MISSION #10

Genetic Algorithms (SIMBot)

Due: Week 14

After you have an experience in designing a reactive control and fuzzy logic control for a very simple simulated robot on previous weeks. In this assignment, you will have to **apply the concept of Genetic Algorithms (GA) for tuning the rules of fuzzy robot programming**. The task and the robot are the same as the previous simulation robot assignment. The GA will be used to help you write the rules. In another words, ***it will be seemed like the computer writes the robot programs for you***. ☺

The appropriate steps in applying this GA system are listed as follows…

- You have to encode the problem and design the representation of the rules, according to the task,
  For example,
  - a rule consists of 9 input bytes and 2 output bytes
  - all bytes are generated randomly [0,255].
  - for input part, the first 8 bytes represent 8 sensory inputs (IR-0, … ,IR-7) and the last byte represents the smell input.
  - each byte will be modulated to indicate if it represents fuzzy set for each input sensor or not.
    For example,
    - if (the first byte % 3 == 1) means the rule will use IR_0_Near()
    - if (the first byte % 3 == 2) means the rule will use IR_0_Far()
    - if (the first byte % 3 == 0) means the rule won't use IR_0
  - for output part, the 2 bytes represent the values of TURN and MOVE.
  - the maximum and minimum values are defined by (Byte % 180 – 90), thus you get -90 to 90

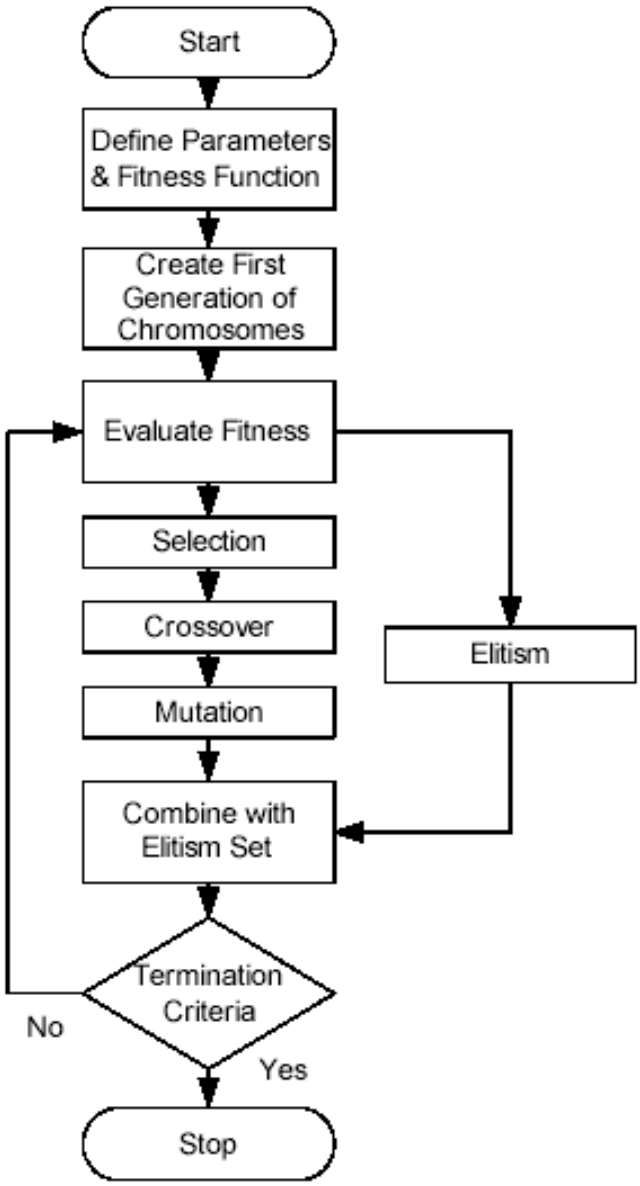
- Write a module to **write** the current rules into to a file (def write_rule(robot, filename):)
- Write a module to **read** the saved rules from a file (def read_rule(robot, filename):)
- Create a new variable for represent rules in the codes
- Modify the **def update(self):** function.
- Implement the process of GA (probably in **def after_simulation(simbot: Simbot):** , please use the guide line on the next page.

- Plot learning curves by recording in every generation with maximum fitness, averaged fitness.

- The learning curve is somewhat look like this graph ….

Genetic Algorithm Operation

The primary function of genetic algorithms is to evolve the feature weights of the evaluation function. Figure below provides an outline of the operation of the GA.



1) Problem Encoding: A binary encoding scheme was selected for chromosomes. Each chromosome consists of 48 bits, 8 bits for each weight – for example.

Fifty chromosomes are randomly created to construct the first population. The population size is constant throughout the run.

2) Fitness Evaluation: The fitness of each chromosome is evaluated using a special form of tournament selection (for example) which assesses the fitness of each individual relative to its peers. More specifically, twenty percent of the population (ten chromosomes) is chosen randomly to form a *fitness set*. Next, each chromosome in the population plays twice against every chromosome in the fitness set, once as north and once as south. The winning player is awarded 2 points for a win, 1 point for a draw and zero points for a loss. The sums of points awarded to a chromosome in those twenty games are equal to the fitness of that chromosome

3) Selection and Elitism: Copies of the best ten percent of the population are placed without changes in the *elitism set*. Elitism ensures that the best chromosomes will not be destroyed during crossover and mutation.

The selection process is then implemented. Fitness proportional selection (with replacement) is used to select chromosomes for the *mating pool*. The size of the mating pool equals ninety percent of the population size.

4) Crossover: Single-point crossover is used with probability of 0.5.

5) Mutation: Bit-flip mutation was used with probability of 0.001.

The chromosomes resulting from crossover and mutation are then combined with the elitism set to construct the next generation.

6) *Termination Criteria*: The GA may run for one hundred generations. Next, the weights extracted from the best chromosome in the last generation are used as the final solution.

```
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)):
                    robot.RULES[i][k] = random.randrange(256)
        # 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.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
    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

    # 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 - 1):
        select1 = select()   # 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)

        # 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.
        pass

    # write the best rule to file
    write_rule(simbot.robots[0], "best_gen{0}.csv".format(simbot.simulation_count))
```

```
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 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)
// Write def read_rule(robot, filename): by yourself
....use something like ...
      simbot.robot[0].RULES = list(csv.reader(open("best_1.csv")))
................................................
```

An Example of Generated Rule File

```
92,141,119,75,134,109,93,185,37,230,59
43,209,176,74,53,239,119,196,214,203,231
191,151,244,150,226,193,14,100,182,104,119
151,114,162,167,117,87,116,164,160,42,32
10,115,192,200,145,230,208,75,108,71,78
15,116,76,139,46,251,78,105,211,137,187
106,169,73,242,83,166,133,75,103,235,211
195,252,9,209,5,51,254,130,37,193,70
95,104,84,127,3,76,64,202,103,95,9
26,254,17,158,52,6,178,35,241,93,33
```

A Table Shows All Possible Values for Each Byte of A Rule

Byte#1	Byte#2	Byte#3	Byte#4	Byte#5	Byte#6	Byte#7	Byte#8	Byte#9	Byte#10	Byte#11
IR-0	IR-1	IR-2	IR-3	IR-4	IR-5	IR-6	IR-7	SMELL	TURN	MOVE
Near	Near	Near	Near	Near	Near	Near	Near	Left	-90	-10
Far	Far	Far	Far	Far	Far	Far	Far	Center	90	10
Don't	Don't	Don't	Don't	Don't	Don't	Don't	Don't	Right		
Don't	Don't	Don't	Don't	Don't	Don't	Don't	Don't	Don't		
Don't	Don't	Don't	Don't	Don't	Don't	Don't	Don't	Don't		
								Don't		

How to Encode a Problem

- **Each rule is encoded with 11 bytes.**
- **Each byte represents different information, with different fuzzy sets.**
- **The process of GA are used to create rules by randomly combine these possible choices for all bytes.**
- **For example, a rule maybe is**

  1,2,4,5,3,5,3,1,6,135,15

  which means

  If [IR-0(Near), IR-1(Far), IR-7(Near)]
  THEN TURN(45) and MOVE(5).