```python
#Part 1
# ENPM661 Project 3 Part 2 Submission
# Rey Roque-Pérez and Dustin Hartnett

# Github Repository: https://github.com/dhartnet/ENPM661-Project-3-Part-2

from queue import PriorityQueue
import time

import numpy as np
import cv2

# PriorityQueues entries in python are not updatable
# This class is an implementation of the PriorityQueue that allows updating
# It does this by keeping a copy of all items in the queue in a dictionary
# It then uses the dictionary to search if an item is in the queue
# It also passes a new argument to the put method (priority, item)
class UpdateableQueue:
    def __init__(self):
        self.queue = PriorityQueue()
        # Maps items to their corresponding queue entries
        self.entry_finder = {}

    # Adds or updates item in PriorityQueue
    def put(self, priority, node):
        # Extracting the key (x, y, theta) from the node
        key = node[0]
        if key in self.entry_finder:
            current_priority, _ = self.entry_finder[key]
            if priority < current_priority:
                # Update the priority in the entry_finder
                self.entry_finder[key] = (priority, node)
                # Update the priority in the queue
                self.queue.put((priority, node))

        else:
            entry = (priority, node)
            self.entry_finder[key] = entry
            self.queue.put(entry)

    def remove(self, key):
        entry = self.entry_finder.pop(key)
        # Return the item removed from the queue
        return entry[1]
```

```python
    def get(self):
        priority, node = self.queue.get()
        # Check if the item is still in entry_finder
        key = node[0]
        if key in self.entry_finder:
            # Remove the item from the entry_finder
            del self.entry_finder[key]
        return priority, node

    def empty(self):
        return self.queue.empty()

    def __contains__(self, key):
        return key in self.entry_finder

###---------------------------------------###
# Check to see if node in question lies within obstacle space
# Return 'False' if in free space, 'True' if in an obstacle or outside the
boundaries
def inObstacle(maybeNode, clearance, radius):

    node = tuple(maybeNode)
    xnode = node[0]
    ynode = node[1]
    vibes = False

    h = 420 # x coord of center of circle obstacle
    k = 120 # y coord of center of circle obstacle
    r = 60 # radius of circle obstacle

    # check if in map
    if xnode < clearance + radius or xnode > spaceX - clearance - radius or ynode
< clearance + radius or ynode > spaceY - clearance - radius:
        vibes = True

    # check first obstacle (rectangle)
    elif xnode > 150 - clearance - radius and xnode < 175 + clearance + radius
and ynode > 100 - clearance - radius:
        vibes = True

    # check second obstacle (rectangle)
    elif xnode > 250 - clearance - radius and xnode < 275 + clearance + radius
and ynode < 100 + clearance + radius:
        vibes = True
```

```
    # check third obstacle (circle)
    elif (xnode - h)**2 + (ynode - k)**2 - (r + clearance + radius)**2 <= 0:
        vibes = True

  # return "vibes". False = node is in free space. True = node is out of map or
in obstacle.
    return vibes

# Check if node is inside of goal boundary
def inGoal(node, goal_node):
    goal_radius = 10 # cm
    x_goal = goal_node[0]
    y_goal = goal_node[1]
    x_node = node[0]
    y_node = node[1]
    return np.sqrt(np.square(x_node-x_goal) + np.square(y_node-y_goal)) <
goal_radius

# Draws start node and end node
def draw_nodes(canvas, start_node, goal_node):
    cv2.rectangle(canvas, (conversion * start_node[0] - 10, visY - conversion *
start_node[1] - 10), (conversion * start_node[0] + 10, visY - conversion *
start_node[1] + 10), color=(0, 250, 0), thickness=cv2.FILLED)
    cv2.rectangle(canvas, (conversion * goal_node[0] - 10, visY - conversion *
goal_node[1] - 10), (conversion * goal_node[0] + 10, visY - conversion *
goal_node[1] + 10), color=(0, 0, 255), thickness=cv2.FILLED)

# Creates a list of the obstacles in the workspace
def obstacle_space():
    obstacle_list = []

    obstacle_list.append(((0,0),(visX,visY)))
    obstacle_list.append(((1500,2000),(1750,1000)))
    obstacle_list.append(((2500,0),(2750,1000)))
    obstacle_list.append((4200,1200,600))

    return obstacle_list

# Populates the canvas with the obstacles
def draw_obstacles(canvas, obstacles, video_output):
    for obstacle in obstacles:
        if len(obstacle) == 2 and obstacle != ((0,0),(visX,visY)):
            start_x = obstacle[0][0]
            # Invert y-value
            start_y = visY - obstacle[0][1]
```

```
            end_x = obstacle[1][0]
            # Invert y-value
            end_y = visY - obstacle[1][1]
            start_coordinate = (start_x, start_y)
            end_coordinate = (end_x, end_y)
            cv2.rectangle(canvas, pt1=start_coordinate, pt2=end_coordinate,
color=(0, 0, 0), thickness=-1)
        elif len(obstacle) == 3:
            cv2.circle(canvas,(obstacle[0],visY-obstacle[1]), obstacle[2],
(0,0,0), -1)
    canvas1 = cv2.resize(canvas, (resizeX, resizeY))
    cv2.imshow('A*', canvas1)
    video_output.write(canvas1)
    cv2.waitKey(100)
    return

# Populates and updates the canvas with explored nodes
def draw_explored(canvas, points, video_output, u1, u2):
    count = 0
    t = 0 # start time
    r =  3.3 * conversion # wheel radius cm
    L =  28.7 * conversion # robot diameter cm
    dt = 0.2 # time step for plotting

    u1 = np.pi * u1 / 30
    u2 = np.pi * u2 / 30

    actions = [[0, u1], [u1, 0], [u1, u1], [0, u2], [u2, 0], [u2, u2], [u1, u2],
[u2, u1]]

    # Start from the second point
    for i in range(0, len(points)-1):
        point = points[i]

        current_node = point[0]

        for action in actions:
            x = int(conversion * current_node[0])
            y = int(conversion * current_node[1])
            theta = np.pi * current_node[2] * 30 / 180 # orientation in radians
            t = 0
            ul = action[0] # left wheel vel
            ur = action[1] # right wheel vel

            while t <= 1.0:
```

```
                if inObstacle((x//conversion, y//conversion), clearance, radius):
                    break
                t = t + dt
                xs = x
                ys = y
                x += int((r/2) * (ul + ur) * np.cos(theta) * dt)
                y += int((r/2) * (ul + ur) * np.sin(theta) * dt)
                theta += (r/L) * (ur - ul) * dt
                cv2.line(canvas, (xs, visY - ys), (x, visY - y), (255,0,0),
conversion) # image, point 1, point 2, color, thickness

            count += 1

        if count % 4000 == 0:
            count = 0
            canvas1 = cv2.resize(canvas, (resizeX, resizeY))
            cv2.imshow('A*', canvas1)
            video_output.write(canvas1)
            # cv2.waitKey(1000//120)
            cv2.waitKey(1)

    canvas1 = cv2.resize(canvas, (resizeX, resizeY))
    cv2.imshow('A*', canvas1)
    video_output.write(canvas1)
    cv2.waitKey(1)
    return

# Populates and updates the canvas with path nodes
def draw_path(canvas, path, video_output):
    count = 0
    prev_point = None  # Initialize previous point
    for i in range(len(path)):
        point = path[i][0]

        # Draw the current path node
        cv2.rectangle(canvas, (conversion * point[0], visY - conversion *
point[1]), (conversion * point[0] + 10, visY - conversion * point[1] + 10),
color=(0, 0, 250), thickness=thickness)

        if prev_point is not None:
            # Draw a line from the previous point to the current point
            cv2.line(canvas, (conversion * prev_point[0], visY - conversion *
prev_point[1]), (conversion * point[0], visY - conversion * point[1]), (0, 0, 0),
thickness)
```

```
        # Create a temporary copy of the canvas
        temp_canvas = canvas.copy()

        # Draw a circle at the current point
        cv2.circle(temp_canvas, (conversion * point[0], visY - conversion *
point[1]), conversion * radius, color=(0, 0, 255), thickness=-1)

        count += 1
        if count % 1 == 0:
            count = 0
            canvas1 = cv2.resize(temp_canvas, (resizeX, resizeY))
            cv2.imshow('A*', canvas1)
            # Adding the same frame to the video multiple times so it goes slower
            video_output.write(canvas1)
            video_output.write(canvas1)
            video_output.write(canvas1)
            video_output.write(canvas1)
            video_output.write(canvas1)
            video_output.write(canvas1)
            video_output.write(canvas1)
            video_output.write(canvas1)
            video_output.write(canvas1)
            video_output.write(canvas1)
            video_output.write(canvas1)
            video_output.write(canvas1)
            video_output.write(canvas1)
            video_output.write(canvas1)
            video_output.write(canvas1)
            cv2.waitKey(1000//10)

        prev_point = point  # Update previous point

    cv2.waitKey(1) # video close time in ms
    return

# Adds blank frames for x amount of seconds at end of video
def add_blank_frames(canvas, video_output, fps, seconds):
    blank_frames = fps * seconds
    canvas1 = cv2.resize(canvas, (resizeX, resizeY))
    for _ in range(blank_frames):
        video_output.write(canvas1)
    return

# Non-holonomic move function
```

```
def newNodes(nodeState, clearance, radius, u1, u2): # (node, lower wheel
velocity, higher wheel velocity) speeds in RPM
  # Extract node information from nodeState value
  node = tuple(nodeState[0]) # Rounded node
  raw_node = tuple(nodeState[3]) # Unrounded node
  xi = node[0] # Rounded node
  yi = node[1] # Rounded node
  thetai = node[2]*30 # deg # Rounded node

  # xi = raw_node[0] # Unrounded node
  # yi = raw_node[1] # Unrounded node
  # thetai = raw_node[2] # Unrounded node

  u1 = np.pi * u1 / 30 # (rad/s)
  u2 = np.pi * u2 / 30# (rad/s)

  # Define action set from wheel rotational velocities
  actions = [[0, u1], [u1, 0], [u1, u1], [0, u2], [u2, 0], [u2, u2], [u1, u2],
[u2, u1]]

  # make empty lists to fill with values
  newNodes = [] # new node information
  c2c = [] # list of costs to come for each new node from parent node

  # define constants and counter values
  t = 0 # time (sec)
  dt = 0.1 # time step for integrating
  r =  3.3 # wheel radius cm
  L =  28.7 # robot diameter cm

  for action in actions:
    ul = action[0] # left wheel rotation velocity (rad/s)
    ur = action[1] # right wheel rotation velocity (rad/s)
    theta = np.pi * thetai / 180 # orientation in radians
    x = xi # set x value to initial node value before integration
    y = yi # set y value to initial node value before integration

    # we let each action run for one second, with a time step of 0.1 seconds for
integration calculation
    while t <= 1.0:
      t = t + dt
      x = x + (r/2) * (ul + ur) * np.cos(theta) * dt
      y = y + (r/2) * (ul + ur) * np.sin(theta) * dt
      theta = theta + (r/L) * (ur - ul) * dt
```

```
    t = 0
    c2c = np.sqrt((xi - x)**2 + (yi - y)**2) # cost to come is linear
displacement, not calculating distance
    newX = int(round(x,0))
    newY = int(round(y,0))

    new_theta = 180 * theta / np.pi #deg
    if new_theta < 0:
      new_theta = 360 + new_theta

    if new_theta >= 360:
      new_theta = new_theta - 360

    theta = new_theta
    new_theta = int((round(new_theta, 0) % 360)/30) # Rounded theta for comparing
nodes

    # v = round((r/2) * (ul + ur), 2)
    # ang = (r/L) * (ur - ul)
    # ang = round(ang, 2) # rpm

    v = (r/2) * (ul + ur)
    ang = (r/L) * (ur - ul)

    if not (newX < clearance + radius or newX > spaceX - clearance - radius or
newY < clearance + radius or newY > spaceY - clearance - radius):
      newNodes.append(((newX, newY, new_theta), round(c2c), (v, ang), (x, y,
theta))) # outputs node, cost to come, and associated linear and ang velocity to
get to each node (to be sent to robot)

  return newNodes

# Calculates cost to go
def heuristic(node, goal_node, weight):
  return weight * np.sqrt(np.square(goal_node[0] - node[0]) +
np.square(goal_node[1] - node[1]))

# Runs A* from start node to goal node and returns visited list and parent
information
def a_star_algorithm(start, goal, weight, rpm1, rpm2, clearance, radius):
    start_node = (int(start[0]), int(start[1]), start[2])
    goal_node = (int(goal[0]), int(goal[1]), goal[2])

    # Create cost_grid and initialize cost to come for start_node
```

```
    cost_grid = [[[float('inf')] * 12 for _ in range(spaceY)] for _ in
range(spaceX)]
    cost_grid[start_node[0]][start_node[1]][start_node[2]] = 0

    # Create grid to store parents
    parent_grid = [[[None] * 12 for _ in range(spaceY)] for _ in range(spaceX)]
    parent_grid[start_node[0]][start_node[1]][start_node[2]] = None

    # Create grid to store parents
    visited_grid = [[[False] * 12 for _ in range(spaceY)] for _ in range(spaceX)]
    visited_list = []

    # Priority queue to store open nodes
    # Cost to come, coordinate values (x,y), parent
    open_queue = UpdateableQueue()
    open_queue.put(0, (start_node,(0),(0,0),
(start_node[0],start_node[1],int(30*start_node[2]))))  # (priority, node)

    while not open_queue.empty():
        _, node_tuple = open_queue.get()
        node = node_tuple[0]
        #raw_node = node_tuple[3]
        visited_grid[node[0]][node[1]][node[2]] = True
        visited_list.append(node_tuple)

        if inGoal(node, goal_node):
            return parent_grid, visited_list

        # Get neighboring nodes
        actions = newNodes(node_tuple, clearance, radius, rpm1, rpm2)
        node_cost = cost_grid[node[0]][node[1]][node[2]]

        for action in actions:
            #print(action)
            action_cost = action[1]
            move = action[0]
            raw_move = action[3]
            if not visited_grid[move[0]][move[1]][move[2]] and not
inObstacle(move, clearance, radius):
                new_cost = node_cost + action_cost
                if new_cost < cost_grid[move[0]][move[1]][move[2]]:
                    cost_grid[move[0]][move[1]][move[2]] = new_cost
                    priority = new_cost + heuristic(raw_move, goal_node, weight)
                    open_queue.put(priority, action)
                    parent_grid[move[0]][move[1]][move[2]] = node_tuple
```

```
    return parent_grid, visited_list, print("Failed to find goal")

# Backtracking using path list created from visited/path dictionary
def find_path(parent_grid, visited_list, start):
    node_tuple = visited_list[-1]
    current_node = node_tuple[0]
    path = [node_tuple]
    start_node = start
    while start_node != current_node:
        temp_node =
parent_grid[int(current_node[0])][int(current_node[1])][current_node[2]]
        current_node = temp_node
        path.insert(0, current_node)
        current_node = current_node[0]
    return path

#### Main ###
# Grid Size Variables - Used as indexes for storing node information
spaceX = 600 # cm
spaceY = 200 # cm

# Visualization Size variables
visX = 6000 # pixels
visY = 2000 # pixels

# Used to scale node coordinates to visualization coordinates
conversion = visX//spaceX

# Resized canvas. We populate a 6000x2000 canvas to draw using the high
resolution-
# Then we resize that drawn canvas to 1200x400-
# Technically we could have used 1200x400 from the start but it doesn't look as
detailed.
resizeX = 1200 # pixels
resizeY = 400 # pixels
thickness = conversion

weight = 1 # option for weighted A*

# Robot Radius
radius = 22 # cm

# Additional clearance to be added to the radius
clearance = 2 # cm
```

```
# Pre-defined start node in Gazebo project
start_node = tuple((100, 100, 0))

#Node used for testing
# goal_node = tuple((575, 100, 0))

# Wheel speeds in RPM
rpm1 = (int(40))
rpm2 = (int(80))

print('Start Node is (100, 100, 0) (centimeters)', '\n')
print('Wheel speeds are 40 and 80 RPM', '\n')

# Get and verify input goal coordinate
xg = int(input('Enter x coordinate value for goal coordinate in centimeters: '))
yg = int(input('Enter y coordinate value for goal coordinate in centimeters: '))
thetag = int(0)//30
goal_node = tuple((xg, yg, thetag)) # cm
while inObstacle(goal_node, clearance, radius):
    print('Node outside workspace or in obstacle. Choose new goal location')
    xg = int(input('Enter x coordinate value for goal location in centimeters:
'))
    yg = int(input('Enter y coordinate value for goal location in centimeters:
'))
    thetag = int(0)//30
    goal_node = tuple((xg, yg, thetag)) # cm
goal_node = tuple((xg, yg, thetag)) # cm

print('Goal Node is ', goal_node, ' in centimeters', '\n')

# Start timer
ti = time.time()

# Run A* Algorithm
print('Exploring nodes...')
explored = a_star_algorithm(start_node, goal_node, weight, rpm1, rpm2, clearance,
radius)
parent_grid = explored[0]
visited_list = explored[1]

# Run BackTracking to generate path
print('Generating path...')
path = find_path(parent_grid, visited_list, start_node)
```

```python
# Get time taken to find path
tf = time.time()
print('Path found in: ', tf-ti)

# ## Initialize Canvas to visualization size
canvas = np.ones((visY, visX, 3), dtype=np.uint8) * 255  # White canvas

# Initialize video. Video is saved to working directory.
v_writer = cv2.VideoWriter_fourcc(*'mp4v')
fps = 60
video_output = cv2.VideoWriter('a_star_output.mp4', v_writer, fps, (resizeX,
resizeY))

# Get obstacle coordinates
obstacles = obstacle_space()
# Initializes Map and Draws the obstacles
draw_obstacles(canvas, obstacles, video_output)

# Draws the Start and End Node
draw_nodes(canvas, start_node, goal_node)

# Draws the explored nodes.
draw_explored(canvas, visited_list, video_output, rpm1, rpm2)

# Draws the Start and End Node covered by the explored lines
draw_nodes(canvas, start_node, goal_node)

# Draws path and circle representing the robot
print('drawing_path')
draw_path(canvas, path, video_output)

# Adds 2 frames at end of video so that it doesn't end instantly
add_blank_frames(canvas, video_output, fps, 2)
video_output.release()



#Part 2

from A_star import * # This imports the path data from the A_Star.py file
import rclpy
from rclpy.node import Node
from geometry_msgs.msg import Twist

class rosControlNode(Node):
```

```
    # Initialize Publisher
    def __init__(self):
        super().__init__('robot_control')
        self.cmd_vel_pub = self.create_publisher(Twist, '/cmd_vel', 10)

    # Send commands to robot

    # We think there may be a conversion difference between truw time using the
time.time() function compaored to the Gazebo simulation time that was
    # introducing an error to the commands. TO fix this we had to add some gains
in places where the error accrued significantly

    def robotControl(self, path):
      i = 0
      # Gains for linear and angular velocity and step time used throughout the
code
      cl = 0.995 # lin gain
      ca = 0.9 # ang gain
      t = 1.55
      velocity_message = Twist()

      for item in path:
        # Extract position, linear velocity and angulaw velocity from the path
        self.coord = item[0]
        self.x = self.coord[0] # x-coordinate of robot for positional awareness
        self.action = item[2]
        self.lin = self.action[0]/100 # Linear velocity at each node converted to
meters/s from centimeters/s
        self.ang = self.action[1] # angular velocity at each node in rad/s
        self.ts =  time.time() #start time
        self.tc = time.time() # current time

        # due to error we had to limit some aggresive turns
        if self.ang > 0.9:
          self.ang = 0.4
        if self.ang < -0.9:
          self.ang = -0.4

        # Add gains when navigating the circle because accrued error was
especially large there
        if self.x > 340 and self.x < 400:
          self.ang = self.ang * 0.7
          self.lin = self.lin * 0.275
```

```
        # Gains for after circle
        if self.x > 400:
          self.lin = self.lin * 0.925

        velocity_message.linear.x = cl*float(self.lin) # m/s
        velocity_message.angular.z = ca*float(self.ang) # rad/s
        self.cmd_vel_pub.publish(velocity_message)

        # Print functions showing linear and angular velocity at each node, as
well as a time stamp and step number
        print('printing  ', self.lin, '  and  ', self.ang, '\n')
        print(self.tc, '\n')
        print('finished step ', i, '\n')
        i = i + 1

        self.ts =  time.time() #start time
        self.tc = time.time()

        # Stop the robot at the end
        while self.tc - self.ts <= t:

          self.tc = time.time()

        velocity_message.linear.x = 0.0 # m/s
        velocity_message.angular.z = 0.0 # rad/s
        self.cmd_vel_pub.publish(velocity_message)

      print('FINISHED', '\n')

def main(args=None):
    rclpy.init(args=args)
    node = rosControlNode()
    node.robotControl(path)
    node.destroy_node()
    rclpy.shutdown()

if __name__ == '__main__':
    main()
```