EAI 320: Practical 2

Lefa K Raleting 14222460

04 March 2020

1 Introduction

In this practical, a genetic algorithm(GA) was required, in order to find a high performance strategy for rps agent. This reports gives an overall breakdown on what GA is and a detailed implementation of the methods used in order to complete the task and the reasons thereto.

2 Background

2.1 What is a Genetic Algorithm?

Genetic algorithm is an algorithm used for optimization, usually used for non linear problems, or non differential problems. It uses concepts from evolutionary biology, and tries to mimic this.

2.2 How does GA work?

GA starts of with an initial population, populated with potential solutions(Genes). These Genes are tested against an objective function using a fitness test. In order to evolve, these Genes go through: **selection step**, **crossover** and **mutation step**. These terms are further explained below.

2.3 How is the selection step in the GA performed?

There are several methods available for use when it comes to selection methods. To understand them, we first look at Darwinian natural selection. Few conditions discussed are:

- Heredity
  Offspring must have a fixed process on how they receive characteristics of their parents.

- Variation
  There must be a way of introducing variation between offspring. Without variation the parents will pass down the same genes to the offspring's and so on.

- Selection
  The must be process to make the fittest offspring parents, also known as survival of the fittest.

The available selection strategies are:

- Roulette wheel selection
- Tournament selection
- Reward-based selection
- Truncation selection
- Random Select

2.3.1 Roulette wheel selection

This is basically the normalisation of fitness scores. Looking at the scores a probabilities. The function used to calculate the probability of a gene being selected is:

$$P_i = \frac{Fit_i}{\sum_{i=1}^{n} Fit_i} \tag{1}$$

where Fit_i is the fitness corresponding to the ith individual.

2.3.2 Tournament Selection

Tournament select is when a pool of genes are selected randomly to compete against each other. The higher probability wins. It can be said that the bigger the pool size, the less likely the low probability genes will win as now the pool would probably have more genes with high probabilities. This is how it works.

Select k individuals(pool size) out of a population randomly. choose the best individual with the best probability p, Then the second best individual with probability p*(1-p) and so on.

To prevent from selecting the same gene twice, after selecting an individual, remove it from the pool.

2.3.3 Truncation Selection

This method is less sophisticated than many other selection methods, and normally isn't used in practice when it comes to the Computer science field.

2.3.4 Random Select

This method, randomly selects k individuals in the population with a probability higher than n.

2.4 How is the crossover step in the GA performed?

2.4.1 One point crossover

In this crossover, a cross over point is selected randomly and the two tails or heads of the genes are then swapped around to create new children.

2.4.2 Multi point crossover

In this crossover, two points are selected randomly, the alternating sections are then swapped around to create new offspring.

2.4.3 Uniform crossover

In this crossover, we don't look at segments but rather chromosomes. We randomly select a chromosome from parent A or parent B, and we repeat this process for all chromosomes in both children.

2.5 How is the mutation step in a GA performed?

Mutation can be performed in several way. A random chromosome can be selected, and randomly changed.

Multiple chromosomes can be selected and randomly changed.

Randomly selected chromosome can be added to a certain constant in some applications.

3 Design

3.1 How is a chromosome represented?

Each chromosome is represented by either a "R","P" or "S".

3.2 How is a gene represented?

Each gene represented by 81 chromosomes.

3.3 How is the population populated?

A population size n is selected, then gene is populated randomly and added to the population, this is repeated until the population size reaches n.

The population is supposed to be greater than or equal to 2.

3.4 How is the fitness score calculated?

Each individual is compared to the whole data set, by comparing each chromosome of the individual with the data set. Every time there is a match then we increase the score of the individual by 1. To normalise it, the score is then divided by the number of chromosomes compared to the data set.

This is done for every individual in the population. It is then added to a list of fitness scores and returned to the caller.

3.5 How is the selection step in the GA performed?

For this Assignment, Random select was used, A threshold of n was selected. Two random individuals are selected with a fitness score above threshold.

3.6 How is the crossover step in the GA performed?

Single point crossover was used. In this crossover, a cross over point is selected randomly and the two tails or heads of the genes are then swapped around to create two new children.

3.7 How is the mutation step in a GA performed?

A random chromosome is selected and changed to either "R" ,"P" or"S" which is selected randomly too, this is to ensure that the agent does not get stuck in a local minimum.

4 Results and Discussion

4.1 How does the size of the population affect the performance of the GA?

4.2 How many generations are required to find a good solution?

It was found that after 40 generations the change became small for small populations.however for a larger population a generation of 200 resulted in greater results.

For greater populations, more generations where needed to find a good solution.

A graph where the fitness of the best individual from each generation is plotted against the number of generations is included below.

4.2.1 Crossover rate = 1

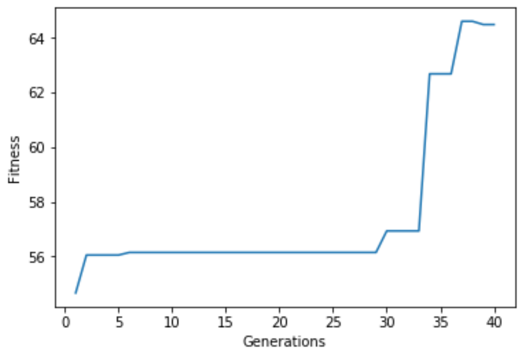

Figure 1: Population: 40; Mutation Rate: 30: Crossover Rate: 1

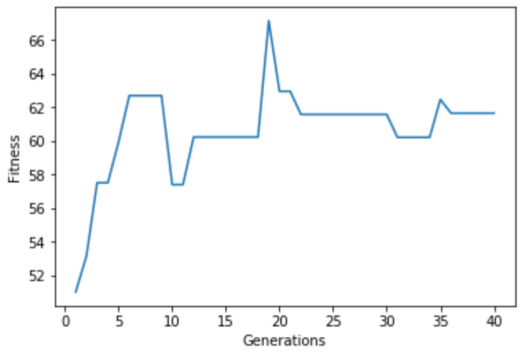

Figure 2: Population: 40; Mutation Rate: 70; Crossover Rate: 1

It is seen that that as the mutation rate increases the more rapidly the the data reaches a solution, however a mutation rate too high could lead to know solution at all.

4.2.2 Mutation Rate = 70

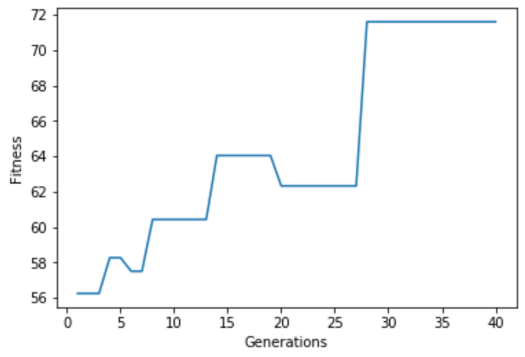

Figure 3: Population: 40; Mutation Rate: 70; Crossover Rate: 10

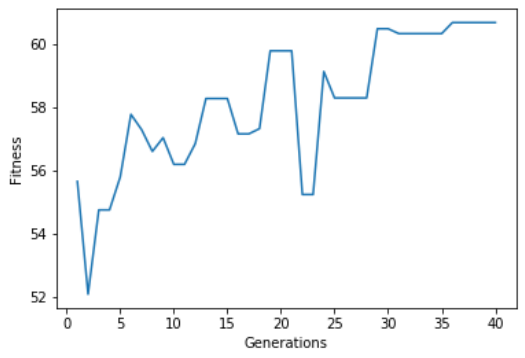

Figure 4: Population: 40; Mutation Rate: 70; Crossover Rate: 70

At a lower crossover rate we notice a less distorted trend , we see constant increases and less dips, this is due to the fact that, at a lower crossover rate,good genes stand a better chance of lasting longer than they would if the would have to mutate with genes of less fitness. At a high crossover rate it is the complete opposite.

4.2.3 Average Fitness and Maximum Fitness versus Number of Generations

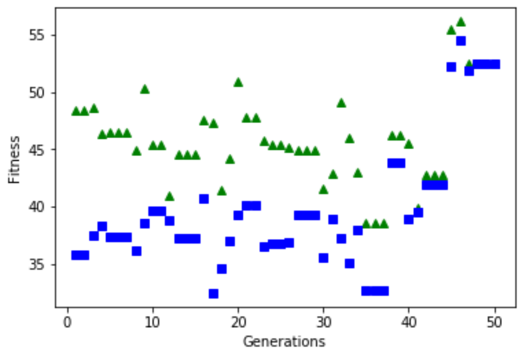

Figure 5: Numeber of Generations: 100

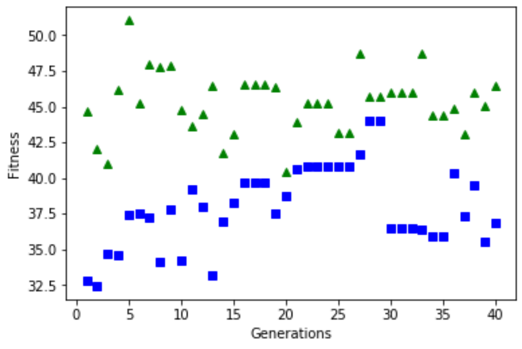

Figure 6: Number of Generations: 40

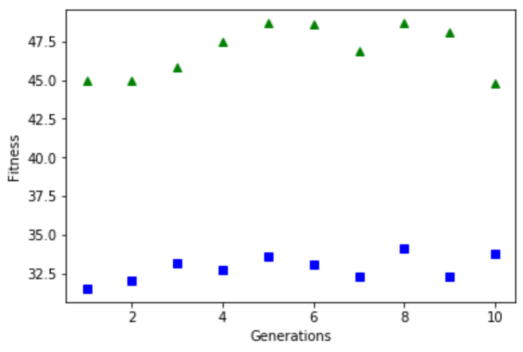

Figure 7: Number of Generations: 10

It is seen that the higher the number of generations that the closer the average fitness gets to the max-fitness of each generation. This show a clear indication of an overall population improvement and a typical example of evolution.

5 Conclusion

It Was found that a mutation rate of 70 and Crossover rate gave us the best solution for its number of generations, and that an increase in population size helped provide variance to the results increasing the gene pool, however this came it a cost of computational cost, as did the number of generations.It was also found that a greater amount of generations lead to a good solution, but after saturation(when the max fitness = average fitness) it was often found that no better solution could be found and resulted in a waste of computational time. Population size with a high crossover rate may cause the agent to never find a solution.A threshold too high leads to problem as the the Ga moves to a local minimum and a threshold too low leads to a problem as it may never reach a optimal solution.

6

References

[1] https://www.tutorialspoint.com/genetic_algorithms/genetic_algorithms$_{c}$*rossover.htm* [Accessed 01 March 2020].

[2] https://en.wikipedia.org/wiki/Selection_(genetic_algorithm)

[3] Krakow(2013). *PROCEEDINGS OF THE FEDCSIS*, Influence of the Population Size on the Genetic Algorithm Performance in Case of Cultivation Process Modelling

[4] Crow Kimura(1979). *Efficiency of truncation selection*

[5] Miller, Brad; Goldberg, David (1995). *Genetic Algorithms, Tournament Selection, and the Effects of Noise*. Complex Systems. 9: 193–212.

7 Appendix

7.1 Code

```
#response = ['P', 'P', 'R', 'P', 'P', 'S', 'S', 'R', 'S', 'R', 'S','P', 'S', 'P'
import numpy as np
import random
import time
import csv
import matplotlib.pyplot as plt
import copy
import collections

# All the possible histories.



#This part of the code. opens up a specified csv file , then adds it to a turple
with open('data2.csv', 'r') as csvFile:
    reader = csv.reader(csvFile)
    line=[tuple(row) for row in reader]
csvFile.close()

#This part of the code then goes on to create a data base of the the data collec
#counting ever instances and classifying it in a dictionary
database=collections.Counter()
database.update(line)


#these are my helper dictionaries to better help navigate my functions

indexmoves= {
           0:'RRRR', 1:'RRRP', 2:'RRRS', 3:'RRPR', 4:'RRPP', 5:'RRPS', 6:'RRSR',
           7:'RRSP', 8:'RRSS', 9: 'RPRR', 10:'RPRP',11:'RPRS', 12:'RPPR', 13:'RP
           14:'RPPS',15:'RPSR',16:'RPSP',17:'RPSS', 18:'RSRR', 19:'RSRP',20: 'RS
           21: 'RSPR', 22: 'RSPP', 23: 'RSPS', 24: 'RSSR', 25: 'RSSP', 26: 'RSSS
           27: 'PRRR', 28: 'PRRP', 29: 'PRRS', 30: 'PRPR', 31: 'PRPP', 32: 'PRPS
           33: 'PRSR', 34: 'PRSP', 35: 'PRSS', 36: 'PPRR', 37: 'PPRP', 38: 'PPRS
           39: 'PPPR', 40: 'PPPP', 41: 'PPPS', 42: 'PPSR', 43: 'PPSP', 44: 'PPSS
           45: 'PSRR', 46: 'PSRP', 47: 'PSRS', 48: 'PSPR', 49: 'PSPP', 50: 'PSPS
           51: 'PSSR', 52: 'PSSP', 53: 'PSSS', 54: 'SRRR', 55: 'SRRP', 56: 'SRRS
           57: 'SRPR', 58: 'SRPP', 59: 'SRPS', 60: 'SRSR', 61: 'SRSP', 62: 'SRSS
           63: 'SPRR', 64: 'SPRP', 65: 'SPRS', 66: 'SPPR', 67: 'SPPP', 68: 'SPPS
           69: 'SPSR', 70: 'SPSP', 71: 'SPSS', 72: 'SSRR', 73: 'SSRP', 74: 'SSRS
           75: 'SSPR', 76: 'SSPP', 77: 'SSPS', 78: 'SSSR', 79: 'SSSP', 80: 'SSSS
plays={0:'R',1:'P',2:'S'}
```

```
players={'R':0 , 'P':1 , 'S':2}

movesIndex={
            "RRRR": 0,"RRRP": 1,"RRRS": 2,"RRPR": 3,"RRPP": 4,"RRPS": 5,"RRSR":
            "RRSP": 7,"RRSS": 8,"RPRR":9,"RPRP": 10,"RPRS":11,"RPPR": 12,"RPPP":
            "RPPS":14,"RPSR":15,"RPSP":16,"RPSS":17,"RSRR":18,"RSRP":19,"RSRS":2
            "RSPR":21,"RSPP":22,"RSPS":23,"RSSR":24,"RSSP":25,"RSSS":26,"PRRR":2
            "PRRP":28,"PRRS":29,"PRPR":30,"PRPP":31,"PRPS":32,"PRSR":33,"PRSP":3
            "PRSS":35,"PPRR":36,"PPRP":37,"PPRS":38,"PPPR":39,"PPPP":40,"PPPS":4
            "PPSR":42,"PPSP":43,"PPSS":44,"PSRR":45,"PSRP":46,"PSRS":47,"PSPR":4
            "PSPP":49,"PSPS":50,"PSSR":51,"PSSP":52,"PSSS":53,"SRRR":54,"SRRP":5
            "SRRS":56,"SRPR":57,"SRPP":58,"SRPS":59,"SRSR":60,"SRSP":61,"SRSS":6
            "SPRR":63,"SPRP":64,"SPRS":65,"SPPR":66,"SPPP":67,"SPPS":68,"SPSR":6
            "SPSP":70,"SPSS":71,"SSRR":72,"SSRP":73,"SSRS":74,"SSPR":75,"SSPP":7
            "SSPS":77,"SSSR":78,"SSSP":79,"SSSS":80
            }



#function to generate a random population
#This function radomly generates positions
def generate_population(size):
    population=[]
    #while population size hasn't reached n then keep on populating
    #a=time.time()
    while len(population)!= size:
        gene=[]
        #for each chromosome, randomly select between R,P,S
and populate gene
        for i in range(0,81):
            gene.append(random.choice(['R', 'P', 'S']))
        #add  The gene into the population
        population.append(gene)
    #b=time.time()

    return population  #RETURN POPULATION



#This function is used to generate a fitness for a single individual
#How it works is that since the individual previous moves are orderd by a DFS ex
 #tree level 4, We know the order of the 81 genes of the invidual/chromosome. we
 #previous moves of the data set and previous moves of the gene. if they are equ
 #the divider, we then go futher by comparing the move of the dataset and the mo
 #Inividual/Gene, If they are equal increase the fitscore. At the end we normali
 #score by dividing the fitscore by the divider.
 #
def Singlefitness(Individual):#81 length
    fitscore=0.0    #this is to keep the fitness score before normalisation

    #print(len(inputs))


    #print(target)
    for i in range(0,len(Individual)):

        fitscore+=database[(indexmoves[i],plays[players[Individual[i]]])]

    return fitscore/1000000.0
      #  divider+=


#This function generates a fitness of the whole population and returns a normali

def Fitness(population):
    score=[]    #The list to keep the normalised fitness score of all the indivi
    #The population


    for ind in population:


        #For every individual in the population
        score.append(Singlefitness(ind))#get normalised score and add it the sco

    return score #return a list of score for  whole population



#This function sets a cut of threshold by fitness and illimates the genes that d
#this threshold. It however adds to biasing genes to avoid local minimums

def selection(population, threshold):
    newpop=[] #this is for storing the new population
    score=Fitness(population) #the fitness score of the current population
    lenpop=len(population)

    lenscore=len(score)-1
    bias=0
    i=0
    while (len(newpop)<lenpop):# while a new popluation isnt full
        #m=max(score)#to save the max score
        if(Singlefitness(population[i])>threshold ):
            newpop.append(population[i])#add to new population
            i+=1
        elif(lenpop*0.50>=bias):
            g=generate_population(1)
            newpop.append(g[0])#add to new population
            i+=1
            bias+=1
        else:
            newpop.append(population[i])#add to new population
            i+=1

    return  newpop

#This functions just returns the cross over point randomly
def crosspoint(individual):
    crosspoint= random.randint(0,len(individual)-1)
    return crosspoint; #returns the crossover point

#This function returns two individuals after crossover (reroduce)
def reproduce(Individual1,Individual2):
        temp1=Individual1
        temp2=Individual2
        fit1=Singlefitness(temp1)
        fit2=Singlefitness(temp2)
        x=crosspoint(Individual1)
        low=Individual1[x:]
        high= Individual2[x:]
        Individual1[x:]=high
        Individual2[x:]=low
        fit3=Singlefitness(Individual1)
        fit4=Singlefitness(Individual2)

        return Individual1, Individual2


#This function mutates every gene in the population
def Mutation(population,threshold):
    m=max(Fitness(population))
    for i in population:
        z=Singlefitness(i)
        if(z>threshold or z==m ):
            continue
        else:
            a=random.randint(0,80)
            i[a]=random.choice(['R', 'P', 'S'])

    return population
#This function runs the Gentic algorthm cylce once
def GenticAlg(Population,threshold,mutationRate,crossRate):
    #creation of a new population
    #print len(Population)
    lenpop=len(Population)
    #print lenpop
    newpoplation=selection(Population, threshold)
    #print len(newpoplation)
    Crosspopulation=[]



    lennewpop=len(newpoplation)-1

    c= random.randint(0,100)
    w=max(Fitness(newpoplation))
    if(c<crossRate):
         while(len(Crosspopulation)<lenpop  ):

             i= random.randint(0,lennewpop)
             j=random.randint(0,lennewpop)
             a=newpoplation[i]
             b=newpoplation[j]
             if(Singlefitness(a)==w):
                  Crosspopulation.append(a)
                  a,b=reproduce(a,b)
                  Crosspopulation.append(b)
             elif(Singlefitness(b)==w ):
                 Crosspopulation.append(b)
                 a,b=reproduce(a,b)
                 Crosspopulation.append(a)
             else:
                  a,b=reproduce(a,b)
                  Crosspopulation.append(a)
                  Crosspopulation.append(b)


    else:
        Crosspopulation=newpoplation
        z=random.randint(0,100)
        if(z<=mutationRate):
            return Mutation(Crosspopulation,threshold)
        else:
            return Crosspopulation


#this is for the training the agent
def training(Population,threshold,mutationRate,crossRate):

    G=GenticAlg(Population,threshold,mutationRate,crossRate)
    store= copy.deepcopy(G)#for storing the the max generation

    storefitness=Fitness(G) #for storing the fitness of the max generation
    maxfit=max(storefitness)+0.01 #for storing the current generation max fitnes
    storemax=maxfit#for storing the max fitness of the max genration
    global generations
    generations=0
    #print len(b)
    n=0

    while(maxfit<1 and n<=2 and generations<40 ):#2+9
        #x=maxfit

        G=GenticAlg(G,threshold,mutationRate,crossRate)

        fitness=Fitness(G)

        #print len(b)
        maxfit=max(fitness)
        maxfit=maxfit/1.0
            #print len(store)
        generations+=1
        mean= (sum(fitness)/len(fitness))/1.0
        #print len(store)

        xaxis.append(generations)
        yaxis.append(maxfit*100)
        #plt.plot(generations,maxfit*100)
        #print (maxfit)

        if((mean/maxfit)>0.999):
            n+=1
        else:
            n=0
            #continue
        #Sprint "n: %d",  n

        if(storemax<maxfit):#temporary storing the generation that gave us the m
            #print storemax,"<",maxfit
            storemax=maxfit
            store=[]
            store=copy.deepcopy(G)


    return G, store


##indv=Popu[0]
#print indv
def getpos(pop):
    val=max(Fitness(pop))
    i=0
    while(val!=Singlefitness(pop[i])):
        i+=1

    return i



xaxis=[]
yaxis=[]


#this is for testing results
x=time.time()


s,m=training(generate_population(200),0.4,25,30) #0.0002 ,mutation rate , cross
    #xaxis.append(mutations)


z=time.time()

maxfit=max(Fitness(m))
#xaxis.append(pop)
#yaxis.append(w)


    #print("time:",z-x)

print ("this is for last generation : ")
print (Fitness(s))

#print s
print( "This is for the maximum generation")

print (maxfit)
t=getpos(m)
print (m[t])


#this is for the plots
#plt.xlabel('Generations')
##plt.ylabel('Fitness')
#plt.yscale('log')
#plt.plot(xaxis,yaxis)
#plt.show()



#Popu= [['R', 'P', 'S', 'S', 'S', 'P', 'P', 'R', 'S', 'R', 'S', 'P', 'S', 'P', '

def winningmove(move):
    if(move=="R" ):
        return "P"
    elif(move=="P"):
        return "S"
    else:
        return "R"
def determinepos(pop):
    fit=Fitness(pop)
    maxfit=max(fit)
    i=0
    while fit[i]!= maxfit:
        i+=1
    return i

def Stringcreater(moves):
    string=''
    for i in moves:
        string+=i
    return string
#print Stringcreater(inputs[0])

#this is for game play functions
"""
if input == '':
    inputs=[]
    target=[]
    history = ['X']*4
"""
"""
    pop=[['R', 'P', 'S', 'S', 'S', 'P', 'P', 'R', 'S', 'R', 'S', 'P', 'S', 'P',
         'S', 'P', 'S', 'R', 'R', 'S', 'R', 'R', 'P', 'R', 'P', 'P', 'R', 'P',
         'R', 'P', 'P', 'S', 'S', 'S', 'P', 'R', 'R', 'R', 'P', 'R', 'S', 'P',
         'S', 'P', 'S', 'R', 'P', 'P', 'P', 'R', 'R', 'P', 'S', 'R', 'R', 'R',
         'S', 'R', 'S', 'R', 'P', 'P', 'S', 'S', 'S', 'S', 'R', 'S', 'P', 'S',
         'P', 'S', 'R', 'S', 'R', 'P', 'P', 'R', 'P', 'R', 'S'], ['R', 'P', 'S'
         'S', 'S', 'P', 'P', 'R', 'S', 'R', 'S', 'P', 'S', 'P', 'S', 'P', 'S',
         'R', 'R', 'S', 'R', 'R', 'P', 'R', 'P', 'P', 'R', 'P', 'R', 'P', 'P',
         'S', 'S', 'S', 'P', 'R', 'R', 'R', 'P', 'R', 'S', 'P', 'S', 'P', 'S',
         'R', 'P', 'P', 'P', 'R', 'R', 'P', 'S', 'R', 'R', 'R', 'S', 'R', 'S',
         'R', 'P', 'P', 'S', 'S', 'S', 'S', 'R', 'S', 'P', 'S', 'P', 'S', 'R',
         'S', 'R', 'P', 'P', 'R', 'P', 'R', 'S'], ['S', 'P', 'R', 'R', 'P', 'R'
         'P', 'R', 'S', 'S', 'P', 'S', 'S', 'R', 'S', 'S', 'R', 'P', 'R', 'R',
         'P', 'S', 'R', 'P', 'S', 'P', 'P', 'P', 'S', 'P', 'P', 'P', 'S', 'R',
         'S', 'P', 'P', 'R', 'P', 'R', 'S', 'S', 'R', 'R', 'P', 'S', 'P', 'P',
         'P', 'P', 'S', 'P', 'S', 'R', 'R', 'R', 'P', 'P', 'P', 'S', 'R', 'R',
         'R', 'R', 'S', 'P', 'S', 'P', 'P', 'S', 'S', 'S', 'S', 'R', 'P', 'S',
         'S', 'R', 'S', 'P', 'R'], ['P', 'P', 'R', 'S', 'R', 'S', 'R', 'P', 'R'
          'R', 'P', 'P', 'R', 'S', 'P', 'S', 'R', 'S', 'R', 'R', 'S', 'S', 'P',
          'P', 'P', 'R', 'R', 'S', 'R', 'P', 'P', 'R', 'S', 'S', 'S', 'R', 'P',
          'R', 'R', 'S', 'S', 'R', 'S', 'S', 'S', 'S', 'R', 'S', 'S', 'P', 'P',
          'P', 'S', 'R', 'R', 'R', 'P', 'S', 'R', 'S', 'R', 'R', 'S', 'R', 'P',
          'R', 'R', 'R', 'P', 'P', 'S', 'S', 'S', 'S', 'P', 'S', 'S', 'S', 'P',
          'S', 'S']]
    """
    #pop=[['R', 'P', 'S', 'S', 'S', 'P', 'P', 'R', 'S', 'R', 'S', 'P', 'S', 'P',
"""
    pop=generate_population(10)
    #fitnessfn=fitness(population,'R')
    MutationRate=30
    CrossoverRate= 25
    thresh=0.0001
    generationz=0
    move = np.random.choice(['R', 'P', 'S'])
    games=0

else:
    if(games<2):
        history.pop(0)
        history.append(input)
        move=np.random.choice(['R', 'P', 'S'])
```

```
            history.pop(0)
            history.append(move)
            games+=1
        else:
            #file1=open("MyFile.txt","a")
            string=Stringcreater(history)#generate the history string
            inputs.append(string)#gathering a data set for prev moves
            target.append(winningmove(input))# gathering a dataset for winning moves
            #z,pop=copy.deepcopy(training(pop,thresh,MutationRate,CrossoverRate))
            for i in range(0,1):
                pop=GenticAlg(pop,thresh,MutationRate,CrossoverRate)

            generationz+=1
            pos=determinepos(pop)
            #file1.write(pop[pos] )
            response=copy.deepcopy(pop[pos])
            move=copy.deepcopy(response[movesIndex[string]])
            history.pop(0)
            history.append(input)
            history.pop(0)
            history.append(move)
            #file1.close()

    output=move
"""
```