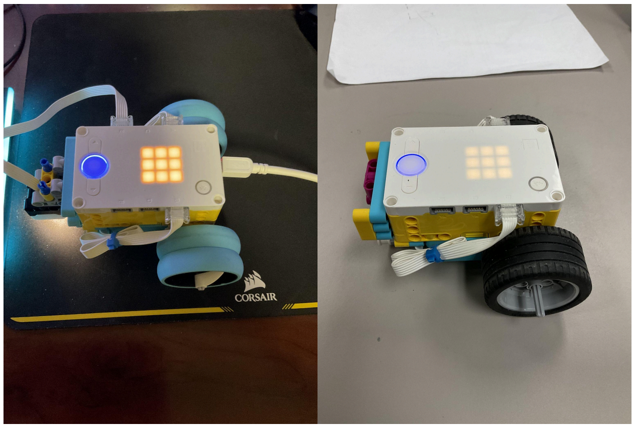
1) The design
    a) PICTURE OF ROBOT

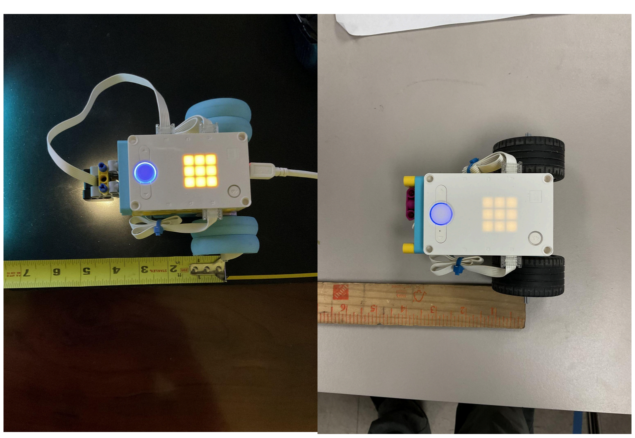


    b) Design choices, we have made the robot footprint as smallest as possible then when are running some test we found out that the robot issue with traction on the ground and we have add another wheel to the robot tried to fix the issued, help with the traction issue little, final robot design we decided to change out the wheel back and front to solve traction issue and we have remove the camera of the robot.

2) Navigation Strategy

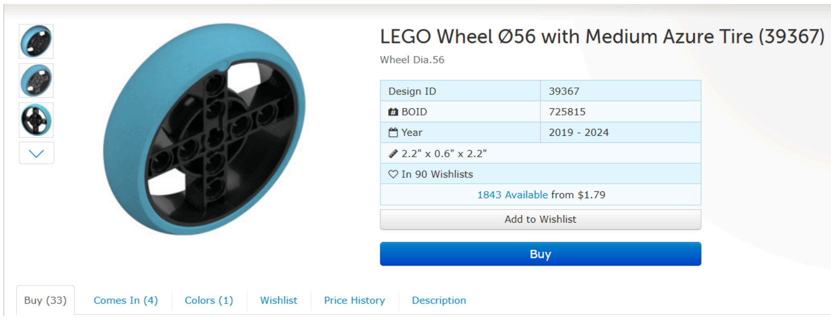
This robot's navigation strategy makes use of the A* algorithm, a well-liked pathfinding technique that finds the best path through a grid-based environment by combining heuristic estimations with travel costs. The f-score, which is the sum of the predicted distance to the objective (h-cost) and the path cost from the start node (g-cost), is determined for each node by the A* algorithm. Avoiding obstacles is essential, as each position is compared against predetermined obstacle coordinates. By calculating Euclidean distances, the check_if_obstacle function confirms that possible neighbouring points are clear. This is crucial for autonomous navigation.

3) Calibration Strategy
    a) ROBOT WITH RULER



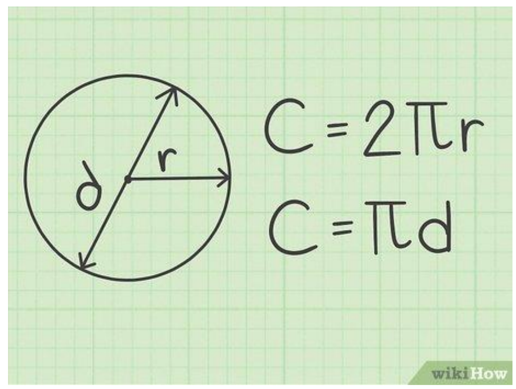
b) To obtain an accurate travel distance from the robot, we found the size of the wheel by finding the part number and checking online to find that it has a diameter of 2.2 inches

[LEGO Wheel Ø56 with Medium Azure Tire (39367) | Brick Owl - LEGO Marketplace]



Knowing the diameter means that we can easily find the circumference, which we can use to translate the distance traveled per wheel rotation. Knowing this plus also knowing the rotation of the wheel from the encoder embedded in the motor, we can find travel distance from the following equation:

$$TraveledDistance = Diameter * pi * (\frac{MotorAngle}{360})$$



Next step was to implement the Kp controller which was implemented as follows into the power parameter of the motor's run functions.

$$Kp * (TargetDistance - TraveledDistance)$$

For turning we implemented the Hub's Imu to measure our rotation. And used the following Kp Controller and inputted it into the motors run functions but opposite, so that they turn in different directions. Angle used was radians to give us a smoother operation instead of degrees

$$Kp * (TargetRotation - TraveledRotation)$$

Visualizer python Code

```
import matplotlib.pyplot as plt

# Define obstacle, start, and goal parameters
obstacle_positions = [
    (0.61, 2.743), (0.915, 2.743), (1.219, 2.743), (1.829, 1.219),
    (1.829, 1.524), (1.829, 1.829), (1.829, 2.134), (2.743, 0.305),
    (2.743, 0.61), (2.743, 0.915), (2.743, 2.743), (3.048, 2.743),
    (3.353, 2.743)
]
start_position = (0.305, 1.219)
goal_position = (3.658, 1.829)
obstacle_radius = 0.61
tile_size = 0.305
goal_tolerance = 0.01

# A* Pathfinding and Visualization Functions
def calculate_distance(point1, point2):
    return math.sqrt((point1[0] - point2[0]) ** 2 + (point1[1] -
point2[1]) ** 2)

def check_if_obstacle(x, y):
    return any(calculate_distance((x, y), obs) < obstacle_radius for obs
in obstacle_positions)

def generate_neighbors(node):
    x, y = node
    step = tile_size
    possible_moves = [
        (x + step, y), (x - step, y),
        (x, y + step), (x, y - step)
    ]
    return [move for move in possible_moves if not
check_if_obstacle(move[0], move[1])]

def is_goal_reached(current, goal, tolerance):
    return calculate_distance(current, goal) < tolerance

def a_star_search(start, goal):
```

```
    if check_if_obstacle(start[0], start[1]) or check_if_obstacle(goal[0],
goal[1]):
        return None
    to_explore = [(start, calculate_distance(start, goal))]
    path_tracking, cost_to_reach = {}, {start: 0}
    while to_explore:
        to_explore.sort(key=lambda x: x[1])
        current_node, _ = to_explore.pop(0)
        if is_goal_reached(current_node, goal, goal_tolerance):
            path = []
            while current_node in path_tracking:
                path.append(current_node)
                current_node = path_tracking[current_node]
            path.reverse()
            return path
        for neighbor in generate_neighbors(current_node):
            new_cost = cost_to_reach[current_node] +
calculate_distance(current_node, neighbor)
            if neighbor not in cost_to_reach or new_cost <
cost_to_reach[neighbor]:
                path_tracking[neighbor] = current_node
                cost_to_reach[neighbor] = new_cost
                estimated_total = new_cost + calculate_distance(neighbor,
goal)
                to_explore.append((neighbor, estimated_total))
    return None

def plot_environment(path, obstacles, start, goal):
    fig, ax = plt.subplots()
    for obs in obstacles:
        ax.add_artist(plt.Circle(obs, obstacle_radius, color='black',
fill=False, linewidth=3))
    if path:
        x_path, y_path = zip(*path)
        ax.plot(x_path, y_path, color='red', linewidth=2, label='Path')
    ax.plot(start[0], start[1], 'go', markersize=10, label='Start')
    ax.plot(goal[0], goal[1], 'bo', markersize=10, label='Goal')

    ax.set_aspect('equal', adjustable='box')
```

```python
    x_min = min(min(x_path), start[0], goal[0]) - 0.5
    x_max = max(max(x_path), start[0], goal[0]) + 0.5
    y_min = min(min(y_path), start[1], goal[1]) - 0.5
    y_max = max(max(y_path), start[1], goal[1]) + 0.5

    plt.xlim(x_min, x_max)
    plt.ylim(y_min, y_max)


    # plt.xlim(0, 4.5)
    # plt.ylim(0, 3.5)

    ax.grid(True)
    for x in range(int(x_min / tile_size) - 1, int(x_max / tile_size) +
1):
        for y in range(int(y_min / tile_size) - 1, int(y_max / tile_size)
+ 1):
            if not check_if_obstacle(x * tile_size, y * tile_size):
                plt.plot(x * tile_size, y * tile_size, 'x', color='cyan',
markersize=5)
    plt.xlabel('X Position')
    plt.ylabel('Y Position')
    plt.legend()
    plt.show()

# Run A* and plot results
path_result = a_star_search(start_position, goal_position)
plot_environment(path_result, obstacle_positions, start_position,
goal_position)

# Execute robot movement commands along the path
if path_result:
    for i in range(1, len(path_result)):
        x1, y1 = path_result[i - 1]
        x2, y2 = path_result[i]
        travel_angle = math.atan2(y2 - y1, x2 - x1)
        travel_distance = calculate_distance((x1, y1), (x2, y2))
        TurnForAngle(travel_angle)          # Turn robot towards next
segment
```

```
        MoveStraightForDistance(travel_distance)  # Move robot to the next
position
else:
    print("No valid path could be found.")
```

a_star_robot python code

```
import umath
#from astar_methods import *           # Import additional A* methods, if
necessary
from CalibrationTest import *          # Import robot movement functions
#import time
#import matplotlib.pyplot as plt
from uerrno import EPERM

# Define obstacle, start, and goal parameters
scale=1
obstacle_positions = [
    (0.61, 2.743), (0.915, 2.743), (1.219, 2.743), (1.829, 1.219),
    (1.829, 1.524), (1.829, 1.829), (1.829, 2.134), (2.743, 0.305),
    (2.743, 0.61), (2.743, 0.915), (2.743, 2.743), (3.048, 2.743),
    (3.353, 2.743)
]
obstacle_positions = [
    (x , y ) for x, y in obstacle_positions
]

start_position = (0.305 , 1.219 )
goal_position = (3.658 , 1.829)
obstacle_radius = 0.61
tile_size = 0.305
goal_tolerance = 0.01
worldX=4.88
worldY=3.05
robotRadius=(4/12)*0.305 # Sets extra buffer around objects to avoid the
robot scraping the side of objects
subdivison=1 # How many times to split the tiles (technically increases
processing time and memory)
print("Staring program...")
```

```
# A* Pathfinding and Visualization Functions
def calculate_distance(point1, point2):
    return umath.sqrt((point1[0] - point2[0]) ** 2 + (point1[1] -
point2[1]) ** 2)

def check_if_obstacle_circle(x, y):
    return any(calculate_distance((x, y), obs) < tile_size for obs in
obstacle_positions)

# Assumes Square Obstacle
def check_if_obstacle(x, y):
    return any(
        abs(x - obs[0]) < (tile_size / 2 + robotRadius) and abs(y -
obs[1]) < (tile_size / 2+robotRadius)
        for obs in obstacle_positions
    )


def generate_neighbors(node):
    x, y = node
    step = tile_size / subdivison
    possible_moves = [
        (x + step, y), (x - step, y),
        (x, y + step), (x, y - step)
    ]
    return [move for move in possible_moves if not
check_if_obstacle(move[0], move[1])]

def is_goal_reached(current, goal, tolerance):
    return calculate_distance(current, goal) < tolerance

def a_star_search(start, goal):
    i=0
    if check_if_obstacle(start[0], start[1]) or check_if_obstacle(goal[0],
goal[1]):
        return None
    to_explore = [(start, calculate_distance(start, goal))]
    path_tracking, cost_to_reach = {}, {start: 0}
    while to_explore:
```

```
        print(f"{i}")
        i=i+1
        to_explore.sort(key=lambda x: x[1])
        current_node, _ = to_explore.pop(0)
        if is_goal_reached(current_node, goal, goal_tolerance):
            path = []
            while current_node in path_tracking:
                path.append(current_node)
                current_node = path_tracking[current_node]
            path.reverse()
            return path
        for neighbor in generate_neighbors(current_node):
            new_cost = cost_to_reach[current_node] +
calculate_distance(current_node, neighbor)
            if neighbor not in cost_to_reach or new_cost <
cost_to_reach[neighbor]:
                path_tracking[neighbor] = current_node
                cost_to_reach[neighbor] = new_cost
                estimated_total = new_cost + calculate_distance(neighbor,
goal)
                to_explore.append((neighbor, estimated_total))

    return None

# # Run A* and plot results
path_result = a_star_search(start_position, goal_position)
print(path_result)

# plot_environment(path_result, obstacle_positions, start_position,
goal_position,subdivisions=subdivison)

#Execute robot movement commands along the path

if path_result:
    print(f"Entering Travel of {len(path_result)} step")
    current_travel_angle=0
    for i in range(1, len(path_result)):

        x1, y1 = path_result[i - 1]
        x2, y2 = path_result[i]
```

```
        travel_angle = umath.atan2(y2 - y1, x2 - x1)
        travel_distance = calculate_distance((x1, y1), (x2, y2))
        print(f"{i}: Angle: {travel_angle*(180/umath.pi)} Distance:
{travel_distance}")
        if abs(current_travel_angle - travel_angle)>(umath.pi/180):
            LeftMotor.brake()
            RightMotor.brake()

TurnForAngle(travel_angle-current_travel_angle,threshold=1*(umath.pi/180),
Kp=250)           # Turn robot towards next segment
            current_travel_angle=travel_angle

        LeftMotor.brake()
        RightMotor.brake()
        wait(100)
        MoveStraightForDistance(travel_distance*0.7,threshold=0.1,Kp=1)   #
Move robot to the next position
        LeftMotor.brake()
        RightMotor.brake()
        wait(100)
        # if i==5:
        #     break
else:
    print("No valid path could be found.")

print("DONEZO")
notes=["D4/16", "D4/16", "D5/8",
"A4/6","Ab4/8","G4/8","F4/8","D4/16","F4/16","G4/16"]
hub.speaker.play_notes(notes,tempo=120)
TurnForAngle(6*umath.pi,threshold=1*(umath.pi/180),Kp=300)
```

Calibration test python code

```
from pybricks.hubs import PrimeHub
from pybricks.pupdevices import Motor, ColorSensor, UltrasonicSensor,
ForceSensor
from pybricks.parameters import Button, Color, Direction, Port, Side, Stop
from pybricks.robotics import DriveBase
from pybricks.tools import wait, StopWatch
```

```python
import umath

hub = PrimeHub()
LeftMotor=Motor(Port.A,Direction.CLOCKWISE)
RightMotor=Motor(Port.B,Direction.COUNTERCLOCKWISE)
# LeftMotor.stop()
# RightMotor.stop()
# LeftMotor.control.limits(actuation=25)
# RightMotor.control.limits(actuation=25)

def MoveStraightForAngleLegacy(desired_angle):
    LeftMotor.reset_angle(0)
    RightMotor.reset_angle(0)
    while desired_angle > LeftMotor.angle() and desired_angle >
RightMotor.angle():
        LeftMotor.run(100)
        RightMotor.run(100)

    LeftMotor.stop()
    RightMotor.stop()

def MoveStraightForDistance(desired_distance,threshold=(1/100),Kp=2):
#Desired Distance in Meters
    Wheel_Radius_in=2.2/2*0.0254
    pi=umath.pi
    desired_angle=(desired_distance/(2*pi*Wheel_Radius_in))*360

    LeftMotor.reset_angle(0)
    RightMotor.reset_angle(0)

#print(abs(desired_distance-Wheel_Radius_in*2*umath.pi*(LeftMotor.angle()/
360)))

#print(abs(desired_distance-Wheel_Radius_in*2*umath.pi*(RightMotor.angle()
/360)))
    while
(abs(desired_distance-Wheel_Radius_in*2*umath.pi*(LeftMotor.angle()/360))
> threshold) and
(abs(desired_distance-Wheel_Radius_in*2*umath.pi*(RightMotor.angle()/360))
> threshold):
```

```
        LeftMotor.run(Kp*(desired_angle-LeftMotor.angle()))
        RightMotor.run(Kp*(desired_angle-RightMotor.angle()))
    #print("Finished Move")
    LeftMotor.stop()
    RightMotor.stop()

def TurnForAngleLegacy(desired_angle):
    wheel_spacing=3.125
    LeftMotor.reset_angle(0)
    RightMotor.reset_angle(0)
    resultant_distance=wheel_spacing*umath.pi*(desired_angle/360)
    ANGLE=(resultant_distance/(2*pi*Wheel_Radius_in))*360

    while ANGLE > LeftMotor.angle() and ANGLE > RightMotor.angle():
        LeftMotor.run(100)
        RightMotor.run(-100)

    LeftMotor.stop()
    RightMotor.stop()
    LeftMotor.brake()
    RightMotor.brake()

def TurnForAngle(desired_angle,threshold=1*(umath.pi/180),Kp=3):
    desired_angle=-desired_angle
    if hub.imu.ready():
        wait(100)
        hub.imu.reset_heading(0)
        wait(100)
        while (abs(desired_angle-hub.imu.heading()*umath.pi/180) >
threshold) and (abs(desired_angle-hub.imu.heading()*umath.pi/180) >
threshold):

LeftMotor.run(-Kp*(desired_angle-hub.imu.heading()*umath.pi/180))

RightMotor.run(Kp*(desired_angle-hub.imu.heading()*umath.pi/180))
            print(desired_angle-hub.imu.heading()*umath.pi/180)
        LeftMotor.stop()
        RightMotor.stop()
        LeftMotor.brake()
```

```
        RightMotor.brake()
    else:
        print("Hub Not Ready")
```