ME 459/5559 – Robotics and Unmanned Systems

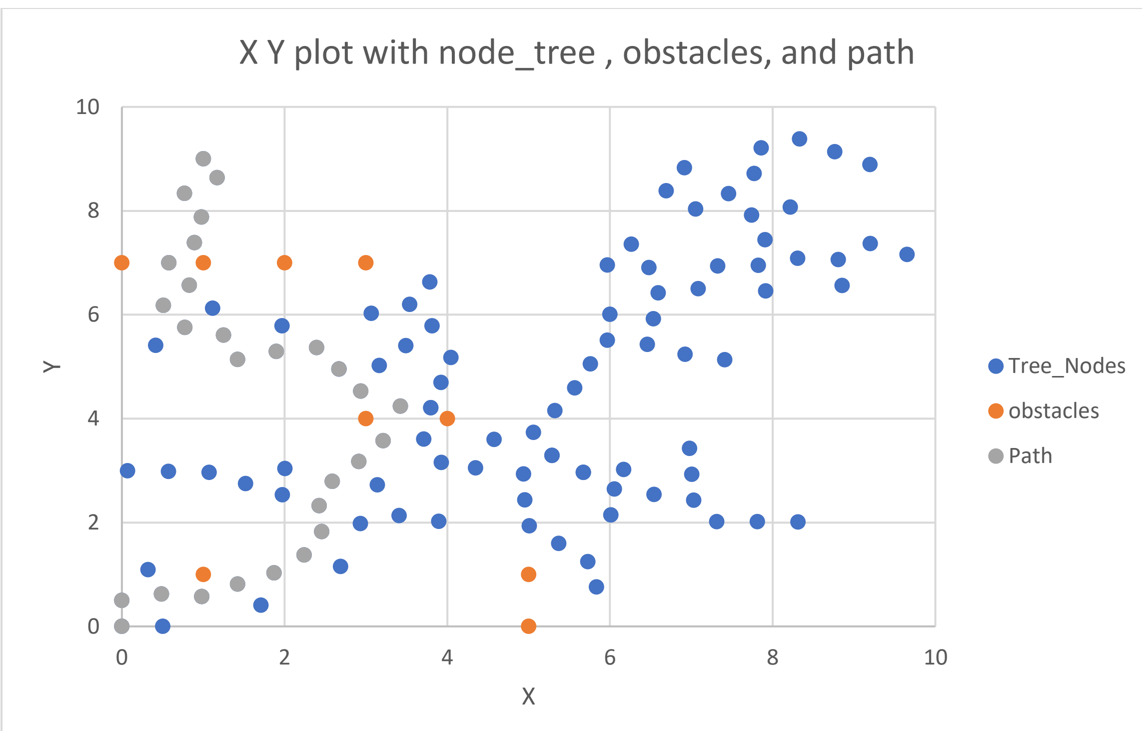
HW #4: DUE March 4th, 2022

Problem 6:

Modify your Dijkstra/A* code (standalone Python script, not ROS) to use the RRT method to get from the start to the goal. Use the same obstacle list and bounding box. Use a distance to jump (from nearest node in the tree) of 0.5.

Create a plot showing the tree (valid nodes) and the corresponding path to get from the start to the goal for the same map as Problem 4.

Submit your Python code.




```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

import rospy
```

```
import numpy as np
import math
import random
from math import atan2
import matplotlib.pyplot as plt # plotting tools
from nav_msgs.msg import Odometry
from tf.transformations import euler_from_quaternion
from geometry_msgs.msg import Point, Twist


class Node():
    def __init__(self, x,y, cost, index):
        self.x = x
        self.y = y
        self.cost = cost
        self.index = index

class Turtle():
    def __init__(self, x, y, step_size):
        self.position  = [x,y]
        self.move_list = [[step_size,0], #move right
                          [-step_size,0], #move left
                          [0,step_size], #move up
                          [0,-step_size],#move down
                          [-step_size,-step_size], #move southwest
                          [step_size,-step_size],#move southeast
                          [step_size,step_size],#move northeast
                          [-step_size,step_size]#move northwest
                          ]

        self.visited_history = {}
        self.not_visited = {}
        self.obstacle_location = {}



class ConfigSpace():

   # sets up a configuration space based on the following inputs:
   # x_bounds = [x_min,x_max]
   # y_bounds = [y_min,y_max]
   # spacing = grid spacing or step size in between values

    def __init__(self, x_bounds, y_bounds, spacing):
        self.x_bounds = x_bounds
```

```
        self.y_bounds = y_bounds
        self.spacing = spacing

    def set_obstacles(self, obstacle_list):
        self.obstacles = obstacle_list

    def set_graph_coords(self):
        """graph coordinates and define the search space"""
        self.x_coords = np.arange(self.x_bounds[0],
self.x_bounds[1]+self.spacing,
                                    self.spacing)

        self.y_coords = np.arange(self.y_bounds[0],
self.y_bounds[1]+self.spacing,
                                    self.spacing)

        self.generate_search_space()

    def get_x_y_coords(self):
        return self.x_coords, self.y_coords

    def generate_search_space(self):
        """generate our search space"""
        self.search_space = np.zeros((len(self.x_coords),len(self.y_coords)))


    def place_obstacles(self, obst_list):
        """places obstacles in grid by inserting a 1"""
        for obstacle in obst_list:
            obs_x = obstacle[0]
            obs_y = obstacle[1]
            self.search_space[obs_x, obs_y]= 1

    def calc_index(self,position):
        """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

        return index

#    def calc_index(self, position_x, position_y):
#        """calculate index """
#        index = (position_y - self.y_bounds[0]) / \
```

```
#           self.spacing * (self.x_bounds[1] - self.x_bounds[0] + self.spacing)/
\
#               self.spacing + (position_x - self.x_bounds[0]) / self.spacing
#
#        return index

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 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"""
    if node_in_question in obstacle_list:
        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 = math.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:
        turtle.obstacle_location[new_index] = new_node
        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
def RRT(x_span, y_span,spacing, start_position, goal_point, obstacle_list,
obstacle_radius,delta_l):

    #%% ##### BUILD WORLD
    configSpace = ConfigSpace(x_span, y_span, spacing)
    configSpace.set_graph_coords()

    x_bounds, y_bounds = configSpace.get_x_y_coords()
    configSpace.set_obstacles(obstacle_list)

     ####  RRT  ###
    current_node = Node(start_position[0],start_position[1], 0, -1) # the start
node
    node_tree = []
    node_tree.append(current_node)
#    print('tree size : ', len(node_tree))
    index_count = 0
    node_tree_count = 0
    while compute_distance([current_node.x, current_node.y] , goal_point) >
delta_l:
```

```
        node_tree_count = 1 + node_tree_count
        rand_x = random.randint(x_span[0],x_span[1])
        print('rand_x : ', rand_x)
        rand_y = random.randint(y_span[0],y_span[1])
        print('rand_y : ', rand_y)
        random_point = [rand_x, rand_y]
        node_dist_list = []
        for node in node_tree:
            node_dist = compute_distance([node.x, node.y] , random_point)
            node_and_distance = [node.x, node.y, node.cost, node.index,
node_dist]
            node_dist_list.append(node_and_distance)
            if len(node_dist_list)==1:
                min_node = node
                min_node1 = node_and_distance
                #print("been Here")
            if min_node1[4] >= node_and_distance[4]:
                temp_min_node = node
                temp_min_node1 = node_and_distance
                rand_ang = math.atan2((rand_y - temp_min_node.y) , (rand_x -
temp_min_node.x))
                print("rand_ang : ", rand_ang)
                print("temp_min_node.x : ", temp_min_node.x)
                print("temp_min_node.y : ", temp_min_node.y)
                temp_new_node_x = (math.cos(rand_ang) * delta_l) +
temp_min_node.x
                print(" temp_new_node_x: ",  temp_new_node_x)
                temp_new_node_y = math.sin(rand_ang) * delta_l + temp_min_node.y
                print(" temp_new_node_y: ",  temp_new_node_y)


                if check_if_obstacle_is_present(obstacle_list,
[temp_new_node_x,temp_new_node_y]) == True:
                    print('obstacle',new_index)
                    continue

                if check_within_obstacle(obstacle_list,
[temp_new_node_x,temp_new_node_y], obstacle_radius) == True:
                    print('node',temp_new_node_x, temp_new_node_y ,'is within
obstacle')
                    continue
                if check_out_bounds([temp_new_node_x,temp_new_node_y], x_span,
y_span) == True:
                    print('node',temp_new_node_x, temp_new_node_y ,'is out of
bounds')
```

```
                    continue

                else:

                    #if temp_new_node is valid then:
                    new_node_x = temp_new_node_x
                    new_node_y = temp_new_node_y
                    min_node = node
                    list_position = node_tree.index(node)
                    print('list position is : ', list_position)
                    min_node1 = temp_min_node1

        new_node_cost = delta_l + min_node.cost
        new_node_parent = list_position
        current_node = Node(new_node_x, new_node_y, new_node_cost,
new_node_parent)
        node_tree.append(current_node)

        print('the node tree has ', len(node_tree), ' nodes in it')
#         if node_tree_count == 5 :  #debug stopper
#             break

    dist_from_cur_2_goal = compute_distance([current_node.x, current_node.y] ,
goal_point)
    cost_2_goal = current_node.cost + dist_from_cur_2_goal
    goal_parent = node_tree.index(current_node)
    print('The goal points parent index is : ', goal_parent)
    goal_node = Node(goal_point[0], goal_point[1], cost_2_goal, goal_parent)
    path_node = Node(goal_point[0], goal_point[1], cost_2_goal, goal_parent)
    print('node tree size:', len(node_tree))
    node_tree.append(goal_node)
    print('final node tree size:', len(node_tree))

    reversed_path_x = []
    reversed_path_y = []

    while path_node.index != -1:
        reversed_path_x.append(path_node.x)
        reversed_path_y.append(path_node.y)
        path_node = node_tree[path_node.index]
    reversed_path_x.append(start_position[0])
    reversed_path_y.append(start_position[1])
    print('last node tree size:', len(node_tree))
    arr = np.array((reversed_path_x, reversed_path_y))
    reverse_path = arr.T
```

```
    path_list = []
    path_x = []
    path_y = []
    print('last node tree size:', len(node_tree))
    print('node_tree_x : ')
    for node in node_tree:
        print(node.x)
    print('node_tree_y : ')
    for node in node_tree:
        print(node.y)
    for path_point in reversed(reverse_path) :
        path_list.append(path_point)
        path_x.append(path_point[0])
        path_y.append(path_point[1])
#         print(path_point[0],path_point[1])
    print('path x : ')
    for x in path_x:
        print(x)
    print('path y : ')
    for y in path_y:
        print(y)
    # return path list
    return  path_x, path_y

if __name__=='__main__':
    #set up parameters
    x_span = [0,10]
    y_span = [0,10]
    spacing = .5
    start_position = [0.0,0.0]
    goal_point = [1.0,9.0]


    obstacle_list = [[1,1], [4,4], [3,4], [5,0], [5,1], [0,7], [1,7], [2,7],
[3,7]] #[[1,1], [4,4], [3,4], [5,0], [5,1], [0,7], [0.5,7], [1,7], [1.5,7],
[2,7], [2.5,7], [3,7]]
    obstacle_radius = 0.25
    delta_l = .5
    path_x, path_y = RRT(x_span, y_span,spacing, start_position, goal_point,
obstacle_list, obstacle_radius,delta_l)



    path_list = []
```

```
    for a, b in zip( path_x, path_y ):

        path_list.append( [ a, b ] )
    print('the path list is : ', path_list)

    #####
    x = 0.0
    y = 0.0
    yaw = 0.0

    def newOdom(msg):
        global x
        global y
        global yaw

        x = msg.pose.pose.position.x
        y = msg.pose.pose.position.y

        rot_q = msg.pose.pose.orientation
        (roll, pitch, yaw) = euler_from_quaternion([rot_q.x, rot_q.y, rot_q.z,
rot_q.w])

    #rospy.init_node("speed_controller")
    rospy.init_node("me_459_hw3_5")

    sub = rospy.Subscriber("/odom", Odometry, newOdom)
    pub = rospy.Publisher("/cmd_vel", Twist, queue_size = 1)

    speed = Twist()

    r = rospy.Rate(20)

    #goal = Point()
    #goal.x = 5
    #goal.y = 5
    end_point = goal_point
    #path_list =[[0.0,0.0],[0.0,1.0],[2.0,2.0],[3.0,-3.0]]
    while not rospy.is_shutdown():
        #while abs(end_point[0] - x) < .15 and abs(end_point[1] - x) < .15:
        #     speed.linear.x(0.0)
        #     pub.publish(speed)
        #     print("you made it!")
        #     r.sleep()
        for point in path_list:
            while abs(point[0]-x) > .01 or abs(point[1]-y) > .01 :
```

```
                inc_x = point[0] - x
                inc_y = point[1] - y

                angle_to_goal = atan2(inc_y, inc_x)

                if (angle_to_goal - yaw) > 0.05:
                    speed.linear.x = 0.0
                    speed.angular.z = 0.2
                elif (angle_to_goal - yaw) < -0.05:
                    speed.linear.x = 0.0
                    speed.angular.z = -0.2

                else:
                    speed.linear.x = 0.15
                    speed.angular.z = 0.0

                pub.publish(speed)
                r.sleep()
        #    break

        #if abs(end_point[0] - inc_x) > .15 and abs(end_point[1] - inc_y) > .15:
        #    speed.linear.x(0.0)
        #    pub.publish(speed)
        #    print("you made it!")
        #    break

    #plotting grid#
    plt.axis([x_span[0], x_span[1] + spacing, y_span[0], x_span[1] + spacing]);
    plt.plot(start_position[0],start_position[0] , marker="x",color="red")
    plt.plot(goal_point[0] ,goal_point[1], marker="x",color="red")
    plt.plot(path_x,path_y)

    #gridlines#
    plt.grid()
    plt.grid(which='minor')
    plt.minorticks_on()

    for y in y_bounds:
        for x in x_bounds:
            temp_node = Node(x,y,0,0)
            node_indx = configSpace.calc_index([x,y])
            if node_indx in turtle.visited_history:
               plt.text(x, y, str(int(node_indx)), color="red", fontsize=8)
    for obst in obstacle_list:
        plt.plot(obst[0], obst[1], marker="x", color= "blue")
```

[redacted]