Dexter R Shepherd

Examining How the Mutation Rate Effects the Performance of a Genetic Algorithm

AI&AB G6042

11-03-2021

Abstract

This pap er describ es how mutation rate effects the performance and speed of a genetic algorithm (GA) converging on a solution. A Genetic algorithm is an alternativ e to conventional metho ds such as brute force and greedy algorithms. The genetic algorithm is applied to a rob otic chassis which learns to walk using the same genot yp e to start with and differen t mutation rates for each exp erimen t. It is found that 20% was the optimal mutation rate. When mutation rate is a high percen tage the algorithm makes more mistak es and when mutation rate is to o low the algorithm do es not mak e the needed changes in time to find an optimal solution.

1 Intro duction

Genetic algorithms (GA) allow systems to adapt randomly to find solutions. This minimizes the time that the developer needs to spend creat ing a solution . Some tasks are to o complicated or time consuming for a developer to code a solution. Tasks such as walking require balance and correct movement all in concurrence with one an other. Using a GA is a known approac h to satisfy- ing a solution while considering both constrain ts. An issue arises where these solutions have an enormous search space and can be impractical for solving a solution quickly [4]. Researc h on bip edal genetic algorithms typically tak es more generations to arriv e at a candidate solution the more motors there are. [2] In this pap er, we will explore how the mutation rate affects the performance of a genetic algorithm, carr y out researc h using a number of mutation rates, and conclude the optimal rate to arriv e at a solution quickly.

This is an interesting topic due to the significan t part physical rob ots will one day have in society. We think very litt le ab out the complex movemen ts we perform everyda y. Being able to master optimization for rob otic s will help get to this stage. In this experiment we will see how key to the performance of an algorithm mutation rate is.

1.1 Problem Statement

The aim of the experiment is to drive a chassis of 4 micro servos to optimize a method of travel. This is a simple low cost chassis which will allow more focus on the GA than on the hardware. Each servo can turn from 0 to 180 degrees. It is the job of the algorithm to mutate movement at a number of rates to find the optimum.

1.2 Previous work

A number of authors within genetic algorithms and robotics have been involving neural networks and deep learning [4] [3]. It was found that neural networks being applied with genetic algorithms was impractical due to the large search space. The experiment found the staged evolution method significantly improved the convergence to a solution.

We will be using a similar method of encoding motor patterns into a robot [4], however without the use of a neural network.

There has been self-organizing systems created in the past which use positive and negative feedback to optimize a robots walk [5]. The paper by Maes took a more constructivist approach than the GA selectionist approach. This method used less memory due to the way it built up information rather than selected out. This experiment tested the algorithm on a simpler physical model which used 6 legs. There was less focus on the need to balance compared to more humanoid chassis. It is the hypothesis of this experiment that GA will work as a general approach which will always give us a sub-optimal solution at minimum and that the mutation rate will decide how quickly the program converges.

2 Apparatus

The chassis chosen, named Bob, uses 4 Tower Pro SG90 micro servos [6]. Each servo has an operating voltage of 3 to 7.2 Volts and operating current of 220Ma. In a worst case scenario that all servos are running, the battery must provide a current of 880ma.

$$4 * 220 = 880ma \tag{1}$$

This can be provided with 4 AAA batteries which provide 1000ma, and in a series circuit configuration we have up to 6V.

For a controller we use the Adafruit Feather m0 [1] which is a CircuitPython running device with limited memory. This memory holds up to around 250 lines of code (including libraries). The controller will be wired to a servo controller and HC-SR04 ultrasonic range finder (distance sensor).

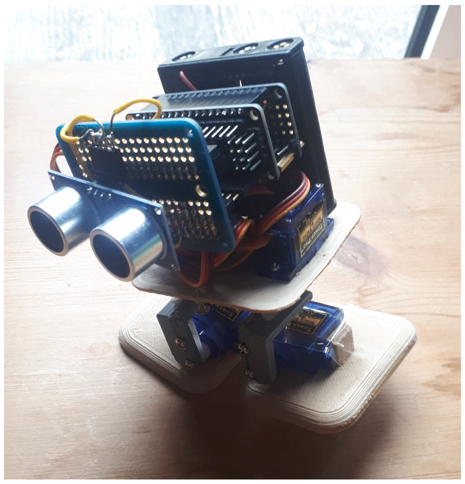

Figure 1: Bob the biped chassis

3 Application of GA

The genetic algorithm will encode a series of motor instructions written as angles. There will be a set number of steps that a robot can take to perform the action of taking a step. This will be presented as a 2-Dimensional array. This will be known as the genotype. At each iteration a mutation function will be applied to the genotype, the genotype will be performed and the fitness is calculated. This will performed a set number of times, which we call generations.

Each angle will be chosen randomly from an array containing integers consisting of many zeros, a 30 and -30. These represent the degree of movement from the current angle. We want the majority of motors to have no movement but give chance for movement in any direction. We chose 30 degrees as it gives enough rotation to make a difference, but not too much that it has a dramatic effect. This was a personal choice.

There are two algorithms that will be applied to the robot, a hill climber and a microbial GA. Diversity of algorithms will give more rigorous results.

Results will be displayed showing mutation rates 10 to 80 percent. We will explore each of the results and data surrounding it such as average fitness.

A method of changing the mutation rate to sample fitnesses will be used outside the algorithms.

Algorithm 1 Hill Climber

```
Require: $FLOAT\ mutationrate$
Ensure: $fitnestgene$
  $genotype \leftarrow generateGene()$
  $best \leftarrow 0$
  for $i = 0 to 15$ do
    $currentgene \leftarrow mutate(genotype, Mrate = rate)$
    $fit \leftarrow fitness(currentgene)$
    if $fit > best$ then
      $fit \leftarrow best$
      $genotype \leftarrow currentgene$
    end if
  end for
  return $genotype$
```

Algorithm 2 Microbial GA

```
Require: $FLOAT\ mutationrate, INT n$
  $genotypePop \leftarrow generateGenePop()$
  $best \leftarrow 0$
  for $i = 0 to 15$ do
    $n1 \leftarrow random(n)$
    $n2 \leftarrow random(n)$
    $currentgene1 \leftarrow genotypePop[n1]$
    $currentgene2 \leftarrow genotypePop[n2]$
    $currentgene1 \leftarrow mutate(currentgene1, Mrate = rate)$
    $currentgene2 \leftarrow mutate(currentgene2, Mrate = rate)$
    $currentgene1, currentgene2 \leftarrow tournament(currentgene1, currentgene2)$

    $genotypePop[n1] \leftarrow currentgene1$
    $genotypePop[n2] \leftarrow currentgene2$
  end for
```

3.1 Fitness

The aim of the robot is to walk as far as possible within the limited sequence space, by using a distance sensor. Fitness will be measured in millimeters, rather than percentage. These millimeters represent how far the robot got when stepping out.

$$fitness = initialDistance - currentDistance \tag{2}$$

Any fitness below 4 will be classed as 0 as the robot may have tilted to face the ground. This is not an optimal solution. This does mean the robot must have a reasonable distance between itself and the wall. For this experiment we will give it 1 meter.

Algorithm 3 Calculate Fitness

```
Require: $INT startDistance, INT endDistance$
Ensure: $fitnessinmm$
  $gained \leftarrow startDistance - endDistance$ 1
  if $gained > 4$ then
    return $gained$
  end if
  return 0
```

The above pseudo-code shows the fitness where the algorithm is penalized for being too close or further away.

3.2 Mutation

Mutation is defined within the algorithm as a percentage chance that each gene (motor angle positions) will be mutated. If it is to be mutated a random position will be chosen, and this position will be changed to another value picked from our selection of potential angles, specified at the beginning of this section.

Algorithm 4 Mutate

```
Require: $Listgenotype, floatrate, listpossibleMoves$
Ensure: $Listmutatedgenotype$
  for $i = 0togenotype.size$ do
    if $random_percentage() < rate$ then
      $step \leftarrow genotype[i]$
      $step[random_position()] \leftarrow possibleMoves[random_position()]$
      $genotype[1] \leftarrow step$
    end if
  end for
  return $genotype$
```

3.3 Implemen tation

There are two loops, one represen ting an increasing mutation rate and another for the generations. Each task starts off with the same genot ype in order to fairly compare the changes made through mutation. This prev ents a randomly generated genot ype being closer to a walking algorithm than what was generated for an other experimen t. Each will ha ve a starting fitness of 0 before the generation begins. We will gather from mutation rates between (inclusiv e) 10% and 80%. There will be 15 generations to generate a walking pattern. This small num ber is appropriate as there is a limited search space and it is applied to all exp erimen ts. When implemen ting the microbial GA the population size could not exceed 3 due to memory issues. As the population size was larger than two the algorithm could run, as it differed from a hill climber.

A few changes were made to the existing functions from the hill climber, in the microbial algorithm for memory . This did not effect the end solution as the functions still performed the same tasks .

At the end of the experimen t, significan t mutation rates will be selected and ran again multiple times to calculate averages. This will be to determined how effective a mutation rate is and remo ve fluke values. By the end, three exp erimen ts will be run on each selected value.

4 Results

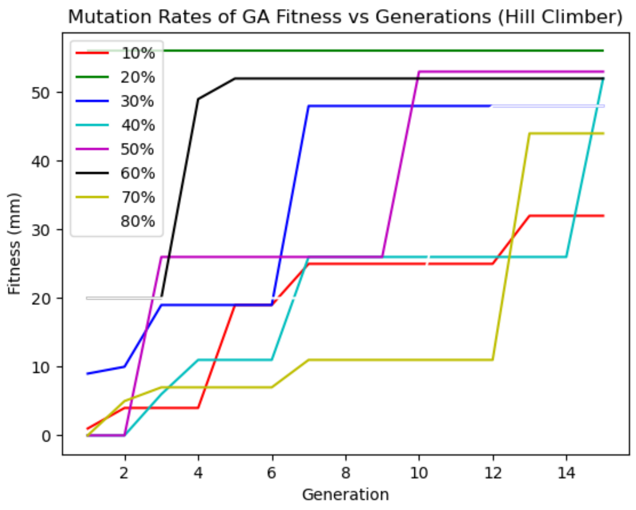

Figure 2: Generation versus Fitness of Hill Clim ber

Plotting each fitness of all generations for every percentage didn't show any obvious trends. There is a larger grouping of low fitnesses mainly for rates 30%, 40%, 70% and 80%.

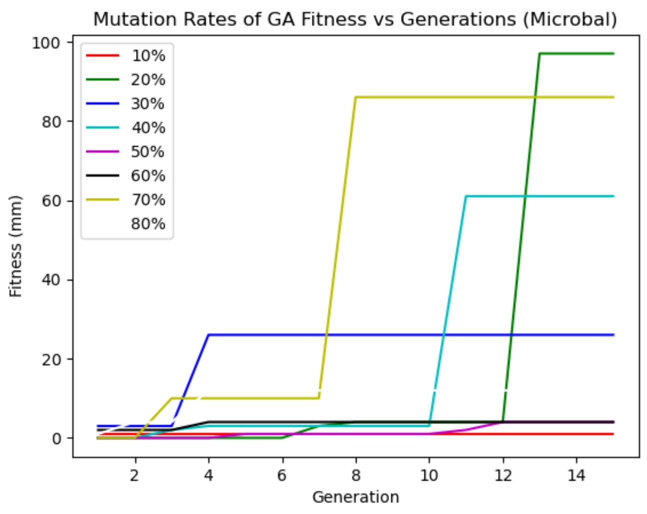

Figure 3: Generation versus Fitness of Hill Climber. Blue points show the microbial points and red shows the hill climber

Figure 3.2 shows the same format of data for the microbial GA. It shows similar information where the majority of higher mutation rates performed lower throughout. With 40%, 70% and 20% showing high fitness at the end.

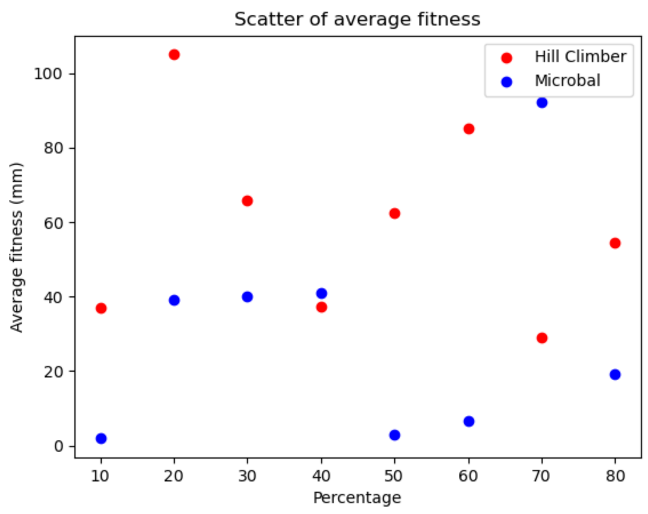

Figure 4: Average fitness of each percentage mutation rate

By compressing figure 4 into the average fitness for each generation, we see 20% is considerably higher compar ed to the next top fitness. The hill climber out performed the microbial in this test case. 20% was still on the high end of the microbial fitness.

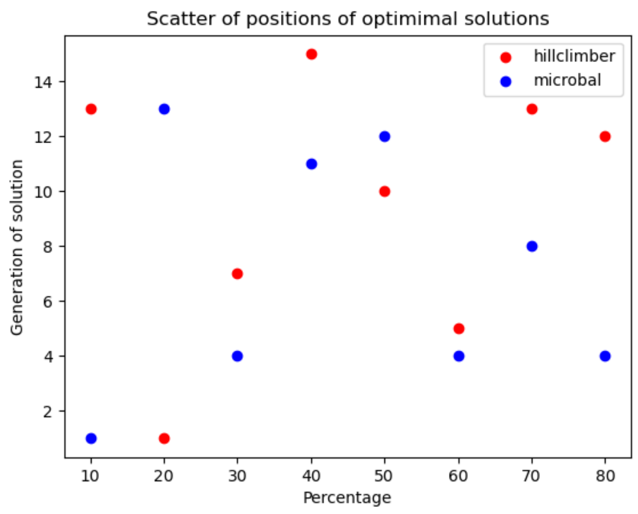

Figure 5: Lowest generation with solution for each mutation rate

The plots of the percentages of mutation rate against the earliest generation of the top solution (mainly sub-optimal) found within each experiment shows how quickly a solution was found. The average generation of finding a solution was generation 9 for the hill climber. The microbial average generation of solution was 7. Only one experiment did not find a solution.

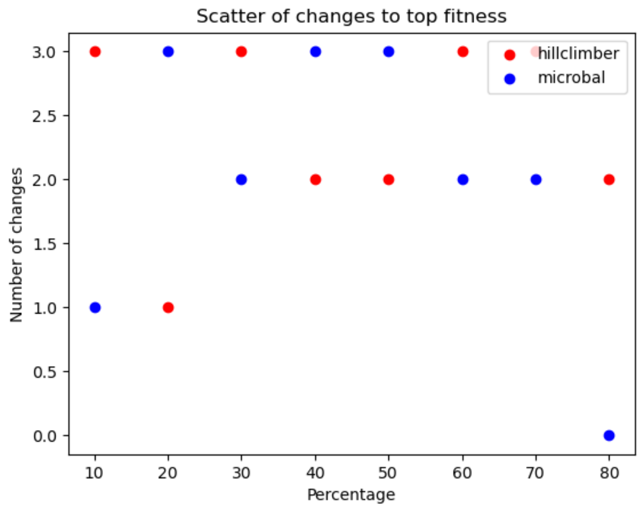

Figure 6: Number of changes of best fitness made for each mutation rate

We plot the number of changes made, where the current fitness is better than the held fitness.

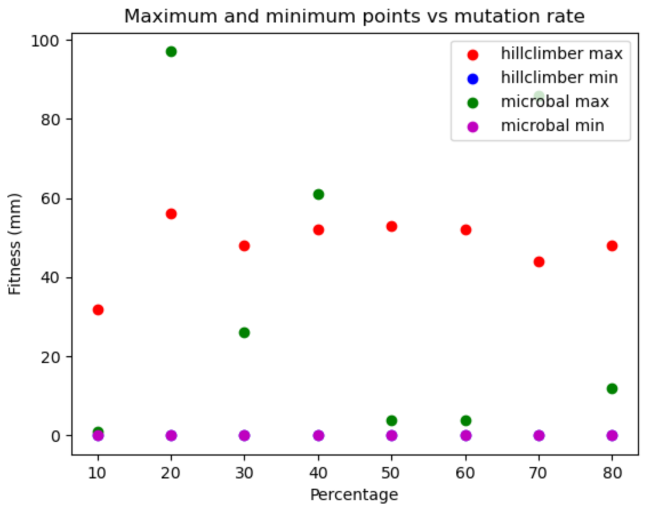

Figure 7: Maximum and minimum fitness of each percentage mutation rate

Finally, we see that every experiment fell over or walked backwards at least once. This is how fitness of 0mm is measured.

After analyzing the results, 20, 40 and 70 % mutation rates were selected to run further experiments per mutation rate.

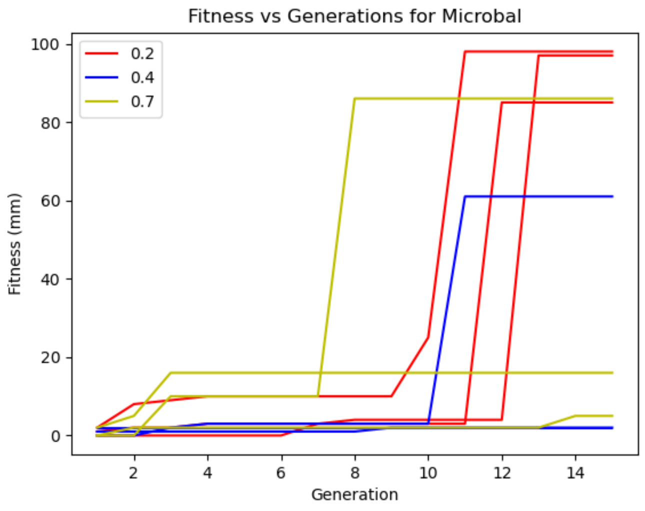

Figure 8: Fitness vs mutations of 3 trials for each significant mutation rate using the Microbial algorithm

Each colour represents the same mutation rate, but a different trial. It is clear to see 0.2 as the higher rate.

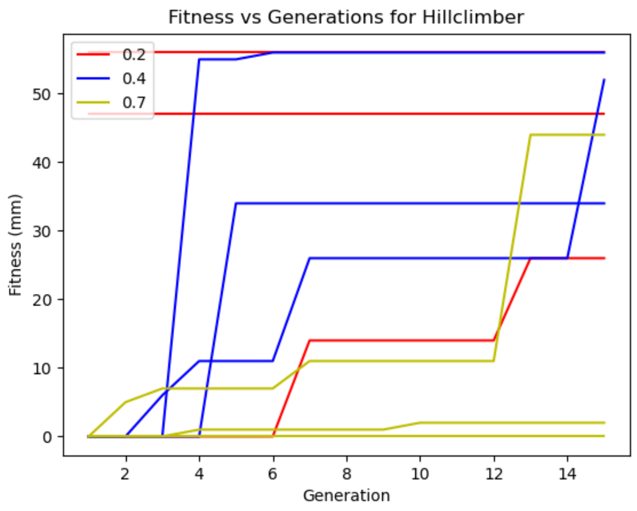

Figure 9: Fitness vs mutations of 3 trials for each significant mutation rate using the Hill climber

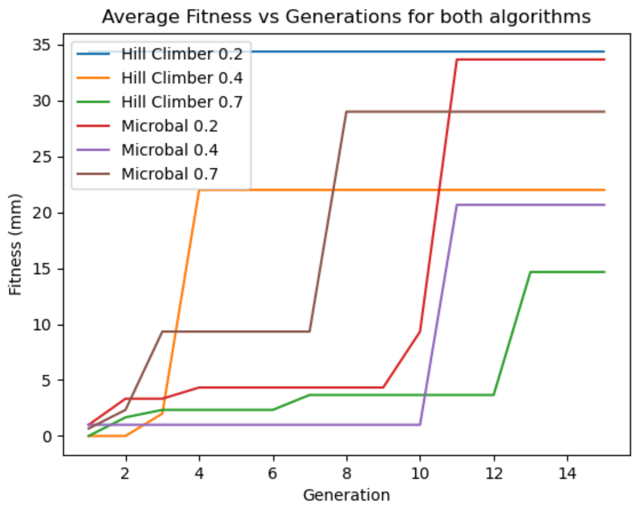

Figure 10: Fitness vs mutations of 3 trials as averages over the 3 trials

Figure 10 helps distinguish fluke values and trials within our experiment.

5 Discussion

The results show the optimal mutation rate for the problem of the 4 joint walking optimization. This was a mutation rate of 20%. The results show that a mutation rate of 10% took a long time to reach a solution within the hill climber and found a low fitness solution in the microbial, most likely due to the fact that little would change in the genotype at each iteration. The average fitness was relatively low in comparison to other results.

With a higher mutation rate the robot would fall over at a higher rate. When using the hill climber it would fail to correct this in the next iteration at a higher mutation rate. We can see in Figure 2 a large number of generations which fails to gain a high fitness. In Figure 3 we see a steady low fitness. Due to the large amounts of change resulting from a higher mutation rate, it is harder for the algorithm to work out what caused the low fitness and what didn't. This suggests a lower mutation rate is more accurate in locating error.

There was no trend within the number of changes in Figure 6. This is likely due to the random nature of genetic algorithms. It seems the average change was from 2 to 3 changes.

Although in Figure 5 many of the higher mutation rates converge to a solution quicker, we see that the average fitness for the respective rates displayed in Figure 4 are low.

In Figure 7 the fitness is much higher for a 20% rate than the other mutation rates of its algorithm. This is unlikely as a result of the mutation function. 20% microbial mutation rate out performed the hill climber results.

After experimenting with 3 trials on each mutation rate, we see that 20% out-performs the other chosen rates (see Figure 8). This backs up what has previously been discussed within this section.

5.1 Future work

The controller board used in the project had limited memory. This held back how in depth the algorithms could go, as well as number of steps and population size. For further development in this project one would need a controller with a larger capacity for longer arrays. This would likely a Raspberry Pi for its Python capabilities.

6 Conclusion

To conclude there is an optimum mutation rate for a genetic algorithm. For the task of moving a 4 joint chassis, the optimum was 20%. At 20% the algorithm converged on average in fewer generations compared to other mutation rates and adapted better to lower fitness than higher mutation rates.

We have demonstrated that the use of a 20% mutation rate is best for a genetic algorithm for the task of walking. If the mutation rate is too high, the algorithm will find it harder to find which mutations were supportive to the task and which were not, increasing the probability of mistake. If the mutation rate is too low, then not enough mutations will take place for the algorithm to reach a optimal or sub-optimal solution in a small number of generations.

20% worked better for both the Microbial and Hill Climber algorithms. This was found to be the case in a series of repeated trials. Advancing this model so that fitness can be taken via a series of sensor methods and a larger population size of genotypes will greatly improve this experiment. This would be the subject of future work.

References

[1] Adafruit. Adafruit feather m0 overview. https://learn.adafruit.com/adafruit-feather-m0-basic-proto .

[2] J. Arabas, Z. Michalewicz, and J. Mulawka. Gavaps-a genetic algorithm with varying population size. In *Proceedings of the First IEEE Conference on Evolutionary Computation. IEEE World Congress on Computational Intelligence*, pages 73–78 vol.1, 1994.

[3] Tuomas Haarnoja. Learning to walk via deep reinforcement learning, 2019.

[4] M. Anthony Lewis. Genetic programming approach to the construction of a neural network for control of a walking robot, 1992.

[5] Pattie Maes and Rodney A. Brooks. Learning to coordinate behaviours, 1999.

[6] Tower Pro. Micro servo datasheet. http://www.ee.ic.ac.uk/pcheung/teaching/DE1_EE/stores/sg90_datasheet.pdf .

7 Appendix

Hill Climber

```
from adafruit_servokit import ServoKit
from board import D9, D6
from random import randint, choice, random
from time import sleep
from adafruit_hcsr04 import HCSR04

sonar = HCSR04(trigger_pin=D9, echo_pin=D6)
kit = ServoKit(channels=8)

#setup motors and genotype
motorStart=[90,100,140,110] #start angles
Mgenotype=[[choice([0,30,-30,0,0]) for x in range(4)] for i in
                                        range(10)] #n steps as max
distNo=0

def getDist(): #get the distane reading
```

```
    dist=None
    while dist==None: #get numeric value
        try:
            dist=sonar.distance
        except RuntimeError: #do not return error
            pass
    return int(dist)
def move(angles):
    for i,ang in enumerate(angles): #move the servos by the given
                                        angles
        if kit.servo[i].angle+ang>=0 and kit.servo[i].angle+ang<=
                                            180: #validate the change
                                             is in range
            kit.servo[i].angle+=ang #set each servo to change
            sleep(0.2) #prevent over current draw
    sleep(1) #give time to rest

def st(angles): #move all servors to the given angles
    for i,ang in enumerate(angles):
        kit.servo[i].angle=ang #set servo angle
def mutate(geno,rate=0.2): #mutate the genotype with the given rate
    for i in range(len(geno)):
        if random() < rate: #if rate% chance
            pos=geno[i]
            n2=randint(0,len(pos)-1) #mutate again
            c=[0,30,-30,0,0]
            c.remove(pos[n2])
            pos[n2]=choice(c) #unique
            geno[i]=pos.copy()
    return geno #return mutated
print("start")
for z in range(10): #loop through the sample size of rates (10%, 20
                                    %, ..., N%)
    st(motorStart)
    lastFitness=0
    store=[]
    genotype=Mgenotype.copy() #copy over the main genotype to give
                                        a fair experiment
    for i in range(15): #total of 15 generations
        sleep(1) #time to reset if fallen over
        distNo=getDist() #start dist
        print("Generation",i+1)
        currentGeno=mutate(genotype.copy(),rate=((z+1)/10)) #mutate
                                             via rate defined by z

        for j,step in enumerate(currentGeno): #move motors by each
                                            position specified in
                                            genotype
            move(step)

        cd=getDist() #get the new distance
        fit = distNo-cd

        print(cd,distNo)
        if cd<=distNo and cd>10: #fitness function not written as
                                            function to conserve
                                            memory
            print(fit)
            if fit>lastFitness: #if fitness is better
                print("fitter")
                genotype=currentGeno.copy() #set the fitter
                                                 genotype
                lastFitness=fit #store the best fitness
        else:
            fit=0
        store.append(fit) #store the fitness of each round
        st(motorStart) #reset to start position to fairly test the
                                          next generation
        sleep(1)
    print(j,store) #show console what is going on
```

microbial

```
from adafruit_servokit import ServoKit
from board import D9, D6
from random import randint, choice, random
from time import sleep
from adafruit_hcsr04 import HCSR04

sonar = HCSR04(trigger_pin=D9, echo_pin=D6)
kit = ServoKit(channels=8)

#setup motors and genotype
"""
PopGenotype=[[[-30, 0, 30, 0], [0, 0, -30, 0], [0, 0, 0, 0], [30, 0
                                  , 30, 30], [30, 0, 30, 0], [0, 0,
                                   -30, 0]], [[30, 0, 0, 0], [0, 30
                                  , 0, -30], [0, 0, 0, 0], [0, 0,
                                  30, 0], [30, 0, -30, -30], [30, 0
                                  , 0, -30]], [[0, 0, 0, 0], [0, 0,
                                   30, 0], [0, 0, 30, 0], [0, 30, 0
                                  , -30], [0, 0, -30, 0], [0, 0, 0,
                                   -30]]]

PopGenotype=[]
popsize=3
for i in range(popsize): #population size 10
    PopGenotype.append([[choice([0,30,-30,0,0]) for x in range(4)]
                                      for i in range(6)])#n steps
                                      as max
"""

def getDist(): #get the distane reading
    dist=None
    while dist==None: #get numeric value
        try:
            dist=sonar.distance
        except RuntimeError: #do not return error
            pass
    return int(dist)
def move(angles):
    for i,ang in enumerate(angles): #move the servos by the given
                                      angles
        if kit.servo[i].angle+ang>=0 and kit.servo[i].angle+ang<=
                                          180: #validate the change
```

```
                                             is in range
            kit.servo[i].angle+=ang #set each servo to change
            sleep(0.2) #prevent over current draw
    sleep(1) #give time to rest

def st(): #move all servors to the given angles
    kit.servo[0].angle=90 #set servo angle
    kit.servo[1].angle=100 #set servo angle
    kit.servo[2].angle=140 #set servo angle
    kit.servo[3].angle=110 #set servo angle
def mutate(geno,rate=0.2): #mutate the genotype with the given rate
    for i in range(len(geno)):
        if random() < rate: #if rate% chance
            pos=geno[i]
            n2=randint(0,len(pos)-1) #mutate again
            c=[0,30,-30,0,0]
            c.remove(pos[n2])
            pos[n2]=choice(c) #unique
            geno[i]=pos.copy()
    return geno #return mutated

for z in range(10): #loop through the sample size of rates (10%, 20
                                     %, ..., N%)
    lastFitness=0
    PopGenotype=[[[-30, 0, 30, 0], [0, 0, -30, 0], [0, 0, 0, 0], [
                                         30, 0, 30, 30], [30, 0, 30, 0
                                         ], [0, 0, -30, 0]], [[30, 0,
                                         0, 0], [0, 30, 0, -30], [0, 0
                                         , 0, 0], [0, 0, 30, 0], [30,
                                         0, -30, -30], [30, 0, 0, -30]
                                         ], [[0, 0, 0, 0], [0, 0, 30,
                                         0], [0, 0, 30, 0], [0, 30, 0,
                                          -30], [0, 0, -30, 0], [0, 0,
                                          0, -30], [0, 0, 30, 0], [0,
                                         30, 0, -30], [0, 0, -30, 0],
                                         [0, 30, 0, -30]]]

    for i in range(15): #total of 15 generations
            print("Generation",i+1)
            st()
            sleep(2) #time to reset if fallen over
            distNo1=getDist() #start dist

            n1=randint(0,popsize-1) #gather the population samples
            n2=randint(0,popsize-1)

            current=PopGenotype[n1].copy()
            current=mutate(current,rate=(z+1)/10)

            for step in current: #move motors by each position
                                                specified in genotype
                move(step)

            cd1=getDist() #get the new distance

            current=PopGenotype[n2].copy()
            current=mutate(current,rate=(z+1)/10)
```

```
                st() #reset to start position to fairly test the next
                                                        generation
                sleep(2) #time to reset if fallen over
                distNo2=getDist() #start dist

                for step in current: #move motors by each position
                                                        specified in genotype
                    move(step)
                cd2=getDist() #get the new distance

                print(max(distNo1-cd1,distNo2-cd2))

                if distNo1-cd1>distNo2-cd2: #tournament selection
                    PopGenotype[n2]=PopGenotype[n1].copy()
                else:
                    PopGenotype[n1]=PopGenotype[n2].copy()

        print((z+1)*10) #show console what is going on
```