3, y2_polygon3], [x1_polygon3, y2_polygon3]], np.int32)
pts_polygon3 =
pts_polygon3.reshape((-1, 1, 2))

pts_polygon4 = np.array([[x1_polygon4, y1_polygon4],
[x2_polygon4, y1_polygon4], [x2_polygon4, y2_polygon4], [x1_polygon4, y2_polygon4]],
np.int32)
pts_polygon4 = pts_polygon4.reshape((-1, 1, 2))
```

```
pts_polygon5 =
np.array([[x1_polygon5, y1_polygon5], [x2_polygon5, y1_polygon5], [x2_polygon5, y2_polygon5],
[x1_polygon5, y2_polygon5]], np.int32)
pts_polygon5 = pts_polygon5.reshape((-1, 1, 2))
#
Create a blank image with size 1190x490
img_ori = np.zeros((500, 1200 ,3),
dtype=np.uint8)

# Define the vertices of the first polygon
pts_polygon1 =
np.array([[x1_polygon1, y1_polygon1], [x2_polygon1, y1_polygon1], [x2_polygon1, y2_polygon1],
[x1_polygon1, y2_polygon1]], np.int32)
pts_polygon1 = pts_polygon1.reshape((-1, 1, 2))

#
Define the vertices of the second polygon
pts_polygon2 = np.array([[x1_polygon2, y1_polygon2],
[x2_polygon2, y1_polygon2], [x2_polygon2, y2_polygon2], [x1_polygon2, y2_polygon2]],
np.int32)
pts_polygon2 = pts_polygon2.reshape((-1, 1, 2))

# Fill the first polygon with
white color
cv2.fillPoly(img_check, [pts_polygon1], (255 , 0 , 0))

# Fill the second
polygon with white color
cv2.fillPoly(img_check, [pts_polygon2], (255 , 0 ,
0))


cv2.fillPoly(img_check, [pts_polygon3], (255 , 0 , 0))

cv2.fillPoly(img_check,
[pts_polygon4], (255 , 0 , 0))

cv2.fillPoly(img_check, [pts_polygon5], (255 , 0 ,
0))

cv2.fillPoly(img_check, [pts_hexagon], (255 , 0 , 0))


def possible_moves(tup ,
step_size):
    x_old, y_old, theta_old = tup
    move_list = []
    angles = [0, 30, 60,
-30, -60]

    for angle in angles:
        theta = (theta_old + angle)
        if (theta
== 0):
            theta = 0
        elif (theta == 360):
            theta = 360

else :
            theta = theta % 360
        x = x_old + step_size *
math.cos(np.radians(theta))
        y = y_old + step_size * math.sin(np.radians(theta))  # Use
sin for y-coordinate
        move_list.append((x, y, theta))

    return move_list

def
is_in_check(tup , visited):
    x , y , theta = tup
    if ((x , y)) in visited :

 thetas = visited[x, y]
            for theta_c in thetas:
```

```
                if abs(theta_c -
theta) < 30:
                    return True
    return False



def
is_move_legal(tup , img_check, total_clearence):
    x , y = tup
    #pixel_value =
img_check[y, x]
#     pixel_value = tuple(pixel_value)
    if x < total_clearence  or x
> 1199 - total_clearence or y < total_clearence or y > 499 - total_clearence:

return False
    elif tuple(img_check[int(round(y)), int(round(x))]) == (255 , 255 , 255) :

      #print(f"Point {point} is in the free region.(here 6)")
        return False

   else :
        return True

def algorithm(start , goal, step_size, image,
total_clearence) :
    #Please note that in the line that follows you will need to change the
video driver to an
    #appropriate one that is available on your system.
    fourcc =
cv2.VideoWriter_fourcc(*'MP4V')
    ##Please change the path to the desired path.
    out =
cv2.VideoWriter(r'C:\Users\sachi\Downloads\test2.mp4', fourcc, 20, (1200, 500))


x_start , y_start , theta_start = start

    x_goal , y_goal , theta_goal = goal


  h = math.sqrt((x_start - x_goal)**2 + (y_start - y_goal)**2)

    queue_open =  [(h ,
(start , 0))]

    heapq.heapify(queue_open)

    visited = {}
    visited_parent = {}

 path = []
    info_dict = {start : ((None , None) , 0)}
    # frame_list = []
    ite = 0

   move_list = []
    reached = 0
    while queue_open :
        element =
heapq.heappop(queue_open)
        t_c , tup = element
        node , c_2_c = tup
#
print(node)
        info = info_dict[node]
        parent , c_2_c_p = info

visited_parent[node] = parent
        x , y , theta = node
        theta = theta % 360

  x_int = int(round(x))
        y_int = int(round(y))
```

```
        #node = (x_int , y_int ,
theta)
        #figure out where exactly to store the theta value
        if (x_int , y_int)
in visited :
            thetas = visited[x_int , y_int]
            thetas.append(theta)

     else:
            thetas = []
            thetas.append(theta)

visited[x_int , y_int] = thetas
#         visited_parent[node] = parent
        if
(math.sqrt((node[0] - goal[0])**2 + (node[1] - goal[1])**2) <= 1.5 and (abs(node[2] -
goal[2]) <= 15 ))  :
            #print(reached)
            path.append(node)


            while node != start :
                parent = visited_parent[node]

     path.append(parent)
                node = parent


path.reverse()
            #print(path)
            for point in (path) :
                x,
y, theta = point
                x_int = int(round(x))
                y_int =
int(round(y))
                cv2.circle(img_ori , (x_int , y_int) , 1 , (0 , 255 , 0) , -1)

           print('reached')
            img_ori_copy = img_ori.copy()

flipped_vertical = cv2.flip(img_ori_copy, 0)
            for i in range (100):

 out.write(flipped_vertical)
            reached = 1
            break
        moves =
possible_moves((x_int , y_int , theta) , step_size)
        for move in (moves) :

x , y, theta = move
            x_int = int(round(x))
            y_int = int(round(y))

       Bool1 = is_move_legal((x_int , y_int) , image, total_clearence)
            #move =
(x_int , y_int , theta)
            if (Bool1 == True):
                #print(move)

        Bool2 = is_in_check((x_int , y_int , theta) , visited)
                if (Bool2 ==
False):
                    cv2.circle(img_ori , (x_int , y_int) , 1 , (255 , 0 , 0) , -1)

                 move_list.append((move[0] , move[1]))
                    if (ite % 10000) ==
0 :
                            img_ori_copy = img_ori.copy()

flipped_vertical = cv2.flip(img_ori_copy, 0)

out.write(flipped_vertical)
                    if move in info_dict :
```

```
 info = info_dict[move]
                        parent , c_2_c_p = info

  c_2_c_n = c_2_c + 1
                        if (c_2_c_n < c_2_c_p):

       #total cost calculation
                            total_cost = c_2_c_n +
math.sqrt((move[0] - goal[0]) ** 2 + (move[1] - goal[1]) ** 2)

info_dict[move] = (node , c_2_c_n)
                            queue_open = [(k, v) for k, v
in queue_open if v[0] != move]
                            heapq.heapify(queue_open)

                    heapq.heappush(queue_open , (total_cost , (move , c_2_c_n)))

     elif move not in info_dict :
                        total_cost = (c_2_c + 1) +
math.sqrt((move[0] - goal[0]) ** 2 + (move[1] - goal[1]) ** 2)

info_dict[move] = (node , c_2_c + 1)
                        heapq.heappush(queue_open ,
(total_cost , (move , c_2_c + 1)))
        ite += 1

    #out.release()
    return
visited_parent, reached, path, move_list

'''
Function to plot all the Visited
Nodes
'''
def animate_search(visited):
    fig, ax = plt.subplots(figsize=(12, 5)) #set
animate to 12:5 match map shape
    ax.set_xlim(0, 1200) #set animate x axis

ax.set_ylim(0, 500) #set animate y axis

    #show obstacles
    for polygons in
obstacles:
        polygon = plt.Polygon(polygons, facecolor="red",
edgecolor='black')
        ax.add_patch(polygon)

    points = ax.scatter([], [], s=1,
color='blue')

    def init():
        points.set_offsets(np.empty((0, 2)))

return points,

    def update(frame):
        skip = 50000 #set flames skip
        frame
*= skip
        visited_points = np.array(visited[:frame+1])

points.set_offsets(visited_points)
        return points,

    ani = FuncAnimation(fig,
update, frames=len(visited), init_func=init, blit=True, interval=1)

plt.show()


'''
Animate the path
'''
```

```
def animate_path(path):
    fig, ax =
plt.subplots(figsize=(12, 5))
    ax.set_xlim(0, 1200)
    ax.set_ylim(0, 500)

    for
polygons in obstacles:
        polygon = plt.Polygon(polygons, facecolor="gray",
edgecolor='black')
        ax.add_patch(polygon)

    line, = ax.plot([], [], 'b-', lw=2)  #
Path line

    def init():
        line.set_data([], [])
        return line,

    def
update(frame):
        skip = 20 #set flames skip
        frame *= skip
        x, y =
zip(*path[:frame+1]) #get path
        line.set_data(x, y)
        return line,

    ani =
FuncAnimation(fig, update, frames=len(path), init_func=init, blit=True, interval=50)

plt.show()


'''
Obstacle coordinates
'''
obstacles = [
    [(100, 100), (100, 500),
(175, 500), (175, 100)],

    [(275, 0), (275, 400), (350, 400), (350, 0)],


[(650-150*np.cos(np.pi/6), 400-150-150*0.5),
     (650-150*np.cos(np.pi/6), 400-150*0.5),

 (650, 400),
     (650+150*np.cos(np.pi/6), 400-150*0.5),
     (650+150*np.cos(np.pi/6),
400-150-150*0.5),
     (650, 100)],

    [(900, 450), (1100, 450), (1100, 50), (900, 50),
(900, 125), (1020, 125), (1020, 375), (900, 375)]
]

start_x = float(input("Enter the
start x position: "))
start_x = int(round(start_x))
start_y = float(input("Enter
the start y position: "))
start_y = int(round(start_y))
start_theta =
int(input("Enter the start orientation: "))
goal_x = float(input("Enter the
goal x position: "))
goal_x = int(round(goal_x))
goal_y = float(input("Enter the
goal y position: "))
goal_y = int(round(goal_y))
goal_theta = int(input("Enter the
goal orientation: "))

start = (start_x , start_y, start_theta)
```

```
goal = (goal_x ,
goal_y, goal_theta)
Bool1 = is_move_legal((start[0] , start[1]) , img_check,
total_clearence)
Bool2 = is_move_legal((goal[0] , goal[1]) , img_check, total_clearence)

if
Bool1 == True and Bool2 == True :
    print('correct positions entered algo is executing')

 visited_parent , reached, path, move_list = algorithm (start , goal, step_size, img_check,
total_clearence)
    if reached == 1 :
        print('Path is available')
    else  :

  print('Did not reach')
else :
    print('please run the code cell again and enter valid
start and goal positions')


'''
Storing all the path coordinates in a list
'''
node =
path[0]
while node != start :
    parent = visited_parent[node]
    path.append(parent)

node = parent
# path.reverse()
print(path)

coord_list = []
for point in path :
    x , y
, theta = point
    coord_list.append((x ,
y))

animate_search(move_list)

animate_path(coord_list)
```