

~/Desktop/Proj3\_FNU\_aryan\_keyur/a\_star\_FNU\_aryan\_keyur.py

```

4 #ENPM 661: Planning for Robotics
5 #Project 3 - Phase 1
6 #authors: Keyur Borad, Aryan Mishra, FNU Koustubh
7 # GitHub link: https://github.com/koustubh1012/A-Star-path-planning
8
9
10
11 # import libraries
12
13 import cv2
14 import numpy as np
15 import heapq as hq
16 import math
17 import time
18
19 canvas = np.ones((501,1201,3)) # creating a frame for video generation
20 obstacle_set = set() # set to store the obstacle points
21 obstacle_list = [] # list to store the obstacle points in order for
  videp
22
23 c2c_node_grid = [[float('inf')] * 500 for _ in range(1200)] # create a 2D array
  for storing cost to come
24 tc_node_grid = [[float('inf')] * 500 for _ in range(1200)] # create a 2D array
  for storing cost to come
25 closed_set = [] # set to store the value of visited and closed points
26 closed_list = np.zeros((1200, 500, 12))
27 visited={}
28
29 ...
30 Loop to define the obstacle points in the map
31 ...
32 C = int(input("Enter the clearance from the obstacle in pixel: ")) # Get
  clearance from the user
33 R = int(input("Enter the radius of the robot in pixel: ")) # Get the
  robot radius from user
34
35
36 x_goal = 0 # Initialize the goal x coordinate
37 y_goal = 0 # Initialize the goal y coordinate
38 x_start = 0 # Initialize the start x coordinate
39 y_start = 0 # Initialize the start y coordinate
40
41 # Funtion to update the visted nodes
42 def visited_node(node):
43     visited.update({node[2]:node[4]})
44
45 def move_forward(node): # Function to move the robot
  forward
46     new_heading = node[5] # get the current heading of
  the robot
47     x = node[4][0] + L*np.cos(np.deg2rad(new_heading)) # calculate the new x
  coordinate
48     y = node[4][1] + L*np.sin(np.deg2rad(new_heading)) # calculate the new y
  coordinate
49     x = round(x)

```

```

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

55
56
57 def move_30(node):                                     # Function to move the
robot by 30 degrees
58     new_heading = (node[5] + 30) % 360                 # calculate the new heading
59     x = node[4][0] + L*np.cos(np.deg2rad(new_heading)) # calculate the new x
coordinate
60     y = node[4][1] + L*np.sin(np.deg2rad(new_heading)) # calculate the new y
coordinate
61     x = round(x)
62     y = round(y)
63     c2c = node[1]+L                                     # calculate the cost to
come
64     c2g = math.sqrt((y_goal-y)**2 + (x_goal-x)**2)      # calculate the cost to
goal
65     tc = c2c + c2g                                     # calculate the total cost
66     return (x,y),new_heading,tc,c2c                    # return the new node's
coordinates, heading, total cost and cost to come

67
68
69 def move_minus_30(node):                               # Function to move the
robot by -30 degrees
70     new_heading = (node[5] - 30) % 360                 # calculate the new heading
71     x = node[4][0] + L*np.cos(np.deg2rad(new_heading)) # calculate the new x
coordinate
72     y = node[4][1] + L*np.sin(np.deg2rad(new_heading)) # calculate the new y
coordinate
73     x = round(x)
74     y = round(y)
75     c2c = node[1]+L                                     # calculate the cost to
come
76     c2g = math.sqrt((y_goal-y)**2 + (x_goal-x)**2)      # calculate the cost to
goal
77     tc = c2c + c2g                                     # calculate the total cost
78     return (x,y),new_heading,tc,c2c                    # return the new node's
coordinates, heading, total cost and cost to come

79
80
81 def move_60(node):                                     # Function to move the
robot by 60 degrees
82     new_heading = (node[5] + 60) % 360                 # calculate the new heading
83     x = node[4][0] + L*np.cos(np.deg2rad(new_heading)) # calculate the new x
coordinate
84     y = node[4][1] + L*np.sin(np.deg2rad(new_heading)) # calculate the new y
coordinate
85     x = round(x)
86     y = round(y)
87     c2c = node[1]+L                                     # calculate the cost to
come
88     c2g = math.sqrt((y_goal-y)**2 + (x_goal-x)**2)      # calculate the cost to
goal
89     tc = c2c + c2g                                     # calculate the total cost
90     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

```

```

194         print("Invalid coordinates, Enter again: ") #
print error message
195     else:
196         goal = (x_goal, y_goal) #
create the goal node
197         valid_goal = True # set
the flag to true
198
199     valid_step = False # flag
to check if the step size is valid
200
201     while not valid_step: # loop
to check if the step size is valid
202         L = int(input("Enter step size: ")) # get
the step size from the user
203         if not (1 <= L <= 10): #
check if the step size is between 1 and 10
204             print("Invalid Step Size, Enter Again: ") #
print error message
205         else:
206             valid_step = True # set
the flag to true
207     start_time = time.time()
208     new_index = 1
209     open_list = []
210     hq.heappush(open_list, initial_node) # Push initial node to the list
211     hq.heapify(open_list) # covers list to heapq data type
212     while(open_list):
213         # total cost ,cost to come, index, parent node index, coordinate values (x,y),
orientation
214         node = hq.heappop(open_list) # pop the node with lowest cost to come
215         closed_set.append(node[4]) # add the node coordinates to closed set
216         closed_list[int(node[4][0]), int(node[4][1]), int(node[5]/30)] = 1 # add
the node to the closed list
217         visited_node(node) # add the node to the visited list
218         index = node[2] # store the index of the current node
219         parent_index = node[3] # store the parent index list of current node
220         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
221         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
222             print("Goal reached")
223             break
224         point, new_heading, tc, c2c = move_30(node) # get the
new node's coordinates and cost to come
225         # print(int(point[0]), int(point[1]), int(new_heading/30))
226         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
227             x = point[0] # get the x
coordinate of the new node
228             y = point[1] # get the y
coordinate of the new node
229             if tc<tc_node_grid[x][y]: # check if
the new cost to come is less than original cost to come
230                 new_parent_index = parent_index.copy() # copy the
parent index list of the current node
231                 new_parent_index.append(index) # Append the
current node's index to the new node's parent index list
232                 new_index+=1 # increment
the index

```



```

233         tc_node_grid[x][y] = tc                                # Update the
new total cost
234         c2c_node_grid[x][y] = c2c                            # Update the
new cost to come
235         new_node = (tc, c2c, new_index, new_parent_index, point, new_heading) #
create the new node
236         hq.heappush(open_list, new_node)                      # push the
new node to the open list
237         # print(new_node)
238
239         point, new_heading, tc, c2c = move_60(node)            # get the new
node's coordinates and cost to come
240         # print(int(point[0]), int(point[1]), int(new_heading/30))
241         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
242             x = point[0]                                       # get the x
coordinate of the new node
243             y = point[1]                                       # get the y
coordinate of the new node
244             if tc < tc_node_grid[x][y]:                        # check if
the new cost to come is less than original cost to come
245                 new_parent_index = parent_index.copy()        # copy the
parent index list of the current node
246                 new_parent_index.append(index)                # Append the
current node's index to the new node's parent index list
247                 new_index += 1                                # increment
the index
248                 tc_node_grid[x][y] = tc                        # Update the
new total cost
249                 c2c_node_grid[x][y] = c2c                    # Update the
new cost to come
250                 new_node = (tc, c2c, new_index, new_parent_index, point, new_heading)
251                 hq.heappush(open_list, new_node)              # push the
new node to the open list
252                 # print(new_node)
253
254
255         point, new_heading, tc, c2c = move_forward(node)      # get the new
node's coordinates and cost to come
256         # print(int(point[0]), int(point[1]), int(new_heading/30))
257         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
258             x = point[0]                                       # get the x
coordinate of the new node
259             y = point[1]                                       # get the y
coordinate of the new node
260             if tc < tc_node_grid[x][y]:                        # check if the
new cost to come is less than original cost to come
261                 new_parent_index = parent_index.copy()        # copy the
parent index list of the current node
262                 new_parent_index.append(index)                # Append the
current node's index to the new node's parent index list
263                 new_index += 1                                # increment
the index
264                 tc_node_grid[x][y] = tc                        # Update the
new total cost
265                 c2c_node_grid[x][y] = c2c                    # Update the
new cost to come
266                 new_node = (tc, c2c, new_index, new_parent_index, point, new_heading)
267                 hq.heappush(open_list, new_node)              # push the new
node to the open list

```

```

268         # print(new_node)
269
270
271     point, new_heading, tc, c2c = move_minus_30(node)
272     # get the new node's coordinates and cost to come
273     # print(int(point[0]), int(point[1]), int(new_heading/30))
274     if point not in obstacle_set and closed_list[int(point[0]), int(point[1]),
275     int(new_heading/30)] == 0: # check if the new node is in the obstacle set
276     or visited list
277         x = point[0] # get the x
278         coordinate of the new node
279         y = point[1] # get the y
280         coordinate of the new node
281         if tc < tc_node_grid[x][y]: # check if
282         the new cost to come is less than original cost to come
283         new_parent_index = parent_index.copy() # copy the
284         parent index list of the current node
285         new_parent_index.append(index) # Append the
286         current node's index to the new node's parent index list
287         new_index += 1 # increment
288         the index
289         tc_node_grid[x][y] = tc # Update the
290         new total cost
291         c2c_node_grid[x][y] = c2c # Update the
292         new cost to come
293         new_node = (tc, c2c, new_index, new_parent_index, point, new_heading)
294         hq.heappush(open_list, new_node) # push the
295         new node to the open list
296         # print(new_node)
297
298
299     point, new_heading, tc, c2c = move_minus_60(node)
300     # get the new node's coordinates and cost to come
301     # print(int(point[0]), int(point[1]), int(new_heading/30))
302     if point not in obstacle_set and closed_list[int(point[0]), int(point[1]),
303     int(new_heading/30)] == 0: # check if the new node is in the obstacle set
304     or visited list
305         x = point[0] # get the x
306         coordinate of the new node
307         y = point[1] # get the y
308         coordinate of the new node
309         if tc < tc_node_grid[x][y]: # check if
310         the new cost to come is less than original cost to come
311         new_parent_index = parent_index.copy() # copy the
312         parent index list of the current node
313         new_parent_index.append(index) # Append
314         the current node's index to the new node's parent index list
315         new_index += 1 # increment
316         the index
317         tc_node_grid[x][y] = tc # Update
318         the new total cost
319         c2c_node_grid[x][y] = c2c # Update
320         the new cost to come
321         new_node = (tc, c2c, new_index, new_parent_index, point, new_heading) #
322         create the new node
323         hq.heappush(open_list, new_node) # push the
324         new node to the open list
325         # print(new_node)
326         # print(open_list)
327         # break
328
329     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.cvtColor(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
347
348 '''
349 Loop to mark the path created
350 '''
351 print("Backtracking")
352
353 for index in path:                                # loop to
mark the path                                     # get the
354     coord=visited[index]                           # mark the
coordinates of the node                           path with black color
355     cv2.circle(canvas, (coord[0],coord[1]), 1, [0,0,0], -1)
356
357     canvas_flipped = cv2.flip(canvas,0)             # flip the
frame                                              # convert
358     canvas_flipped_uint8 = cv2.convertScaleAbs(canvas_flipped)
the frame to uint8                               # write the
359     # cv2.imshow('window',canvas_flipped_uint8)
360     # cv2.waitKey(1)
361
362     video_writer.write(canvas_flipped_uint8)
frame to video
363
364
365
366 '''
367 Loop to add some additional frames at the end of the video
368 '''
369 for i in range(150):
370     video_writer.write(canvas_flipped_uint8)        # write
the frame to video
371
372 print("Video Processed")                          # print
message
373 # cv2.waitKey(0)
374 video_writer.release()                            # release
the video writer
375 # cv2.destroyAllWindows()
376 end_time = time.time()                             # get the
end time of the program
377 print(f"The runtime of my program is {end_time - start_time} seconds.") # print
the runtime of the program
```