a_star_adam_brendan.py

```
## Brendan Neal and Adam Lobo
## ENPM661 Project 3 A*

##------------------------Importing Libraries-------------------------##
import numpy as np
import cv2 as cv
from matplotlib import pyplot as plt
import math
import timeit
import queue
from queue import PriorityQueue


##------------Defining Node Class (From Previous Project)-----------------##

class Node():
    #Initializing Function
    def __init__(self, state, parent, move, C2C, TotalCost):
        self.state = state
        self.parent = parent
        self.move = move
        self.C2C = C2C
        self.TotalCost = TotalCost

    #---Methods for this Class---#
    def ReturnState(self): #Returns Node State X and Y
        return self.state

    def ReturnParent(self): #Returns the Parent Node
        return self.parent

    def ReturnParentState(self): #Returns the Parent Node's State
        if self.ReturnParent() is None:
            return None
        return self.ReturnParent().ReturnState()

    def ReturnMove(self): #Returns Move
        return self.move

    def ReturnC2C(self): # Returns C2C
        return self.C2C

    def ReturnTotalCost(self): #Returns the Total Cost
        return self.TotalCost

    def __lt__(self, other): #OOP Definition for Less than. Required for Priority
Queue.
        return self.TotalCost < other.TotalCost

    ##--------------BACKTRACKING FUNCTION Integrated into Class--------##
    def ReturnPath(self):
```

```
        CompletedMoves = [] #Initialize Move Array
        NodePath = [] #Initialize the Node Path
        CurrentNode = self
        while(CurrentNode.ReturnMove() is not None): #For move that a Node has made
            CompletedMoves.append(CurrentNode.ReturnMove()) #Append the previous move
            NodePath.append(CurrentNode) #Append Node to Path
            CurrentNode = CurrentNode.ReturnParent() #Backtrack to the Parent before
repeating Process
        NodePath.append(CurrentNode) #Append the starting point after path is derived.
        NodePath.reverse() #Reverse Order to get front to back path
        CompletedMoves.reverse() #Reverse Order to get front to back path

        return CompletedMoves, NodePath

##-----------------------Defining Obstacle Space Setup Functions--------------------##
def setup(robotradius):

    global arena

    #Colors
    white = (255, 255, 255)
    gray = (177, 177, 177)
    darkGray = (104, 104, 104)

    #Draw Radial Clearance
    for x in range(0, 600):

        for y in range(0, 250):

            if checkClearance(x, y, robotradius):
                arena[y, x] = darkGray

    #Draw Obstacle Borders
    for x in range(0, 600):

        for y in range(0, 250):

            if checkBorder(x, y):
                arena[y, x] = gray

    #Draw Obstacles
    for x in range(0, 600):

        for y in range(0, 250):

            if checkObstacle(x, y):
                arena[y, x] = white

##--------------------------Obstacle Setup Function------------------------##
def checkObstacle(x, y):

    #Both Rectangles
    if x >= 100 and x <= 150:

        if y < 100 or y >= 150:
```

```
            return True

    #Pentagon (Left Half)
    if x >= 235 and x <= 300:

        if (y >= (-38/65)*x + (2930/13)) and (y <= (38/65)*x + (320/13)):
            return True

    #Pentagon (Right Half)
    if x >= 300 and x <= 366:

        if (y >= (38/65)*x + (-1630/13)) and (y <= (-38/65)*x + (4880/13)):
            return True

    #Triangle
    if x >= 460 and x <= 510:

        if (y >= 2*x - 895) and (y <= -2*x + 1145):
            return True

    return False

##----------------------------Border Check Function--------------------##
def checkBorder(x, y):

    triHeight = int(round(5/math.cos(math.radians(63.4))))
    hexHeight = int(round(5/math.cos(math.radians(30.3))))

    #Both Rectangles
    if x >= 100 - 5 and x <= 150 + 5:

        if y < 100 + 5 or y >= 150 - 5:
            return True

    #Pentagon (Left Half)
    if x >= 235 - 5 and x <= 300:

        if (y >= (-38/65)*x + (2930/13) - hexHeight) and (y <= (38/65)*x + (320/13) +
hexHeight):
            return True

    #Pentagon (Right Half)
    if x >= 300 and x <= 366 + 5:

        if (y >= (38/65)*x + (-1630/13) - hexHeight) and (y <= (-38/65)*x + (4880/13)
+ hexHeight):
            return True

    #Triangle
    if x >= 460 - 5 and x <= 510 + 5:

        if (y >= 2*x - 895 - triHeight) and (y <= -2*x + 1145 + triHeight) and (y >=
25 - 5) and (y <= 225 + 5):
            return True

    return False
```

```
##------------------------------Defining Radial Clearance Function--------------##
def checkClearance(x, y, r):

    rr = r+1

    if rr == 0:
        return False

    triHeight = int(round((5 + rr)/math.cos(math.radians(63.4))))
    hexHeight = int(round((5 + rr)/math.cos(math.radians(30.3))))

    #Both Rectangles
    if x >= 100 - 5 - rr and x <= 150 + 5 + rr:

        if y < 100 + 5 + rr or y >= 150 - 5 - rr:
            return True

    #Pentagon (Left Half)
    if x >= 235 - 5 - rr and x <= 300:

        if (y >= (-38/65)*x + (2930/13) - hexHeight) and (y <= (38/65)*x + (320/13) +
hexHeight):
            return True

    #Pentagon (Right Half)
    if x >= 300 and x <= 366 + 5 + rr:

        if (y >= (38/65)*x + (-1630/13) - hexHeight) and (y <= (-38/65)*x + (4880/13)
+ hexHeight):
            return True

    #Triangle
    if x >= 460 - 5 - rr and x <= 510 + 5 + rr:

        if (y >= 2*x - 895 - triHeight) and (y <= -2*x + 1145 + triHeight) and (y >=
25 - 5 - rr) and (y <= 225 + 5 + rr):
            return True

    return False

##----------------------------------------Defining Check Valid Move
Function----------------------##
#Checks to see if a point is valid (by checking obstacle, border, and clearance, as
well as making sure the point is within arena bounds)
def checkValid(x, y, r):

    if checkObstacle(x, y):
        return False

    if checkBorder(x, y):
        return False

    if checkClearance(x, y, r):
        return False
```

```
    if (x < 0 or x >= 600 or y < 0 or y >= 250):
        return False

    return True

##--------------------------------Defining my Action
Set------------------------------------##
def MoveMaxTurnLeft(Current_State, Step_Size, RobotRadius):

    RobotTheta = Current_State[2]
    MoveTheta = RobotTheta + 60 #Adjust the Angle Offset

    if MoveTheta >=360: #Wrapping to 0-360
        MoveTheta = MoveTheta -360
    if MoveTheta <0:
        MoveTheta = MoveTheta + 360



    ChangeX = Step_Size * np.cos(np.radians(MoveTheta)) #Change in X
    ChangeY = Step_Size * np.sin(np.radians(MoveTheta)) #Change in Y

    NewNodeState = [Current_State[0] + ChangeX, Current_State[1] + ChangeY, MoveTheta]
#Generate the New State
    if checkValid(NewNodeState[0], NewNodeState[1], RobotRadius) == False: #Check if
move takes us into obstacle space, or outside the workspace.
        return None

    return NewNodeState

def MoveTurnLeft(Current_State, Step_Size, RobotRadius):
    RobotTheta = Current_State[2]
    MoveTheta = RobotTheta + 30 #Angle Offset from current angle

    if MoveTheta >=360: #Wrap to 0-360
        MoveTheta = MoveTheta -360
    if MoveTheta <0:
        MoveTheta = MoveTheta + 360

    ChangeX = Step_Size * np.cos(np.radians(MoveTheta)) #Change in X
    ChangeY = Step_Size * np.sin(np.radians(MoveTheta)) #Change in Y

    NewNodeState = [Current_State[0] + ChangeX, Current_State[1] + ChangeY, MoveTheta]
#Generate New State
    if checkValid(NewNodeState[0], NewNodeState[1], RobotRadius) == False: #Check if
move takes us into obstacle space, or outside the workspace.
        return None

    return NewNodeState

def MoveStraight(Current_State, Step_Size, RobotRadius):
    RobotTheta = Current_State[2]
    MoveTheta = RobotTheta #Angle Offset

    if MoveTheta >=360: #Wrap to 0-360
        MoveTheta = MoveTheta -360
```

```
    if MoveTheta <0:
        MoveTheta = MoveTheta + 360

    ChangeX = Step_Size * np.cos(np.radians(MoveTheta)) #Change in X
    ChangeY = Step_Size * np.sin(np.radians(MoveTheta)) #Change in Y

    NewNodeState = [Current_State[0] + ChangeX, Current_State[1] + ChangeY, MoveTheta]
#Generate New Node State
    if checkValid(NewNodeState[0], NewNodeState[1], RobotRadius) == False: #Check if
move takes us into obstacle space, or outside the workspace.
        return None

    return NewNodeState

def MoveMaxTurnRight(Current_State, Step_Size, RobotRadius):
    RobotTheta = Current_State[2]
    MoveTheta = RobotTheta - 60 #Angle Offset

    if MoveTheta >=360: #Wrap to 360
        MoveTheta = MoveTheta -360
    if MoveTheta <0:
        MoveTheta = MoveTheta + 360

    ChangeX = Step_Size * np.cos(np.radians(MoveTheta)) #Change in X
    ChangeY = Step_Size * np.sin(np.radians(MoveTheta)) #Change in Y

    NewNodeState = [Current_State[0] + ChangeX, Current_State[1] + ChangeY, MoveTheta]
#Generate New Node State
    if checkValid(NewNodeState[0], NewNodeState[1], RobotRadius) == False: #Check if
move takes us into obstacle space, or outside the workspace.
        return None

    return NewNodeState

def MoveTurnRight(Current_State, Step_Size, RobotRadius):
    RobotTheta = Current_State[2]
    MoveTheta = RobotTheta - 30 #Angle Offset

    if MoveTheta >=360: #Wrap to 360
        MoveTheta = MoveTheta -360
    if MoveTheta <0:
        MoveTheta = MoveTheta + 360

    ChangeX = Step_Size * np.cos(np.radians(MoveTheta)) #Change in X
    ChangeY = Step_Size * np.sin(np.radians(MoveTheta)) #Change in Y

    NewNodeState = [Current_State[0] + ChangeX, Current_State[1] + ChangeY, MoveTheta]
#Generate New Node State
    if checkValid(NewNodeState[0], NewNodeState[1], RobotRadius) == False: #Check if
move takes us into obstacle space, or outside the workspace.
        return None

    return NewNodeState

##------------------Concacts All Possible Actions into Single
List---------------------##
def GeneratePossibleMoves(Current_Node, StepSize, Robot_Radius):
```

```
    Current_Node_State = Current_Node.ReturnState()
    New_Node_Locations = []
    #Append all new states to an array
    New_Node_Locations.append(MoveMaxTurnLeft(Current_Node_State, StepSize,
Robot_Radius))
    New_Node_Locations.append(MoveTurnLeft(Current_Node_State, StepSize,
Robot_Radius))
    New_Node_Locations.append(MoveStraight(Current_Node_State, StepSize,
Robot_Radius))
    New_Node_Locations.append(MoveMaxTurnRight(Current_Node_State, StepSize,
Robot_Radius))
    New_Node_Locations.append(MoveTurnLeft(Current_Node_State, StepSize,
Robot_Radius))

    #If the action set is an invalid move, it returns none. This removes the "nones"
from the possible new states
    Possible_New_States = [Location for Location in New_Node_Locations if Location is
not None]

    return Possible_New_States



##-------------------------Defining my Cost to Go
Calculation---------------------------##
def Calculate_C2G(CurrentNodeState, GoalNodeState):
    C2G = 0.0
    X_Current = CurrentNodeState[0]
    Y_Current = CurrentNodeState[1]
    X_Goal = GoalNodeState[0]
    Y_Goal = GoalNodeState[1]
    if CurrentNodeState is not None:
        C2G = np.sqrt((X_Goal-X_Current)**2 + (Y_Goal- Y_Current)**2) #Euclidian
Distance Heuristic function
    return C2G

##----------------------Defining my Compare to Goal
Function--------------------------##

def CompareToGoal(Current_Node_Position, Goal_Node_Position, ErrorThreshold):
    Dist2Goal = (Goal_Node_Position[0] - Current_Node_Position[0])**2 +
(Goal_Node_Position[1] - Current_Node_Position[1])**2 #Euclidian Distance
    if Dist2Goal < ErrorThreshold**2 and Current_Node_Position[2] ==
Goal_Node_Position[2]: #Error less than threshold PLUS the angle has to be equal
        return True
    else:
        return False

##-----------------------Defining my Round to Half
Function------------------------##
''' This function is Required for "Check Visited" Capabilities'''
def Round2Half(number):
    testvalue = np.round(2*number)/2
    if (testvalue == 10):
        testvalue = testvalue - 0.5
    return testvalue

##-------------------------Defining my Check Visited
Function----------------------##
```

```
def CheckIfVisited(Current_Node_State, Node_Array, XYThreshold, ThetaThreshold):
    X = Current_Node_State[0]
    Y = Current_Node_State[1]
    Theta = Current_Node_State[2]
    X = int(Round2Half(X)/XYThreshold)
    Y = int(Round2Half(Y)/XYThreshold)
    Theta = int(Round2Half(Theta)/ThetaThreshold)
    if Node_Array[Y,X,Theta] == 1:
        result = True
    else:
        result = False
    return result


##------------------------Defining my GetInitialState
Function---------------------##
def GetInitialState():
    print("Enter Initial Node X, Y, and Theta separated by spaces: ")
    Init_State=[int(x) for x in input().split()]
    return Init_State

##------------------------Defining my GetGoalState
Function------------------------##
def GetGoalState():
    print("Enter Goal Node X and Y, and Theta separated by spaces: ")
    Goal_State=[int(x) for x in input().split()]
    return  Goal_State
##-------------------------Defining my Get Robot Radius
Function-------------------##
def GetRobotRadius():
    print("Enter Robot Radius.")
    Robot_Radius=int(input())
    return  Robot_Radius

##-------------------------Defining my Get Step Size Function-----------------------##
def GetStepSize():
    print("Enter Robot Step Size (L = 1 to L = 10)")
    StepSize=int(input())
    return  StepSize

##--------------------------Defining my Plotting Function-------------------------##
'''For Floats'''
def Plotter(CurrentNodeState, ParentNodeState, Color):
    plt.plot([ParentNodeState[0], CurrentNodeState[0]],[ParentNodeState[1],
CurrentNodeState[1]], Color, linewidth = 0.75)




'''For Integers'''
def WSColoring(Workspace, Location, Color):
    x,_,_ = Workspace.shape #Get Shape of Workspace
    translation_y = Location[0] #Where in Y
    translation_x = x - Location[1] - 1 #Where in X - (Shifts origin from top left to
bottom right when plotting!)
    Workspace[translation_x,translation_y,:] = Color #Change the Color to a set Color
    return Workspace
```

```
##-----------------------------Main
Function-----------------------------------------##
arena = np.zeros((250, 600, 3), dtype = "uint8")
InitState = GetInitialState()
GoalState =GetGoalState()
RobotRadius = GetRobotRadius()
StepSize = GetStepSize()

if not checkValid(InitState[0], InitState[1], RobotRadius):
    print("Your initial state is inside an obstacle or outside the workspace. Please
retry.")
    exit()
if not checkValid(GoalState[0], GoalState[1], RobotRadius):
    print("Your goal state is inside an obstacle or outside the workspace. Please
retry.")
    exit()

setup(RobotRadius)
WSColoring(arena, InitState, (0,255,0))
WSColoring(arena, GoalState, (0,255,0))

plt.imshow(arena)
plt.show()

#Initialize Arena and Thresholds
SizeArenaX = 600
SizeArenaY = 250
ThreshXY = 0.5
ThreshTheta = 30
ThreshGoalState = 1.5

# Initialize Node Array
node_array = np.array([[[ 0 for k in range(int(360/ThreshTheta))]
                        for j in range(int(SizeArenaX/ThreshXY))]
                        for i in range(int(SizeArenaY/ThreshXY))])

Open_List = PriorityQueue() #Initialize list using priority queue.
traversed_nodes = [] #Traversed nodes is for visualization later.
starting_node_Temp = Node(InitState, None, None, 0, 0) #Generate starting node based
on the initial state given above. The Temp node is needed for plotting.
starting_node = Node(InitState, starting_node_Temp, None, 0, Calculate_C2G(InitState,
GoalState)) #Generate starting node based on the initial state given above.
Open_List.put((starting_node.ReturnTotalCost(), starting_node)) #Add to Open List


GoalReach = False #Initialze Goal Check Variable

Closed_List= np.array([])#Initialize Closed List of nodes. Closed list is based on
node states

##-----------------------CONDUCT A*---------------------##

starttime = timeit.default_timer() #Start the Timer when serch starts
print("A* Search Starting!!!!")
```

```
while not (Open_List.empty()):
    current_node = Open_List.get()[1] #Grab first (lowest cost) item from Priority
Queue.
    traversed_nodes.append(current_node) #Append the explored node (for visualization
later)
    Plotter(current_node.ReturnState(), current_node.ReturnParentState(), 'g') #Plot
the move
    print(current_node.ReturnState(), current_node.ReturnTotalCost()) #Print to show
search is working.
    np.append(Closed_List, current_node.ReturnState()) #Append to Closed List
    goalreachcheck = CompareToGoal(current_node.ReturnState(), GoalState,
ThreshGoalState) #Check if we have reached goal.

    if goalreachcheck: #If we have reached goal node.
        print("Goal Reached!")
        print("Total Cost:", current_node.ReturnTotalCost()) #Print Total Cost
        MovesPath, Path = current_node.ReturnPath() #BackTrack to find path.
        for nodes in Path: #For Each node in ideal path
            Plotter(nodes.ReturnState(), nodes.ReturnParentState(), 'm') #Plot in
Magenta



    else: #If you have NOT reached the goal node
        NewNodeStates = GeneratePossibleMoves(current_node, StepSize,
RobotRadius)#Generate New Nodes from the possible moves current node can take.
        ParentC2C = current_node.ReturnC2C() #Get Parent C2C
        if NewNodeStates not in Closed_List: #Check to see if the new node position is
currently in the closed list
            for State in NewNodeStates: #For each new node generated by the possible
moves.
                ChildNode_C2C = ParentC2C + StepSize #Get C2C for the child node
                ChildNode_Total_Cost = ChildNode_C2C + Calculate_C2G(State, GoalState)
#Get Total Cost for Child Node
                NewChild = Node(State, current_node, "Move" ,ChildNode_C2C,
ChildNode_Total_Cost) #Generate New Child Node Class
                if CheckIfVisited(NewChild.ReturnState(), node_array, ThreshXY,
ThreshTheta) ==  False: #If the node has not been visited before
                    #Mark in Node Array
                    node_array[int(Round2Half(NewChild.ReturnState()[1])/ThreshXY),
int(Round2Half(NewChild.ReturnState()[0])/ThreshXY),
int(Round2Half(NewChild.ReturnState()[2])/ThreshTheta)] = 1
                    Open_List.put((NewChild.ReturnTotalCost() , NewChild)) #Put it
into the Open list
                if CheckIfVisited(NewChild.ReturnState(), node_array, ThreshXY,
ThreshTheta) ==  True: #If you have visited before:
                        if NewChild.ReturnTotalCost() > current_node.ReturnC2C() +
StepSize: #If the current total cost is greater than the move
                            NewChild.parent = current_node #Update Parent
                            NewChild.C2C = current_node.ReturnC2C() + StepSize #Update
C2C

                            NewChild.TotalCost = NewChild.ReturnC2C() +
Calculate_C2G(NewChild.ReturnState(), GoalState) #Update Total Cost

    if goalreachcheck: #If you reach goal
        break #Break the Loop

stoptime = timeit.default_timer() #Stop the Timer, as Searching is complete.
```

```python
    print("That took", stoptime - starttime, "seconds to complete")


    #Show the Completed Searched Arena
    plt.imshow(arena, origin='lower')
    plt.show()


    ##--------------------Visualization--------------------##
    print("Visualization Starting!")
    plt.plot(InitState[0], InitState[1], 'go', markersize = 0.5) #plot init state
    plt.imshow(arena, origin = 'lower')

    for node in traversed_nodes: #Plots the search area
        curr_node_state = node.ReturnState()
        parent_node_state = node.ReturnParentState()
        Plotter(curr_node_state, parent_node_state, 'g')
        plt.pause(0.000001)


    for node in Path: #Plots the ideal path
        curr_node_state = node.ReturnState()
        parent_node_state = node.ReturnParentState()
        Plotter(curr_node_state, parent_node_state, 'm')
        plt.pause(0.0001)

    plt.show()
    plt.close()
```