COURSEWORK 1, GA

Acquired Intelligence and Adaptive portfolio

JUNE 3, 2020

UNIVERSITY OF SUSSEX
Candidate no. 181334

Introduction

We created the following algorithms in python:

- Hillclimber
- Steady State Genetic Algorithm
- Spatial Genetic Algorithm
- Full Microbial Genetic Algorithm.

Each algorithm was pitted to solve the knapsack problem. The knapsack problem is such that we are aiming to maximize value:

$$\Sigma_i^N B_i$$

FIGURE 1 [1]

Subject to constraint:

$$\Sigma_i^N V_i \leq V.$$

FIGURE 2 [1]

We do this using the data:

Item	a	b	c	d	e	f	g	h	i	j
B	5	6	1	9	2	8	4	3	7	10
V	3	2	4	5	8	9	10	1	6	7

We Implemented a population of hill climbers to solve a resource allocation(or knapsack) problem and design a suitable fitness function and selection method. This allowed us to investigate how the number of generations or repetitions of the algorithm effects the generated solutions through evolution.

Hill Climber

Each hill climbing individual increases its fitness through trial and error by creating random solutions by mutating the genotype and comparing it with the current genotype.

We coded and implemented a single hill climbing individual to solve the above task. We ran our algorithm for 100 generations and observed that the population converged towards the optimal solution but then entered a local minima most of the time. We coded and implemented a solution to allow a population of hill climbers to solve the task in parallel.

Steady State GA with tournament selection.

Here we aim to implement a fully functional steady state genetic algorithm. We investigated the effects of population size, mutation rate and recombination rate on evolution. We discovered that it performed RELATIVE TO HILL CLIMBER FROM LAST WEEK. THE EFFECT OF THE MUTATION RATE IS.

Spatial GA

Here we implemented a spatial genetic algorithm and evaluated comparatively to the steady state genetic algorithm with tournament selection. We evaluated this algorithm in comparison to the Steady-State genetic algorithm. We found that it evolves similarly as fast. We found that it does get stuck in local minima more or less often.

Full Microbial Genetic Algorithm

Method

Hill Climber

To create the hill climber, we a hill climber class. The class contained:

- RandomiseGenotye(Genotype)
- getTotalVolumeOfGenotype(Genotype)
- getTotalbenefitOfGenotype(Genotype, arrayType)
- mutate(Genotype)
- mutateEntirePopulationByN(population, N)
- fitnessBefore(Genotype)
- fitnessAfter(Genotype)

We created a population of 10 initial genotypes. We then ran the model using the brute force method to produce 100 generations. The fitness of every mutation of every genotype in the population and plotted a graph of the fitness score. The fitness score calculated how close each benefit is to the ideal solution.

Steady State GA with tournament selection

We created a Steady state GA with tournament selection in python. We generated a random population of size 50-100 and plotted 100 generations. We picked 2 individuals from the population at random and evaluated them to find a winner and loser. We copied the winner over the loser then mutated the loser 20 times. If the loser scored a fitness higher than the winner, we update the winner as the mutated loser that scored higher. This will give up after 20 mutations.

Spatial GA implementation.

We created a spatial GA in python. We plotted population of N individuals on a 1D array. We initialized random pop P across an array. We then picked an element at random and then a second individual k items away from the randomly picked element. We then calculated fitness of both genotypes and compared them to find a winner. We copy the winner over the loser and add a mutation to the loser up to 20 times. If a better solution/gentype, scored using the fitness function.

Full Microbial Genetic Algorithm

We implemented the full microbial genetic algorithm in python. We created a 1D grid the same as the spatial GA. We evaluated two randomly selected elements of the 1D array within K distance of each other. We copy the winner over the loser and add a Pcrossover mutation to the loser up to 20

times. If a better solution/gentype, scored using the fitness function, we update the winner as the newfound value.

Results

Hill Climber

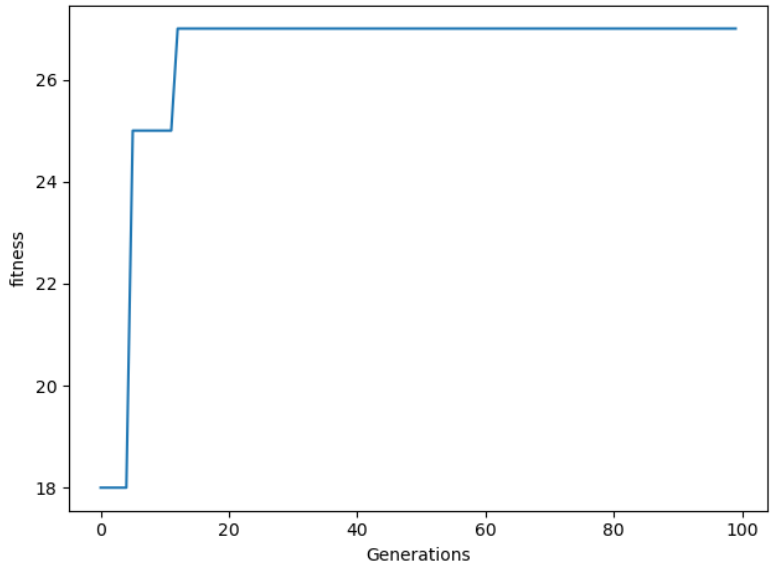

FIGURE 3 1 SINGLE HILL CLIMBER

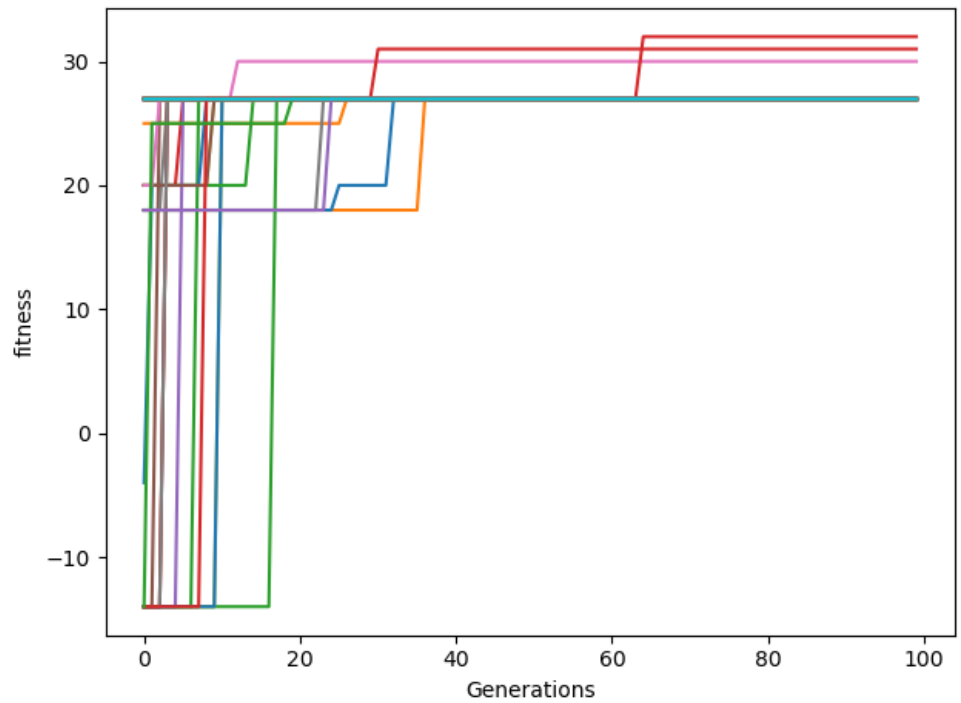

FIGURE 4 100 HILL CLIMBERS OVER 100 GENERATIONS.

Steady State Genetic Algorithm

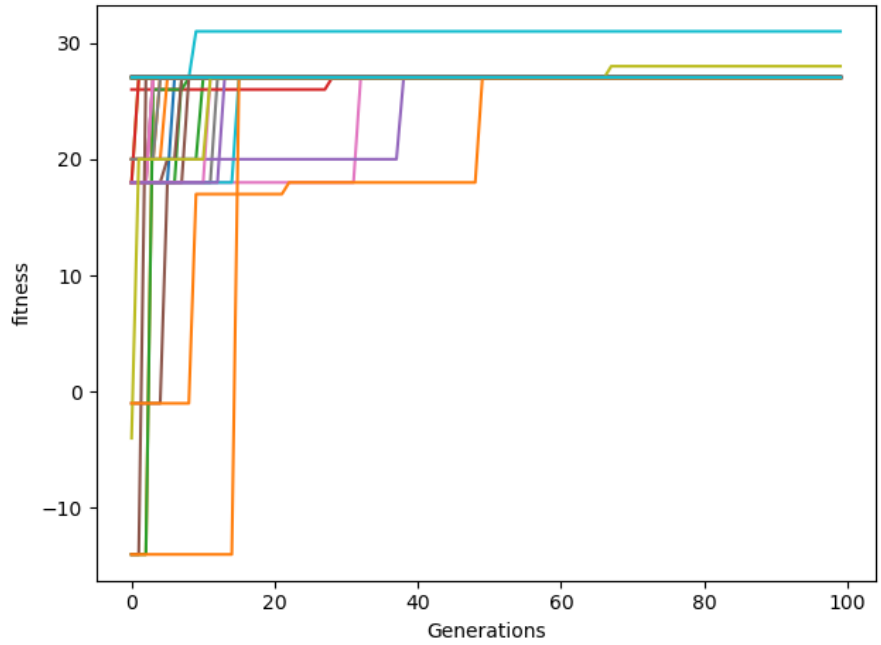

FIGURE 5 STEADY STATE GENETIC ALGORITHM OVER 100 GENERATIONS

Spatial Genetic Algorithm

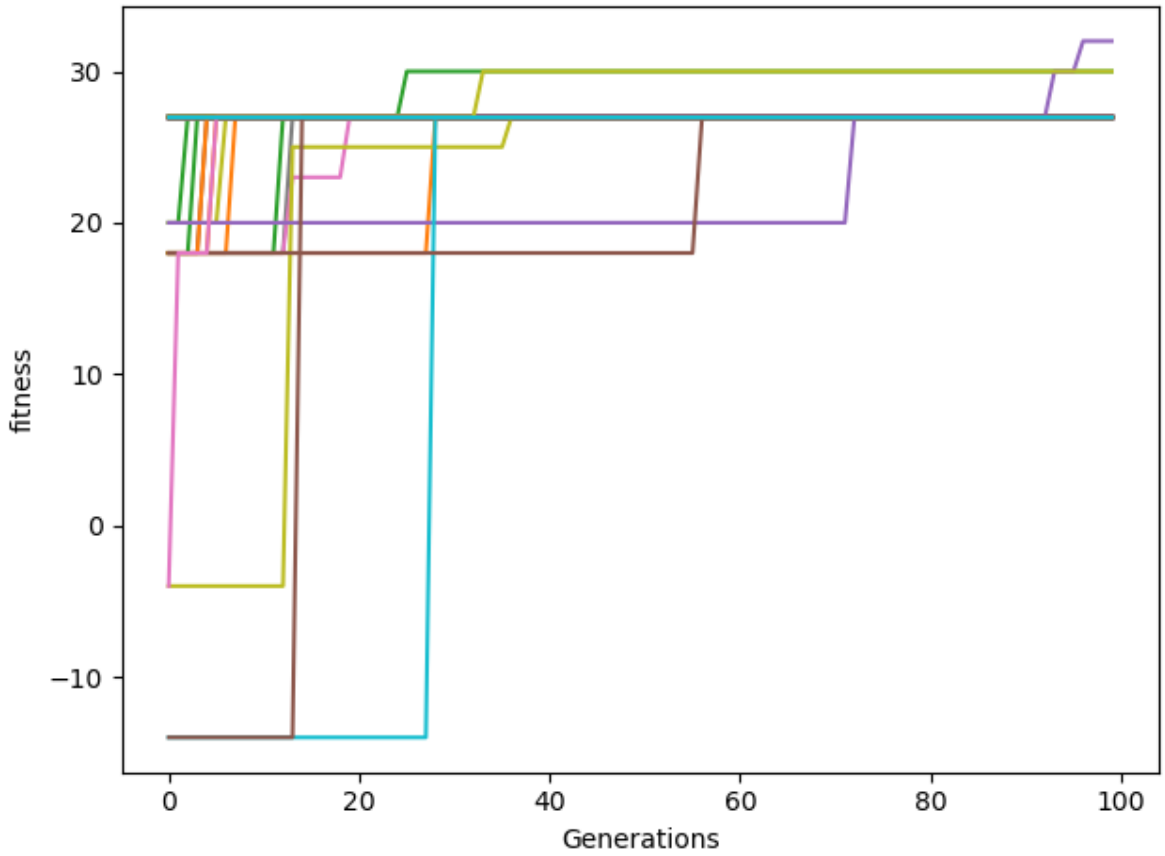

FIGURE 6 SPATIAL GENETIC ALGORITHM OVER 100 GENERATION

Full Microbial Genetic Algorithm

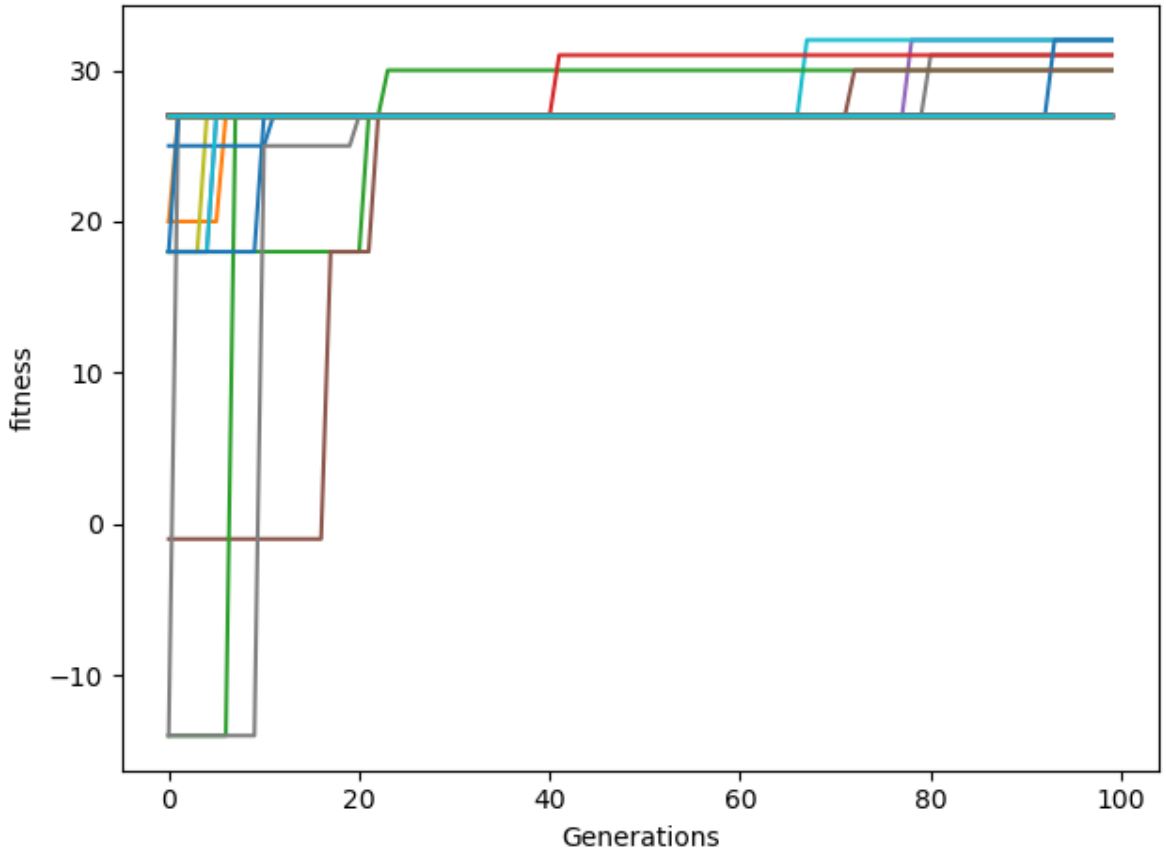

FIGURE 7 3 OPTIMAL SOLUTIONS

Discussion/Conclusions

Hill climber

As we can see in figure 3, some hill climbers get stuck at a suboptimal solution or 'minima'. We did not find any solutions that get worse over time due to our code design filtering out worse mutations. Therefore, oue hillclimber cannot get worse over time. This can be observed when plotting multiple hillclimbers in parallel as we can see after 100 generations, they finish with different final fitness scores after 100 generations. We reached a minima 99/100 times.

Coding a population of hill climbers is more computationally expensive as many climbers are required to run in parallel. From this observation, we can predict that the more complex the problem, for example, if the knapsack had 100 items, and a max volume of 100, it would be more computationally expensive/ require more processing power to finish running the algorithm.

Steady State Genetic Algorithm

We found this solution to evolve at a very similar rate to the hill climbers. This reached a minima 98 out of 100 times. In two instances it reached an optimal solution of fitness 32.

Spatial Genetic Algorithm

This spatial genetic algorithm appears to evolve slower than the steady state genetic algorithm and get stuck in local minima more often than the steady state genetic algorithm.

Full Microbial Genetic Algorithm

This algorithm enables faster evolution in some cases. However for the most part the evolution speed seems to be faster than the hill climber, steady state and special as it seems to reach either a local minima or an optimal solution by only 40 generations. The Full Microbial Genetic Algorithm reaches an optimal solution 3 times out of 100, and a minima 97 out of 100. We found Pcrossover to have no substantial effect on the speed of evolution.

References

[1] Sign in to your account. 2020. Sign in to your account. [ONLINE] Available at: https://canvas.sussex.ac.uk/courses/8727/files/1106094?module_item_id=653784. [Accessed 10 June 2020].

Appendix

Hill Climber

```
#Hillclimber Algorithm

import matplotlib.pyplot as plt
import random as random
import numpy as np

Matrix = np.array([['a', 5, 3, 0], ['b', 6, 2, 0], ['c', 1, 4, 0], ['d', 9, 5, 0], ['e', 2, 8, 0], ['f', 8, 9, 0], ['g', 4, 10, 0], ['h', 3, 1, 0], ['i', 7, 6, 0], ['j', 10, 7, 0]], dtype = object)
Genotype = Matrix



class hillclimber:#
    optimalCount = 0
    def _init_(self, Genotype, object):
        hillclimber.Genotype = Matrix
        hillclimber.Genotype = self.CreateGenotype(Genotype)
        hillclimber.hillClimberCount += 1
        Genotype = Matrix
        self.optimalCount = 0

    def RandomiseGenotype(self, Genotype):
        for i in Genotype:
            vol = self.getTotalVolumeOfGenotype(Genotype)
            count = 0
            while (vol + i[1] <= 20):
                randomChoice = random.choice(list(Genotype))
                comparison = i == randomChoice
                equal_arrays = comparison.all()
                if (equal_arrays == True):
                    i[3] = 1
                count += 1
                if (count == 30):
                    break

        return (Genotype)

    def displayGenotype(self, Genotype):
        print(Genotype)

    def getTotalVolumeOfGenotype(self, Genotype):
        volume = 0
        for i in Genotype:
            if (i[3] == 1):
                volume = volume + i[1]
        return volume

    def getTotalBenefitOfGenotype(self, Genotype):
        benefit = 0
        for i in Genotype:
            if (i[3] == 1):
                benefit = benefit + i[2]
        return benefit

    def fitness(self, Genotype):
```

```
        if (self.getTotalVolumeOfGenotype(Genotype) < 20):
            return self.getTotalBenefitOfGenotype(Genotype)
        else:
            return 20 - self.getTotalBenefitOfGenotype(Genotype)
        if(getTotalBenefitOfGenotype(Genotype) == 31):
            self.optimalCount += 1

    def CreateGenotype(self, Genotype):  # function to create a genotype with VOLUME <= 20
        Genotype = np.array([])
        Genotype = Matrix
        Genotype = self.RandomiseGenotype(Genotype)
        #print(Genotype)
        return Genotype
#
    def mutate(self, Genotype):
        num = random.randint(0, 9)
        if (Genotype[num][3] == 0):
            Genotype[num][3] = 1
            if (self.getTotalVolumeOfGenotype(Genotype) > 20):
                Genotype[num][3] = 0
        if (Genotype[num][3] == 1):
            Genotype[num][3] = 0
            self.RandomiseGenotype(Genotype)

        return Genotype

    def climber(self, Genotype, G):
        fitness = []
        fitnessGeno = self.fitness(Genotype)
        for i in range(0, G):
            mutatedGeno = self.mutate(Genotype)
            fitnessMutatedGeno = self.fitness(mutatedGeno)
            #if(fitnessMutatedGeno < fitnessGeno):
            #    fitnessGeno = fitnessGeno
            if(fitnessGeno < fitnessMutatedGeno):
                fitnessGeno = fitnessMutatedGeno
            fitness.append(fitnessGeno)
            #print(len(fitness))
        return fitness


hillClimber0 = hillclimber()
geno = hillClimber0.CreateGenotype(Genotype)
fitness0Climber = hillClimber0.climber(geno, 100)
#print(fitness0Climber)

P = []
listOfFitness = []
for i in range(0, 100): # create 100 genotypes
    geno = hillClimber0.CreateGenotype(Genotype)
    fitness0Climber = hillClimber0.climber(geno, 100)
    P.append(geno)
    listOfFitness.append(fitness0Climber[-1])
    plt.plot(fitness0Climber)
    plt.xlabel("Generations")
    plt.ylabel("fitness")

plt.show()
print(listOfFitness)
optimalCount = 0
for i in listOfFitness:
    if(i == 32):
        optimalCount += 1

print(optimalCount)
```

Steady State Genetic Algorithm

```
#Steady State Genetic Algorithm

import matplotlib.pyplot as plt
import random as random
import numpy as np
from hillclimber import hillclimber

Matrix = np.array([['a', 5, 3, 0], ['b', 6, 2, 0], ['c', 1, 4, 0], ['d', 9, 5, 0], ['e', 2, 8, 0], ['f', 8, 9, 0], ['g', 4, 10, 0], ['h', 3, 1, 0], ['i', 7, 6, 0], ['j', 10, 7, 0]], dtype = object)
Genotype = Matrix

class GA:
    def __init__(self, Genotype, populationSize, numberOfGen):
        self.pop = []
        randomSize = random.randint(50, populationSize)
        for i in range(randomSize):
            hillClimber0 = hillclimber()
            hillClimber = hillClimber0.CreateGenotype(Genotype)
```

```
            self.pop.append(hillClimber)

        self.noOfGen = numberOfGen
        self.Geno = Genotype
        print(self.Geno)
        print(self.noOfGen)

    def mutate(self):
        meanFitnessAtEachEvaluation = []
        for i in range(self.noOfGen):
            lenPop = ((len(self.pop)) - 1)
            num1 = random.randint(0, lenPop)
            num2 = random.randint(0, lenPop)
            a = self.pop[num1]
            b = self.pop[num2]
            aclimb = hillclimber0.climber(self.Geno, self.noOfGen)
            bclimb = hillclimber0.climber(self.Geno, self.noOfGen)
            #last item of fitness arrays
            #compare last item
            aFit = aclimb[-1]
            bFit = bclimb[-1]

            if (aFit == bFit):
                winner = a
                loser = b
            if (aFit > bFit):
                winner = a
                loser = b
            if (bFit > aFit):
                winner = b
                loser = a
            # copy W over L
            a = winner
            b = winner
            self.pop[num1] = a
            self.pop[num2] = b
            #print(a)
            # add a mutation to the loser
            for i in range(20):
                winnerFit = hillclimber0.fitness(winner)
                mutatedLoser = hillclimber0.mutate(loser)
                mutatedLoserFit = hillclimber0.fitness(mutatedLoser)
                if(mutatedLoserFit > winnerFit):
                    winner = mutatedLoser

            meanFitnessAtEachEvaluation.append(winner)
        return meanFitnessAtEachEvaluation


hillclimber0 = hillclimber()
ga0 = GA(Genotype, 100, 100)
fitnessPlot = ga0.mutate()

plt.plot(fitnessPlot)
plt.show()
```

Spatial Genetic Algorithm

```
#Spatial Genetic ALgorithm

import matplotlib.pyplot as plt
import random as random
import numpy as np
from hillclimber import hillclimber

Matrix = np.array([['a', 5, 3, 0], ['b', 6, 2, 0], ['c', 1, 4, 0], ['d', 9, 5, 0], ['e', 2, 8, 0], ['f', 8, 9, 0], ['g', 4, 10, 0], ['h', 3, 1, 0], ['i', 7, 6, 0], ['j', 10, 7, 0]], dtype = object)
Genotype = Matrix



class spatialGA:
    def __init__(self, Genotype, populationSize, numberOfGen):
        self.pop = []
        randomSize = random.randint(50, populationSize)
        for i in range(randomSize):
            hillClimber0 = hillclimber()
            hillClimber = hillClimber0.CreateGenotype(Genotype)
            self.pop.append(hillClimber)

        self.noOfGen = numberOfGen
        self.Geno = Genotype


    def mutate(self, k):
        meanFitnessAtEachEvaluation = []
```

```
        lenPop = ((len(self.pop)) - 1)
        # pick random individual
        num = random.randint(0, lenPop)
        posOrNeg = random.randint(0, 1)
        if(posOrNeg == 0):
            k = -k
        if(posOrNeg == 1):
            k = k


        aIndividual = self.pop[num]
        bIndividual = self.pop[num + k]
        if ((num + k) > lenPop):
            bIndividual = self.pop[(num - k)]
        if ((num + k) < 0):
            bIndividual = self.pop[(num + k)]


        print(aIndividual)
        print(bIndividual)

        aFit = hillClimber0.fitness(aIndividual)
        bFit = hillClimber0.fitness(bIndividual)

        loser = aFit
        winner = bFit
        if (aFit == bFit):
            winner = aIndividual
            loser = bIndividual
        if (aFit > bFit):
            winner = aIndividual
            loser = bIndividual
        if (bFit > aFit):
            winner = bIndividual
            loser = aIndividual
        # copy W over L
        aIndividual = winner
        bIndividual = winner

        self.pop[num] = winner
        self.pop[num + k] = winner
        # add a mutation to the loser
        # loserFit = fitnessBefore(loser, 0)
        winnerFit = hillClimber0.fitness(winner)
        # print(type(loser))
        j = 0
        while (j < 20):
            winnerFit = hillClimber0.fitness(winner)
            mutatedLoser = hillClimber0.mutate(loser)
            mutatedLoserFit = hillClimber0.fitness(mutatedLoser)
            if (mutatedLoserFit > winnerFit):
                winner = mutatedLoser
            j += 1

        meanFitnessAtEachEvaluation.append(winnerFit)

        return meanFitnessAtEachEvaluation


hillClimber0 = hillclimber()
ga0 = spatialGA(Genotype, 100, 300)
fitnessPlot = ga0.mutate(5)

plt.plot(fitnessPlot)
```

Full Microbial Genetic Algorithm

```
# Full Microbial Genetic Algorithm

import matplotlib.pyplot as plt
import random as random
import numpy as np
from hillclimber import hillclimber

Matrix = np.array(
    [['a', 5, 3, 0], ['b', 6, 2, 0], ['c', 1, 4, 0], ['d', 9, 5, 0], ['e', 2, 8, 0], ['f', 8, 9, 0], ['g', 4, 10, 0],
     ['h', 3, 1, 0], ['i', 7, 6, 0], ['j', 10, 7, 0]], dtype=object)
Genotype = Matrix

class microbialGA:
    def __init__(self, Genotype, populationSize, numberOfGen):
        self.pop = []
        randomSize = random.randint(50, populationSize)
        for i in range(randomSize):
```

```
            hillClimber0 = hillclimber()
            hillClimber = hillClimber0.CreateGenotype(Genotype)
            self.pop.append(hillClimber)

        self.noOfGen = numberOfGen
        self.Geno = Genotype

    def mutate(self, k):
        meanFitnessAtEachEvaluation = []
        lenPop = ((len(self.pop)) - 1)
        # pick random individual
        num = random.randint(0, lenPop)
        posOrNeg = random.randint(0, 1)

        if (posOrNeg == 0):
            k = -k
        if (posOrNeg == 1):
            k = k

        aIndividual = self.pop[num]
        bIndividual = self.pop[num + k]
        if ((num + k) > lenPop):
            bIndividual = self.pop[(num - k)]
        if ((num + k) < 0):
            bIndividual = self.pop[(num + k)]


        print(aIndividual)
        print(bIndividual)

        aFit = hillClimber0.fitness(aIndividual)
        bFit = hillClimber0.fitness(bIndividual)

        loser = aFit
        winner = bFit
        if (aFit == bFit):
            winner = aIndividual
            loser = bIndividual
        if (aFit > bFit):
            winner = aIndividual
            loser = bIndividual
        if (bFit > aFit):
            winner = bIndividual
            loser = aIndividual
        # copy W over L
        aIndividual = winner
        bIndividual = winner

        self.pop[num] = winner
        self.pop[num + k] = winner
        # add a mutation to the loser
        # loserFit = fitnessBefore(loser, 0)
        winnerFit = hillClimber0.fitness(winner)
        # print(type(loser))
        j = 0
        while (j < 20):
            winnerFit = hillClimber0.fitness(winner)
            for i in range(len(winner)):
                a = random.randint(0,9)
                if(a < 5):
                    loser[i] = winner[i]
                else:
                    loser[i] = loser[i]
            mutatedLoserFit = hillClimber0.fitness(loser)
            if (mutatedLoserFit > winnerFit):
                winner = loser
            j += 1

        meanFitnessAtEachEvaluation.append(winnerFit)

        return meanFitnessAtEachEvaluation


hillClimber0 = hillclimber()
ga0 = microbialGA(Genotype, 100, 100)

fitnessPlot = ga0.mutate(5)
plt.plot(fitnessPlot)
```