Aadiv Shah
WPI ID 413770228

Assignment 5

RBE 550
Motion Planning

Wildfire

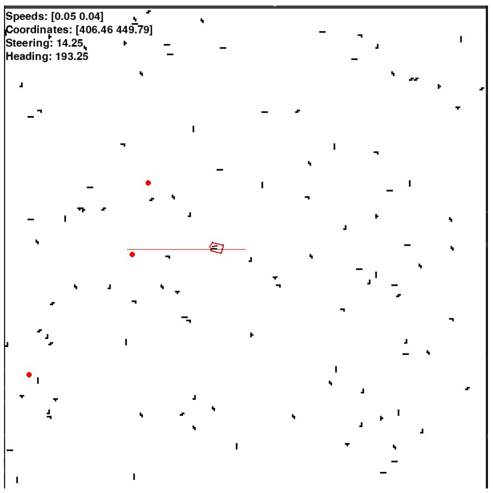
Introduction

Implement a planner that utilizes two forms of motion planning algorithms to navigate a firetruck through a cluttered, maze-like environment. Test two types of motion planning algorithms, first a combinatorial search algorithm, specifically A* or a variant, and then a Probabilistic RoadMap planner.

Implementation

Environment Generation

Using **Pygame**, I simulated the continuous space of the environment with all obstacles as shown in Fig. 1.



```
Speeds: [0.05 0.04]
Coordinates: [406.46 449.79]
Steering: 14.25
Heading: 193.25
```

Figure 1: Continuous space with the robot randomly spawned and fires starting to break out.

Robot Framework

Keeping in mind modularity, I developed several classes keeping in mind the properties of the robot e.g *Robot* class, *Controller* class and *Planner* class. I also developed a *World* and *Visualiser* class to define the world and display the environment + robot on pygame. Finally all the classes and their respective methods are called sequentially in the main function.

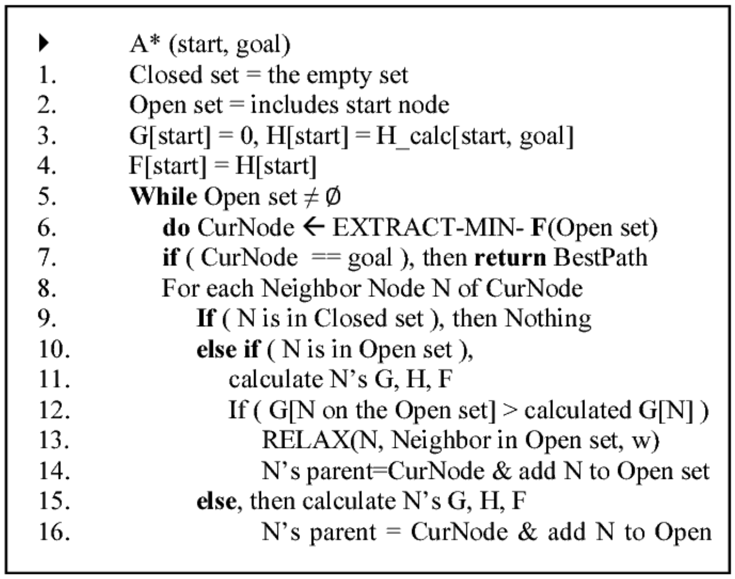
Planning Algorithms

A* Algorithm

The A* planning algorithms was inspired by Sakai, Atushi, et al. [1]. The robots may move to any $8-connected$ neighbour. Obstacle avoidance is achieved by inflating the obstacles. *Collision checking* is performed by observing if any of the robot lines are intersecting with the lines of the obstacle. The code to perform line intersection is inspired from intersects [2].

Separate class files were created to keep the code organized. Inside a while loop, the controller is *stepped* and the position of the robots is updated. Finally the current state of the world and robots is updated and displayed.

A* Search algorithm. A* is an informed search algorithm, or a best-first search, meaning that it is formulated in terms of weighted graphs: starting from a specific starting node of a graph, it aims to find a path to the given goal node having the smallest cost (least distance travelled, shortest time, etc.). It does this by maintaining a tree of paths originating at the start node and extending those paths one edge at a time until its termination criterion is satisfied.



```
▸     A* (start, goal)
1.    Closed set = the empty set
2.    Open set = includes start node
3.    G[start] = 0, H[start] = H_calc[start, goal]
4.    F[start] = H[start]
5.    While Open set ≠ Ø
6.        do CurNode ← EXTRACT-MIN- F(Open set)
7.        if ( CurNode  == goal ), then return BestPath
8.        For each Neighbor Node N of CurNode
9.            If ( N is in Closed set ), then Nothing
10.           else if ( N is in Open set ),
11.               calculate N's G, H, F
12.               If ( G[N on the Open set] > calculated G[N] )
13.                   RELAX(N, Neighbor in Open set, w)
14.                   N's parent=CurNode & add N to Open set
15.           else, then calculate N's G, H, F
16.                   N's parent = CurNode & add N to Open
```

Figure 1. A* Algorithm pseudo code

Probabilistic Roadmap (PRM)

The basic idea behind PRM is to take random samples from the configuration space of the robot, testing them for whether they are in the free space, and use a local planner to attempt to connect these configurations to other nearby configurations. The starting and goal configurations are added in, and a graph search algorithm is applied to the resulting graph to determine a path between the starting and goal configurations.

This algorithm is very useful but, it does not give the optimal solution every time. PRM inspired from [1].

After the planner returns the trajectory, I fed these waypoints to the robot controller and this moved the robot to the specific waypoints.

Results

The results for the ratios are shown in the figures below -

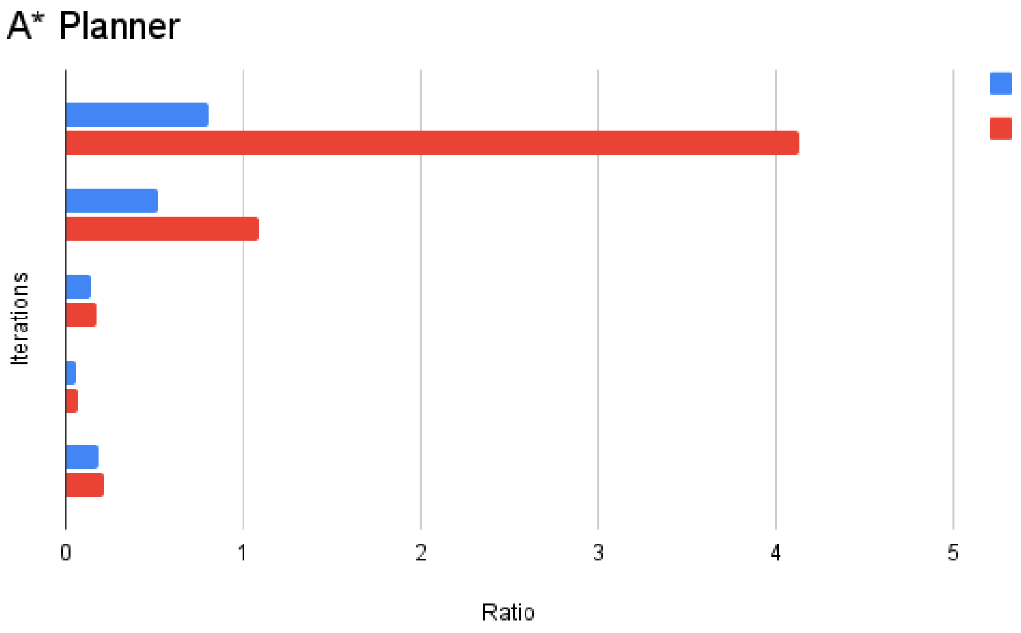


Figure 2: In blue: ratio of intact to total, in red: ratio of extinguished to burned

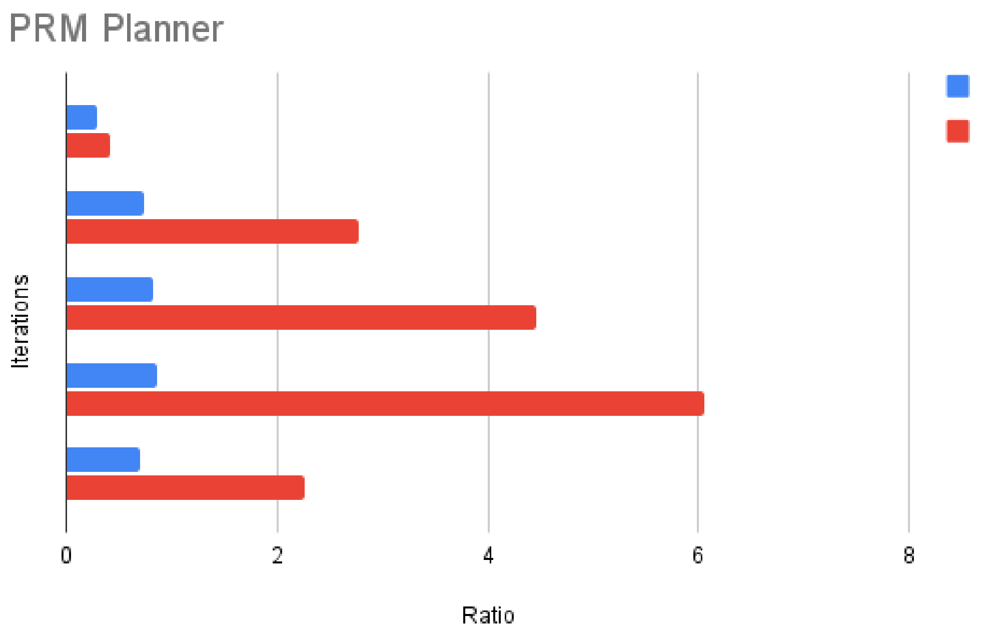


Figure 3: In blue: ratio of intact to total, in red: ratio of extinguished to burned

Finally, comparing the average CPU times and graphing them -

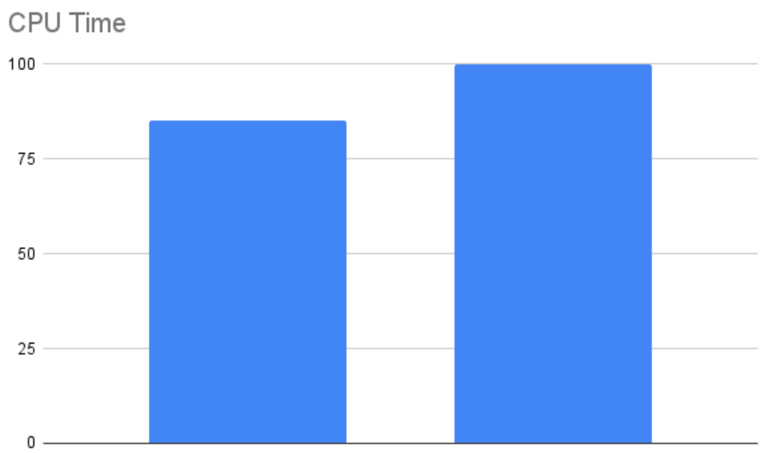


Figure 4: Average CPU times: A* (left) and PRM (right)

Discussion

From the results we see that the PRM performs better in extinguishing the fires as it has a higher ratio of extinguished to burned compared to the A*. Although the PRM does not necessarily plan the most efficient path, it still performs pretty well in extinguishing the fires. A* always gave the most optimal path. In a static world, computing the roadmap apriori is better as there is lesser strain on the CPU to perform realtime operations. The PRM method took up more resources in time and computational power (I could hear my laptop fan working overtime!).

References

[1] Sakai, Atsushi, et al. "Pythonrobotics: a python code collection of robotics algorithms." arXiv preprint arXiv:1808.10703 (2018).

[2] How to check if two given line segments intersect?

Appendix

Source code -

```
from pickletools import uint8
import time
import threading
from typing import List, Tuple, Union
from copy import copy
from cv2 import CV_16UC1, CV_8UC1
from matplotlib import pyplot as plt
import numpy as np
import pygame
from math import floor, sin, cos, tan, atan, atan2, pi, sqrt,
    hypot
import cv2
from prm import prm_planning
from astar import AStarPlanner

WIDTH = 300*3
HEIGHT = 300*3

def euclidian(p1, p2):
    return ((p1[0]-p2[0])**2+(p1[1]-p2[1])**2)**0.5
```

```
def rot_points(mat, degrees: float):
    degrees = degrees * pi/180
    rot_mat = np.array([[cos(degrees), sin(degrees)],[-sin(
       degrees), cos(degrees)]])

    return mat @ rot_mat

def convert_to_display(mat):
    mat[:,1] = HEIGHT - mat[:,1]
    return mat


class Robot:
    x_coord: int
    y_coord: int
    vel_r: int
    vel_l: int
    width: int = 20
    height: int = 15
    wheel: int = 5
    angle = 0
    points = []
    dt = 0


    def __init__(self, x, y, theta) -> None:
        self.x_coord = x
        self.y_coord = y
        self.angle = theta
        self.phi = 0
        self.vel_l = 0
        self.vel_r = 0
        self.max_turn = 85
        self.rad = self.width / tan(self.max_turn * pi/180)


    def get_robot_points(self):
        points = []

        points.append([self.width+5,0])
        points.append([self.width, self.height/2])
        points.append([0, self.height/2])
        points.append([0, -self.height/2])
        points.append([self.width, -self.height/2])


        # right side
        points.append([0, self.height/2])
        points.append([self.width, self.height/2])
        # left wheel
        points.append([0, -self.height/2])
        points.append([self.width, -self.height/2])

        # front wheels
        points.append(rot_points([-self.wheel,0], -self.phi) +
           np.array([self.width-self.wheel,self.height/2-self.
           wheel]))
        points.append(rot_points([self.wheel,0], -self.phi) +
           np.array([self.width-self.wheel,self.height/2-self.
           wheel]))
        points.append(rot_points([-self.wheel,0], -self.phi) +
           np.array([self.width-self.wheel,-self.height/2+self.
           wheel]))
        points.append(rot_points([self.wheel,0], -self.phi) +
           np.array([self.width-self.wheel,-self.height/2+self.
           wheel]))

        # normal line
        points.append(rot_points([0,141], -self.phi) + np.array
           ([self.width,0]))
        points.append(np.array([self.width,0]))

        points.append([0, self.rad])
        points.append([0, -self.rad])


        points = rot_points(points, self.angle) + np.array([
           self.x_coord, self.y_coord])
        # points = convert_to_display(points)

        return points

    def move(self, speed: float, phi: float = 0) -> None:
        self.phi = phi

        p = abs(self.phi * pi/180)
        w = self.height
        h = self.width
        vel_l = speed
        vel_r = vel_l * ((2*h - w*tan(p))/(2*h + w*tan(p)))

        if phi > 0:
            self.vel_r = vel_r
            self.vel_l = vel_l
        else:
            self.vel_l = vel_r
            self.vel_r = vel_l



    def turn(self, speed:float, gain: float, diff_heading:
       float) -> None:
        angle_thresh = self.max_turn
        # +ve is RIGHT
        # -ve is LEFT
        if diff_heading > 0:
            if diff_heading > 180:
                # turn right
                if gain*diff_heading > angle_thresh: self.move(
                   speed, angle_thresh)
                else: self.move(speed, gain*diff_heading)
            else:
                # turn left
                if gain*diff_heading > angle_thresh: self.move(
                   speed, -angle_thresh)
                else: self.move(speed, -gain*diff_heading)
        else:
            if diff_heading > -180:
                # turn right
                if gain*diff_heading < -angle_thresh: self.move
                   (speed, angle_thresh)
                else: self.move(speed, -gain*diff_heading)
            else:
                # turn left
                if gain*diff_heading < -angle_thresh: self.move
```

```
                    (speed, -angle_thresh)
                else: self.move(speed, gain*diff_heading)

    def get_position(self) -> Tuple[float, float]:
        return self.x_coord, self.y_coord

    def get_speed(self) -> Tuple[float, float]:
        return self.vel_l, self.vel_r

    def set_position(self, pos: Tuple[float, float]) -> None:
        self.x_coord = pos[0]
        self.y_coord = pos[1]



class Controller:

    def __init__(self, robot: Robot) -> None:
        self.robot = robot

    def step(self):
        # theta = pi - self.robot.angle * pi/180
        theta = self.robot.angle * pi/180
        self.robot.x_coord += ((self.robot.vel_l+self.robot.
           vel_r)/2)*cos(theta) * self.robot.dt
        self.robot.y_coord += ((self.robot.vel_l+self.robot.
           vel_r)/2)*sin(theta) * self.robot.dt
        self.robot.angle += atan2((self.robot.vel_r - self.
           robot.vel_l),self.robot.height)*180/pi * self.robot.
           dt

        self.robot.angle = self.robot.angle % 359

    def check_success(self, goal: Tuple[float, float]) -> bool:
        return np.allclose(self.robot.get_position(), goal,
           atol=8.5)


class World:

    shape = np.array([[[0,0], [1,0], [2,0], [2,1]],
            [[0,0], [1,0], [2,0], [3,0]],
            [[0,0], [0,1], [0,2], [1,1]],
            [[0,0], [0,1], [1,1], [1,2]]],dtype=object)

    visited = []

    def __init__(self, robot: Robot, width: int, height: int)
       -> None:
        self.width = width
        self.total = 0
        self.height = height
        self.robot = robot
        self.traj = []
        self.obstacle = []
        self.env = np.zeros((300,300),dtype=int)
        self.surface = np.zeros((self.width,self.height,3),
           dtype=int)

    def set_surface(self):
        # print(type(self.env[0,0]))
        # self.surface[:,:,0] = np.array(cv2.resize(self.env
           *255, (self.width,self.height), interpolation=cv2.
           INTER_AREA),dtype=int)
        self.surface[:,:,0] = np.array(cv2.resize(self.env*255,
            (self.width,self.height), interpolation=cv2.
           INTER_AREA),dtype=int)
        self.surface[:,:,1] = self.surface[:,:,0]
        self.surface[:,:,2] = self.surface[:,:,0]
        self.surface = 255 - self.surface

    def generate_random_tetromino(self) -> None:
        possible_starts = np.argwhere(self.env == 0)
        start = possible_starts[np.random.randint(0,len(
           possible_starts))]

        choose_shape = np.random.randint(0,4)
        # choose_shape = 2
        choose_orientation = np.random.randint(0,2)*(pi/2)
        # choose_orientation = 2 * pi/2

        rot_matrix = np.array([[cos(choose_orientation), sin(
           choose_orientation)],
                                [-sin(choose_orientation), cos(
                                            choose_orientation)]])


        new_obs = np.array(self.shape[choose_shape] @
           rot_matrix + start, dtype=int)

        original_env = copy(self.env)

        greater = new_obs > 299
        less = new_obs < 0

        if greater.sum()>0 or less.sum()>0:
            # print("Out of bounds")
            flag = 0
            self.env = original_env
        else:
            for obs in new_obs:
                flag = 1
                if self.env[obs[0], obs[1]] == 1 or (obs[1]==0
                   and obs[0]==0) or (obs[1]==299 and obs
                   [0]==299):
                    self.env = original_env
                    flag = 0
                    break
                else:
                    self.env[obs[0], obs[1]] = 1

        if flag:
            self.set_surface()

class Planner:
    def __init__(self, robot: Robot, world: World) -> None:
        self.robot = robot
        self.world = world

    def get_trajectory(self):
        pos = self.robot.get_position()
        # print(self.world.obstacle[:][0])
        obs = [row[0] for row in self.world.obstacle]
        nodes = np.asarray(obs)
        dist_2 = np.sum((nodes - pos)**2, axis=1)
        return np.argmin(dist_2)
    def get_heading(self, dx, dy) -> float:
        heading = atan2(dy,dx) * 180/pi
        # print("head: ", heading)
        if heading < 0:
            return 360 + heading
        else:
            return heading

class Visualizer:
    BLACK: Tuple[int, int, int] = (0, 0, 0)
    RED: Tuple[int, int, int] = (255, 0, 0)
    GREEN: Tuple[int, int, int] = (0, 255, 0)
    INV_RED: Tuple[int, int, int] = (0, 255, 255)
    WHITE: Tuple[int, int, int] = (255, 255, 255)
    BLUE: Tuple[int, int, int] = (0, 0, 255)

    def __init__(self, robot: Robot, controller: Controller,
       world: World) -> None:
        pygame.init()
        pygame.font.init()
        self.world = world
        self.robot = robot
        self.controller = controller
        self.screen = pygame.display.set_mode((world.width,
           world.height))
        pygame.display.set_caption('Tetromino Challenge')
        self.font = pygame.font.SysFont('freesansbolf.tff', 30)
        self.robot_path = []

    def display_robot(self):
        robot_points = self.robot.get_robot_points()
        pygame.draw.circle(self.screen, self.BLACK, (self.robot
           .x_coord, HEIGHT-self.robot.y_coord),1)
        # pygame.draw.circle(self.screen, self.BLACK, (self.
           robot.x_coord, HEIGHT-self.robot.y_coord),40,2)
        for i in range(5):
            if i == 4:
                pygame.draw.line(self.screen, self.RED,
                   robot_points[i], robot_points[0], 2)
            else:
                pygame.draw.line(self.screen, self.RED,
                robot_points[i], robot_points[i+1], 2)

        # left wheel
        pygame.draw.circle(self.screen, self.BLUE, robot_points
           [5], 1)
        pygame.draw.circle(self.screen, self.BLUE, robot_points
           [6], 1)
        # right wheel
        pygame.draw.circle(self.screen, self.BLACK,
           robot_points[7], 1)
        pygame.draw.circle(self.screen, self.BLACK,
           robot_points[8], 1)

        # front wheel
        pygame.draw.line(self.screen, self.BLACK, robot_points
           [9], robot_points[10], 2)
        pygame.draw.line(self.screen, self.BLACK, robot_points
           [11], robot_points[12], 2)

        # normal line
        # pygame.draw.line(self.screen, self.BLUE, robot_points
           [11], robot_points[12], 2)

        self.robot_path.append([self.robot.x_coord, self.robot.
           y_coord])
        robot_path = convert_to_display(np.array(self.
           robot_path))

        # for r in robot_path:
        #     pygame.draw.circle(self.screen, self.BLACK, r,
           0.3)

        p1 = robot_points[-2]
        p2 = robot_points[-1]

        # pygame.draw.line(self.screen, self.GREEN, p1, (self.
           robot.x_coord, 900-self.robot.y_coord), 4)
        # pygame.draw.line(self.screen, self.GREEN, p2, (self.
           robot.x_coord, 900-self.robot.y_coord), 4)


        coor = 'Coordinates: ' + str(np.round(self.robot.
            get_position(),2))
        speeds = 'Speeds: ' + str(np.round(self.robot.get_speed
            (),2))
        angle = 'Steering: ' + str(np.round(self.robot.phi,2))
        self_angle = 'Heading: ' + str(np.round(self.robot.
            angle,2))
        text = self.font.render(coor, True, self.BLACK)
        self.screen.blit(text, (1, 30))
        text1 = self.font.render(speeds, True, self.BLACK)
        self.screen.blit(text1, (1, 5))
        text2 = self.font.render(angle, True, self.BLACK)
        self.screen.blit(text2, (1, 55))
        text3 = self.font.render(self_angle, True, self.BLACK)
        self.screen.blit(text3, (1, 80))


    def display_world(self):
        # print((255 - self.world.surface))
        surf_array = self.world.surface
        surf = pygame.pixelcopy.make_surface(surf_array)
        self.screen.blit(surf, (0,0))

        # center = self.world.convert_to_display(self.robot.
            x_coord, self.robot.y_coord)
        # pygame.draw.circle(self.screen, self.RED, center, 10,
             2)

        for i in range(len(self.world.traj)-1):
            pygame.draw.line(self.screen, self.RED, self.world.
                traj[i], self.world.traj[i+1], 1)

        for obs in self.world.obstacle:
            pygame.draw.circle(self.screen, self.RED, [obs
                [0][1], obs[0][0]], 10)


    def update_display(self) -> bool:

        self.display_world()

        self.display_robot()
        for event in pygame.event.get():
            # Keypress
            if event.type == pygame.KEYDOWN:
                if event.key == pygame.K_ESCAPE: # if escape is
                    pressed, quit the program
                    return False

        pygame.display.flip()

        return True

    def cleanup(self) -> None:
        pygame.quit()


class Runner:
    def __init__(self, robot: Robot, controller: Controller,
       world: World, planner: Planner, vis: Visualizer, coverage
       : int) -> None:
        self.robot = robot
        self.world = world
        self.vis = vis
        self.controller = controller
        self.planner = planner
        self.coverage = coverage

    def run(self):
        self.world.env = np.zeros((300,300))
        running = True
        generate_new_map = True
        counter = -1
        flag = True
        lasttime = pygame.time.get_ticks()
        idxs = []
        idx = -100
        time_tick = 1.5
        new_goal = True
        traj_counter = 0

        try:
            np.load("wild.npy")
            print("Use previous map")
            generate_new_map = True
        except:
            generate_new_map = True

        while running:

            self.controller.step()

            if generate_new_map:
                while self.world.env.sum()/(300*300) < self.
                    coverage:
                     self.world.generate_random_tetromino()
                     print("Generating environment, percent
                         covered: ", (self.world.env.sum()/(self.
                         coverage*300*300))*100, end='\r')
                # np.save("wild.npy", self.world.env)
                print("\n")

            if counter == -1:
                # obs = np.load("wild.npy")
                # self.world.env = obs
                obs = self.world.env
                obs_idx = np.argwhere(obs == 1)
                ox = np.array(obs_idx[:,0])
                oy = 299 - np.array(obs_idx[:,1])
                obs_map = np.flip(obs, axis=1)

                self.world.set_surface()

                thresh = self.world.surface[:,:,0]
                thresh = thresh.astype("uint8")
                output = cv2.connectedComponentsWithStats(255-
                    thresh, 4, cv2.CV_32S)
                centroids = list(output[3])
                # print(centroids[10])
                # print(len(obs_idx))
                generate_new_map = False

                self.world.total = len(centroids)

                # cv2.imshow("amu", thresh)
                # cv2.waitKey()
                # cv2.destroyAllWindows()

                planning = AStarPlanner(ox, oy, obs_map)

            # do trajectory here

            if len(self.world.obstacle) > 1 and new_goal ==
                True:
                # print("New goal: ",self.world.obstacle[self.
                   planner.get_trajectory()])
                goal_idx = self.planner.get_trajectory()
                goal = self.world.obstacle[goal_idx][0]
                # print(goal)
                rx, ry = planning.planning(self.robot.x_coord
                   /3, self.robot.y_coord/3, goal[1]/3-4, goal
                   [0]/3-4)
                # rx, ry = prm_planning(self.robot.x_coord/3,
                   self.robot.y_coord/3, goal[1]/3-4, goal
                   [0]/3-4, ox, oy, 2)
                x = HEIGHT/300
                self.world.traj = np.flip(np.vstack((rx*x,ry*x)
                   ).T, axis=0)
                new_goal = False
                traj = self.world.traj
                # traj = traj)
                traj_counter = 0

            # do moving stuff here

            elif len(self.world.obstacle) > 1:
                # print("INSIDE HERE")
                if traj_counter < len(traj):
                    # self.robot.set_position(traj[traj_counter
                       ])
                    success = self.controller.check_success(
                       traj[traj_counter])
                    goal_check = False
                    # print(type(self.world.obstacle))
                    # traj_counter += 1
                # print("Counter: ", counter)
                else:
                    success = True
                    goal_check = True
                    new_goal = True
                    self.world.obstacle.pop(goal_idx)


                if not goal_check:
                    dy = traj[traj_counter][1] - self.robot.
                       y_coord
                    dx = traj[traj_counter][0] - self.robot.
                       x_coord
                    # print(counter_loop)


                heading = np.round(self.planner.get_heading(dx,
                   dy),0)
                diff_heading = heading - self.robot.angle
                # print(diff_heading)

                if abs(diff_heading) > 0.5 and not goal_check:
                    # print(counter)
                    if traj_counter < len(traj)-1:
                        points = self.robot.get_robot_points()
                        p1 = points[-2]
                        p2 = points[-1]
                        p3 = [traj[traj_counter+1][0], HEIGHT-
                           traj[traj_counter+1][1]]
                        buffer = self.robot.rad+5
                        if (euclidian(p1, p3)<buffer or
                           euclidian(p2, p3)<buffer) and
                           traj_counter!=1:
                            # print("Rad")
                            traj_counter += 1
                            # print(counter)
                    self.robot.turn(0.05, 1, diff_heading)

                elif not goal_check:
                    # print("Move towards goal")
                    speed = 0.05
                    self.robot.move(speed, 0)

                if success and not goal_check:
                    # do stuff for new goal
```

```
                    # stop robot
                    # if not goal_check:
                    self.robot.move(0,0)
                    # print("WAYPOINT")
                    traj_counter += 1
                    gain = 2.5


            if floor(pygame.time.get_ticks()/1000) % time_tick
               == 0 and flag:
                # print("Time seconds: ", pygame.time.get_ticks
                   ())
                flag = False
                for i in range(3):
                    idx = np.random.randint(0,len(centroids)-1)
                    centroids.pop(idx)
                    # print(len(self.world.obstacle))
                    idxs.append(idx)
                    self.world.obstacle.append([centroids[idx],
                        floor(pygame.time.get_ticks()/1000)])

            elif floor(pygame.time.get_ticks()/1000) %
               time_tick != 0:
                flag = True


            # dt
            counter += 1
            self.robot.dt = (pygame.time.get_ticks() - lasttime
               )
            lasttime = pygame.time.get_ticks()

            running = self.vis.update_display()
            time.sleep(0.001)
def main():

    robot = Robot(330, 550, 0)

    controller = Controller(robot)

    world = World(robot, WIDTH, HEIGHT)
    planner = Planner(robot, world)
    vis = Visualizer(robot, controller, world)

    coverage = 0.5
    runner = Runner(robot, controller, world, planner, vis,
       coverage/100)



    try:
        runner.run()
    except AssertionError as e:
        print(f'ERROR: {e}, Aborting.')
    except KeyboardInterrupt:
        pass
    finally:
        vis.cleanup()
        print("Obstacles on fire (burning): ", len(world.
           obstacle))
        print("Obstacles in all (total): ", world.total)
        print("Obstacles not burning (intact): ", world.total-
           len(world.obstacle))

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