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.

```python
# -*- coding: utf-8 -*-
"""
Created on Sat Feb  5 22:59:24 2022

@author: alexa
"""
import numpy as np
import math as m
import matplotlib.pyplot as plt # plotting tools

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 = {}


            
```

```python
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
```

```python

    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 = 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:
        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


if __name__=='__main__':
    #set up parameters
    x_span = [0,10]
    y_span = [0,10]
    spacing = 0.5
    starting_point = [0,0]
    goal_point = [8,9]
#%% ##### BUILD WORLD
    configSpace = ConfigSpace(x_span, y_span, spacing)
    configSpace.set_graph_coords()

    x_bounds, y_bounds = configSpace.get_x_y_coords()

    obstacle_list = [[1,1], [4,4], [3,4], [5,0], [5,1], [0,7], [1,7], [2,7], [3,7]]
    obstacle_radius = 0.25
    configSpace.set_obstacles(obstacle_list)
#%% Build dijkstra


    turtle = Turtle(0,0,spacing)
    start_position = [0,0]
    goal_point = [8,9]

    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.index])
        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)
```

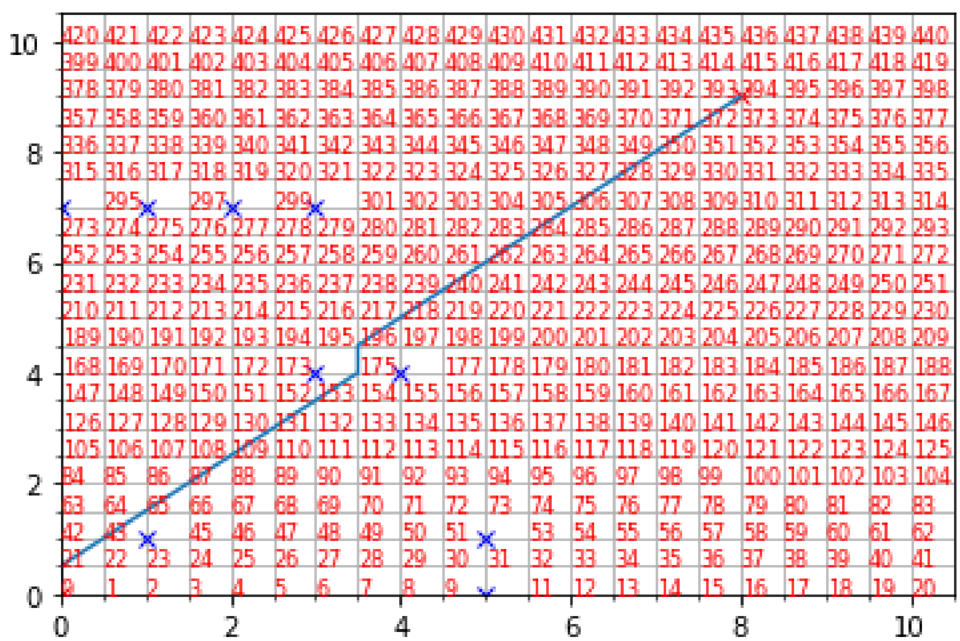
```python
while turtle.visited_history[path_index].index != -1:
    path_index = turtle.visited_history[path_index].index
    path_x.append(turtle.visited_history[path_index].x)
    path_y.append(turtle.visited_history[path_index].y)

print(path_x,path_y)
#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")
```

```
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]
```



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).

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

```
In [76]: runfile('C:/Users/alexa/ME_459_HW_2_P3.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, 0.5, 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]
```