git link
https://github.com/sbadshah96/ENPM661_Project3_phase1.git

Animation link
https://drive.google.com/file/d/1adSHrzEtFxdMqjhhDuRlv7y5UIpLrIMB/view

```
import math
import heapdict
import numpy as np
import time
import vidmaker
from sortedcollections import OrderedSet
import pygame

# Calculate new 'C' value for new obstacle definition.
def calculate_new_c(m, c, buffer_val):
    if m > 0 and c < 0:
        c_new = c - ((buffer_val) * np.sqrt(1 + (m ** 2)))
        return c_new
    elif m < 0 and c > 0:
        if c > 300:
            c_new = c + ((buffer_val) * np.sqrt(1 + (m ** 2)))
            return c_new
        elif c < 300:
            c_new = c - ((buffer_val) * np.sqrt(1 + (m ** 2)))
            return c_new
    elif m > 0 and c > 0:
        c_new = c + ((buffer_val) * np.sqrt(1 + (m ** 2)))
        return c_new
    else:
       return None

# Obstacles for PyGame Mapping only
def pygame_obstacles(obstacle_buffer):
    obstacles = []

    buffer_val = obstacle_buffer

    c1_rec1 = 100
    m1_rec1 = 0
    c1_rec1_new = c1_rec1 - buffer_val
    obstacles.append((m1_rec1, c1_rec1_new))

    c2_rec1 = 150
```

```
m2_rec1 = 0
c2_rec1_new = c2_rec1 + buffer_val
obstacles.append((m2_rec1, c2_rec1_new))

c3_rec1 = 100
m3_rec1 = 0
c3_rec1_new = c3_rec1 + buffer_val
obstacles.append((m3_rec1, c3_rec1_new))

c1_rec2 = 100
m1_rec2 = 0
c1_rec2_new = c1_rec2 - buffer_val
obstacles.append((m1_rec2, c1_rec2_new))

c2_rec2 = 150
m2_rec2 = 0
c2_rec2_new = c2_rec2 + buffer_val
obstacles.append((m2_rec2, c2_rec2_new))

c3_rec2 = 150
m3_rec2 = 0
c3_rec2_new = c3_rec2 - buffer_val
obstacles.append((m3_rec2, c3_rec2_new))

c1_hex = -123.21
m1_hex = 0.577
c1_hex_new = calculate_new_c(m1_hex, c1_hex, buffer_val)
obstacles.append((m1_hex, c1_hex_new))

c2_hex = 364.95
m2_hex = 0
c2_hex_new = c2_hex + buffer_val
obstacles.append((m2_hex, c2_hex_new))

c3_hex = 373.21
m3_hex = -0.577
c3_hex_new = calculate_new_c(m3_hex, c3_hex, buffer_val)
obstacles.append((m3_hex, c3_hex_new))

c4_hex = 26.78
m4_hex = 0.577
c4_hex_new = calculate_new_c(m4_hex, c4_hex, buffer_val)
obstacles.append((m4_hex, c4_hex_new))
```

```
c5_hex = 235.05
m5_hex = 0
c5_hex_new = c5_hex - buffer_val
obstacles.append((m5_hex, c5_hex_new))

c6_hex = 223.21
m6_hex = -0.577
c6_hex_new = calculate_new_c(m6_hex, c6_hex, buffer_val)
obstacles.append((m6_hex, c6_hex_new))

c1_tri = 460
m1_tri = 0
c1_tri_new = c1_tri - buffer_val
obstacles.append((m1_tri, c1_tri_new))

c2_tri = 25
m2_tri = 0
c2_tri_new = c2_tri - buffer_val
obstacles.append((m2_tri, c2_tri_new))

c3_tri = -895
m3_tri = 2
c3_tri_new = calculate_new_c(m3_tri, c3_tri, buffer_val)
obstacles.append((m3_tri, c3_tri_new))

c4_tri = 1145
m4_tri = -2
c4_tri_new = calculate_new_c(m4_tri, c4_tri, buffer_val)
obstacles.append((m4_tri, c4_tri_new))

c5_tri = 225
m5_tri = 0
c5_tri_new = c5_tri + buffer_val
obstacles.append((m5_tri, c5_tri_new))

c1_bound = 0 + buffer_val
c2_bound = 600 - buffer_val
c3_bound = 0 + buffer_val
c4_bound = 250 - buffer_val
obstacles.append((0, c1_bound))
obstacles.append((0, c2_bound))
obstacles.append((0, c3_bound))
obstacles.append((0, c4_bound))
```

```
    return obstacles

# Define new obstacles based on user input buffer
def obstacles(obstacle_buffer, robot_size):
    obstacles = []

    buffer_val = obstacle_buffer + robot_size

    c1_rec1 = 100
    m1_rec1 = 0
    c1_rec1_new = c1_rec1 - buffer_val
    obstacles.append((m1_rec1, c1_rec1_new))

    c2_rec1 = 150
    m2_rec1 = 0
    c2_rec1_new = c2_rec1 + buffer_val
    obstacles.append((m2_rec1, c2_rec1_new))

    c3_rec1 = 100
    m3_rec1 = 0
    c3_rec1_new = c3_rec1 + buffer_val
    obstacles.append((m3_rec1, c3_rec1_new))

    c1_rec2 = 100
    m1_rec2 = 0
    c1_rec2_new = c1_rec2 - buffer_val
    obstacles.append((m1_rec2, c1_rec2_new))

    c2_rec2 = 150
    m2_rec2 = 0
    c2_rec2_new = c2_rec2 + buffer_val
    obstacles.append((m2_rec2, c2_rec2_new))

    c3_rec2 = 150
    m3_rec2 = 0
    c3_rec2_new = c3_rec2 - buffer_val
    obstacles.append((m3_rec2, c3_rec2_new))

    c1_hex = -123.21
    m1_hex = 0.577
    c1_hex_new = calculate_new_c(m1_hex, c1_hex, buffer_val)
    obstacles.append((m1_hex, c1_hex_new))

    c2_hex = 364.95
```

```
m2_hex = 0
c2_hex_new = c2_hex + buffer_val
obstacles.append((m2_hex, c2_hex_new))

c3_hex = 373.21
m3_hex = -0.577
c3_hex_new = calculate_new_c(m3_hex, c3_hex, buffer_val)
obstacles.append((m3_hex, c3_hex_new))

c4_hex = 26.78
m4_hex = 0.577
c4_hex_new = calculate_new_c(m4_hex, c4_hex, buffer_val)
obstacles.append((m4_hex, c4_hex_new))

c5_hex = 235.05
m5_hex = 0
c5_hex_new = c5_hex - buffer_val
obstacles.append((m5_hex, c5_hex_new))

c6_hex = 223.21
m6_hex = -0.577
c6_hex_new = calculate_new_c(m6_hex, c6_hex, buffer_val)
obstacles.append((m6_hex, c6_hex_new))

c1_tri = 460
m1_tri = 0
c1_tri_new = c1_tri - buffer_val
obstacles.append((m1_tri, c1_tri_new))

c2_tri = 25
m2_tri = 0
c2_tri_new = c2_tri - buffer_val
obstacles.append((m2_tri, c2_tri_new))

c3_tri = -895
m3_tri = 2
c3_tri_new = calculate_new_c(m3_tri, c3_tri, buffer_val)
obstacles.append((m3_tri, c3_tri_new))

c4_tri = 1145
m4_tri = -2
c4_tri_new = calculate_new_c(m4_tri, c4_tri, buffer_val)
obstacles.append((m4_tri, c4_tri_new))
```

```
    c5_tri = 225
    m5_tri = 0
    c5_tri_new = c5_tri + buffer_val
    obstacles.append((m5_tri, c5_tri_new))

    c1_bound = 0 + buffer_val
    c2_bound = 600 - buffer_val
    c3_bound = 0 + buffer_val
    c4_bound = 250 - buffer_val
    obstacles.append((0, c1_bound))
    obstacles.append((0, c2_bound))
    obstacles.append((0, c3_bound))
    obstacles.append((0, c4_bound))

    return obstacles


# Check if the robot is in obstacle space.
def check_obstacles(x, y):
    c1_rec1 = obstacles[0][1]
    c2_rec1 = obstacles[1][1]
    c3_rec1 = obstacles[2][1]

    c1_rec2 = obstacles[3][1]
    c2_rec2 = obstacles[4][1]
    c3_rec2 = obstacles[5][1]

    m1_hex = obstacles[6][0]
    c1_hex = obstacles[6][1]

    c2_hex = obstacles[7][1]

    m3_hex = obstacles[8][0]
    c3_hex = obstacles[8][1]

    m4_hex = obstacles[9][0]
    c4_hex = obstacles[9][1]

    c5_hex = obstacles[10][1]

    m6_hex = obstacles[11][0]
    c6_hex = obstacles[11][1]

    c1_tri = obstacles[12][1]
```

```
    c2_tri = obstacles[13][1]

    m3_tri = obstacles[14][0]
    c3_tri = obstacles[14][1]

    m4_tri = obstacles[15][0]
    c4_tri = obstacles[15][1]

    c5_tri = obstacles[16][1]

    c1_bound = obstacles[17][1]
    c2_bound = obstacles[18][1]
    c3_bound = obstacles[19][1]
    c4_bound = obstacles[20][1]

    if (((c1_rec1) <= x <= (c2_rec1)) and (0 <= y <= (c3_rec1))):
        return False
    elif (((c1_rec2) <= x <= (c2_rec2)) and ((c3_rec2) <= y <= 250)):
        return False
    elif ((int(y - (m1_hex * x)) >= c1_hex) and
          (x <= (c2_hex)) and
          (int(y - (m3_hex * x)) <= c3_hex) and
          (int(y - (m4_hex * x)) <= c4_hex) and
          (x >= c5_hex) and
          (int(y - (m6_hex * x)) >= c6_hex)):
        return False
    elif ((x >= c1_tri) and
          (y >= c2_tri) and
          (int(y - (m3_tri * x)) >= c3_tri) and
          (int(y - (m4_tri * x)) <= c4_tri) and
          (y <= c5_tri)):
        return False
    elif ((x <= c1_bound) or (x >= c2_bound) or (y <= c3_bound) or (y >= c4_bound)):
        return False
    else:
        return True

# Custom rounding off function for coordinates
def custom_coord_round(a):
    if a - int(a) <= 0.25:
        return int(a)
    elif 0.25 < a - int(a) <= 0.75:
        return int(a) + 0.5
```

```
    elif 0.75 < a - int(a) < 1:
        return int(a) + 1

# Custom rounding off function for angle
def custom_ang_round(b):
    if b >= 360:
        b = b % 360
    elif -360 < b < 0:
        b += 360
    elif b <= -360:
        b = b % 360 + 360
    c = b % 30
    if c < 15:
        return int(b - c)
    else:
        return int(b - c + 30)

# Visited nodes threshold
def visited_nodes_threshold_check(x, y, theta):
    if visited_nodes[int(2 * x)][int(2 * y)][int(theta / 30)]:
        return False
    elif visited_nodes[int(2 * (x + 0.5))][int(2 * y)][int(theta / 30)]:
        return False
    elif visited_nodes[int(2 * (x - 0.5))][int(2 * y)][int(theta / 30)]:
        return False
    elif visited_nodes[int(2 * (x))][int(2 * (y + 0.5))][int(theta / 30)]:
        return False
    elif visited_nodes[int(2 * (x))][int(2 * (y - 0.5))][int(theta / 30)]:
        return False
    elif visited_nodes[int(2 * (x + 0.5))][int(2 * (y + 0.5))][int(theta / 30)]:
        return False
    elif visited_nodes[int(2 * (x - 0.5))][int(2 * (y + 0.5))][int(theta / 30)]:
        return False
    elif visited_nodes[int(2 * (x + 0.5))][int(2 * (y - 0.5))][int(theta / 30)]:
        return False
    elif visited_nodes[int(2 * (x - 0.5))][int(2 * (y - 0.5))][int(theta / 30)]:
        return False
    else:
        return True

# Check new node based on action set and making decisions to adding it to visited nodes list
def check_new_node(x, y, theta, total_cost, cost_to_go, cost_to_come):
    if visited_nodes_threshold_check(x, y, theta):
        if (x, y, theta) in explored_nodes:
```

```
        if explored_nodes[(x, y, theta)][0] >= total_cost:
            explored_nodes[(x, y, theta)] = total_cost, cost_to_go, cost_to_come
            node_records[(x, y, theta)] = (pop[0][0], pop[0][1], pop[0][2])
            visited_nodes_track.add((x, y, theta))
            return None
        else:
            return None
    explored_nodes[(x, y, theta)] = total_cost, cost_to_go, cost_to_come
    node_records[(x, y, theta)] = (pop[0][0], pop[0][1], pop[0][2])
    explored_mapping.append((x, y))
    visited_nodes_track.add((x, y, theta))

# Action1 - moving robot to a step size at 60deg
def action1(pop):
    x, y, theta = pop[0]
    cost_to_come = pop[1][2]
    theta_new = custom_ang_round(theta + 60)
    x_new = custom_coord_round(x + step_size * (np.cos(np.deg2rad(theta_new))))
    y_new = custom_coord_round(y + step_size * (np.sin(np.deg2rad(theta_new))))
    obs = check_obstacles(x_new, y_new)

    if obs:
        new_cost_to_go = np.sqrt(((x_new - x_f) ** 2) + ((y_new - y_f) ** 2))
        new_cost_to_come = cost_to_come + step_size
        new_total_cost = new_cost_to_go + new_cost_to_come
        check_new_node(x_new, y_new, theta_new, new_total_cost, new_cost_to_go,
new_cost_to_come)

# Action2 - moving robot to a step size at 30deg
def action2(pop):
    x, y, theta = pop[0]
    cost_to_come = pop[1][2]
    theta_new = custom_ang_round(theta + 30)
    x_new = custom_coord_round(x + step_size * (np.cos(np.deg2rad(theta_new))))
    y_new = custom_coord_round(y + step_size * (np.sin(np.deg2rad(theta_new))))
    obs = check_obstacles(x_new, y_new)

    if obs:
        new_cost_to_go = np.sqrt(((x_new - x_f) ** 2) + ((y_new - y_f) ** 2))
        new_cost_to_come = cost_to_come + step_size
        new_total_cost = new_cost_to_go + new_cost_to_come
        check_new_node(x_new, y_new, theta_new, new_total_cost, new_cost_to_go,
new_cost_to_come)
```

```
# Action3 - moving robot to a step size at 0deg
def action3(pop):
   x, y, theta = pop[0]
   cost_to_come = pop[1][2]
   theta_new = custom_ang_round(theta)
   x_new = custom_coord_round(x + step_size * (np.cos(np.deg2rad(theta_new))))
   y_new = custom_coord_round(y + step_size * (np.sin(np.deg2rad(theta_new))))
   obs = check_obstacles(x_new, y_new)

   if obs:
      new_cost_to_go = np.sqrt(((x_new - x_f) ** 2) + ((y_new - y_f) ** 2))
      new_cost_to_come = cost_to_come + step_size
      new_total_cost = new_cost_to_go + new_cost_to_come
      check_new_node(x_new, y_new, theta_new, new_total_cost, new_cost_to_go,
new_cost_to_come)

# Action4 - moving robot to a step size at 330deg
def action4(pop):
   x, y, theta = pop[0]
   cost_to_come = pop[1][2]
   theta_new = custom_ang_round(theta - 30)
   x_new = custom_coord_round(x + step_size * (np.cos(np.deg2rad(theta_new))))
   y_new = custom_coord_round(y + step_size * (np.sin(np.deg2rad(theta_new))))
   obs = check_obstacles(x_new, y_new)

   if obs:
      new_cost_to_go = np.sqrt(((x_new - x_f) ** 2) + ((y_new - y_f) ** 2))
      new_cost_to_come = cost_to_come + step_size
      new_total_cost = new_cost_to_go + new_cost_to_come
      check_new_node(x_new, y_new, theta_new, new_total_cost, new_cost_to_go,
new_cost_to_come)

# Action5 - moving robot to a step size at 300deg
def action5(pop):
   x, y, theta = pop[0]
   cost_to_come = pop[1][2]
   theta_new = custom_ang_round(theta - 60)
   x_new = custom_coord_round(x + step_size * (np.cos(np.deg2rad(theta_new))))
   y_new = custom_coord_round(y + step_size * (np.sin(np.deg2rad(theta_new))))
   obs = check_obstacles(x_new, y_new)

   if obs:
      new_cost_to_go = np.sqrt(((x_new - x_f) ** 2) + ((y_new - y_f) ** 2))
      new_cost_to_come = cost_to_come + step_size
```

```
        new_total_cost = new_cost_to_go + new_cost_to_come
        check_new_node(x_new, y_new, theta_new, new_total_cost, new_cost_to_go,
new_cost_to_come)

# Backtrack to find the optimal path
def backtracking(x, y, theta):
    backtrack.append((x, y, theta))
    key = node_records[(x, y, theta)]
    backtrack.append(key)
    while key != init_pos:
        key = node_records[key]
        backtrack.append(key)
    return backtrack[::-1]

# Find intersecting coordinates based on (m,c) values
def find_intersection(m1, m2, c1, c2, a, b):
    A = np.array([[-m1, a], [-m2, b]])
    B = np.array([c1, c2])
    X = np.linalg.solve(A, B)
    return X

# Convert coordinates into pygame coordinates
def to_pygame(coords, height):
    return coords[0], height - coords[1]

# Convert an object's coordinates into pygame coordinates
def rec_pygame(coords, height, obj_height):
    return coords[0], height - coords[1] - obj_height

# Plot arrow
def arrow(screen, lcolor, tricolor, start, end, trirad):
    pygame.draw.line(screen, lcolor, start, end, 1)
    rotation = math.degrees(math.atan2(start[1]-end[1], end[0]-start[0]))+90
    pygame.draw.polygon(screen, tricolor, ((end[0]+trirad*math.sin(math.radians(rotation)),
end[1]+trirad*math.cos(math.radians(rotation))),
                                          (end[0]+trirad*math.sin(math.radians(rotation-120)),
                                           end[1]+trirad*math.cos(math.radians(rotation-120))),
                                          (end[0]+trirad*math.sin(math.radians(rotation+120)),
end[1]+trirad*math.cos(math.radians(rotation+120)))))

#######Global initializations######
obstacle_buffer = int(input('Obstacle buffer value in integer (ex. 2): '))
robot_size = int(input('Size of Robot in integer (ex. 2): '))
obstacles = obstacles(obstacle_buffer,robot_size)
```

```
py_obstacles = pygame_obstacles(obstacle_buffer)

init_pos = input('Initial position (x, y & theta separated by spaces - Example : 10 95 5): ')
init_pos = tuple(int(i) for i in init_pos.split(" "))
x_s = custom_coord_round(init_pos[0])
y_s = custom_coord_round(init_pos[1])
theta_s = custom_ang_round(init_pos[2])
init_pos = (x_s,y_s,theta_s)

goal_pos = input('Goal position (x, y & theta separated by spaces - Example : 170 220 90): ')
goal_pos = tuple(int(i) for i in goal_pos.split(" "))
x_f = goal_pos[0]
y_f = goal_pos[1]
theta_f = goal_pos[2]

step_size = int(input('Step size between and including 1 and 10, in integer: '))

# Global variable initialization
explored_nodes = heapdict.heapdict()
explored_mapping = []
visited_nodes = np.zeros((1200, 500, 12))
visited_nodes_track = OrderedSet()
backtrack = []
node_records = {}
pop = []
index = 0
the_path = []

# The A* algorithm
if __name__ == '__main__':
    start = time.time()

    if check_obstacles(x_s, y_s) and check_obstacles(x_f, y_f):
        print('A-starring........')
        init_cost_to_go = round(np.sqrt(((x_s - x_f) ** 2) + ((y_s - y_f) ** 2)), 1)
        init_cost_to_come = 0
        init_total_cost = init_cost_to_come + init_cost_to_go
        explored_nodes[(x_s, y_s, theta_s)] = init_total_cost, init_cost_to_go, init_cost_to_come
        explored_mapping.append((x_s, y_s))
        while len(explored_nodes):
            pop = explored_nodes.popitem()
            index += 1
            if not (x_f - 0.75 < pop[0][0] < x_f + 0.75 and y_f - 0.75 < pop[0][1] < y_f + 0.75):
                if visited_nodes_threshold_check(pop[0][0], pop[0][1], pop[0][2]):
```

```
                visited_nodes[int(2 * pop[0][0])][int(2 * pop[0][1])][int(pop[0][2] / 30)] = 1

                action1(pop)

                action2(pop)

                action3(pop)

                action4(pop)

                action5(pop)

            else:
                print('Goal Reached!')
                print('Last Pop: ', pop)
                the_path = backtracking(pop[0][0], pop[0][1], pop[0][2])
                print('Backtracking: ', the_path)
                end = time.time()
                print('Time: ', round((end - start), 2), 's')
                break

        if not len(explored_nodes):
            print('No solution found.')
            print('Last Pop: ', pop)

    elif not check_obstacles(x_s, y_s):
        print('Cannot A-star, starting node in an obstacle space.')
    elif not check_obstacles(x_f, y_f):
        print('Cannot A-star, goal node in an obstacle space.')

####Pygame Visualization####
pygame.init()
video = vidmaker.Video("a_star_shreejay_aaqib.mp4", late_export=True)
size = [600, 250]
d = obstacle_buffer
monitor = pygame.display.set_mode(size)
pygame.display.set_caption("Arena")

Done = False
clock = pygame.time.Clock()
while not Done:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            Done = True
```

```
    monitor.fill("black")

    # Walls
    pygame.draw.rect(monitor, "red", [0, 0, d, 250], 0)
    pygame.draw.rect(monitor, "red", [0, 0, 600, d], 0)
    pygame.draw.rect(monitor, "red", [0, 250-d, 600, d], 0)
    pygame.draw.rect(monitor, "red", [600-d, 0, d, 250], 0)

    # Rectangles
    x, y = rec_pygame([100-d, 0], 250, 100+d)
    pygame.draw.rect(monitor, "red", [x, y, 50+2*d, 100+d], 0)

    x, y = rec_pygame([100-d, 150-d], 250, 100+d)
    pygame.draw.rect(monitor, "red", [x, y, 50+2*d, 100+d], 0)

    x, y = rec_pygame([100, 0], 250, 100)
    pygame.draw.rect(monitor, "orange", [x, y, 50, 100], 0)

    x, y = rec_pygame([100, 150], 250, 100)
    pygame.draw.rect(monitor, "orange", [x, y, 50, 100], 0)

    # Triangle
    T1 = find_intersection(-1,0,py_obstacles[12][1],py_obstacles[13][1],0,1)
    T2 =
find_intersection(py_obstacles[13][0],py_obstacles[14][0],py_obstacles[13][1],py_obstacles[14]
[1],1,1)
    T3 =
find_intersection(py_obstacles[14][0],py_obstacles[15][0],py_obstacles[14][1],py_obstacles[15]
[1], 1, 1)
    T4 = find_intersection(py_obstacles[15][0],0,py_obstacles[15][1],py_obstacles[16][1], 1, 1)
    T5 = find_intersection(-1,py_obstacles[16][0],py_obstacles[12][1],py_obstacles[16][1], 0, 1)

    a, b = to_pygame(T1, 250)
    c, d = to_pygame(T2, 250)
    e, f = to_pygame(T3, 250)
    g, h = to_pygame(T4, 250)
    i, j = to_pygame(T5, 250)
    pygame.draw.polygon(monitor, "red", ([a, b], [c, d], [e, f], [g, h], [i, j]), 0)

    a, b = to_pygame([460, 25], 250)
    c, d = to_pygame([460, 225], 250)
    e, f = to_pygame([510, 125], 250)
    pygame.draw.polygon(monitor, "orange", [[a, b], [c, d], [e, f]], 0)
```

```
    # Hexagon
    H1 = find_intersection(py_obstacles[6][0], -1, py_obstacles[6][1], py_obstacles[7][1], 1, 0)
    H2 = find_intersection(-1, py_obstacles[8][0], py_obstacles[7][1], py_obstacles[8][1], 0, 1)
    H3 = find_intersection(py_obstacles[8][0], py_obstacles[9][0], py_obstacles[8][1],
py_obstacles[9][1], 1, 1)
    H4 = find_intersection(py_obstacles[9][0], -1, py_obstacles[9][1], py_obstacles[10][1], 1, 0)
    H5 = find_intersection(-1, py_obstacles[11][0], py_obstacles[10][1], py_obstacles[11][1], 0, 1)
    H6 = find_intersection(py_obstacles[11][0], py_obstacles[6][0],py_obstacles[11][1],
py_obstacles[6][1], 1, 1)

    a, b = to_pygame(H1, 250)
    c, d = to_pygame(H2, 250)
    e, f = to_pygame(H3, 250)
    g, h = to_pygame(H4, 250)
    i, j = to_pygame(H5, 250)
    k, l = to_pygame(H6, 250)
    pygame.draw.polygon(monitor, "red", [[a, b], [c, d], [e, f], [g, h], [i, j], [k, l]], 0)

    pygame.draw.polygon(monitor, "orange", ((235.05, 87.5), (300, 50),(364.95, 87.5), (364.95,
162.5), (300, 200),
                                            (235.05, 162.5)))

    # Simulation of visited nodes and Backtracking
    for l in range(len(visited_nodes_track) - 2):
        m = visited_nodes_track[l]
        n = node_records[m]
        m = to_pygame(m, 250)
        n = to_pygame(n, 250)
        video.update(pygame.surfarray.pixels3d(monitor).swapaxes(0, 1), inverted=False)
        arrow(monitor, "white", (0, 0, 0),[m[0], m[1]], [n[0], n[1]], 0.5)
        pygame.display.flip()
        clock.tick(300)
    for i in the_path:
        pygame.draw.circle(monitor, (0, 255, 0), to_pygame(i, 250), robot_size)
        video.update(pygame.surfarray.pixels3d(monitor).swapaxes(0, 1), inverted=False)
        pygame.display.flip()
        clock.tick(20)

    pygame.display.flip()
    pygame.time.wait(10000)
    Done = True

pygame.quit()
video.export(verbose=True)
```