ME 459 HW4 Code #4 & #5

4

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

import rospy
import numpy as np
import math
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 Dijkstra(x_span, y_span,spacing, start_position, goal_point, obstacle_list,
obstacle_radius):

    obstacle_list = [[1,1], [4,4], [3,4], [5,0], [5,1], [0,7], [1,7], [2,7],
[3,7]]
    obstacle_radius = 0.25
    #%% ##### 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)
```

```
#%% Build dijkstra


    turtle = Turtle(0,0,spacing)

    current_node = Node(turtle.position[0], turtle.position[1], 0, -1)
    current_index = configSpace.calc_index(turtle.position)
    turtle.not_visited[current_index] = current_node
    new_node_list = []
    node_cost_transaction_list= []
    while len(turtle.not_visited) != 0:

        current_node_index = min(turtle.not_visited, key=lambda
x:turtle.not_visited[x].cost)
#         current_node_index = min(turtle.not_visited)
#         current_node_index = min(turtle.not_visited,  key=lambda
x:turtle.not_visited[x].cost)
        current_node = turtle.not_visited[current_node_index]
#         print(current_node.x,current_node.y,current_node.cost,current_node.index
)
        turtle.position = [current_node.x, current_node.y]
        turtle.visited_history[current_node_index] = current_node
        del turtle.not_visited[current_node_index]

#         if [current_node.x, current_node.y]  == goal_point:
#             #Have method to return path
#             print("I've arrived!", current_node.x, current_node.y)
#             break

#         print("turtle position is", turtle.position)

        for move in turtle.move_list:
            new_position = [turtle.position[0] + move[0],
                            turtle.position[1] +move[1]]
            new_index = configSpace.calc_index(new_position)
            greedy_cost = compute_distance(new_position, [current_node.x,
current_node.y]) + current_node.cost
            new_node = Node(new_position[0], new_position[1], greedy_cost,
current_node_index)

            new_node_list.append([new_node.x,new_node.y,new_node.cost,new_node.in
dex])
            if new_index in turtle.visited_history:
                continue
```

```
            if check_out_bounds(new_position, x_span, y_span) == True:

                continue

            if check_if_obstacle_is_present(obstacle_list, new_position) == True:
#                print('obstacle',new_index)
                continue

            if check_within_obstacle(obstacle_list, new_position,
obstacle_radius) == True:
                continue
            if new_index not in turtle.not_visited:
                turtle.not_visited[new_index] = new_node
                continue
            if new_node.cost < turtle.not_visited[new_index].cost:
                node_cost_transaction_list.append([])
                turtle.not_visited[new_index].cost = new_node.cost
                turtle.not_visited[new_index].index = new_node.index
                continue
    path_x = []
    path_y = []

    goal_node = Node(goal_point[0],goal_point[1],0,0)
    path_index = configSpace.calc_index([goal_node.x,goal_node.y])
    path_x.append(turtle.visited_history[path_index].x)
    path_y.append(turtle.visited_history[path_index].y)
#     print (path_index)
#     print(turtle.visited_history[394].index)
    while turtle.visited_history[path_index].index != -1:
        path_index = turtle.visited_history[path_index].index
        path_x.append(turtle.visited_history[path_index].x)
        path_y.append(turtle.visited_history[path_index].y)

    print('The x path is:',path_x)
    print('The y path is:',path_y)

    return path_x, path_y

if __name__=='__main__':
    #set up parameters
    x_span = [0,10]
    y_span = [0,10]
    spacing = 1
    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]]
    obstacle_radius = 0.25
    path_x, path_y = Dijkstra(x_span, y_span,spacing, start_position, goal_point,
obstacle_list, obstacle_radius)
    arr = np.array((path_x, path_y))
    reverse_path = arr.T
    path_list = []
    for path_point in reversed(reverse_path) :
        path_list.append(path_point)
        print('the path is:', path_point)

    #####
    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
```

```python
    #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")
```

#5

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

import rospy
import numpy as np
import math
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]])
```

```python
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 Astar(x_span, y_span,spacing, start_position, goal_point, obstacle_list,
obstacle_radius,bot_radius):
```

```
    #%% ##### 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)

#%% Build Astar


    turtle = Turtle(start_position[0],start_position[1],spacing)

    current_node = Node(turtle.position[0], turtle.position[1], 0, -1)
    current_index = configSpace.calc_index(turtle.position)
    turtle.not_visited[current_index] = current_node
    new_node_list = []
    node_cost_transaction_list= []
    while len(turtle.not_visited) != 0:

        current_node_index = min(turtle.not_visited, key=lambda
x:turtle.not_visited[x].cost)
#         current_node_index = min(turtle.not_visited)
#         current_node_index = min(turtle.not_visited,  key=lambda
x:turtle.not_visited[x].cost)
        current_node = turtle.not_visited[current_node_index]
#         print(current_node.x,current_node.y,current_node.cost,current_node.index
)
        turtle.position = [current_node.x, current_node.y]
        turtle.visited_history[current_node_index] = current_node
        del turtle.not_visited[current_node_index]

        if [current_node.x, current_node.y]  == goal_point:
            #Have method to return path
            print("I've arrived!", current_node.x, current_node.y)
            break

        print("turtle position is", turtle.position)

        for move in turtle.move_list:
            new_position = [turtle.position[0] + move[0],
                            turtle.position[1] +move[1]]

            new_index = configSpace.calc_index(new_position)
            heuristic = compute_distance(new_position, goal_point)
```

```
            greedy_cost = compute_distance(new_position, [current_node.x,
current_node.y]) + current_node.cost + heuristic
            new_node = Node(new_position[0], new_position[1], greedy_cost,
current_node_index)
            new_node_list.append([new_node.x,new_node.y,new_node.cost,new_node.in
dex])

            if new_index in turtle.visited_history:
                continue

            if check_out_bounds(new_position, x_span, y_span) == True:

                continue

            if check_if_obstacle_is_present(obstacle_list, new_position) == True:
#                print('obstacle',new_index)
                continue

            if check_within_obstacle(obstacle_list, new_position,
obstacle_radius) == True:
                continue
            if new_index not in turtle.not_visited:
                turtle.not_visited[new_index] = new_node
                continue
            if new_node.cost < turtle.not_visited[new_index].cost:
                node_cost_transaction_list.append([])
                turtle.not_visited[new_index].cost = new_node.cost
                turtle.not_visited[new_index].index = new_node.index
                continue
    path_x = []
    path_y = []

    goal_node = Node(goal_point[0],goal_point[1],0,0)
    path_index = configSpace.calc_index([goal_node.x,goal_node.y])
    path_x.append(turtle.visited_history[path_index].x)
    path_y.append(turtle.visited_history[path_index].y)
#     print (path_index)
#     print(turtle.visited_history[394].index)
    while turtle.visited_history[path_index].index != -1:
        path_index = turtle.visited_history[path_index].index
        path_x.append(turtle.visited_history[path_index].x)
        path_y.append(turtle.visited_history[path_index].y)

    print('The x path is:',path_x)
    print('The y path is:',path_y)
```

```python
    return path_x, path_y

if __name__=='__main__':
    #set up parameters
    x_span = [0,10]
    y_span = [0,10]
    spacing = 1
    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]]
    obstacle_radius = 0.5
    bot_radius = .5
    path_x, path_y = Astar(x_span, y_span,spacing, start_position, goal_point,
obstacle_list, obstacle_radius,bot_radius)
    arr = np.array((path_x, path_y))
    reverse_path = arr.T
    path_list = []
    for path_point in reversed(reverse_path) :
        path_list.append(path_point)
        print('the path is:', path_point)

    #####
    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")
```

```python
    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
```

```python
#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")
```