```
#   RBE550 - Motion Planning HMWK 4
#   Author: <NATHANAEL> <VICKERY>
#   Date: Nov. 2, 2021

#   ---------------------------------------------
#   Wildfire - Sampling Based Planner
#   Current Space Scale: 2x
#   Current Time Scale: 10x
#   Change Random.seed in line 64 for variations
#   If planner fails, hit "Q" to quit
#   ---------------------------------------------

import pygame
import math
import numpy as np
import random
from time import *


# Build Workspace Grid
grid_size = 18
width,height = 30,30 #15,15 #4, 4 #Pixel size of blocks
margin = 0 #1 #Pixel size of margin
screen_size = grid_size*(width+margin) + margin
count = 0
cpu_time = 0
curr_state = []
obst = []
timing = 2 #20s scaled down 10x
sim_time = 362 # 3600s of sim plus 20s setup scaled down 10x
velocity = 10 # [m/s] truck velocity
extinguish_radius = 1 #12 * 2 # 10 meter extinguish radius scaled 2x
BLUE = (0, 0, 255)      # history of paths explored

# Obstacle states:
# Intact = 1 or black
# Burned = 3 or red
# Extinguished = 4 or purple
class obstacle:
    def __init__(self, x, y, state):
        self.gridx = x
        self.gridy = y
        self.state = state
    def changestate(self, newstate):
        self.state = newstate


# ObstacleField() generates a maze filled with obstacles.
# It also creates a border and start/end points.
def ObstacleField(coverage):
```

```
    I = [[1],[1],[1],[1]] # tetraminoes shapes
    L = [[1,1],[0,1],[0,1]]
    S = [[1,0],[1,1],[0,1]]
    T = [[0,1],[1,1],[0,1]]
    obst_cells = round(coverage*(grid_size**2))
    obstacles = round(obst_cells/4)
    maze = [[0 for x in range(grid_size)] for y in range(grid_size)]
    for i in range(grid_size): # Create outer border
        for j in range(grid_size):
            maze[0][j] = 1
            maze[i][0] = 1
            maze[grid_size-1][j] = 1
            maze[i][grid_size-1] = 1
    random.seed(1234)    # randomize obstacle placement
    for n in range(0,obstacles):
        x = random.randrange(1,grid_size-2)
        y = random.randrange(1,grid_size-4)
        tetra = random.choice([I,L,S,T])
        for i in range(len(tetra[0])):
            for j in range(len(tetra)):
                maze[y+j][x+i] = tetra[j][i]
    for i in range(1,grid_size-1):
        for j in range(1,grid_size-1):
            if maze[i][j] == 1:
                obst.append(obstacle(i,j,1))
    return maze


# ScreenUpdate() is pretty straightforward.
# It updates pygame's screen.
def ScreenUpdate(screen, current = []):
    BLACK = (0, 0, 0)       # obstacles
    GREEN = (0, 128, 0)     # start
    RED = (255, 0, 0)       # end
    BLUE = (0, 0, 255)      # history of paths explored
    PURPLE = (165,31,219)  # Current Path
    if len(current) > 1:
        if current[-1][0] == endx & current[-1][1] == endy:
            for coordinates in range(len(current)-1):
                x_new = current[coordinates][0]
                y_new = current[coordinates][1]
                grid[x_new] [y_new] = 5
    column = range(grid_size)
    row = range(grid_size)
    for i in column:
        for j in row:
            if grid[j][i] == 1:
                pygame.draw.rect(screen, BLACK, [(margin+width)*i+margin,
(margin+height)*j+margin, width, height])
            if grid[j][i] == 2:
```

```
                pygame.draw.rect(screen, GREEN, [(margin+width)*i+margin,
(margin+height)*j+margin, width, height])
            if grid[j][i] == 3:
                pygame.draw.rect(screen, RED, [(margin+width)*i+margin,
(margin+height)*j+margin, width, height])
            if grid[j][i] == 4:
                pygame.draw.rect(screen, PURPLE, [(margin+width)*i+margin,
(margin+height)*j+margin, width, height])
            if grid[j][i] == 5:
                pygame.draw.rect(screen, BLUE, [(margin+width)*i+margin,
(margin+height)*j+margin, width, height])
    pygame.display.flip() # Update Screen


# Spreads Fire Wildly
def wildfire(fcount):
    for block in obst:
        if block.state == 3:
            x, y = block.gridx, block.gridy
            spread = [(x-1,y),(x+1,y),(x,y-1),(x,y+1)]
#,(x-1,y-1),(x+1,y+1),(x+1,y-1),(x-1,y+1)]
            for other in obst:
                for growth in spread:
                    if other.state == 1 and other.gridy == growth[1] and other.gridx
== growth[0]:
                            other.changestate(5)
    for blck in obst:
        if blck.state == 5:
            grid[blck.gridx][blck.gridy] = 3
            blck.changestate(3)
    test = fcount % 3
    if test == 0:
        find = True
        while find:
            fire = random.randrange(len(obst))
            if obst[fire].state == 1:
                grid[obst[fire].gridx][obst[fire].gridy] = 3
                obst[fire].changestate(3)
                print('Fire! : ', obst[fire].gridx, obst[fire].gridy)
                ScreenUpdate(screen)
                find = False
                fcount += 1
    else:
        fcount += 1
    return fcount


def metrics(cpu_time):
    total = len(obst)
    intact = 0
```

```
    burned = 0
    extinguished = 0
    for block in obst:
        if block.state == 1:
            intact += 1
        if block.state == 3 or block.state == 4:
            burned += 1
        if block.state == 4:
            extinguished += 1
    print(' Total: ', total, '\n Burned: ', burned, '\n Extinguished: ',
extinguished, '\n')
    print(' Survival Ratio: ', intact/total, '\n Extinguished Ratio: ',
extinguished/burned, '\n')
    print(' CPU Time: ', cpu_time, 'Seconds \n')

def obstcl_setup():
    ox, oy = [], []
    for items in obst:
        ox.append(items.gridx)
        oy.append(items.gridy)
    for i in range(grid_size): # Bottom Wall
        ox.append(i)
        oy.append(0)
    for i in range(grid_size): # Top Wall
        ox.append(i)
        oy.append(grid_size)
    for i in range(grid_size): # Left Wall
        ox.append(0)
        oy.append(i)
    for i in range(grid_size): # Right Wall
        ox.append(grid_size)
        oy.append(i)
    return ox, oy


def fire_seeker(curr, dead, PRM):
    fires = []
    skip = False
    for blck in obst:
        if blck.state == 3:
            for bam in dead:
                if blck.gridx == bam.gridx and blck.gridy == bam.gridy:
                    skip = True
            if skip == True:
                skip = False
                continue
            targetx, targety = blck.gridx, blck.gridy
            sides =
[(targetx-1,targety),(targetx+1,targety),(targetx,targety-1),(targetx,targety+1)]
            for neighbors in sides:
```

```
                if grid[neighbors[0]][neighbors[1]] == 0:
                    fires.append(blck)
    if len(fires) == 0:
        return None, dead
    seekx, seeky =  fires[0].gridx, fires[0].gridy
    dist = math.hypot(abs(fires[0].gridx - curr[0]), abs(fires[0].gridy - curr[1]))
    for fire in range(1,len(fires)):
        if math.hypot(abs(fires[fire].gridx - curr[0]), abs(fires[fire].gridy -
curr[1])) <= dist:
            seekx, seeky = fires[fire].gridx, fires[fire].gridy
            dist = math.hypot(fires[fire].gridx - curr[0], fires[fire].gridy -
curr[1])
            dead.append(fires[fire])
    print('Target Fire @: ', targetx, targety)
    goal_orient = 0 #round(math.atan2(curr[0]-targetx, curr[1]-targety), 2)
    free = []
    column = range(grid_size)
    row = range(grid_size)
    for i in column:
        for j in row:
            if grid[j][i] == 0:
                free.append([j,i])
    sides =
[(targetx-1,targety),(targetx+1,targety),(targetx,targety-1),(targetx,targety+1),(ta
rgetx-1,targety-1),(targetx+1,targety+1),(targetx+1,targety-1),(targetx-1,targety+1)
]
    for neighbors in sides:
        for elem in free:
            if elem[0] ==  neighbors[0] and elem[1] == neighbors[1]:
                test_x, test_y = neighbors[0], neighbors[1]
                print('Target: ', test_x, test_y)
                goal_x, goal_y = PRM[0][0], PRM[0][1]
                dist = math.hypot(abs(PRM[0][0] - test_x), abs(PRM[0][1] - test_y))
                for node in PRM:
                    if math.hypot(abs(node[0] - test_x), abs(node[1] - test_y)) <=
dist:
                        goal_x, goal_y = node[0], node[1]
                        dist = math.hypot(node[0] - test_x, node[1] - test_y)
                goal = [goal_x, goal_y, goal_orient]
                print('PRM Goal: ', goal)
                return goal, dead
    print('Target-Neighbor Goal Selection Error: Default to Current')
    return None, dead


def generate_start(PRM):
##    free = []
##    column = range(grid_size)
##    row = range(grid_size)
##    for i in column:
```

```
##        for j in row:
##            if grid[j][i] == 0:
##                free.append([j,i])
    start = random.choice(PRM)
    grid[start[1]][start[0]] = 2
    print('Start: ',[start[1], start[0], 0])
    return [start[1], start[0], 0]


# The Class "node" builds node relationships
# for each coordinate in the grid. It also
# retains the position, parent relation,
# and the distance of the node from the start
class node:
    def __init__(self, position:(), orientation:(), parent:()):
        self.position = position
        self.orientation = orientation % 360
        self.parent = parent
        self.length = 0
        self.time = 0
        self.g = 0
        self.h = 0
        self.f = 0

    def __eq__(self, other):
        return self.position == other.position

    def __lt__(self, other):
         return self.f < other.f

    def path_length(self, newlength):
        self.length = newlength
        self.time = self.length/velocity


# A* Search algorithm.
# It searches the maze (grid) to find
# the lowest cost path and updates
# the pygame screen as it searches.
def astar(start_state, goal_state):
    global count
    beginning = time()
    open_nodes = []
    closed_nodes = []
    startx, starty, start_theta = start_state[0], start_state[1], start_state[2]
    endx, endy = goal_state[0], goal_state[1]
    start = node((startx, starty), start_theta, None)
    end = node((endx, endy), 0, None)
    open_nodes.append(start)
    while len(open_nodes) > 0:
```

```
        count = count + 1
        open_nodes.sort()
        current = open_nodes.pop(0)
        closed_nodes.append(current)
        (x,y) = current.position
        theta = current.orientation
        if current.position == end.position:   # Found ending
            ending = time()
            path = []
            while current.position != start.position:
                path.append([current.position, current.orientation])
                current = current.parent
            timer = ending-beginning
            final = path[::-1]
            return final, timer
        neighbors = []
        if current.orientation % 180 == 0:
            backward = (x,y-1,theta)
            if grid[backward[0]][backward[1]] == 0:
                neighbors.append((x,y-1,theta))
                neighbors.append((x+1,y-1,theta+90))
                neighbors.append((x-1,y-1,theta-90))
            forward = (x,y+1,theta)
            if grid[forward[0]][forward[1]] == 0:
                neighbors.append((x,y+1,theta))
                neighbors.append((x-1,y+1,theta+90))
                neighbors.append((x+1,y+1,theta-90))
        else:
            backward = (x-1,y,theta)
            if grid[backward[0]][backward[1]] == 0:
                neighbors.append((x-1,y,theta))
                neighbors.append((x-1,y-1,theta+90))
                neighbors.append((x-1,y+1,theta-90))
            forward = (x+1,y,theta)
            if grid[forward[0]][forward[1]] == 0:
                neighbors.append((x+1,y,theta))
                neighbors.append((x+1,y-1,theta-90))
                neighbors.append((x+1,y+1,theta+90))
        for new in neighbors:
            if grid[new[0]][new[1]] != 0:
                continue
            neighbor = node((new[0],new[1]), new[2], current)
            if neighbor.orientation == current.orientation:
                neighbor.path_length(15) #15 meters to cross to next block
            else:
                neighbor.path_length(15*np.deg2rad(90)) # 23ish meters to cross to
next cornerblock
            if neighbor in closed_nodes:
                continue
            neighbor.g =
```

```
np.sqrt((neighbor.position[0]-start.position[0])**2+(neighbor.position[1]-start.posi
tion[1])**2+(neighbor.orientation-start.orientation)**2)
            neighbor.h =
np.sqrt((end.position[0]-neighbor.position[0])**2+(end.position[1]-neighbor.position
[1])**2+(end.orientation-neighbor.orientation)**2)
            neighbor.f = neighbor.g + neighbor.h
            if add2open(open_nodes, neighbor) == True:
                open_nodes.append(neighbor)
    ending = time()
    timer = round(ending-beginning)
    return [], timer

# A function that checks whether it should add the node
# to list of open nodes
def add2open(open_nodes, neighbor):
    for node in open_nodes:
        if neighbor == node and neighbor.f >= node.f:
            return False
    return True


def path_length(x, y):
    length = 0
    for elem in range(len(x)):
        if x[elem] == x[-1]:
            length += 0
            return length
        length += math.hypot(x[elem+1]-x[elem], y[elem+1]-y[elem])


def fire_fighter(next_fight, goal, curr_state):
    print('Goal State: ', goal)
    if curr_state[0] == goal[0] and curr_state[1] == goal[1]:
        for block in obst:
            if block.state == 3:
                targetx = block.gridx
                targety = block.gridy
                sides =
[(targetx-1,targety),(targetx+1,targety),(targetx,targety-1),(targetx,targety+1),(ta
rgetx-1,targety-1),(targetx+1,targety+1),(targetx+1,targety-1),(targetx-1,targety+1)
]
                for hits in sides:
                    if curr_state[0] == hits[0] and curr_state[1] == hits[1]:
                        grid[block.gridx][block.gridy] = 4
                        block.changestate(4)
                        return curr_state, [], 0
    path, planner_time = astar(curr_state, goal)
    print('Path Planning Complete')
    if not path:
        print('Error - Path Planner Failure')
```

```
        print('Waiting')
        pause = round((timing-planner_time)*1000)
        pygame.time.delay(pause)
        return curr_state, [], 0
    else:
        x = [curr_state[0]]
        y = [curr_state[1]]
        theta = [curr_state[2]]
        for i in range(len(path)):
            x.append(path[i][0][0])
            y.append(path[i][0][1])
            theta.append(path[i][1])
        PATH = [x,y]
        path_dist = path_length(x, y)
        real_path_time = path_dist / velocity
        sim_path_time = real_path_time / 10
        real_planner_time = planner_time
        sim_time = real_planner_time / 10
        execution_time = sim_time + sim_path_time
        if execution_time <= timing:
            if len(next_fight) > 0:
                for fires in next_fight:
                    for block in obst:
                        if fires.gridy == block.gridy and fires.gridx ==
block.gridx:
                            grid[block.gridx][block.gridy] = 4
                            block.changestate(4)
            next_fires = []
            for points in range(len(PATH[0])):
                    for block in obst:
                        if block.state == 3:
                            targetx = block.gridx
                            targety = block.gridy
                            sides =
[(targetx-1,targety),(targetx+1,targety),(targetx,targety-1),(targetx,targety+1),(ta
rgetx-1,targety-1),(targetx+1,targety+1),(targetx+1,targety-1),(targetx-1,targety+1)
]
                            for hits in sides:
                                if PATH[0][points] == hits[0] and PATH[1][points] ==
hits[1]:
                                    grid[block.gridx][block.gridy] = 4
                                    block.changestate(4)
            curr_state = [goal[0], goal[1], path[-1][1]]
        if execution_time > timing:
            for points in path:
                    for block in obst:
                        if math.hypot(block.gridx - points[0], block.gridy -
points[1]) <= extinguish_radius:
                            next_fight.append(block)
            return curr_state, next_fight, real_planner_time
```

```
        pathline = []
        for elem in range(len(x)):
            pathline.append(((margin+width)*y[elem]+round(width/2),
(margin+height)*x[elem]+round(width/2)))
        pygame.draw.lines(screen, BLUE, False, pathline)
        pygame.display.flip()
        ScreenUpdate(screen)
        pause = round((timing-execution_time)*1000)
        pygame.time.delay(pause)
        return curr_state, [], real_planner_time


def build_PRM():
    free = []
    column = range(grid_size)
    row = range(grid_size)
    for i in column:
        for j in row:
            if grid[j][i] == 0:
                free.append([j,i])
    n = 20 # Number of samples
    Nodes = []
    while len(Nodes) < n:
        q = random.choice(free)
        Nodes.append(q)
    print('PRM Samples: ', Nodes)
    return Nodes


# runSim() is the main code and calls all the other functions.
# It creates the maze (grid) and updates it as the DFS algorithm
# runs.
def runSim():
    coverage = 0.2 # 20% coverage
    global grid, screen, fire_count, ox, oy, curr_state, cpu_time
    fire_count = 0
    num = 0
    next_fight = []
    grid = ObstacleField(coverage)
    begin = time()
    PRM = build_PRM()
    stop = time()
    curr_state = generate_start(PRM)
    start = obstacle(curr_state[0], curr_state[1], 0)
    dead = [start]
    fight = 0
    ox, oy = obstcl_setup()
    cpu_time += stop-begin
    pygame.init()
    size = (screen_size, screen_size) # Set dimension of the screen [width, height]
```

```
    screen = pygame.display.set_mode(size)
    pygame.display.set_caption("PRM - Obstacle Map(Coverage: %s %%)"%(coverage*100))
    done = False # closes loop when exited
    clock = pygame.time.Clock() # Clock for screen updates
    arson = pygame.USEREVENT
    pygame.time.set_timer(arson, timing*1000) #SET TO 20 FOR SIM!
    while not done: # -------- Main Program Loop -----------
        for event in pygame.event.get(): # --- Main event loop
            if event.type == pygame.QUIT:
                done = True
            elif event.type == pygame.KEYUP:
                if event.key == pygame.K_q or event.key == ord('q'):
                    metrics(cpu_time)
                    print("Closing Pygame - Unfinished Sim")
                    done = True
            elif event.type == arson:
                fire_count = wildfire(fire_count)
                burned = 0
                extinguished = 0
                for block in obst:
                    if block.state == 3:
                        burned += 1
                    if block.state == 4:
                        extinguished += 1
                lost = burned+extinguished
                if lost == len(obst):
                    print("The firetruck failed... All is lost")
                    metrics(cpu_time)
                    done = True
                    continue
                goal, dead = fire_seeker(curr_state, dead, PRM)
                if goal != None:
                    curr_state, next_fight, newtime = fire_fighter(next_fight, goal,
curr_state)
                    cpu_time += newtime
                    print('Current State: ', curr_state)
            elif pygame.time.get_ticks()/1000 >= sim_time: #event.type == end:
                metrics(cpu_time)
                print("Simulation Complete - Closing Pygame")
                done = True
        WHITE = (255, 255, 255)
        screen.fill(WHITE)  # populate map with grid data
##        for coor in PRM:
##            grid[coor[0]][coor[1]] = 5
##        ScreenUpdate(screen)
##        pygame.image.save(screen, "PRM_MAP.png")
##        print("PRM Map saved as .png file")
##        for coor in PRM:
##            grid[coor[0]][coor[1]] = 0
##        done = True
```

```
        ScreenUpdate(screen)
        clock.tick(60)
    pygame.quit() # Close the window and quit.

# MAIN CODE for Wildfire PRM Path Planner
runSim()
```