```python
# ========================================
# ENPM661 Spring 2023: Robotic Path Planning
# Project #3 Phase 1
# Maze Search with Obstacles with A* Algorithm
#
# Author: Doug Summerlin (dsumm1001@gmail.com, dsummerl@umd.edu)
# UID: 114760753
# Directory ID: dsummerl

# Author: Vignesh Rajagopal(vickyrv570@gmail.com, vigneshr@umd.edu)
# UID: 119476192
# Directory ID: vigneshr
# ========================================
# Run as 'python3 a_star_douglas_vignesh.py'
# Github link: https://github.com/dsumm1001/nonholonomic-Astar-maze-search.git
# Results link: https://docs.google.com/document/d/1odwbrP457jTVn1dUarhkSgPyvDlK8KBRE_ITPUqQi6A/edit?usp=sharing
# Press CTRL+C for exit

import numpy as np
import matplotlib.pyplot as plt
import cv2
import math
from queue import PriorityQueue
import time
import sys

def getValidRobotRadius():
    while True:
        try:
            robotRadius = int(input("Enter the radius of the robot as an integer from 1-12: "))
        except ValueError:
            print("Sorry, results invalid. Please try again, entering the input as an integer between 1-12. ")
            continue
        if robotRadius <= 0 or robotRadius >= 13:
            print("Sorry, results invalid. Please try again, entering the input as an integer between 1-12. ")
            continue
        else:
            break
    return robotRadius

def getValidCoords(type, maze, robotRadius, x2Maze):
    while True:
        try:
            coordInput = input("Enter " + type + " node coordinates in x, y format, separated by a comma: ")
            coords = tuple(int(item) for item in coordInput.split(','))
        except ValueError:
            print("Sorry, results invalid. Please try again, entering two integer inputs within the maze space. ")
            continue
        if coords[0] < 0 + robotRadius or coords[0] > 600 - robotRadius or coords[1] < 0 + robotRadius or coords[1] > 250 - robotRadius:
            print("Sorry, results invalid. Please try again, entering two integer inputs within the maze space. ")
            continue
        if all(maze[(int(coords[1]), int(coords[0]))] == [255,255,255]) == False:
            print("Sorry, results invalid. Please try again, making sure to not place the start or goal in an obstacle space.")
            continue
        else:
            break

    while True:
        try:
            theta = int(input("Enter " + type + " node orientation as an integer between 0-360, using increments of 30 deg: "))
            if theta % 30 !=0:
                raise ValueError
        except ValueError:
            print(("Sorry, entry invalid. Please try again, entering an integer input between 0-360 in increments of 30. "))
            continue
        if not searchNode((coords,theta), x2Maze):
            if type == "start":
                print(("Sorry, entry invalid. Please try again, entering an integer input between 0-360 in increments of 30, oriented toward the center of the mazespace. "))
                continue
            else:
                break
        else:
            break

    nodeState = (coords, theta)
    return nodeState

def getValidStepSize():
    while True:
        try:
            stepSize = int(input("Enter the step size of the robot as an integer from 1-10: "))
        except ValueError:
            print("Sorry, results invalid. Please try again, entering the input as an integer between 1-10. ")
            continue
        if stepSize <= 0 or stepSize >= 11:
            print("Sorry, results invalid. Please try again, entering the input as an integer between 1-10. ")
            continue
        else:
            break
    return stepSize

#calculate the distance between the current node to goal node
def euclideanCostToGo(curr, goal):
    eucCost = math.sqrt(math.pow(goal[0] - curr[0], 2) + math.pow(goal[1] - curr[1], 2))
    return eucCost #float

def drawMaze(robotRadius):
    mazeSize = (250,600)

    # Create blank maze
    maze = np.zeros((mazeSize[0], mazeSize[1], 3), dtype = np.uint8)
    maze[:] = (0, 255, 0)
    cv2.rectangle(maze, pt1=(robotRadius,robotRadius), pt2=(mazeSize[1]-robotRadius,mazeSize[0]-robotRadius), color=(255,255,255), thickness= -1)

    # draw rectangle obstacles
    cv2.rectangle(maze, pt1=(100-robotRadius,0), pt2=(150 + robotRadius, 100 + robotRadius), color=(0,255,0), thickness= -1)
    cv2.rectangle(maze, pt1=(100-robotRadius, 150-robotRadius), pt2=(150+robotRadius, mazeSize[1]), color=(0,255,0), thickness= -1)

    cv2.rectangle(maze, pt1=(100,0), pt2=(150,100), color=(0,0,255), thickness= -1)
    cv2.rectangle(maze, pt1=(100,150), pt2=(150,mazeSize[1]), color=(0,0,255), thickness= -1)

    # draw hexagonal boundary
    hexRad = math.radians(30)
    hexBoundPts = np.array([[300, 49 - robotRadius],
                            [365 + robotRadius, math.floor(125-37.5) - math.floor(robotRadius*math.sin(hexRad))],
                            [365 + robotRadius, math.ceil(125+37.5) + math.ceil(robotRadius*math.sin(hexRad))],
                            [300, 201 + robotRadius],
                            [235 - robotRadius, math.ceil(125+37.5) + math.ceil(robotRadius*math.sin(hexRad))],
                            [235 - robotRadius, math.floor(125-37.5) - math.floor(robotRadius*math.sin(hexRad))]])
    cv2.fillConvexPoly(maze, hexBoundPts, color=(0, 255, 0))
    # draw hexagonal obstacle
    hexPts = np.array([[300, 50], [365, math.ceil(125-37.5)],
                        [365, math.floor(125+37.5)], [300, 125+75],
                        [235, math.floor(125+37.5)], [235, math.ceil(125-37.5)]])
    cv2.fillConvexPoly(maze, hexPts, color=(0, 0, 255))

    # draw triangular boundary
    cv2.circle(maze, (460, 25), robotRadius, color=(0, 255, 0), thickness=-1)
    cv2.circle(maze, (460, 225), robotRadius, color=(0, 255, 0), thickness=-1)
    cv2.circle(maze, (510, 125), robotRadius, color=(0, 255, 0), thickness=-1)

    cv2.rectangle(maze, pt1=(460 - robotRadius,25), pt2=(460,225), color=(0,255,0), thickness= -1)

    triRad = math.radians(26.565)
    triUpperBoundPts = np.array([[460, 25],
                             [460 + int(robotRadius*math.cos(triRad)), 25 - int(robotRadius*math.sin(triRad))],
                             [510 + int(robotRadius*math.cos(triRad)), 125 - int(robotRadius*math.sin(triRad))],
                             [510, 125]])
    cv2.fillConvexPoly(maze, triUpperBoundPts, color=(0, 255, 0))

    triLowerBoundPts = np.array([[510, 125],
                             [510 + int(robotRadius*math.cos(triRad)), 125 + int(robotRadius*math.sin(triRad))],
                             [460 + int(robotRadius*math.cos(triRad)), 225 + int(robotRadius*math.sin(triRad))],
                             [460, 225]])
    cv2.fillConvexPoly(maze, triLowerBoundPts, color=(0, 255, 0))

    # draw triangular obstacle
    triPts = np.array([[460, 25], [460, 225], [510, 125]])
    cv2.fillConvexPoly(maze, triPts, color=(0, 0, 255))
    return maze

def checkObstacle(xyCoords, maze):
    try:
        if all(maze[(int(2*xyCoords[1]), int(2*xyCoords[0]))] == [255,255,255]):
            return False
        else:
            return True
    except IndexError:
        return True

def roundCoord(val):
    if (val - math.floor(val)) < 0.25:
        val = math.floor(val)
    elif 0.25 <= (val - math.floor(val)) < 0.75:
        val = math.floor(val) + 0.5
    elif (val - math.floor(val)) >= 0.75:
        val = math.ceil(val)
    return val

def checkAngle(ang):
    temp = ang % 360
    ang = temp
    return ang

# [cost, index, coords, c2c]
def actZero(node, maze):
    xZero = roundCoord(node[0][0] + stepSize*math.cos(math.radians(node[1])))
    yZero = roundCoord(node[0][1] + stepSize*math.sin(math.radians(node[1])))
    angZero = checkAngle(node[1])

    if (checkObstacle((xZero,yZero), maze) == False):
        newThetaZero = [None, None, ((xZero, yZero), angZero), None]
        return newThetaZero
    else:
        return None

def actPlus30(node, maze):
    xP30 = roundCoord(node[0][0] + stepSize*math.cos(math.radians(node[1] + 30)))
    yP30 = roundCoord(node[0][1] + stepSize*math.sin(math.radians(node[1] + 30)))
    angP30 = checkAngle(node[1] + 30)

    if (checkObstacle((xP30,yP30), maze) == False):
        newThetaPlus30 = [None, None, ((xP30, yP30), angP30), None]
        return newThetaPlus30
    else:
        return None

def actMinus30(node, maze):
    xM30 = roundCoord(node[0][0] + stepSize*math.cos(math.radians(node[1] - 30)))
    yM30 = roundCoord(node[0][1] + stepSize*math.sin(math.radians(node[1] - 30)))
    angM30 = checkAngle(node[1] - 30)

    if (checkObstacle((xM30,yM30), maze) == False):
        newThetaMinus30 = [None, None, ((xM30, yM30), angM30), None]
        return newThetaMinus30
    else:
        return None

def actPlus60(node, maze):
    xP60 = roundCoord(node[0][0] + stepSize*math.cos(math.radians(node[1] + 60)))
    yP60 = roundCoord(node[0][1] + stepSize*math.sin(math.radians(node[1] + 60)))
    angP60 = checkAngle(node[1] + 60)

    if (checkObstacle((xP60,yP60), maze) == False):
        newThetaPlus60 = [None, None, ((xP60, yP60), angP60), None]
        return newThetaPlus60
    else:
        return None

def actMinus60(node, maze):
    xM60 = roundCoord(node[0][0] + stepSize*math.cos(math.radians(node[1] - 60)))
    yM60 = roundCoord(node[0][1] + stepSize*math.sin(math.radians(node[1] - 60)))
    angM60 = checkAngle(node[1] - 60)

    if (checkObstacle((xM60,yM60), maze) == False):
        newThetaMinus60 = [None, None, ((xM60, yM60), angM60), None]
        return newThetaMinus60
    else:
        return None

def searchNode(nodeCoords, maze):
    zero = actZero(nodeCoords, maze)
    plus30 = actPlus30(nodeCoords, maze)
    minus30 = actMinus30(nodeCoords, maze)
    plus60 = actPlus60(nodeCoords, maze)
    minus60 = actMinus60(nodeCoords, maze)

    results = []
    if zero is not None:
        results.append(zero)
    if plus30 is not None:
        results.append(plus30)
    if minus30 is not None:
        results.append(minus30)
    if plus60 is not None:
        results.append(plus60)
    if minus60 is not None:
        results.append(minus60)

    return results

def generatePath(nodeIndex, nodeCoords, maze):
    pathIndices = []
    pathCoords = []

    while nodeIndex is not None:
        pathIndices.append(nodeIndex)
        pathCoords.append(nodeCoords)
        tempX = int(2*nodeCoords[0][0])
        tempY = int(2*nodeCoords[0][1])
        cv2.circle(maze, (tempX, tempY), 5, color=(0,255,255), thickness=-1)
        nodeCoords = coordDict[nodeIndex]
        nodeIndex = parentDict[nodeIndex]

    return pathIndices, pathCoords

def simulateBot(pathCoords, emptyMaze, robotRadius):
    for i in pathCoords:
        tempX = int(2*i[0][0])
        tempY = int(2*i[0][1])
        cv2.circle(emptyMaze, (tempX, tempY), 3, color=(0,255,255), thickness=-1)
        outVid.write(cv2.flip(emptyMaze,0))

    pathCoords.reverse()

    for i in pathCoords:
        emptyMazeCopy = emptyMaze.copy()
        tempXR = int(2*i[0][0])
        tempYR = int(2*i[0][1])
        currCirc = cv2.circle(emptyMazeCopy, (tempXR,tempYR), 2*robotRadius, color=(255,0,255), thickness=-1)
        outVid.write(cv2.flip(currCirc,0))

    index = 30
    while index >=0:
        index -= 1
        outVid.write(cv2.flip(currCirc,0))

print("\nWelcome to the A* Maze Finder Program! \n")

fourcc = cv2.VideoWriter_fourcc(*'mp4v')
outVid = cv2.VideoWriter('output.mp4', fourcc, 30, (1200,500))

robotRadius = getValidRobotRadius()
stepSize = getValidStepSize()

maze = drawMaze(robotRadius)
doubleMaze = cv2.resize(maze, (maze.shape[1]*2, maze.shape[0]*2), interpolation = cv2.INTER_LINEAR)
blankMaze = doubleMaze.copy()

counter = 30
while counter >=0:
    counter -= 1
    outVid.write(cv2.flip(blankMaze,0))

# get start and goal nodes
start = getValidCoords("start", maze, robotRadius, doubleMaze)
goal = getValidCoords("goal", maze, robotRadius, doubleMaze)
print()
print("Pathfinding... \n")

startTime = time.time()
solved = False

openList = PriorityQueue()
openSet = set()

# intialize data containers for backtracking
parentDict = {1:None}
coordDict = {1:start}
costDict = {1:0}
c2cDict = {1:0}
closedSet = set()
closedList = []

# initialize pathfinding matrix
threshXY = 0.5
threshTheta = 30
graph = np.zeros((int(600/threshXY), int(250/threshXY), int(360/threshTheta)))

# [cost, index, coords/theta, c2c]
startNode = [0, 1, start, 0]
index = startNode[1]

openList.put(startNode)
openSet.add(start)

while not openList.empty() and solved == False:
    first = openList.get()
    openSet.remove(first[2])
    closedSet.add(first[2])
    closedList.append(first[2])
    graph[int(2*first[2][0][0])][int(2*first[2][0][1])][int(first[2][1]/30)] = 1
    #print("Current Node: ", first)

    if euclideanCostToGo(first[2][0], goal[0]) <= 1.5:
        elapsedTime = time.time() - startTime
        print ("Yay! Goal node located... Operation took ", elapsedTime, " seconds.")
        print("Current node index: ", first[1], " and cost: ", round(first[3],2), "\n")
        solved = True

        dispMaze = doubleMaze.copy()

        pathIndices, pathCoords = generatePath(first[1], first[2], dispMaze)
        print("Displaying generated path... close window to continue \n")

        # # display the path image using opencv
        dispMaze = cv2.flip(dispMaze, 0)
        cv2.imshow('Generated Path', dispMaze)
```

```python
        cv2.waitKey(0)

        print("Generating simulation...")
        simulateBot(pathCoords, doubleMaze, robotRadius)
        print("Simulation complete! \n")
        break

    results = searchNode(first[2], doubleMaze)

    for i in results:
        if graph[int(2*i[2][0][0])][int(2*i[2][0][1])][int(i[2][1]/30)] == 0:
            if not i[2] in openSet:
                index += 1
                i[1] = index
                i[3] = first[3] + stepSize
                i[0] = i[3] + euclideanCostToGo(i[2][0], goal[0])

                parentDict[i[1]] = first[1]
                coordDict[i[1]] = i[2]
                costDict[i[1]] = i[0]
                c2cDict[i[1]] = i[3]

                openList.put(i)
                openSet.add(i[2])

                cv2.arrowedLine(doubleMaze, (int(2*first[2][0][0]),int(2*first[2][0][1])),
                                  (int(2*i[2][0][0]),int(2*i[2][0][1])),
                                  color = [255,0,0],
                                  thickness = 1)

                counter += 1
                if counter >= 50:
                    outVid.write(cv2.flip(doubleMaze,0))
                    counter = 0

        else:
            #print("Gotcha, ", i)
            tempIndex = {j for j in coordDict if coordDict[j] == i[2]}
            tempIndex = tempIndex.pop()
            if costDict[tempIndex] > first[3] + stepSize:
                parentDict[tempIndex] = first[1]
                c2cDict[tempIndex] = first[3] + stepSize
                costDict[tempIndex] = first[3] + stepSize + euclideanCostToGo(i[2][0], goal[0])

if solved == False:
    print ("Failure! Goal node not found")

print("Saving video... ")
outVid.release()

# play simulation video
print("Video saved successfully! Displaying video... \n")
cap = cv2.VideoCapture('output.mp4')

if cap.isOpened() == False:
    print("Error File Not Found")

while cap.isOpened():
    ret,frame= cap.read()
    if ret == True:
        cv2.imshow('frame', frame)
        if cv2.waitKey(25) & 0xFF == ord('q'):
            break
    else:
        break

cap.release()
print("Video displayed successfully! Program termination  \n")
cv2.destroyAllWindows()

# Resources
# https://www.programiz.com/dsa/priority-queue
# https://bobbyhadz.com/blog/python-input-tuple
# https://stackoverflow.com/questions/23294658/asking-the-user-for-input-until-they-give-a-valid-response
# https://www.w3schools.com/python/python_sets.asp
# https://www.freecodecamp.org/news/python-set-how-to-create-sets-in-python/#:~:text=How%20to%20Add%20Items%20to%20a%20Set%20in%20Python,passed%20in%20as%20a%20parameter.&text=We%20add
# https://stackoverflow.com/questions/30103077/what-is-the-codec-for-mp4-videos-in-python-opencv
# https://docs.opencv.org/3.4/dd/d43/tutorial_py_video_display.html
# https://www.geeksforgeeks.org/python-play-a-video-using-opencv/
# https://www.geeksforgeeks.org/python-opencv-cv2-arrowedline-method/
```