```
#!/usr/bin/python3
#GA
from pysimbotlib.core import PySimbotApp, Simbot, Robot, Util
from kivy.logger import Logger
from kivy.config import Config
import copy
import random
import csv
import os
import matplotlib.pyplot as plt
from statistics import mean
maxFNV = []
meanFNV = []
# # Force the program to show user's log only for "info" level or more. The info
log will be disabled.
# Config.set('kivy', 'log_level', 'debug')
Config.set('graphics', 'maxfps', 10)

class StupidRobot(Robot):

    RULE_LENGTH = 11
    NUM_RULES = 10

    def __init__(self, **kwarg):
        super(StupidRobot, self).__init__(**kwarg)
        self.RULES = [[0] * self.RULE_LENGTH for _ in range(self.NUM_RULES)]

        # initial list of rules
        self.rules = [0.] * self.NUM_RULES
        self.turns = [0.] * self.NUM_RULES
        self.moves = [0.] * self.NUM_RULES

        self.fitness = 0

    def update(self):
        ''' Update method which will be called each frame
        '''
        self.ir_values = self.distance()
        self.S0, self.S1, self.S2, self.S3, self.S4, self.S5, self.S6, self.S7 =
self.ir_values
        self.target = self.smell()

        for i, RULE in enumerate(self.RULES):
            self.rules[i] = 1.0
            for k, RULE_VALUE in enumerate(RULE):
```

```
            if k < 8:
                if RULE_VALUE % 5 == 1:
                    if k == 0: self.rules[i] *= self.S0_near()
                    elif k == 1: self.rules[i] *= self.S1_near()
                    elif k == 2: self.rules[i] *= self.S2_near()
                    elif k == 3: self.rules[i] *= self.S3_near()
                    elif k == 4: self.rules[i] *= self.S4_near()
                    elif k == 5: self.rules[i] *= self.S5_near()
                    elif k == 6: self.rules[i] *= self.S6_near()
                    elif k == 7: self.rules[i] *= self.S7_near()
                elif RULE_VALUE % 5 == 2:
                    if k == 0: self.rules[i] *= self.S0_far()
                    elif k == 1: self.rules[i] *= self.S1_far()
                    elif k == 2: self.rules[i] *= self.S2_far()
                    elif k == 3: self.rules[i] *= self.S3_far()
                    elif k == 4: self.rules[i] *= self.S4_far()
                    elif k == 5: self.rules[i] *= self.S5_far()
                    elif k == 6: self.rules[i] *= self.S6_far()
                    elif k == 7: self.rules[i] *= self.S7_far()
            elif k == 8:
                temp_val = RULE_VALUE % 6
                if temp_val == 1: self.rules[i] *= self.smell_left()
                elif temp_val == 2: self.rules[i] *= self.smell_center()
                elif temp_val == 3: self.rules[i] *= self.smell_right()
            elif k==9: self.turns[i] = (RULE_VALUE % 181) - 90
            elif k==10: self.moves[i] = (RULE_VALUE % 21) - 10

        answerTurn = 0.0
        answerMove = 0.0
        for turn, move, rule in zip(self.turns, self.moves, self.rules):
            answerTurn += turn * rule
            answerMove += move * rule

        self.turn(answerTurn)
        self.move(answerMove)

    def S0_near(self):
        if self.S0 <= 0: return 1.0
        elif self.S0 >= 100: return 0.0
        else: return 1 - (self.S0 / 100.0)

    def S0_far(self):
        if self.S0 <= 0: return 0.0
        elif self.S0 >= 100: return 1.0
        else: return self.S0 / 100.0
```

```
    def S1_near(self):
        if self.S1 <= 0: return 1.0
        elif self.S1 >= 100: return 0.0
        else: return 1 - (self.S1 / 100.0)

    def S1_far(self):
        if self.S1 <= 0: return 0.0
        elif self.S1 >= 100: return 1.0
        else: return self.S1 / 100.0

    def S2_near(self):
        if self.S2 <= 0: return 1.0
        elif self.S2 >= 100: return 0.0
        else: return 1 - (self.S2 / 100.0)

    def S2_far(self):
        if self.S2 <= 0: return 0.0
        elif self.S2 >= 100: return 1.0
        else: return self.S2 / 100.0

    def S3_near(self):
        if self.S3 <= 0: return 1.0
        elif self.S3 >= 100: return 0.0
        else: return 1 - (self.S3 / 100.0)

    def S3_far(self):
        if self.S3 <= 0: return 0.0
        elif self.S3 >= 100: return 1.0
        else: return self.S3 / 100.0

    def S4_near(self):
        if self.S4 <= 0: return 1.0
        elif self.S4 >= 100: return 0.0
        else: return 1 - (self.S4 / 100.0)

    def S4_far(self):
        if self.S4 <= 0: return 0.0
        elif self.S4 >= 100: return 1.0
        else: return self.S4 / 100.0

    def S5_near(self):
        if self.S5 <= 0: return 1.0
        elif self.S5 >= 100: return 0.0
        else: return 1 - (self.S5 / 100.0)
```

```
    def S5_far(self):
        if self.S5 <= 0: return 0.0
        elif self.S5 >= 100: return 1.0
        else: return self.S5 / 100.0

    def S6_near(self):
        if self.S6 <= 0: return 1.0
        elif self.S6 >= 100: return 0.0
        else: return 1 - (self.S6 / 100.0)

    def S6_far(self):
        if self.S6 <= 0: return 0.0
        elif self.S6 >= 100: return 1.0
        else: return self.S6 / 100.0

    def S7_near(self):
        if self.S7 <= 0: return 1.0
        elif self.S7 >= 100: return 0.0
        else: return 1 - (self.S7 / 100.0)

    def S7_far(self):
        if self.S7 <= 0: return 0.0
        elif self.S7 >= 100: return 1.0
        else: return self.S7 / 100.0

    def smell_right(self):
        if self.target >= 45: return 1.0
        elif self.target <= 0: return 0.0
        else: return self.target / 45.0

    def smell_left(self):
        if self.target <= -45: return 1.0
        elif self.target >= 0: return 0.0
        else: return 1-(-1*self.target)/45.0

    def smell_center(self):
        if self.target <= 45 and self.target >= 0: return self.target / 45.0
        if self.target <= -45 and self.target <= 0: return 1-(-
1*self.target)/45.0
        else: return 0.0

def write_rule(robot, filename):
    with open(filename, "w") as f:
        writer = csv.writer(f, lineterminator="\n")
```

```
        writer.writerows(robot.RULES)

def read_rule():
    R = []
    oldRule = list(csv.reader(open(r"file name")))
    for k in range(len(oldRule)):
        STI = [eval(i) for i in oldRule[k]]
        R.append(STI)
    return R
# initializing next generation robot list
next_gen_robots = list()

def before_simulation(simbot: Simbot):
    for robot in simbot.robots:
        # random RULES value for the first generation
        if simbot.simulation_count == 0:
            Logger.info("GA: initial population")
            for i, RULE in enumerate(robot.RULES):
                for k in range(len(RULE)):
                    if k==0 :
                        #robot.RULES[i][k] = random.randrange(256)
                        robot.RULES = read_rule()[1]
                    else:
                        robot.RULES[i][k] = random.randrange(256)
            print(robot.RULES)
        # used the calculated RULES value from the previous generation
        else:
            Logger.info("GA: copy the rules from previous generation")
            for simbot_robot, robot_from_last_gen in zip(simbot.robots,
next_gen_robots):
                simbot_robot.RULES = robot_from_last_gen.RULES

def after_simulation(simbot: Simbot):
    Logger.info("GA: Start GA Process ...")

    # There are some simbot and robot calcalated statistics and property during
simulation
    # - simbot.score
    # - simbot.simulation_count
    # - simbot.eat_count
    # - simbot.food_move_count
    # - simbot.score
    # - simbot.scoreStr

    # - simbot.robot[0].eat_count
```

```
    # - simbot.robot[0].collision_count
    # - simbot.robot[0].color

    # evaluation - compute fitness values here
    FNV = []
    for robot in simbot.robots:
        food_pos = simbot.objectives[0].pos
        robot_pos = robot.pos
        distance = Util.distance(food_pos, robot_pos)
        robot.fitness = 1000 - int(distance)
        robot.fitness -= robot.collision_count
        FNV.append(robot.fitness)
        MFNV = max(FNV)
        AVGFNV = mean(FNV)
    # descending sort and rank: the best 10 will be on the list at index 0 to 9
    simbot.robots.sort(key=lambda robot: robot.fitness, reverse=True)

    # empty the list
    next_gen_robots.clear()

    # adding the best to the next generation.
    next_gen_robots.append(simbot.robots[0])

    num_robots = len(simbot.robots)

    def select():
        index = random.randrange(num_robots)
        return simbot.robots[index]

    # doing genetic operations
    for _ in range(num_robots):
        select1 = simbot.robots[0] # design the way for selection by yourself
        select2 = select()   # design the way for selection by yourself

        while select1 == select2:
            select2 = select()

        # Doing crossover
        #     using next_gen_robots for temporary keep the offsprings, later they
will be copy
        #     to the robots
        #next_gen_robots.append(select1)
        NGR = StupidRobot()
        CPR = 1
        CPR_len = 10
```

```
        Logger.info(f"First Robot : \n{select1.RULES}")
        Logger.info(f"Second Robot : \n{select2.RULES}")
        NGR.RULES[:CPR] = copy.deepcopy(select1.RULES[:CPR])
        NGR.RULES[CPR+1:] = copy.deepcopy(select2.RULES[CPR+1:])
        NGR.RULES[CPR][:CPR_len] = copy.deepcopy(select1.RULES[CPR][:CPR_len])
        NGR.RULES[CPR][CPR_len: 1] =
copy.deepcopy(select2.RULES[CPR][CPR_len+1:])
        Logger.info(f"New Gen Robot Rules : \n{NGR.RULES}")
        next_gen_robots.append(NGR)
        print('crossever')
        # Doing mutation
        #     generally scan for all next_gen_robots we have created, and with
very low
        #     propability, change one byte to a new random value.
        if random.randrange(0,99) > 95:
            print("mutation")
            i = random.randrange(0,9)
            j = random.randrange(0,10)
            NGR.RULES[i][j] = random.randrange(0,256)
            next_gen_robots.append(NGR)
        pass

    # write the best rule to file
    write_rule(simbot.robots[0],
"best_gen{0}.csv".format(simbot.simulation_count))
    maxFNV.append(MFNV)
    meanFNV.append(AVGFNV)
    gen = [x for x in range(len(maxFNV))]
    plt.plot(gen, maxFNV)
    plt.plot(gen, meanFNV)
    plt.title('Learning performance')
    plt.ylabel('fitness value')
    plt.xlabel('generation')
    plt.legend(['MaxFitnessValue', 'AVGFitnessValue'], loc='upper left')
    plt.show(block = False)
    plt.pause(3)
    # Make sure to close the plt object once done
    plt.close()
    '''if simbot.simulation_count == 100:
        exit()'''
if __name__ == '__main__':

    app = PySimbotApp(robot_cls=StupidRobot,
                      num_robots=30,
                      theme='default',
```

```
                    simulation_forever=True,
                    max_tick=500,
                    interval=1/100.0,
                    food_move_after_eat=False,
                    customfn_before_simulation=before_simulation,
                    customfn_after_simulation=after_simulation)
    app.run()
```