```python
# ========================================
# ENPM661 Spring 2023: Robotic Path Planning
# Project #3 Phase 2 Part 01
# Maze Search with Turtlebot3 using A* Algorithm with Non-Holonomic constraints
#
# 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 turtlebot3_astar_douglas_vignesh.py'
# Github link:
# Results link:
# 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
from collections import OrderedDict


def getValidRPMs(rpmThresh):
    while True:
        try:
            rpmInput = input(
                "Enter two wheel RPMs [rev per minute] as integer values between "
                + str(rpmThresh[0])
                + " - "
                + str(rpmThresh[1])
                + " , separated by a comma: "
            )
            rpms = tuple(int(item) for item in rpmInput.split(","))
        except (IndexError, ValueError):
            print(
                "Sorry, results invalid. Please try again, entering the wheel RPMs as integer values between "
                + str(rpmThresh[0])
                + " - "
                + str(rpmThresh[1])
                + " , separated by a comma: "
            )
            continue
        if ((rpms[0] or rpms[1]) < rpmThresh[0]) or (
            (rpms[0] or rpms[1]) > rpmThresh[1]
        ):
            print(
                "Sorry, results invalid. Please try again, entering the wheel RPMs as integer values between "
                + str(rpmThresh[0])
                + " - "
                + str(rpmThresh[1])
                + " , separated by a comma: "
            )
            continue
        else:
            break
    return rpms


def getValidClearance(robotRadius):
    while True:
        try:
            print(
                "The radius of the Turtlebot3 burger model is approximately ",
                (robotRadius * 1000),
                " [mm]. ",
            )
            clearance = int(
                input(
                    "Please enter the desired obstacle clearance as an integer value between "
                    + str(robotRadius * 1000)
                    + " and 120 [mm]: "
                )
            )
        except (IndexError, ValueError):
            print(
                "Sorry, results invalid. Please try again, entering the desired obstacle clearance as an integer value between "
                + str(robotRadius * 1000)
                + " and 120 [mm]: "
            )
            continue
        if clearance < robotRadius * 1000 or clearance >= 130:
            print(
                "Sorry, results invalid. Please try again, entering the desired obstacle clearance as an integer value between "
                + str(robotRadius * 1000)
                + " and 120 [mm]: "
            )
            continue
        else:
            break
    clearance = int(round(clearance / 10))
    return clearance


def getValidCoords(type, maze, clearance):
    theta = None

    while True:
        try:
            coordInput = input(
                "Enter "
                + type
                + " node coordinates in x, y format, in [cm], separated by a comma: "
            )
            coords = tuple(int(item) for item in coordInput.split(","))
        except (IndexError, ValueError):
            print(
                "Sorry, results invalid. Please try again, entering two integer inputs within the maze space. "
            )
            continue
        try:
            if (
                coords[0] < 0 + clearance
                or coords[0] > 600 - clearance
                or coords[1] < 0 + clearance
                or coords[1] > 250 - clearance
            ):
                print(
                    "Sorry, results invalid. Please try again, entering two integer inputs within the maze space. "
                )
                continue
        except (IndexError, ValueError):
            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 and type == "start":
        try:
            theta = int(
                input(
                    "Enter "
                    + type
                    + " node orientation as an integer between 0-359, using increments of 1 deg: "
                )
            )
            if theta >= 360 or theta < 0:
                raise ValueError
        except (IndexError, ValueError):
            print(
                (
                    "Sorry, entry invalid. Please try again, entering an integer input between 0-359 in increments of 1 deg. "
                )
            )
            continue
        if searchNode((coords, theta), RPM1, RPM2, maze) == False:
            print(
                (
                    "Sorry, entry invalid. Please try again, entering an integer input between 0-359 in increments of 1 deg, oriented toward the center of the mazespace. "
                )
            )
            continue
        else:
            break

    nodeState = (coords, theta)
    return nodeState


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(clearance):
    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=(clearance, clearance),
        pt2=(mazeSize[1] - clearance, mazeSize[0] - clearance),
        color=(255, 255, 255),
        thickness=-1,
    )

    # draw rectangle obstacles
    cv2.rectangle(
        maze,
        pt1=(100 - clearance, 0),
        pt2=(150 + clearance, 100 + clearance),
        color=(0, 255, 0),
        thickness=-1,
    )
    cv2.rectangle(
        maze,
        pt1=(100 - clearance, 150 - clearance),
        pt2=(150 + clearance, 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 - clearance],
            [
                365 + clearance,
                math.floor(125 - 37.5) - math.floor(clearance * math.sin(hexRad)),
            ],
            [
                365 + clearance,
                math.ceil(125 + 37.5) + math.ceil(clearance * math.sin(hexRad)),
            ],
            [300, 201 + clearance],
            [
                235 - clearance,
                math.ceil(125 + 37.5) + math.ceil(clearance * math.sin(hexRad)),
            ],
            [
                235 - clearance,
                math.floor(125 - 37.5) - math.floor(clearance * 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), clearance, color=(0, 255, 0), thickness=-1)
    cv2.circle(maze, (460, 225), clearance, color=(0, 255, 0), thickness=-1)
    cv2.circle(maze, (510, 125), clearance, color=(0, 255, 0), thickness=-1)

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

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

    triLowerBoundPts = np.array(
        [
            [510, 125],
            [
                510 + int(clearance * math.cos(triRad)),
                125 + int(clearance * math.sin(triRad)),
            ],
            [
                460 + int(clearance * math.cos(triRad)),
                225 + int(clearance * 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[xyCoords[1], xyCoords[0]] == [255, 255, 255]):
            return False
        else:
            return True
    except IndexError:
        return True


def normalizeAngle(ang):
    ang = ang % 360
    return ang


def getPlotPoints(floatPoints):
    roundedPoints = [(round(x), round(y)) for x, y in floatPoints]
    uniqueRoundedPoints = list(OrderedDict.fromkeys(roundedPoints))
    return uniqueRoundedPoints


def plotTrajectory(presentNode, plotPoints, maze):
    for i in range(len(plotPoints) - 1):
        cv2.line(
            maze,
            (plotPoints[i][0], plotPoints[i][1]),
            (plotPoints[i + 1][0], plotPoints[i + 1][1]),
            color=[255, 0, 0],
            thickness=1,
        )


# [cost, index, coords, c2c]
def actionCost(nodeCoords, RPM1, RPM2, maze):
    t = 0
```

```python
    step = 0

    thetaNew = math.pi * nodeCoords[1] / 180  # converts deg to rad
    xNew = nodeCoords[0][0]
    yNew = nodeCoords[0][1]

    validPath = True

    RPS1 = ((2 * math.pi) / 60) * RPM1  # rev per mintue to rad per sec
    RPS2 = ((2 * math.pi) / 60) * RPM2  # rev per mintue to rad per sec

    incrementCoords = []
    incrementCoords.append((xNew, yNew))

    while t < 1:  # DO NOT CHANGE
        t = t + dt

        deltaX = 0.5 * wheelRadius * (RPS1 + RPS2) * math.cos(thetaNew) * dt
        xNew += deltaX * 100
        deltaY = 0.5 * wheelRadius * (RPS1 + RPS2) * math.sin(thetaNew) * dt
        yNew += deltaY * 100
        incrementCoords.append((xNew, yNew))

        deltaTheta = (wheelRadius / wheelBase) * (RPS2 - RPS1) * dt
        thetaNew += deltaTheta

        step += math.sqrt(math.pow(deltaX*100, 2) + math.pow(deltaY*100, 2))

    thetaNew = 180 * (thetaNew) / math.pi

    plotPoints = getPlotPoints(incrementCoords)

    for i in plotPoints:
        if checkObstacle((i[0], i[1]), blankMaze) == True:
            validPath = False

    if validPath == True:
        # [cost, index, coords, c2c, step]
        newNode = [
            None,
            None,
            ((round(xNew), round(yNew)), round(normalizeAngle(thetaNew))),
            None,
            step,
        ]
        plotTrajectory(nodeCoords[0], plotPoints, maze)

        # Realtime livestream of search
        # intermediateMaze = cv2.flip(maze, 0)
        # while True:
        #     cv2.imshow("Maze", intermediateMaze)
        #     key = cv2.waitKey(1) & 0xFF
        #     # If the 'q' key is pressed, quit the loop
        #     if key == ord("q"):
        #         break
        #     break

        return newNode
    else:
        return None


def searchNode(nodeCoords, RPM1, RPM2, maze):
    results = []
    action1 = actionCost(nodeCoords, RPM1, RPM1, maze)
    if action1 is not None:
        results.append(action1)

    action2 = actionCost(nodeCoords, 0, RPM1, maze)
    if action2 is not None:
        results.append(action2)

    action3 = actionCost(nodeCoords, RPM1, 0, maze)
    if action3 is not None:
        results.append(action3)

    action4 = actionCost(nodeCoords, RPM2, RPM2, maze)
    if action4 is not None:
        results.append(action4)

    action5 = actionCost(nodeCoords, RPM1, RPM2, maze)
    if action5 is not None:
        results.append(action5)

    action6 = actionCost(nodeCoords, RPM2, RPM1, maze)
    if action6 is not None:
        results.append(action6)

    action7 = actionCost(nodeCoords, RPM2, 0, maze)
    if action7 is not None:
        results.append(action7)

    action8 = actionCost(nodeCoords, 0, RPM2, maze)
    if action8 is not None:
        results.append(action8)

    return results


def generatePath(nodeIndex, nodeCoords, maze):
    pathIndices = []
    pathCoords = []
    nodeCoords = nodeCoords[0]
    counta = 0

    print("Elements in parent dict: ", len(parentDict))
    print("Elements in coord dict: ", len(coordDict))

    while nodeIndex is not None:
        pathIndices.append(nodeIndex)
        pathCoords.append(nodeCoords)
        tempX = int(nodeCoords[0])
        tempY = int(nodeCoords[1])
        cv2.circle(maze, (tempX, tempY), 5, color=(0, 255, 255), thickness=-1)
        nodeCoords = coordDict[nodeIndex]
        nodeIndex = parentDict[nodeIndex]
        counta += 1
        print("Nodes in path: ", counta)

    return pathIndices, pathCoords


def simulateBot(pathCoords, emptyMaze, clearance):
    timer = 5
    for i in pathCoords:
        tempX = int(i[0])
        tempY = int(i[1])
        cv2.circle(emptyMaze, (tempX, tempY), 3, color=(0, 255, 255), thickness=-1)
        while timer > 0:
            outVid.write(cv2.flip(emptyMaze, 0))
            timer = timer - 1

    pathCoords.reverse()

    for i in pathCoords:
        emptyMazeCopy = emptyMaze.copy()
        tempXR = i[0]
        tempYR = i[1]
        currCirc = cv2.circle(
            emptyMazeCopy,
            (tempXR, tempYR),
            int(round(turtlebot3Radius*100)),
            color=(255, 0, 255),
            thickness=-1,
        )

        timer = 10
        while timer > 0:
            outVid.write(cv2.flip(currCirc, 0))
            timer = timer - 1

    timer = 60
    while timer >= 0:
        timer -= 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, (600, 250))

# hardcode robot params
turtlebot3Radius = 0.105  # [m]
wheelRadius = 0.033  # [m]
wheelBase = 0.160  # [m]
dt = 0.1  # DO NOT CHANGE
goalThresh = 10

# get obstacle clearance
clearance = getValidClearance(turtlebot3Radius)

# draw maze and make reserve
maze = drawMaze(clearance)
blankMaze = maze.copy()
counter = 30
while counter >= 0:
    counter -= 1
    outVid.write(cv2.flip(blankMaze, 0))

# get RPMs
rpmThresh = (1, 200)
RPM1, RPM2 = getValidRPMs(rpmThresh)

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

startTime = time.time()
solved = False

openList = PriorityQueue()
openSet = set()

# intialize data containers for backtracking

parentDict = {1: None}
coordDict = {1: start[0]}
costDict = {1: 0}
c2cDict = {1: 0}
closedSet = set()
closedList = []

# [cost, index, coords/theta, c2c]
startNode = [0, 1, start, 0, 0]
index = startNode[1]
openList.put(startNode)
openSet.add(start[0])

while not openList.empty() and solved == False:
    first = openList.get()
    openSet.remove(first[2][0])

    # print()
    # print("Current Node: ", first)
    # print()

    closedSet.add(first[2][0])
    closedList.append(first[2][0])
```

```python
    if euclideanCostToGo(first[2][0], goal[0]) <= goalThresh:
        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 = maze.copy()

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

        dispMaze = cv2.flip(dispMaze, 0)
        cv2.imshow("Generated Path", dispMaze)
        cv2.waitKey(0)

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

    results = searchNode(first[2], RPM1, RPM2, maze)

    for i in results:
        if not i[2][0] in closedSet:
            if not i[2][0] in openSet:
                index += 1
                i[1] = index
                i[3] = first[3] + i[4]
                i[0] = i[3] + 2 * euclideanCostToGo(i[2][0], goal[0]) # weighted by two

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

                openList.put(i)
                openSet.add(i[2][0])

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

        else:
            tempIndex = {j for j in coordDict if coordDict[j] == i[2][0]}
            tempIndex = tempIndex.pop()

            if c2cDict[tempIndex] > first[3] + i[4]:
                parentDict[tempIndex] = first[1]
                c2cDict[tempIndex] = first[3] + i[4]
                costDict[tempIndex] = (
                    first[3] + i[4] + 2 * euclideanCostToGo(i[2][0], goal[0])  # weighted by two
                )

    # input("Progress to next node?")

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.geeksforgeeks.org/python-get-unique-values-list/
# https://stackoverflow.com/questions/480214/how-do-i-remove-duplicates-from-a-list-while-preserving-order
# https://emanual.robotis.com/docs/en/platform/turtlebot3/features/#:~:text=The%20dimension%20of%20TurtleBot3%20Burger,L%20x%20W%20x%20H).
# ========================================
# ENPM661 Spring 2023: Robotic Path Planning
# Project #3 Phase 2
# Maze Search with Turtlebot3 using A* Algorithm with Non-Holonomic constraints
#
# 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 turtlebot3_astar_douglas_vignesh.py'
# Github link:
# Results link:
# Press CTRL+C for exit

#! /usr/bin/env python
import rospy
from geometry_msgs.msg import Twist
import time
import numpy as np
import matplotlib.pyplot as plt
import cv2
import math
from queue import PriorityQueue
import time
import sys
from collections import OrderedDict
import tf.transformations as tf
from gazebo_msgs.msg import ModelState
from gazebo_msgs.srv import SetModelState
from geometry_msgs.msg import Pose, Quaternion


def getValidRPMs(rpmThresh):
    while True:
        try:
            rpmInput = input(
                "Enter two wheel RPMs [rev per minute] as integer values between "
                + str(rpmThresh[0])
                + " - "
                + str(rpmThresh[1])
                + " , separated by a comma: "
            )
            rpms = tuple(int(item) for item in rpmInput.split(","))
        except (IndexError, ValueError):
            print(
                "Sorry, results invalid. Please try again, entering the wheel RPMs as integer values between "
                + str(rpmThresh[0])
                + " - "
                + str(rpmThresh[1])
                + " , separated by a comma: "
            )
            continue
        if ((rpms[0] or rpms[1]) < rpmThresh[0]) or (
            (rpms[0] or rpms[1]) > rpmThresh[1]
        ):
            print(
                "Sorry, results invalid. Please try again, entering the wheel RPMs as integer values between "
                + str(rpmThresh[0])
                + " - "
                + str(rpmThresh[1])
                + " , separated by a comma: "
            )
            continue
        else:
            break
    return rpms


def getValidClearance(robotRadius):
    while True:
        try:
            print(
                "The radius of the Turtlebot3 burger model is approximately ",
                (robotRadius * 1000),
                " [mm]. ",
            )
            clearance = int(
                input(
                    "Please enter the desired obstacle clearance as an integer value between "
                    + str(robotRadius * 1000)
                    + " and 250 [mm]: "
                )
            )
        except (IndexError, ValueError):
            print(
                "Sorry, results invalid. Please try again, entering the desired obstacle clearance as an integer value between "
                + str(robotRadius * 1000)
                + " and 250 [mm]: "
            )
            continue
        if clearance < robotRadius * 1000 or clearance > 250:
            print(
                "Sorry, results invalid. Please try again, entering the desired obstacle clearance as an integer value between "
                + str(robotRadius * 1000)
                + " and 250 [mm]: "
            )
            continue
        else:
            break
    clearance = int(round(clearance / 10))
    return clearance


def getValidCoords(type, maze, clearance):
    theta = None

    while True:
        try:
            coordInput = input(
                "Enter "
                + type
                + " node coordinates in x, y format, in [cm], separated by a comma: "
            )
            coords = tuple(int(item) for item in coordInput.split(","))
        except (IndexError, ValueError):
            print(
                "Sorry, results invalid. Please try again, entering two integer inputs within the maze space. "
            )
            continue
        try:
            if (
                coords[0] < 0 + clearance
                or coords[0] > 600 - clearance
                or coords[1] < 0 + clearance
                or coords[1] > 200 - clearance
            ):
                print(
                    "Sorry, results invalid. Please try again, entering two integer inputs within the maze space. "
                )
                continue
        except (IndexError, ValueError):
            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 and type == "start":
        try:
            theta = int(
                input(
                    "Enter "
                    + type
                    + " node orientation as an integer between 0-359, using increments of 1 deg: "
                )
            )
            if theta >= 360 or theta < 0:
                raise ValueError
        except (IndexError, ValueError):
            print(
                (
                    "Sorry, entry invalid. Please try again, entering an integer input between 0-359 in increments of 1 deg. "
                )
            )
            continue
        if searchNode((coords, theta), RPM1, RPM2, maze) == False:
            print(
                (
                    "Sorry, entry invalid. Please try again, entering an integer input between 0-359 in increments of 1 deg, oriented toward the center of the mazespace. "
                )
            )
            continue
        else:
            break

    nodeState = (coords, theta)
    return nodeState


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(clearance):
    mazeSize = (200, 600)

    # Create blank maze
    maze = np.zeros((mazeSize[0], mazeSize[1], 3), dtype=np.uint8)
    maze[:] = (0, 255, 0)

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

    # Draw rectangular obstacles
    cv2.rectangle(
        maze,
        pt1=(150 - clearance, 75 - clearance),
        pt2=(165 + clearance, mazeSize[0] + clearance),
        color=(0, 255, 0),
        thickness=-1,
    )

    cv2.rectangle(
        maze,
        pt1=(235 - clearance, clearance),
        pt2=(250 + clearance, 125 + clearance),
        color=(0, 255, 0),
        thickness=-1,
    )

    cv2.rectangle(
        maze, pt1=(150, 75), pt2=(165, mazeSize[0]), color=(0, 0, 255), thickness=-1
    )
    cv2.rectangle(maze, pt1=(235, 0), pt2=(250, 125), color=(0, 0, 255), thickness=-1)

    # Draw circular obstacles
```

```python
    cv2.circle(maze, (400, 110), (50 + clearance), color=(0, 255, 0), thickness=-1)
    cv2.circle(maze, (400, 110), 50, color=(0, 0, 255), thickness=-1)
    return maze


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


def normalizeAngle(ang):
    ang = ang % 360
    return ang


def getPlotPoints(floatPoints):
    roundedPoints = [(round(x), round(y)) for x, y in floatPoints]
    uniqueRoundedPoints = list(OrderedDict.fromkeys(roundedPoints))
    return uniqueRoundedPoints


def plotTrajectory(presentNode, plotPoints, maze):
    for i in range(len(plotPoints) - 1):
        cv2.line(
            maze,
            (plotPoints[i][0], plotPoints[i][1]),
            (plotPoints[i + 1][0], plotPoints[i + 1][1]),
            color=[255, 0, 0],
            thickness=1,
        )


# [cost, index, coords, c2c]
def actionCost(nodeCoords, RPM1, RPM2, maze):
    t = 0
    step = 0

    thetaNew = math.pi * nodeCoords[1] / 180  # converts deg to rad
    xNew = nodeCoords[0][0]
    yNew = nodeCoords[0][1]

    validPath = True

    RPS1 = ((2 * math.pi) / 60) * RPM1  # rev per mintue to rad per sec
    RPS2 = ((2 * math.pi) / 60) * RPM2  # rev per mintue to rad per sec

    incrementCoords = []
    incrementCoords.append((xNew, yNew))
    velSteps = []

    while round(t,1) < 1:  # DO NOT CHANGE
        t = t + dt

        deltaX = 0.5 * wheelRadius * (RPS1 + RPS2) * math.cos(thetaNew) * dt  # meters
        xNew += deltaX * 100  # cm
        deltaY = 0.5 * wheelRadius * (RPS1 + RPS2) * math.sin(thetaNew) * dt  # meters
        yNew += deltaY * 100  # cm
        incrementCoords.append((xNew, yNew))

        deltaTheta = (wheelRadius / wheelBase) * (RPS2 - RPS1) * dt  # radians
        thetaNew += deltaTheta

        step += math.sqrt(math.pow(deltaX * 100, 2) + math.pow(deltaY * 100, 2))

        velSteps.append((deltaX/dt, deltaY/dt, deltaTheta/dt))      #Maybe ydot times (-1)

    thetaNew = 180 * (thetaNew) / math.pi

    plotPoints = getPlotPoints(incrementCoords)

    for i in plotPoints:
        if checkObstacle((i[0], i[1]), blankMaze) == True:
            validPath = False

    if validPath == True:
        # [cost, index, coords, c2c, step]
        newNode = [
            None,
            None,
            ((round(xNew), round(yNew)), round(normalizeAngle(thetaNew))),
            None,
            step,
            velSteps
        ]
        plotTrajectory(nodeCoords[0], plotPoints, maze)

        # Realtime livestream of search
        # intermediateMaze = cv2.flip(maze, 0)
        # while True:
        #     cv2.imshow("Maze", intermediateMaze)
        #     key = cv2.waitKey(1) & 0xFF
        #     # If the 'q' key is pressed, quit the loop
        #     if key == ord("q"):
        #         break
        #     break

        return newNode
    else:
        return None


def searchNode(nodeCoords, RPM1, RPM2, maze):
    results = []
    action1 = actionCost(nodeCoords, RPM1, RPM1, maze)
    if action1 is not None:
        results.append(action1)
    action2 = actionCost(nodeCoords, 0, RPM1, maze)
    if action2 is not None:
        results.append(action2)

    action3 = actionCost(nodeCoords, RPM1, 0, maze)
    if action3 is not None:
        results.append(action3)

    action4 = actionCost(nodeCoords, RPM2, RPM2, maze)
    if action4 is not None:
        results.append(action4)

    action5 = actionCost(nodeCoords, RPM1, RPM2, maze)
    if action5 is not None:
        results.append(action5)

    action6 = actionCost(nodeCoords, RPM2, RPM1, maze)
    if action6 is not None:
        results.append(action6)

    action7 = actionCost(nodeCoords, RPM2, 0, maze)
    if action7 is not None:
        results.append(action7)

    action8 = actionCost(nodeCoords, 0, RPM2, maze)
    if action8 is not None:
        results.append(action8)

    return results


def generatePath(nodeIndex, nodeCoords, maze):
    pathIndices = []
    pathCoords = []
    nodeCoords = nodeCoords[0]
    counta = 0

    print("Elements in parent dict: ", len(parentDict))
    print("Elements in coord dict: ", len(coordDict))

    while nodeIndex is not None:
        pathIndices.append(nodeIndex)
        pathCoords.append(nodeCoords)
        tempX = int(nodeCoords[0])
        tempY = int(nodeCoords[1])
        cv2.circle(maze, (tempX, tempY), 5, color=(0, 255, 255), thickness=-1)
        nodeCoords = coordDict[nodeIndex]
        nodeIndex = parentDict[nodeIndex]
        counta += 1
        print("Nodes in path: ", counta)

    return pathIndices, pathCoords


def simulateBot(pathCoords, emptyMaze, clearance):
    timer = 5
    for i in pathCoords:
        tempX = int(i[0])
        tempY = int(i[1])
        cv2.circle(emptyMaze, (tempX, tempY), 3, color=(0, 255, 255), thickness=-1)
        while timer > 0:
            outVid.write(cv2.flip(emptyMaze, 0))
            timer = timer - 1

    pathCoords.reverse()

    for i in pathCoords:
        emptyMazeCopy = emptyMaze.copy()
        tempXR = i[0]
        tempYR = i[1]
        currCirc = cv2.circle(
            emptyMazeCopy,
            (tempXR, tempYR),
            int(round(turtlebot3Radius * 100)),
            color=(255, 0, 255),
            thickness=-1,
        )

        timer = 10
        while timer > 0:
            outVid.write(cv2.flip(currCirc, 0))
            timer = timer - 1

    timer = 60
    while timer >= 0:
        timer -= 1
        outVid.write(cv2.flip(currCirc, 0))


def actuateTurtlebot(pathIndices, pathCoords, startPosition):
    pathVels = []
    pathIndices.reverse()
    velIndex = 0
    print("Final path indices: ", pathIndices, "\n")

    print("Getting path dynamics... \n")
    for i in pathIndices:
        print("Getting velocities of node: ", i)
        vels = stepDict[i]
        for j in vels:
            pathVels.append((((j[0] ** 2) + (j[1] ** 2)) ** 0.5, j[2]))

    print()
    print("Intializing ROS node... \n")
    rospy.init_node('robot_talker', anonymous=True)
    rate = rospy.Rate(10)  #10hz

    print("Setting Turtlebot position in map... \n")

    pubModelState = rospy.Publisher('/gazebo/set_model_state', ModelState, queue_size=10)
    rospy.wait_for_service('/gazebo/set_model_state')

    try:
        modelStateMsg = ModelState()
        modelStateMsg.model_name = 'turtlebot3_burger'
        modelStateMsg.pose.position.x = startPosition[0][0]/100 - 0.50
        modelStateMsg.pose.position.y = startPosition[0][1]/100 - 1.00
        modelStateMsg.pose.position.z = 0

        roll = 0.0
        pitch = 0.0
        yaw = math.pi * startPosition[1] / 180    # rotation around z-axis in RADIANS

        quaternion = tf.quaternion_from_euler(roll, pitch, yaw)

        modelStateMsg.pose.orientation.x = quaternion[0]
        modelStateMsg.pose.orientation.y = quaternion[1]
        modelStateMsg.pose.orientation.z = quaternion[2]
        modelStateMsg.pose.orientation.w = quaternion[3]

        setModelState = rospy.ServiceProxy('/gazebo/set_model_state', SetModelState)
        response = setModelState(modelStateMsg)

    # catch exceptions
    except rospy.ServiceException as e:
        print("Service call failed: ", e)

    # pub_model_state.publish(modelStateMsg)

    print("Position set!")
    input("Press enter to articulate the robot!")

    velMsg = Twist()
    pubCmdVel = rospy.Publisher('/cmd_vel',Twist,queue_size=10)

    print("\n rticulating Turtlebot3 in gazebo...")
    while not rospy.is_shutdown():
        for i in pathVels:
            velMsg.angular.z = i[1] * 1.02
            if (i[1] != 0):
                velMsg.linear.x = i[0] * 1.02
            else:
                velMsg.linear.x = i[0] * 1.0
            pubCmdVel.publish(velMsg)
            rate.sleep()

        velMsg.angular.z = 0
        velMsg.linear.x = 0
        pubCmdVel.publish(velMsg)
        print("Finished articulating Turtlebot! Program Termination.")
        print()
        break


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

fourcc = cv2.VideoWriter_fourcc(*"mp4v")
outVid = cv2.VideoWriter("output.mp4", fourcc, 30, (600, 200))

# hardcode robot params
turtlebot3Radius = 0.105  # [m]
wheelRadius = 0.033  # [m]
wheelBase = 0.160  # [m]
dt = 0.1  # DO NOT CHANGE
goalThresh = 10

# get obstacle clearance
clearance = getValidClearance(turtlebot3Radius)

# draw maze and make reserve
maze = drawMaze(clearance)
blankMaze = maze.copy()
counter = 30
while counter >= 0:
    counter -= 1
    outVid.write(cv2.flip(blankMaze, 0))

# get RPMs
rpmThresh = (1, 200)
RPM1, RPM2 = getValidRPMs(rpmThresh)

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

startTime = time.time()
solved = False

openList = PriorityQueue()
openSet = set()

# intialize data containers for backtracking

parentDict = {1: None}
coordDict = {1: start[0]}
stepDict = {1: [(0,0,0)]}
costDict = {1: 0}
c2cDict = {1: 0}
closedSet = set()
closedList = []

# [cost, index, coords/theta, c2c]
startNode = [0, 1, start, 0, 0]
index = startNode[1]
openList.put(startNode)
openSet.add(start[0])

while not openList.empty() and solved == False:
    first = openList.get()
    openSet.remove(first[2][0])

    # print()
    # print("Current Node: ", first)
    # print()

    closedSet.add(first[2][0])
    closedList.append(first[2][0])

    if euclideanCostToGo(first[2][0], goal[0]) <= goalThresh:
        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 = maze.copy()

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

        dispMaze = cv2.flip(dispMaze, 0)
        cv2.imshow("Generated Path", dispMaze)
        cv2.waitKey(0)

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

    results = searchNode(first[2], RPM1, RPM2, maze)

    for i in results:
        if not i[2][0] in closedSet:
            if not i[2][0] in openSet:
                index += 1
                i[1] = index
                i[3] = first[3] + i[4]
                i[0] = i[3] + 2 * euclideanCostToGo(i[2][0], goal[0])  # weighted by two

                parentDict[i[1]] = first[1]
                coordDict[i[1]] = i[2][0]
                costDict[i[1]] = i[0]
                c2cDict[i[1]] = i[3]
                stepDict[i[1]] = i[5]

                openList.put(i)
                openSet.add(i[2][0])

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

        else:
            tempIndex = {j for j in coordDict if coordDict[j] == i[2][0]}
            tempIndex = tempIndex.pop()

            if c2cDict[tempIndex] > first[3] + i[4]:
                parentDict[tempIndex] = first[1]
                c2cDict[tempIndex] = first[3] + i[4]
                costDict[tempIndex] = (
                    first[3]
                    + i[4]
                    + 2 * euclideanCostToGo(i[2][0], goal[0])  # weighted by two
                )

    #input("Progress to next node?")

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! \n")
cv2.destroyAllWindows()


if __name__=='__main__':
    actuateTurtlebot(pathIndices,pathCoords,start)

# Resources:
# https://www.geeksforgeeks.org/python-get-unique-values-list/
# https://stackoverflow.com/questions/480214/how-do-i-remove-duplicates-from-a-list-while-preserving-order
# https://emanual.robotis.com/docs/en/platform/turtlebot3/features/#:~:text=The%20dimension%20of%20TurtleBot3%20Burger,L%20x%20W%20x%20H).
# http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber%28python%29
# https://answers.gazebosim.org//question/22125/how-to-set-a-models-position-using-gazeboset_model_state-service-in-python/
# http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber%28python%29
# https://de3-panda-wall.readthedocs.io/en/latest/spawn_doc.html
```