# ME 401/5501 – Robotics and Unmanned Systems HW #2: DUE February 8th, 2022

### LATE HOMEWORK WILL BE PENALIZED 10% PER DAY

### Problem 1 (10 pts):

Create a function that checks if the current **node** is valid based upon the list of obstacles, grid boundaries, and current location.

Using an obstacle list of (1,1), (4,4), (3,4), (5,0), (5,1), (0,7), (1,7), (2,7), and (3,7); and a bounding box of 0 to 10 for both x and y, and step size of 0.5, verify that the location (2,2) is valid. Assume the obstacles have a diameter of 0.5 (only occupy the node at which they reside).

Pass the obstacle list, node, and map boundaries/step size, and return a Boolean True/False depending on if the node location is valid (reachable).

```
Submit your Python code.
My Script:
# -*- coding: utf-8 -*-
Created on Sat Feb 5 18:21:29 2022
@author: alex
import numpy as np
import math as m
def check_within_obstacle(obstacle_list, current_position, obstacle_radius):
  """check if I am within collision of obstacle return True if it is
  false if I'm not"""
  for obstacle in obstacle list:
     distance = compute_distance(current_position, obstacle)
     if abs(distance)<=obstacle radius:
       return True
     else:
       return False
def check_if_obstacle_is_present(obstacle_list, node_in_question):
  """check to see if an obstacle is in the way"""
  for obstacle in obstacle_list:
     if obstacle == node_in_question:
       return True
```

```
def check_obstacle_exists(obstacle_list):
  """sanity check to see if obstacle exists"""
  for obst in obstacle list:
     if configSpace.search_space[obst[0],obst[1]] == 1:
        print("yes", configSpace.search_space[obst[0],obst[1]])
def compute_distance(current_pos, another_pos):
  """compute distance"""
  dist = m.sqrt((another\_pos[0] - current\_pos[0])**2 + (another\_pos[1] - current\_pos[1])**2)
  return dist
  #return dist(current_pos, another_pos)
def check_out_bounds( current_position, x_bounds, y_bounds):
     """check out of bounds of configuration space"""
     if current\_position[0] < x\_bounds[0] or current\_position[0] > x\_bounds[1]:
       return True
     if current_position[1] < y_bounds[0] or current_position[1] > y_bounds[1]:
       return True
     return False
def check_node_validity(obstacle_list, node_in_question, x_bounds, y_bounds, turtle_radius):
  """ check if current node is valid """
  if check_out_bounds(node_in_question, x_bounds, y_bounds) == True:
     print("the node in question is out of bounds")
     return False
  elif check_if_obstacle_is_present(obstacle_list, node_in_question) == True:
     print("the node in question is an obstacle")
     return False
  elif check_within_obstacle(obstacle_list, node_in_question, turtle_radius) == True:
     print("the node in question is too close to an obstacle")
     return False
  else:
     print("the node in question is valid")
     return True
if name ==' main ':
  #set up parameters
  x_{span} = [0,10]
  y_span = [0,10]
  spacing = 0.5
  obstacle_list=[(1,1), (4,4), (3,4), (5,0), (5,1), (0,7), (1,7), (2,7), (3,7)]
```

```
turtle_radius = float(input("Enter the turtle's radius please: "))
   val_x, val_y = [int(x) for x in input("Enter the node in question with no commas and I'll tell you if
it's valid: ").split()]
   node_in_question = [val_x, val_y]
   check_node_validity(obstacle_list, node_in_question, x_span, y_span, turtle_radius)

Running Script and inputing parameters when prompted:
In [59]: runfile('C:/Users/alexa/ME_459_HW_2_P1.py', wdir='C:/Users/alexa')

Enter the turtle's radius please: .25

Enter the node in question with no commas and I'll tell you if it's valid: 2 2
the node in question is valid
```

#### Problem 2 (30 pts):

# See pseudocode on Canvas for significant assistance in the implementation of Dijkstra's Algorithm.

Integrate your functions into a script with a function that runs Dijkstra's algorithm given the starting location, goal location, grid information, and obstacle list. The function should end with a plot showing the grid space with obstacles (can just use markers for the obstacles) and the desired path from the start location to the goal location. Your function should return this list of waypoints for the desired path. The waypoint list will be used for the Turtlebots later in the semester.

Use the grid information above (Problem 1) with a starting location of (0,0) and a goal location of (8, 9).

Show the plot of the grid and desired path.

Submit your Python code.

```
1# -*- coding: utf-8 -*-
 3Created on Sat Feb 5 22:59:24 2022
 4
 5@author: alexa
 7 import numpy as np
 8 import math as m
 9 import matplotlib.pyplot as plt # plotting tools
10
11 class Node():
      def __init__(self, x,y, cost, index):
12
          self.x = x
13
14
            self.y = y
15
           self.cost = cost
            self.index = index
16
17
18class Turtle():
       def __init__(self, x, y, step_size):
    self.position = [x,y]
19
20
21
            self.move_list = [[step_size,0], #move right
                                [-step_size,0], #move Left
[0,step_size], #move up
[0,-step_size], #move down
22
23
24
25
                                 [-step_size,-step_size], #move southwest
26
                                 [step_size,-step_size],#move southeast
[step_size,step_size],#move northeast
27
                                 [-step_size,step_size]#move northwest
28
29
30
            self.visited_history = {}
31
32
            self.not_visited = {}
33
            self.obstacle_location = {}
34
35
36
```

```
37 class ConfigSpace():
38
39
     # sets up a configuration space based on the following inputs:
     # x_bounds = [x_min,x_max]
# y_bounds = [y_min,y_max]
40
41
42
    # spacing = grid spacing or step size in between values
43
      def __init__(self, x_bounds, y_bounds, spacing):
44
45
           self.x_bounds = x_bounds
46
           self.y_bounds = y_bounds
47
           self.spacing = spacing
48
49
      def set_obstacles(self, obstacle_list):
50
           self.obstacles = obstacle_list
51
52
      def set_graph_coords(self):
            ""graph coordinates and define the search space"""
53
           self.x_coords = np.arange(self.x_bounds[0], self.x_bounds[1]+self.spacing,
54
55
                                     self.spacing)
56
57
           self.y_coords = np.arange(self.y_bounds[0], self.y_bounds[1]+self.spacing,
58
                                     self.spacing)
59
60
           self.generate_search_space()
61
      def get_x_y_coords(self):
62
63
           return self.x_coords, self.y_coords
64
65
      def generate_search_space(self):
            ""generate our search space"""
66
67
           self.search_space = np.zeros((len(self.x_coords),len(self.y_coords)))
68
69
70
      def place_obstacles(self, obst_list):
            ""places obstacles in grid by inserting a 1"""
71
72
           for obstacle in obst_list:
73
              obs x = obstacle[0]
               obs_y = obstacle[1]
74
75
               self.search_space[obs_x, obs_y]= 1
```

```
77
        def calc_index(self, position):
 78
             """calculate index """
            index = (position[1] - self.y_bounds[0]) / \
    self.spacing * (self.x_bounds[1] - self.x_bounds[0] + self.spacing)/ \
    self.spacing + (position[0] - self.x_bounds[0]) / self.spacing
 79
 80
 81
 82
            return index
 83
 84
        def calc_index(self, position_x, position_y):
 85#
 86#
              """calculate index "
              index = (position_y - self.y_bounds[0]) / \
 87#
                88#
 89#
 90#
 91#
             return index
 93 def check_within_obstacle(obstacle_list, current_position, obstacle_radius):
94 """check if I am within collision of obstacle return True if it is
        false if I'm not"""
 95
        for obstacle in obstacle_list:
 96
 97
            distance = compute_distance(current_position, obstacle)
            if distance<=obstacle_radius:</pre>
 98
 99
                 return True
100
            else:
101
                 return False
103 def check_if_obstacle_is_present(obstacle_list, node_in_question):
        """check to see if an obstacle is in the way"
104
        if node_in_question in obstacle_list:
105
106
            return True
107
108 def check_obstacle_exists(obstacle_list):
        """sanity check to see if obstacle exists"""
109
        for obst in obstacle_list:
110
111
             if configSpace.search_space[obst[0],obst[1]] == 1:
112
                 print("yes", configSpace.search_space[obst[0],obst[1]])
113
114
```

```
115 def compute_distance(current_pos, another_pos):
116
        """compute distance""
        dist = m.sqrt((another_pos[0] - current_pos[0])**2+(another_pos[1]- current_pos[1])**2)
117
118
119
        return dist
120
        #return dist(current_pos, another_pos)
121
122 def check_out_bounds( current_position, x_bounds, y_bounds):
123
            """check out of bounds of configuration space"
124
125
            if current_position[0] < x_bounds[0] or current_position[0] > x_bounds[1]:
126
                return True
127
128
            if current position[1] < y bounds[0] or current position[1] > y bounds[1]:
129
130
131
            return False
132
133 def check_node_validity(obstacle_list, node_in_question, x_bounds, y_bounds, turtle_radius):
        """ check if current node is valid """
134
135
136
        if check_out_bounds(node_in_question, x_bounds, y_bounds) == True:
137
            print("the node in question is out of bounds")
138
            return False
139
140
        elif check if obstacle is present(obstacle list, node in question) == True:
141
            turtle.obstacle_location[new_index] = new_node
142
            print("the node in question is an obstacle")
143
            return False
144
145
        elif check within obstacle(obstacle list, node in question, turtle radius) == True:
146
            print("the node in question is too close to an obstacle")
147
            return False
148
149
        else:
150
            print("the node in question is valid")
151
            return True
152
153
154 if __name__=='__main__':
155
      #set up parameters
      x_{span} = [0,10]
156
157
      y_{span} = [0,10]
158
       spacing = 0.5
159
       starting point = [0,0]
160
      goal\_point = [8,9]
161 #%% ##### BUILD WORL
162
      configSpace = ConfigSpace(x_span, y_span, spacing)
163
       configSpace.set_graph_coords()
164
165
      x_bounds, y_bounds = configSpace.get_x_y_coords()
166
167
       obstacle_list = [[1,1], [4,4], [3,4], [5,0], [5,1], [0,7], [1,7], [2,7], [3,7]]
168
       obstacle_radius = 0.25
       configSpace.set_obstacles(obstacle_list)
169
170 #%% Build dijkstra
171
172
173
       turtle = Turtle(0,0,spacing)
174
       start_position = [0,0]
175
       goal_point = [8,9]
176
177
       current_node = Node(turtle.position[0], turtle.position[1], 0, -1)
178
       current_index = configSpace.calc_index(turtle.position)
179
       turtle.not_visited[current_index] = current_node
180
       new_node_list = []
181
       node_cost_transaction_list= []
```

```
182
       while len(turtle.not_visited) != 0:
183
184#
            current_node_index = min(turtle.not_visited, key=lambda x:turtle.not_visited[x].cost)
185
           current_node_index = min(turtle.not_visited)
186#
            current node index = min(turtle.not visited, key=lambda x:turtle.not visited[x].cost)
           current_node = turtle.not_visited[current_node_index]
187
188#
            print(current_node.x,current_node.y,current_node.cost,current_node.index)
189
           turtle.position = [current_node.x, current_node.y]
190
           turtle.visited_history[current_node_index] = current_node
191
           del turtle.not visited[current node index]
192
193#
            if [current_node.x, current_node.y] == goal_point:
194#
                #Have method to return path
195#
                print("I've arrived!", current_node.x, current_node.y)
196#
                break
197
198#
            print("turtle position is", turtle.position)
199
200
           for move in turtle.move_list:
201
               new_position = [turtle.position[0] + move[0],
                               turtle.position[1] +move[1]]
202
               new_index = configSpace.calc_index(new_position)
203
204
               greedy_cost = compute_distance(new_position, [current_node.x, current_node.y]) + current_node.cost
205
               new_node = Node(new_position[0], new_position[1], greedy_cost, current_node_index)
206
207
               new node list.append([new node.x,new node.y,new node.cost,new node.index])
               if new_index in turtle.visited_history:
208
209
                   continue
210
211
               if check_out_bounds(new_position, x_span, y_span) == True:
212
213
                   continue
214
215
               if check_if_obstacle_is_present(obstacle_list, new_position) == True:
216
                   print('obstacle',new_index)
217
                   continue
218
219
               if check_within_obstacle(obstacle_list, new_position, obstacle_radius) == True:
                   continue
220
221
               if new_index not in turtle.not_visited:
                   turtle.not visited[new index] = new node
222
223
                   continue
               if new_node.cost < turtle.not_visited[new_index].cost:</pre>
224
225
                   node_cost_transaction_list.append([])
226
                   turtle.not_visited[new_index].cost = new_node.cost
227
                   turtle.not_visited[new_index].index = new_node.index
228
229
       path_x = []
230
       path_y = []
231
       goal_node = Node(goal_point[0],goal_point[1],0,0)
232
       path_index = configSpace.calc_index([goal_node.x,goal_node.y])
233
       path_x.append(turtle.visited_history[path_index].x)
234
       path_y.append(turtle.visited_history[path_index].y)
       print (path_index)
235
       print(turtle.visited_history[394].index)
236
```

```
while turtle.visited_history[path_index].index != -1:
238
            path index = turtle.visited history[path index].index
239
            path_x.append(turtle.visited_history[path_index].x)
240
            path_y.append(turtle.visited_history[path_index].y)
241
242
        print(path_x,path_y)
243
244
        plt.axis([x_span[0], x_span[1] + spacing, y_span[0], x_span[1] + spacing]);
245
        plt.plot(start_position[0],start_position[0] , marker="x",color="red")
246
        plt.plot(goal_point[0] ,goal_point[1], marker="x",color="red")
247
        plt.plot(path_x,path_y)
248
249
        #aridLines#
250
        plt.grid()
251
        plt.grid(which='minor')
252
        plt.minorticks_on()
253
254
        for y in y_bounds:
255
            for x in x_bounds:
256
                temp_node = Node(x,y,0,0)
                node_indx = configSpace.calc_index([x,y])
257
258
                if node_indx in turtle.visited_history:
259
                  plt.text(x, y, str(int(node_indx)), color="red", fontsize=8)
        for obst in obstacle_list:
260
261
            plt.plot(obst[0], obst[1], marker="x", color= "blue")
In [72]: runfile('C:/Users/alexa/ME_459_HW_2_P2.py', wdir='C:/Users/alexa')
The x path is: [8.0, 7.5, 7.0, 6.5, 6.0, 5.5, 5.0, 4.5, 4.0, 3.5, 3.5, 3.0, 2.5, 2.0, 1.5, 1.0, 0.5, 0, 0]
The y path is: [9.0, 8.5, 8.0, 7.5, 7.0, 6.5, 6.0, 5.5, 5.0, 4.5, 4.0, 3.5, 3.0, 2.5, 2.0, 1.5, 1.0, 0.5, 0]
        20 421 422 423 424 425 426 427 428 429 430 431 432 433 434 435 436 437 438 439 440
  10
         9 400 401 402 403 404 405 406 407 408 409 410 411 412 413 414 415 416 417 418 419
        378 379 380 381 382 383 384 385 386 387 388 389 390 391 392 393
        57 358 359 360 361 362 363 364 365 366 367 368 369 370 371 372 373 374 375 376 377
        36 337 338 339 340 341 342 343 344 345 346 347 348 349 350 351 352 353 354 355 356
   8
        115 | 316 | 317 | 318 | 319 | 320 | 321 | 322 | 323 | 324 | 325 | 326 | 327
                                    301 302 303 304 305
                                                          306 307 308 309 310 311 312 313 314
                                             240 241 242 243 244 245 246 247 248 249 250 251
           211 212 213 214 215 216 217 218 219 220 221 222 223 224 225 226 227 228 229 230
         89 190 191 192 193 194 195 196 197 198 199 200 201 202 203 204 205 206 207 208
                                             177 178 179 180 181 182 183 184 185 186 187 188
                                   3 | 154 | 155 | 156 | 157 | 158 | 159 | 160 | 161 | 162 | 163 | 164 | 165 | 166 | 167
                            231 132 133 134 135 136 137 138 139 140 141 142 143 144 145 146
                        109 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 12
                                                                  98 99
                                             93 94
                                                         96 97
   2
                                             72 73 74 75
                                    70
                                        71
                                                             76 77 78
                                                                          79 80
                                    49 50 51
                            47
                        46
                                48
                                                     53 54
                                                              55
                                                                  56
                                                                      57
                                                                              59
                                                                                   60
                                27
                                             30
                                                         33
                                                                          37
```

## Problem 3 (10 pts):

Assuming the robot has an actual size, you will need to inflate the graph such that the algorithm does not plan such that it would contact the robot. Use a robot diameter of 1.0. This inflation can occur in several different methods including: 1) inflate the obstacle list and grid boundary, 2) when checking for a collision between a node and the grid boundary or obstacle, see if the distance to the obstacle is less than the robot diameter (rather than checking for distance = 0).

10

Rerun the same grid/configuration used for Problem 2 and show the results.

