~/Desktop/Proj3_FNU_aryan_keyur/a_star_FNU_aryan_keyur.py

```
#ENPM 661: Planning for Robotics
#Project 3 - Phase 1
#authors: Keyur Borad, Aryan Mishra, FNU Koustubh
# GitHub link: https://github.com/koustubh1012/A-Star-path-planning



# import libraries

import cv2
import numpy as np
import heapq as hq
import math
import time

canvas = np.ones((501,1201,3))   # creating a frame for video generation
obstacle_set = set()             # set to store the obstacle points
obstacle_list = []               # list to store the obstacle points in order for videp

c2c_node_grid = [[float('inf')] * 500 for _ in range(1200)]       # create a 2D array for storing cost to come
tc_node_grid = [[float('inf')] * 500 for _ in range(1200)]        # create a 2D array for storing cost to come
closed_set = []               # set to store the value of visited and closed points
closed_list = np.zeros((1200, 500, 12))
visited={}

'''
Loop to define the obstacle points in the map
'''
C = int(input("Enter the clearance from the obstacle in pixel: "))     # Get clearance from the user
R = int(input("Enter the radius of the robot in pixel: "))             # Get the robot radius from user


x_goal = 0  # Initialize the goal x coordinate
y_goal = 0  # Initialize the goal y coordinate
x_start = 0 # Initialize the start x coordinate
y_start = 0 # Initialize the start y coordinate

# Funtion to update the visted nodes
def visited_node(node):
    visited.update({node[2]:node[4]})

def move_forward(node):                                  # Function to move the robot forward
    new_heading = node[5]                                # get the current heading of the robot
    x = node[4][0] + L*np.cos(np.deg2rad(new_heading)) # calculate the new x coordinate
    y = node[4][1] + L*np.sin(np.deg2rad(new_heading)) # calculate the new y coordinate
    x = round(x)
```

```
    y = round(y)
    c2c = node[1]+L                                 # calculate the cost to come
    c2g = math.sqrt((y_goal-y)**2 + (x_goal-x)**2)  # calculate the cost to goal
    tc = c2c + c2g                                  # calculate the total cost
    return (x,y),new_heading,tc,c2c                 # return the new node's coordinates, heading, total cost and cost to come


def move_30(node):                                  # Function to move the robot by 30 degrees
    new_heading = (node[5] + 30) % 360              # calculate the new heading
    x = node[4][0] + L*np.cos(np.deg2rad(new_heading))  # calculate the new x coordinate
    y = node[4][1] + L*np.sin(np.deg2rad(new_heading))  # calculate the new y coordinate
    x = round(x)
    y = round(y)
    c2c = node[1]+L                                 # calculate the cost to come
    c2g = math.sqrt((y_goal-y)**2 + (x_goal-x)**2)      # calculate the cost to goal
    tc = c2c + c2g                                  # calculate the total cost
    return (x,y),new_heading,tc,c2c                 # return the new node's coordinates, heading, total cost and cost to come


def move_minus_30(node):                            # Function to move the robot by -30 degrees
    new_heading = (node[5] - 30) % 360              # calculate the new heading
    x = node[4][0] + L*np.cos(np.deg2rad(new_heading))  # calculate the new x coordinate
    y = node[4][1] + L*np.sin(np.deg2rad(new_heading))  # calculate the new y coordinate
    x = round(x)
    y = round(y)
    c2c = node[1]+L                                 # calculate the cost to come
    c2g = math.sqrt((y_goal-y)**2 + (x_goal-x)**2)      # calculate the cost to goal
    tc = c2c + c2g                                  # calculate the total cost
    return (x,y),new_heading,tc,c2c                 # return the new node's coordinates, heading, total cost and cost to come


def move_60(node):                                  # Function to move the robot by 60 degrees
    new_heading = (node[5] + 60) % 360              # calculate the new heading
    x = node[4][0] + L*np.cos(np.deg2rad(new_heading))  # calculate the new x coordinate
    y = node[4][1] + L*np.sin(np.deg2rad(new_heading))  # calculate the new y coordinate
    x = round(x)
    y = round(y)
    c2c = node[1]+L                                 # calculate the cost to come
    c2g = math.sqrt((y_goal-y)**2 + (x_goal-x)**2)      # calculate the cost to goal
    tc = c2c + c2g                                  # calculate the total cost
    return (x,y),new_heading,tc,c2c                 # return the new node's coordinates, heading, total cost and cost to come
```

```
91
92  def move_minus_60(node):                                    # Function to move the robot
    by -60 degrees
93      new_heading = (node[5] - 60) % 360                      # calculate the new heading
94      x = node[4][0] + L*np.cos(np.deg2rad(new_heading))      # calculate the new x
    coordinate
95      y = node[4][1] + L*np.sin(np.deg2rad(new_heading))      # calculate the new y
    coordinate
96      x = round(x)
97      y = round(y)
98      c2c = node[1]+L                                          # calculate the cost to
    come
99      c2g = math.sqrt((y_goal-y)**2 + (x_goal-x)**2)           # calculate the cost to
    goal
100     tc = c2c + c2g                                           # calculate the total cost
101     return (x,y),new_heading,tc,c2c                          # return the new node's
    coordinates, heading, total cost and cost to come
102
103 T = C + R
104
105 for y in range(500):                                         # loop to define the
    obstacle points : x
106     for x in range(1200):                                    # loop to define the
    obstacle points : y
107         canvas[y,x] = [255,255,255]                          # mark the points in the
    frame with white color
108         if (0<=y<=T):                                        # points in the bottom
    boundary
109             obstacle_set.add((x,y))                          # add the points to the
    obstacle set
110             obstacle_list.append((x,y))                      # add the points to the
    obstacle list
111             c2c_node_grid[x][y] = -1                         # mark the points in the
    cost to come grid with -1
112             tc_node_grid[x][y] = -1                          # mark the points in the
    total cost grid with -1
113         elif (0<=x<=T):                                      # points in the left
    boundary
114             obstacle_set.add((x,y))                          # add the points to the
    obstacle set
115             obstacle_list.append((x,y))                      # add the points to the
    obstacle list
116             c2c_node_grid[x][y] = -1                         # mark the points in the
    cost to come grid with -1
117             tc_node_grid[x][y] = -1                          # mark the points in the
    total cost grid with -1
118         elif (500-T<=y<500):                                 # points in the top
    boundary
119             obstacle_set.add((x,y))                          # add the points to the
    obstacle set
120             obstacle_list.append((x,y))                      # add the points to the
    obstacle list
121             c2c_node_grid[x][y] = -1                         # mark the points in the
    cost to come grid with -1
122             tc_node_grid[x][y] = -1                          # mark the points in the
    total cost grid with -1
123         elif (1200-T<=x<1200):                               # points in the right
    boundary
124             obstacle_set.add((x,y))                          # add the points to the
    obstacle set
125             obstacle_list.append((x,y))                      # add the points to the
    obstacle list
```

```
126             c2c_node_grid[x][y] = -1                        # mark the points in the
    cost to come grid with -1
127             tc_node_grid[x][y] = -1                         # mark the points in the
    total cost grid with -1
128         elif (100-T<=x<=175+T) and (100-T<=y<500):          # points in first
    rectangle
129             obstacle_set.add((x,y))                         # add the points to the
    obstacle set
130             obstacle_list.append((x,y))                     # add the points to the
    obstacle list
131             c2c_node_grid[x][y] = -1                        # mark the points in the
    cost to come grid with -1
132             tc_node_grid[x][y] = -1                         # mark the points in the
    total cost grid with -1
133         elif (275-T<=x<=350+T and 0<=y<=400+T):             # points in second
    rectangle
134             obstacle_set.add((x,y))                         # add the points to the
    obstacle set
135             obstacle_list.append((x,y))                     # add the points to the
    obstacle list
136             c2c_node_grid[x][y] = -1                        # mark the points in the
    cost to come grid with -1
137             tc_node_grid[x][y] = -1                         # mark the points in the
    total cost grid with -1
138
139         # Points int the Concave shaped obstacle
140         elif (900-T<=x<=1015+T) and (50-T<=y<=125+T):       # points in the first
    rectangle of Concave shaped obstacle
141             obstacle_set.add((x,y))                         # add the points to the
    obstacle set
142             obstacle_list.append((x,y))                     # add the points to the
    obstacle list
143             c2c_node_grid[x][y] = -1                        # mark the points in the
    cost to come grid with -1
144             tc_node_grid[x][y] = -1                         # mark the points in the
    total cost grid with -1
145         elif (1020-T<=x<1100+T and 50-T<=y<=450+T):         # points in the second
    rectangle of C-shaped obstacle
146             obstacle_set.add((x,y))                         # add the points to the
    obstacle set
147             obstacle_list.append((x,y))                     # add the points to the
    obstacle list
148             c2c_node_grid[x][y] = -1                        # mark the points in the
    cost to come grid with -1
149             tc_node_grid[x][y] = -1                         # mark the points in the
    total cost grid with -1
150         elif (900-T<=x<=1015+T and 375-T<=y<=450+T):        # points in the third
    rectangle of C-shaped obstacle
151             obstacle_set.add((x,y))                         # add the points to the
    obstacle set
152             obstacle_list.append((x,y))                     # add the points to the
    obstacle list
153             c2c_node_grid[x][y] = -1                        # mark the points in the
    cost to come grid with -1
154             tc_node_grid[x][y] = -1                         # mark the points in the
    total cost grid with -1
155
156         # Points in the hexagon obstacle
157         elif(520-T<=x<=780+T) and (175<=y<=325):            # points in the hexagon
    obstacle
158             obstacle_set.add((x,y))                         # add the points to the
    obstacle set
```

```
159             obstacle_list.append((x,y))                     # add the points to the obstacle list
160             c2c_node_grid[x][y] = -1                        # mark the points in the cost to come grid with -1
161             tc_node_grid[x][y] = -1                         # mark the points in the total cost grid with -1
162         elif(325<=y<=400+T) and (y+0.577*x-775.05-(T/math.sin(np.deg2rad(60)))<=0) and (y-0.577*x-24.95-(T/math.sin(np.deg2rad(60)))<=0) and (520-T<=x<=780+T):
163             # points in the hexagon obstacle above the center
164             obstacle_set.add((x,y))                         # add the points to the obstacle set
165             obstacle_list.append((x,y))                     # add the points to the obstacle list
166             c2c_node_grid[x][y] = -1                        # mark the points in the cost to come grid with -1
167             tc_node_grid[x][y] = -1                         # mark the points in the total cost grid with -1
168         elif(100-T<=y<=175) and (y+0.577*x-475.05+(T/math.sin(np.deg2rad(60))))>=0 and (y-0.577*x+275.05+(T/math.sin(np.deg2rad(60)))>=0) and (520-T<=x<=780+T):
169             # points in the hexagon obstacle below the center
170             obstacle_set.add((x,y))                         # add the points to the obstacle set
171             obstacle_list.append((x,y))                     # add the points to the obstacle list
172             c2c_node_grid[x][y] = -1                        # mark the points in the cost to come grid with -1
173             tc_node_grid[x][y] = -1                         # mark the points in the total cost grid with -1
174
175 valid_start = False                                                 # flag to check if the start point is valid
176
177 while not valid_start:                                             # loop to check if the start point is valid
178         x_start = int(input("Enter the start x coordinate: "))      # get the start x coordinate from the user
179         y_start = int(input("Enter the start y coordinate: "))      # get the start y coordinate from the user
180         theta_start = int(input("Enter the initial heading: "))     # get the start heading from the user
181         if (x_start, y_start) in obstacle_set:                      # check if the start point is in the obstacle set
182             print("Invalid coordinates, Enter again: ")             # print error message
183         else:
184             initial_node = (0, 0, 1, [], (x_start, y_start), theta_start)       # create the initial node
185             valid_start = True                                      # set the flag to true
186
187 valid_goal = False                                                  # flag to check if the goal point is valid
188
189 while not valid_goal:
190         x_goal = int(input("Enter the goal x coordinate: "))        # get the goal x coordinate from the user
191         y_goal = int(input("Enter the goal y coordinate: "))        # get the goal y coordinate from the user
192         theta_goal = int(input("Enter the goal heading: "))         # get the goal heading from the user
193         if (x_goal, y_goal) in obstacle_set:                        # check if the goal point is in the obstacle set
```

```
            print("Invalid coordinates, Enter again: ")                         # print error message
        else:
            goal = (x_goal, y_goal)                                             # create the goal node
            valid_goal = True                                                   # set the flag to true

valid_step = False                                                              # flag to check if the step size is valid

while not valid_step:                                                           # loop to check if the step size is valid
        L = int(input("Enter step size: "))                                     # get the step size from the user
        if not (1 <= L <= 10):                                                  # check if the step size is between 1 and 10
            print("Invalid Step Size, Enter Again: ")                           # print error message
        else:
            valid_step = True                                                   # set the flag to true
start_time = time.time()
new_index = 1
open_list = []
hq.heappush(open_list,initial_node)        # Push initial node to the list
hq.heapify(open_list)                      # covers list to heapq data type
while(open_list):
    # total cost ,cost to come, index, parent node index, coordinate values (x,y), orientation
    node = hq.heappop(open_list)       # pop the node with lowest cost to come
    closed_set.append(node[4])            # add the node coordinates to closed set
    closed_list[int(node[4][0]), int(node[4][1]), int(node[5]/30)] = 1          # add the node to the closed list
    visited_node(node)                 # add the node to the visited list
    index = node[2]                    # store the index of the current node
    parent_index = node[3]             # store the parent index list of current node
    node_dist = math.sqrt((node[4][0]-x_goal)**2 + (node[4][1]-y_goal)**2)     # calculate the distance between the current node and goal node
    if node_dist<3 and (abs(node[5]-theta_goal)<=30 or abs(theta_goal-node[5])<=30): # if the node is goal position, exit the loop
        print("Goal reached")
        break
    point, new_heading, tc, c2c = move_30(node)                          # get the new node's coordinates and cost to come
    # print(int(point[0]), int(point[1]), int(new_heading/30))
    if point not in obstacle_set and closed_list[int(point[0]), int(point[1]), int(new_heading/30)] == 0:          # check if the new node is in the obstacle set or visited list
        x = point[0]                                                            # get the x coordinate of the new node
        y = point[1]                                                            # get the y coordinate of the new node
        if tc<tc_node_grid[x][y]:                                               # check if the new cost to come is less than original cost to come
            new_parent_index = parent_index.copy()                              # copy the parent index list of the current node
            new_parent_index.append(index)                                      # Append the current node's index to the new node's parent index list
            new_index+=1                                                        # increment the index
```

```
            tc_node_grid[x][y] = tc                                     # Update the new total cost
            c2c_node_grid[x][y] = c2c                                   # Update the new cost to come
            new_node = (tc, c2c, new_index, new_parent_index, point, new_heading) # create the new node
            hq.heappush(open_list, new_node)                            # push the new node to the open list
            # print(new_node)

    point, new_heading, tc, c2c = move_60(node)                         # get the new node's coordinates and cost to come
    # print(int(point[0]), int(point[1]), int(new_heading/30))
    if point not in obstacle_set and closed_list[int(point[0]), int(point[1]), int(new_heading/30)] == 0:    # check if the new node is in the obstacle set or visited list
        x = point[0]                                                    # get the x coordinate of the new node
        y = point[1]                                                    # get the x coordinate of the new node
        if tc<tc_node_grid[x][y]:                                       # check if the new cost to come is less than original cost to come
            new_parent_index = parent_index.copy()                      # copy the parent index list of the current node
            new_parent_index.append(index)                              # Append the current node's index to the new node's parent index list
            new_index+=1                                                # increment the index
            tc_node_grid[x][y] = tc                                     # Update the new total cost
            c2c_node_grid[x][y] = c2c                                   # Update the new cost to come
            new_node = (tc, c2c, new_index, new_parent_index, point, new_heading)
            hq.heappush(open_list, new_node)                            # push the new node to the open list
            # print(new_node)


    point, new_heading, tc, c2c = move_forward(node)                    # get the new node's coordinates and cost to come
    # print(int(point[0]), int(point[1]), int(new_heading/30))
    if point not in obstacle_set and closed_list[int(point[0]), int(point[1]), int(new_heading/30)] == 0:           # check if the new node is in the obstacle set or visited list
        x = point[0]                                                    # get the x coordinate of the new node
        y = point[1]                                                    # get the y coordinate of the new node
        if tc<tc_node_grid[x][y]:                                       # check if the new cost to come is less than original cost to come
            new_parent_index = parent_index.copy()                      # copy the parent index list of the current node
            new_parent_index.append(index)                              # Append the current node's index to the new node's parent index list
            new_index+=1                                                # increment the index
            tc_node_grid[x][y] = tc                                     # Update the new total cost
            c2c_node_grid[x][y] = c2c                                   # Update the new cost to come
            new_node = (tc, c2c, new_index, new_parent_index, point, new_heading)
            hq.heappush(open_list, new_node)                            # push the new node to the open list
```

```
            # print(new_node)


    point, new_heading, tc, c2c = move_minus_30(node)
# get the new node's coordinates and cost to come
    # print(int(point[0]), int(point[1]), int(new_heading/30))
    if point not in obstacle_set and closed_list[int(point[0]), int(point[1]),
int(new_heading/30)] == 0:           # check if the new node is in the obstacle set
or visited list
        x = point[0]                                                    # get the x
coordinate of the new node
        y = point[1]                                                    # get the y
coordinate of the new node
        if tc<tc_node_grid[x][y]:                                       # check if
the new cost to come is less than original cost to come
            new_parent_index = parent_index.copy()                      # copy the
parent index list of the current node
            new_parent_index.append(index)                              # Append the
current node's index to the new node's parent index list
            new_index+=1                                                # increment
the index
            tc_node_grid[x][y] = tc                                     # Update the
new total cost
            c2c_node_grid[x][y] = c2c                                   # Update the
new cost to come
            new_node = (tc, c2c, new_index, new_parent_index, point, new_heading)
            hq.heappush(open_list, new_node)                            # push the
new node to the open list
            # print(new_node)


    point, new_heading, tc, c2c = move_minus_60(node)
# get the new node's coordinates and cost to come
    # print(int(point[0]), int(point[1]), int(new_heading/30))
    if point not in obstacle_set and closed_list[int(point[0]), int(point[1]),
int(new_heading/30)] == 0:           # check if the new node is in the obstacle set
or visited list
        x = point[0]                                                      # get the x
coordinate of the new node
        y = point[1]                                                      # get the y
coordinate of the new node
        if tc<tc_node_grid[x][y]:                                         # check if
the new cost to come is less than original cost to come
            new_parent_index = parent_index.copy()                        # copy the
parent index list of the current node
            new_parent_index.append(index)                                # Append
the current node's index to the new node's parent index list
            new_index+=1                                                  # increment
the index
            tc_node_grid[x][y] = tc                                       # Update
the new total cost
            c2c_node_grid[x][y] = c2c                                     # Update
the new cost to come
            new_node = (tc, c2c, new_index, new_parent_index, point, new_heading) #
create the new node
            hq.heappush(open_list, new_node)                              # push the
new node to the open list
            # print(new_node)
    # print(open_list)
    # break

print("The Robot reached at ", node[4],"with orientation ",node[5]," degrees")
```

```
305
306 # Mark the obstacle points in the frame, including points after bloating
307 for point in obstacle_list:                              # loop to mark the obstacle
    points
308     canvas[point[1],point[0]] = [255, 0, 0]              # mark the obstacle points
    with blue color
309
310 # Draw the obstacles in the frame, excluding the points after bloating
311 cv2.rectangle(canvas, (100, 499), (175, 100), (0 , 0, 255), -1)   # draw the first
    rectangle
312 cv2.rectangle(canvas, (275, 400), (350, 0), (0 , 0, 255), -1)     # draw the second
    rectangle
313 cv2.rectangle(canvas, (900, 125), (1100, 50), (0 , 0, 255), -1)   # draw the first
    rectangle of C-shaped obstacle
314 cv2.rectangle(canvas, (900, 450), (1100, 375), (0 , 0, 255), -1)  # draw the third
    rectangle of C-shaped obstacle
315 cv2.rectangle(canvas, (1020, 450), (1100, 50), (0, 0, 255), -1)   # draw the second
    rectangle of C-shaped obstacle
316 pts = np.array([[650, 400], [780, 325],                          # draw the hexagon
    obstacle
317                [780, 175], [650, 100],
318                [520, 175], [520, 325]],
319                np.int32)
320 canvas = cv2.fillPoly(canvas, [np.array(pts)], color=(0, 0, 255)) # fill the hexagon
    obstacle with red color
321 cv2.circle(canvas,(x_start, y_start), 5, (0,0,255),-1)           # mark the start
    point with red color
322 cv2.circle(canvas,(x_goal, y_goal), 5, (0,0,255), -1)           # mark the goal
    point with red color
323
324
325 print("Processing Video...")
326
327 path = node[3]           # Get the parent node list
328 counter = 0              # counter to count the frames to write on video
329
330 fourcc = cv2.VideoWriter_fourcc(*'mp4v')  # Codec for MP4 format
331 video_writer = cv2.VideoWriter('output.mp4', fourcc, 60, (1200, 500)) # Video writer
    object
332
333 '''
334 Loop to mark the explored nodes in order on the frame
335 '''
336 print("Exploring map")
337
338 for node in closed_set:                                          # loop to
    mark the explored nodes
339     canvas[node[1], node[0]] = [0, 255, 0]                       # mark the
    explored nodes with green color
340     counter +=1                                                  # increment
    the counter
341     if counter%500 == 0 or counter == 0:                         # check if
    the counter is divisible by 500
342         canvas_flipped = cv2.flip(canvas,0)                      # flip the
    frame
343         canvas_flipped_uint8 = cv2.convertScaleAbs(canvas_flipped)       # convert
    the frame to uint8
344         # cv2.imshow('window',canvas_flipped_uint8)
345         # cv2.waitKey(1)
346         video_writer.write(canvas_flipped_uint8)                 # write the
```

```
frame to video

'''
Loop to mark the path created
'''
print("Backtracking")

for index in path:                                                  # loop to mark the path
    coord=visited[index]                                            # get the coordinates of the node
    cv2.circle(canvas, (coord[0],coord[1]), 1, [0,0,0], -1)         # mark the path with black color

    canvas_flipped = cv2.flip(canvas,0)                             # flip the frame
    canvas_flipped_uint8 = cv2.convertScaleAbs(canvas_flipped)      # convert the frame to uint8
    # cv2.imshow('window',canvas_flipped_uint8)
    # cv2.waitKey(1)

    video_writer.write(canvas_flipped_uint8)                        # write the frame to video



'''
Loop to add some additional frames at the end of the video
'''
for i in range(150):
    video_writer.write(canvas_flipped_uint8)                         # write the frame to video

print("Video Processed")                                             # print message
# cv2.waitKey(0)
video_writer.release()                                               # release the video writer
# cv2.destroyAllWindows()
end_time = time.time()                                               # get the end time of the program
print(f"The runtime of my program is {end_time - start_time} seconds.")    # print the runtime of the program
```