Contents

1 Introduction

This report will discuss the implementation of a genetic algorithm designed to beat a group of specialised bots by training against a large CSV data set. The genetic algorithm should be programmed on python 3 and should make use of common genetic algorithm techniques such as selection, crossover and mutation.

1.1 Aim of the genetic algorithm

The genetic algorithm has 1 main goal and 4 desirable peripheral goals. The main aim of the genetic algorithm is to win as many matches as possible against the other specialised bots. The other 4 sub goals are: the genetic algorithm should reach a specified fitness within a reasonable amount of time; the algorithm should also not produce too much over head; win as many rounds as possible and create a gene sequence that is a created from the beating the given data sets.

1.2 Genetic algorithm requirements

The genetic algorithm must build a gene sequence of length 81 that the agent.py program will use to play against the other bots, the gene sequence must be saved to a text file. This gene sequence must be created by a Python program that uses the CSV data sets 1 and 2 respectively as a fitness function for the genetic algorithm. The algorithm should at least make use of some form of crossover and mutation techniques when reproducing.

1.3 Expected outcome

The genetic algorithm is expected to produce a gene sequence for each data set. These gene sequences are expected to win more than 92% of matches against the pool of specialised bots. Ultimately the genetic algorithm is expected to converge to the global maximum of the fitness function generated by the CSV.

2 Theoretical background

2.1 What are genetic algorithms

Genetic algorithms are optimisation algorithms that are used for non linear problems that can not be solved using differentiation optimisation techniques. Genetic algorithms are based off of the evolutionary techniques of nature. The algorithm starts with a initialised population of any given size. This population is then tested against a fitness function, the most fit individuals are then used for the creation of the next generation by using selection, crossover and mutation techniques.This process repeats until the population or individual reaches or converges to the desired fitness.[1][2]

2.2 Local vs Global maximum

When any population is initialised in a variable plain "work space" the population runs the risk of converging to a local maximum. A local maximum is a point on the variable plain where any action by the individual reduces the fitness of the individual. The problem is the individual may be able to exit the local maximum but it will need to reduce its fitness for a certain amount of generations before working up again. When the population get stuck in a local maximum they think they have reached peak fitness but in reality they haven't since there is another peak that is better thus it is the global maximum.[2][3]

2.3 Implemented methodology

To ensure the genetic algorithm does not get stuck in a local maximum or starts to stagnate a portion of the population may be randomised and mutated at a certain rate per generation.

3 Methodology

The program makes use of basic genetic algorithm techniques such as selection, crossover and mutation. The fitness function is based off of the compressed CSV data sets 1 and 2, the fitness function thus consists of a 81 gene sequence that the individuals play against to measure their fitness.[3]

3.1 Overall program structure

The program is created with modulation and ease of manipulation in mind. The parameters that the genetic algorithm uses should be changed easily to test different variable configurations. The program is also split into 3 main parts: CSV manipulation; population initialisation; reproduction through crossover and mutation and convergence to goal fitness.

Firstly the CSV data sets are compressed to reduce overhead for the program when the fitness of an individual is calculated. The CSV data set is compressed into a 81 gene sequence array, each node in the gene corresponds to the history of the data set. For example the history RRRR corresponds to the 0 index (1^{st} node) in the CSV_OPTIMAL array and SSSS corresponds to the 80^{th} index (81^{st} node) in the array. Also during compression the program tallies the amount of different responses to a given history in the data set. For example RRRR has 98000 R response 2000 P responses and 0 S responses. This process is done for each history in the data set. Finally the response that occurred the most for a given history are entered into the corresponding position in the CSV_OPTIMAL array. In the case discussed above index 0 of the array will hold the R response as the R responses were occurred the most given the RRRR history.

After the compression is complete the CSV_OPTIMAL array essentially acts as the fitness function, each individual is tested against this array and is awarded a fitness rating accordingly. A set portion of the most fit individuals within the population is selected as the elites and are just "copied" into the next generation.Another set portion of the population is then a product of crossover between 2 distinct elite individuals. Each gene in these individual's gene sequences have a chance of mutating, not all mutations will change the mutated gene since the mutation is preformed by randomly changing into R,P or S, this means the gene can mutate into the same response it already is.Lastly the remaining portion of the new generation is a product of randomness since it consists of essentially reinitialised individuals. This ensures the population will never get stuck in a local maximum.

This process is repeated until an individual reaches the desired fitness predetermined in the program.

3.2 Definition functionality

This subsection examines each function within the code, the aim is to explain how each of these functions contributes to the genetic algorithm.

3.2.1 Compression

The compression function has two goals: reduce the overhead of the entire program as well as the fitness calculation process and to reduce the data sets provided into a more manageable form. As explained above data set is reduced to an array of length 81. Each history component from RRRR to SSSS corresponds to a location in the array. As the data from the CSV is worked through the amount of responses for each history is tallied. The tallies are broken up into 3 parts: R, P and S. The responses that are the most present for any given history is then placed into the appropriate slot in the array. Ultimately this process creates a optimal array called CSV_OPTIMAL that acts as the objective function. The overhead is also reduced since the individuals don't need to go through the entire data set each time. Even though this process relies on probability since each history has responses that are proportional to each other it still acts as a good fitness function.

3.2.2 Population initialisation

In the beginning of the program the population is initialised with a specified amount of child objects. Each child object consist of a gene sequence, a fitness and an index. The first generation of the population is made up of child object with randomly generated gene sequences, this acts as a starting point for the population. The population is manipulated by using an array this array thus holds all of the children within any given generation.

3.2.3 Calculating fitness

The fitness of an individual is determined quite easily. Each individual's gene sequence is compared to the CSV_OPTIMAL array one index (gene) at a time. If the individual's gene beats the CSV's corresponding gene the fitness of the individual is increased by 1 on the other hand if the CSV's gene beats the individual's gene then the individual's fitness is reduced by 1 (nothing chances if they are the same). Thus an individual can have a maximum fitness of 81. Once every individual goes through the fitness calculation process the individuals are ranked and arranged from fittest to least fit within the population array, this allows for easier reproduction in the crossover and mutation function.Take note that the gene sequence's fitness is retained not a single gene's. Thus the individual is judged as a whole.

3.2.4 Selection

The selection process is broken up into 3 part: the elite individuals, individuals that are the product of crossover as well as mutation and the reinitialised individuals. Each of these 3 parts form a portion of the entire population. The elite individuals are selected from the portion of the population with the highest fitness. This portion is not more than 10% and is not less than 2%. The elite individuals are essentially copied into the new generation, they are not altered in any way. (The population also needs more than 1 elite individual to ensure crossover can take place.)

This ensures that the population always moves towards a higher fitness or at least stagnates since since the next generation can not have a fittest individual weaker than the previous generation's fittest individual. The second portion of the crossover/mutation individuals make up 60% to 70% of the population. The final part of the population consists of reinitialised individuals, these individuals have gene sequences that are randomly generated. This portion of the population acts as the safety net since this portion of the population will never get stuck in a local minimum. It can act as a break out individual that may randomly have a higher fitness as the elite individuals breaking stagnation. This is why this portion of the population is so important. Even though it will take a long time there is a 1 in 3^{81} chance that an individual will randomly initialise with the perfect sequence.

3.2.5 Crossover

The process of creating an individual through crossover has three stages. Firstly ever individual is a product of 2 distinct elite individuals, thus for every individual a pair of elite individual parents are chosen at random. Then the crossover point is determined, the crossover point can be a random one of three predetermined points: index 26, index 41 or index 54 of the gene sequence array. This method is used since it introduces some randomness to the crossover process and reduces the amount of redundant individuals within the population. All in all this increases diversity within the cross over subset of the population. Lastly the first parent fills up the child's gene sequence up to and including the crossover point. Once the first parent is done the second parent fills in the rest but it starts just after the crossover point. Crossover helps the population progress towards a greater fitness by combining genes of the elite, each crossover individual ultimately has a good foundation to work off of. The idea is to potentially combine the best aspect of the two parents into one child.

3.2.6 Mutation

The mutation aspect of the genetic algorithm is very basic. The only portion of the population that experiences mutation is the individuals that are a product of crossover. As the crossover individuals are created each index (gene) within the gene sequence has a fixed percentage chance of mutating. Thus every gene within the gene sequence has a chance of mutating. The only catch is the mutation is random since it only has a set percentage of happening and a gene can mutate but produce the same response since the mutate does not stipulate whether or not if a mutation is triggered the gene must change into one of the other options. Ultimately the genetic algorithm is left with this caveat since it merely reduces the mutation rate. For example if a gene has a 10% chance of mutating it actually has a 6.66% chance of changing it's response at the gene.

4 Results

The result are gathered from 2 sources: the pyplot graphs that are generated after every successful convergence of the gene sequence and win rates that are achieved by the sequence when it has played a set number of rounds and matches against the bots.

4.1 Analysing the pyplots

During the testing of the genetic algorithm many interesting characteristics of a typical genetic algorithm became evident. These characteristics effected many aspects of the program, some of these effects altered the speed of convergence, increased the programs running over head, caused long term stagnation and even resulted in non-convergence.

In order to show how different variable effected the genetic algorithm a benchmark needed to be set that will act as the control in terms of variable values. After many iterations became evident that best benchmark variable values are:

Population size = 100
Mutation rate = 5%
Elite-portion = 10%
Crossover portion = 70%
Convention fitness = 80 to 81 (98,2% to 100%) As the graph above illustrate the benchmark

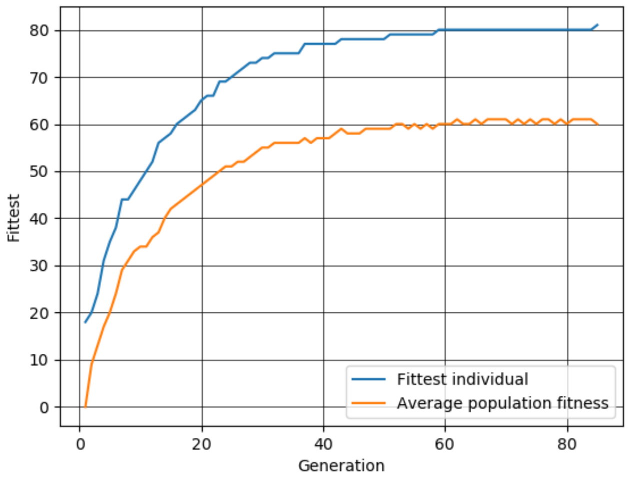

Figure 1: Bench mark graph that plots the fittest individual and average population fitness for each generation.

converges to a optimal gene sequence within a respectable 90 generations even though the population is so small. If the main concern was overhead this would be a very good result for a convergence rate since the overhead is miner and the generation count is respectable. Over all 9000 individuals are created. The main issue of the benchmark is the huge slow down in fitness progression after 40 generations, it take just as much time to increase it fitness by 6 within the last 50 generation then it does to increase its fitness by 55 within the first 40 generations. The factors that might cause this long stagnation might be the small population size, small mutation rate or small initialisation pool.

The results above can now act as a good starting point. Firstly the effects of changing the population sizes should be explored thus the only variables changed in the 2 graphs below are the population sizes.

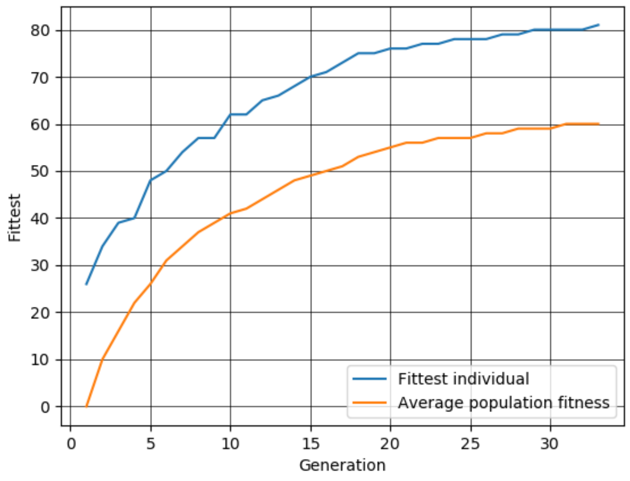

Figure 2: The population size is 1000 per generation

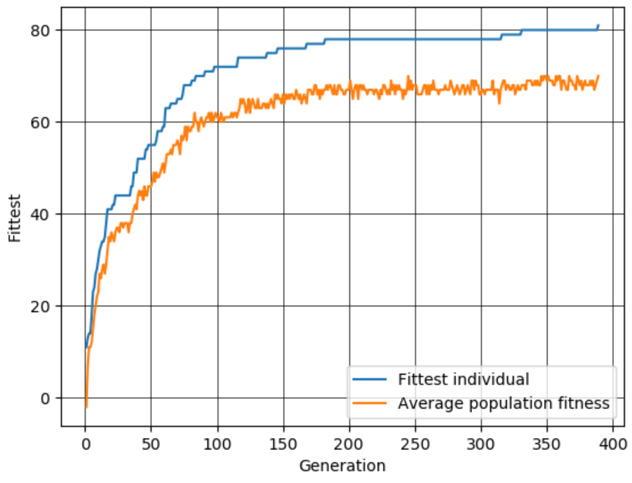

Figure 3: The population size is 10 per generation.

It is clear to see that the population size is a very clear indication of the speed of convergence. When the population per generation is increased to a 1000 the amount of generations needed for a complete convergence is only 37, this is less than half of the 100 population simulation. Never the less when the population is only 10 per generation it takes about 375 generations for full convergence. This is more than 4 times longer than the 100 population simulation. Even though the speed of convergence increases as the population size increases the over head also increases.

It took 36000 individuals for the 1000 population simulation and only 3750 individuals for the 10 population simulation. It can thus be assumed that the larger the population the faster the convergence but also the larger the overhead for the computer.

The next variable to consider is the mutation rate, thus revert back the population size to 100 to ensure the mutation rate is isolated.

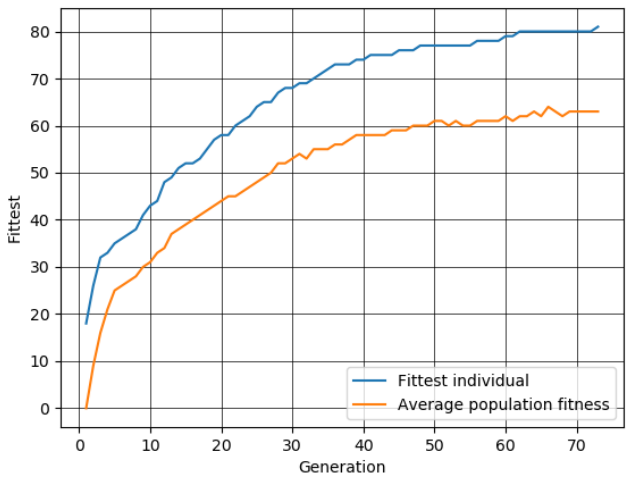

Figure 4: The mutation rate is 1%.

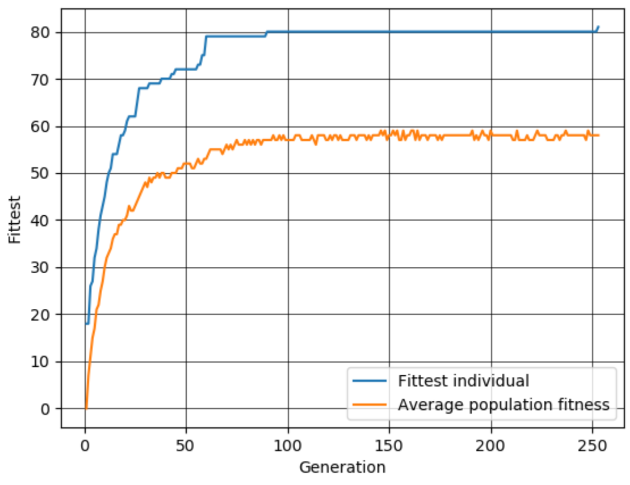

Figure 5: The mutation rate is 10%.

Interestingly when the mutation rate is reduced the amount of time that the genetic algorithm takes to converge is not effected that much as indicated above, the convergence falls within the 75- 95 generation range thus a smaller mutation rate may or may not be beneficial. On the other hand when the mutation rate is increased to 10% the convergence take much longer. This might be due to the fact that the data is becoming scrabbled and the purpose of the crossover technique is lost since the preferred genes have a 10% chance of being altered. These results indicate that an increased mutation rate also increases the chaotic nature of the system and reverts it back to a random chance system, since the elite portion of the population will ensure a positive gain it will converge but it will take longer the higher the mutation rate is, At 33% mutation rate the system basically becomes a random chance system and the genetic algorithm is destroyed.

Lastly the proportions of the various population sections can be altered.

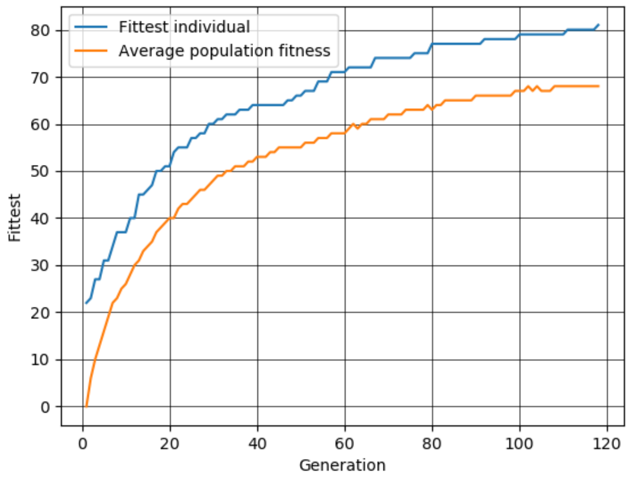

Figure 6: The elite population makes up 30% and the crossover portion makes up 40%

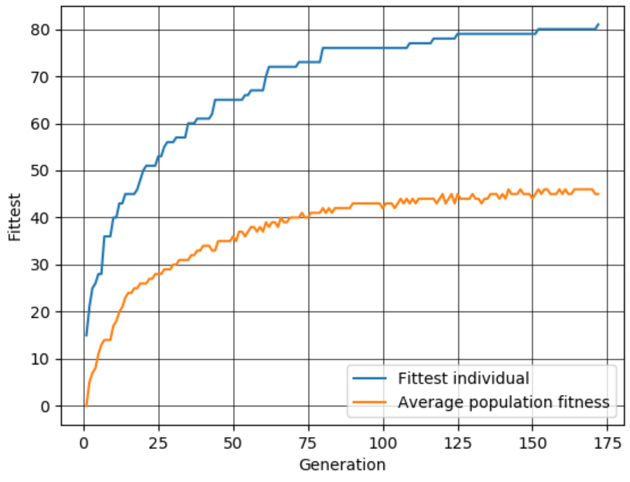

Figure 7: The elite population makes up 30% and the crossover portion makes up 30%

As expected altering these variables have little effect on the system as a whole, but it is clear that the elite portion of the population should not be too large since you are only wasting a portion of the population as it takes up space. Interestingly when the portions become very similar the system takes longer to converge and the gap between the strongest individual and average individual increases. When considering all of the possible variable configurations a trend emerges: when the gap between the strongest and average individuals increases the system is moving to a system that will not (will take long) to converge. This can be seen in all three configuration pairs,

4.2 Competition results

Even though all the configurations mentioned above have different converging rate and over head they all resulted in the same gene sequences for Data set 1 and 2. They also all ensure a match win rate of above 92%. Also note worthy is the fact that a 85 convergence has produced multiple gene sequences that have won 100% of matches against the pool of bots. This might indicate that the CSV data sheets do not have the optimal gene sequences.

The gene sequence produced using Data1.csv:
P,S,S,P,R,S,P,P,R,R,S,P,S,S,P,P,S,R,R,S,R,S,R,R,S,P,R,P,R,S,P,P,R,P,S,P,P,S,S,R,S,R,P,S,P,P,S,R,P,R,R,
R,R,R,P,S,R,P,P,P,P,R,R,S,S,S,R,S,S,P,S,R,S,S,R,R,P,R,P,P,R

The gene sequence produced using Data2.csv:

P,S,P,P,R,R,P,P,P,R,S,P,S,S,P,P,S,S,R,S,R,R,R,R,S,P,R,P,P,S,P,P,R,P,P,P,R,S,S,P,S,R,R,S,P,R,R,R,S,R,R,
R,R,R,P,R,R,P,S,S,P,P,R,S,S,S,R,S,S,P,S,S,S,R,R,R,P,R,P,S,R

Both of the sequences produce on average a 94% match win rate.

5 Discussion

Throughout the report the implementation and results have been discussed and examined. The implementation of the selection, crossover and mutation techniques used throughout the genetic algorithm is very simple to deconstruct. As mentioned in the methodology section the selection makes use of elite individuals, the crossover produces a individual by combining two elite individuals and mutating their genes by a fixed percentage

During the planning phase of the genetic algorithm many different configurations of variables were considered when designing the pseudo-optimal genetic algorithm, ultimately the best configuration was the bench mark conditions excluding the population size. A population size of 1000 is better since it halves the amount of generations needed for the system to fully converge and the overhead is not to large, any computer can handle the strain. Once the population increases past the over head and the time it takes to complete outways the small amount of generation count reduction.

On the other hand if the generation count must be as low as possible the population size can be increased indefinitely. Mathematically if the population is 3^{81} large theoretically the genetic algorithm will converge within the first generation every time.

Lastly even though the annealing technique was suggested it is not really necessary for the current genetic algorithm and problem since the rate of convergence of the genetic algorithm does not differ much with different mutation rates (if small) also the reinitialised population proportion allows for greater increases in fitness early on.

6 Conclusion

The report demonstrates the planning, design and implementation process when using a genetic algorithm to solve a problem. The results obtained from the graphs and match win rates of the gene sequences indicate that the genetic algorithm converged correctly using the CSV_OPTIMAL array as the fitness function. All in all the gene sequences produced by data set 1 and 2 produced favourable results when they are used in the game against all of the specialised bots, as the average win rate is 94%. Another point of note is that the population size has a huge impact on the rate of convergence to the global maximum, thus the population should always be considered when designing a genetic algorithm.

7 References

(1) MATLAB, YouTube. (2020). What is a Genetic Algorithm. [online] Available at: https://www. youtube.com/watch?v=1i8muvzZkPw [Accessed 2 Mar. 2020].
(2) Alldatasheet.com. (2020). PT331C Datasheet(PDF) - Everlight Electronics Co., Ltd. [online] Available at: https://www.alldatasheet.com/datasheet-pdf/pdf/91033/EVERLIGHT/PT331C.html [Accessed 2 Mar. 2020].
(3) Alldatasheet.com. (2020). PT331C Datasheet(PDF) - Everlight Electronics Co., Ltd. [online] Available at: https://www.alldatasheet.com/datasheet-pdf/pdf/91033/EVERLIGHT/PT331C.html [Accessed 2 Mar. 2020].

8 Appendix I

```python
# Author:   H. van der Westhuizen, u18141235
# Date:     28 February 2020
# Revision: 29 February 2020
# Genetic algorithm.

# Data set 1 output sequence:
# P,S,S,P,R,S,P,P,R,R,S,P,S,S,P,P,S,R,R,S,R,S,R,R,S,P,R,P,R,S,P,P,R,P,S,P,P,S,S,R,S,R,P,
# S,P,P,S,R,P,R,R,R,R,R,P,S,R,P,P,P,P,R,R,S,S,S,R,S,S,P,S,R,S,S,R,R,P,R,P,P,R
# Data set 2 output sequence:
# P,S,P,P,R,R,P,P,P,R,S,P,S,S,P,P,S,S,R,S,R,R,R,R,S,P,R,P,P,S,P,P,R,P,P,P,R,S,S,P,S,R,
# R,S,P,R,R,R,S,R,R,R,R,R,P,R,R,P,S,S,P,P,R,S,S,S,R,S,S,P,S,S,S,R,R,R,P,R,P,S,R

import csv
import random
from operator import attrgetter
from matplotlib import pyplot as plt

optimal_CSV = []  # This array holds the compressed 81 length sequence of the data sets.
pop_size = 100  # This is the population size of each generation.
population = []  # This array holds the child objects of each individual within the current generation.
beat = {"R": "P", "P": "S", "S": "R"}  # This is used to determine the best move to play against the opponent.
bestChildArray = []  # This array holds the fittest child for every generation in array of size: total generations. Plot
averageChildArray = []  # This array holds average fitness of a population every generation.(Plot)


# The class below creates the object of the child/individual. It holds the individuals fitness, gene sequence and
# index.
class child():
    def __init__(self, childIndex=0):
        self.fitness = 0  # This is the total fitness of the child/ individual.
        self.gene_sequence = []  # This is the child gene
            sequence of length 81.
        self.childIndex = childIndex  # This index is used
            to keep track of certain individuals and used in
            sorting.


# This dictionary is used to read from the CSV files during
    the construction of the optimal_CSV array.
dictionary = [['R', 'R', 'R', 'R'], ['R', 'R', 'R', 'P'],
              ['R', 'R', 'R', 'S'], ['R', 'R', 'P', 'R'],
              ['R', 'R', 'P', 'P'], ['R', 'R', 'P', 'S'],
              ['R', 'R', 'S', 'R'], ['R', 'R', 'S', 'P'],
              ['R', 'R', 'S', 'S'], ['R', 'P', 'R', 'R'],
              ['R', 'P', 'R', 'P'], ['R', 'P', 'R', 'S'],
              ['R', 'P', 'P', 'R'], ['R', 'P', 'P', 'P'],
              ['R', 'P', 'P', 'S'], ['R', 'P', 'S', 'R'],
              ['R', 'P', 'S', 'P'], ['R', 'P', 'S', 'S'],
              ['R', 'S', 'R', 'R'], ['R', 'S', 'R', 'P'],
              ['R', 'S', 'R', 'S'], ['R', 'S', 'P', 'R'],
              ['R', 'S', 'P', 'P'], ['R', 'S', 'P', 'S'],
              ['R', 'S', 'S', 'R'], ['R', 'S', 'S', 'P'],
              ['R', 'S', 'S', 'S'], ['P', 'R', 'R', 'R'],
              ['P', 'R', 'R', 'P'], ['P', 'R', 'R', 'S'],
              ['P', 'R', 'P', 'R'], ['P', 'R', 'P', 'P'],
              ['P', 'R', 'P', 'S'], ['P', 'R', 'S', 'R'],
              ['P', 'R', 'S', 'P'], ['P', 'R', 'S', 'S'],
              ['P', 'P', 'R', 'R'], ['P', 'P', 'R', 'P'],
              ['P', 'P', 'R', 'S'], ['P', 'P', 'P', 'R'],
              ['P', 'P', 'P', 'P'], ['P', 'P', 'P', 'S'],
              ['P', 'P', 'S', 'R'], ['P', 'P', 'S', 'P'],
              ['P', 'P', 'S', 'S'], ['P', 'S', 'R', 'R'],
              ['P', 'S', 'R', 'P'], ['P', 'S', 'R', 'S'],
              ['P', 'S', 'P', 'R'], ['P', 'S', 'P', 'P'],
              ['P', 'S', 'P', 'S'], ['P', 'S', 'S', 'R'],
              ['P', 'S', 'S', 'P'], ['P', 'S', 'S', 'S'],
              ['S', 'R', 'R', 'R'], ['S', 'R', 'R', 'P'],
              ['S', 'R', 'R', 'S'], ['S', 'R', 'P', 'R'],
              ['S', 'R', 'P', 'P'], ['S', 'R', 'P', 'S'],
              ['S', 'R', 'S', 'R'], ['S', 'R', 'S', 'P'],
              ['S', 'R', 'S', 'S'], ['S', 'P', 'R', 'R'],
              ['S', 'P', 'R', 'P'], ['S', 'P', 'R', 'S'],
              ['S', 'P', 'P', 'R'], ['S', 'P', 'P', 'P'],
              ['S', 'P', 'P', 'S'], ['S', 'P', 'S', 'R'],
              ['S', 'P', 'S', 'P'], ['S', 'P', 'S', 'S'],
              ['S', 'S', 'R', 'R'], ['S', 'S', 'R', 'P'],
              ['S', 'S', 'R', 'S'], ['S', 'S', 'P', 'R'],
                  ['S', 'S', 'P', 'P'], ['S', 'S', 'P', 'S'],
                  ['S', 'S', 'S', 'R'], ['S', 'S', 'S', 'P'],
                  ['S', 'S', 'S', 'S']]


# The function below compresses the information in the data sets of 1 and 2. This allows for less overhead and
# ease of comparison. Ultimately the compressed array holds an 81 gene sequence that will help in determining
# the fitness of the individuals.
def compressData():
    # This for loop creates the 81 gene array containing the csv file, History and amount of a response, R=0 , P=0 ,S=0
    global optimal_CSV
    for i in range(81):
        optimal_CSV.append([dictionary[i], 0, 0, 0])
    # The code below allows the code too read from the data CSV files row by row which is divided by (,).
    with open('data1.csv') as csv_data:
        readCSV = (csv.reader(csv_data, delimiter=','))
        # The for loop below goes through all of the rows (1000000) and counts the R,P and S resp for each history.
        for row in readCSV:
            CSV_History = list(row[0])  # This variable stores the History (last four moves) such as RRRR.
            CSV_Response = list(row[1])[0]  # This variable stores the Response given the history such as R or P.
            # The if statements below check the history and increment the R,P,S counter for the given history and resp
            # For example [RRRR,9000,200,0].The given history thus shows the responses were 9000 R, 200 P and 0 S.
            if CSV_Response == 'R':
                optimal_CSV[dictionary.index(CSV_History)][1] = optimal_CSV[dictionary.index(CSV_History)][1] + 1
            elif CSV_Response == 'P':
                optimal_CSV[dictionary.index(CSV_History)][2] = optimal_CSV[dictionary.index(CSV_History)][2] + 1
            else:
```

```python
                optimal_CSV[dictionary.index(CSV_History)][3] = optimal_CSV[dictionary.index(CSV_History)][3] + 1
    # The for loop below now just creates the 81 length gene gathered from the CSV file, this will be used in the
    # calculation of the GA's fitness. It is constructed according to the amount of R,P and S responses were noted
    # per history.
    for i in range(81):
        if (optimal_CSV[i][1] > optimal_CSV[i][2]) and (optimal_CSV[i][1] > optimal_CSV[i][3]):
            optimal_CSV[i] = 'R'
        elif (optimal_CSV[i][2] > optimal_CSV[i][1]) and (optimal_CSV[i][2] > optimal_CSV[i][3]):
            optimal_CSV[i] = 'P'
        elif (optimal_CSV[i][3] > optimal_CSV[i][1]) and (optimal_CSV[i][3] > optimal_CSV[i][2]):
            optimal_CSV[i] = 'S'
        else:
            optimal_CSV[i] = random.choice(["R", "P", "S"])  # Not all histories are represented thus its randomised.


# The function below initialises the population by creating random children/individuals that will represent
# the first generation. This is done by randomising their gene sequences.
def initialisePopulation():
    global pop_size  # This is the population size (number of children)
    global population
    #  Below the population is created with the size of pop_size and consists of child objects.
    population = [child() for i in range(pop_size)]
    # The for loop below initialises the entire population by randomising there gene_sequence and clears fitness.
    for x in range(len(population)):
        for y in range(81):
            population[x].gene_sequence.append(random.choice(["R", "P", "S"]))
        population[x].fitness = 0
        population[x].childIndex = x

```

```python

# This function is used to calculate each child/individuals
    fitness by comparing there entire gene sequence to the
# optimal_CSV gene sequence. The population is then sorted
   according to there fitness which will add the crossover
# and mutation stages.
def calculateFitness():
    global pop_size
    global bestChildArray
    global averageChildArray
    global population
    averageChildCounter = 0
    # The for loop below calculates the fitness of each
       child. By comparing each of the child's genes to the
       "optimal
    # gene" (optimal_CSV) in the sequence.
    for x in range(pop_size):
        fitnessCounter = 0  # This variable keeps track of
           the child's fitness.
        for y in range(81):
            # If the child's gene beats the optimal gene
               then it increases its fitness by 1 and if it
               is beaten it
            # decreases by 1.
            if population[x].gene_sequence[y] == beat[
               optimal_CSV[y]]:
                fitnessCounter += 1
            elif optimal_CSV[y] == beat[population[x].
               gene_sequence[y]]:
                fitnessCounter -= 1
        population[x].fitness = fitnessCounter
    # Here the population is ranked and sorted by their
       fitness. It is ranked from best to worst. This
       process will
    # aid during crossover and mutation.
    population = sorted(population, key=attrgetter("fitness
       "), reverse=True)
    bestChildCounter = population[0].fitness  # The fittest
        individual is determined.(Plot)
    # Below the average population fitness is determined.(
       Plot)
    for x in range(pop_size):
        population[x].childIndex = x
        if bestChildCounter < population[x].fitness:
            bestChildCounter = population[x].fitness
        averageChildCounter += population[x].fitness
    averageChildCounter = int(averageChildCounter / pop_size)
    bestChildArray.append(bestChildCounter)
    averageChildArray.append(averageChildCounter)


# The function below does the bulk of the genetic algorithm implementation as it is in charge of crossover, mutation
# and reproduction of the new generation. The system is split into 3 sections elite, products of crossover and mutation
# and random population insertion.
def crossover_mutation():
    global population
    global pop_size
    # This variable keeps track of the portion of the population already accounted for, ensures stable population size
    popAccounted = 0
    mutation_rate = (5 / 100)  # This is the percentage chance that a gene of the sequence will mutate
    elite_percentage = (10 / 100)  # This is the percentage of parents that will stay unaltered into the next generation
    COC = (70 / 100)  # This is the amount of children that are products of crossover and potential mutation
    crossover_point = [int(81 / 3), 40, int((2 * 81) / 3)]  # This array hold the three possible crossover points
    tempPopulation = []  # This array hold the children of the new generation as it is produced.
    tempChild = child()
    tempChild.gene_sequence = ["X"] * 81
    # The popAccounted now accounts for the elite individuals.
    popAccounted += int(pop_size * elite_percentage)
    # The elite individuals are transferred to the next generation as is with no alterations. Small portion of pop.
    for i in range(int(pop_size * elite_percentage)):
        tempPopulation.append(population[i])
    # The popAccounted accounts for the elite individuals and individuals that are products of crossover and mutation.
    popAccounted += int(pop_size * COC)
    # The loop below creates the proportion of the population that is a product of crossover and
```

```
        mutation.
    # The elite individuals act as the parents, 2 random elite individuals are crossovered together to make a COC child
    for i in range(int(pop_size * COC)):
        tempChild = child()
        tempChild.gene_sequence = ["X"] * 81
        parent1 = tempPopulation[random.randrange(int(pop_size * elite_percentage))]
        parent2 = tempPopulation[random.randrange(int(pop_size * elite_percentage))]
        while (parent1 == parent2):
            parent2 = tempPopulation[random.randrange(int(pop_size * elite_percentage))]
        point_cross = crossover_point[random.randrange(len(crossover_point))]
        countPosition = 0
        # Cross over takes place at 1 of three positions. The parent 1 fills in from one side then second parent
        # fills in the rest.
        for x in range(81):
            if countPosition <= point_cross:
                tempChild.gene_sequence[x] = parent1.gene_sequence[x]
            else:
                tempChild.gene_sequence[x] = parent2.gene_sequence[x]
            # During every gene crossover the gene has a set chance of mutating into any of the tree options,
            # even the one it currently is thus it has (mutation rate/100)*0.667 chance of changing.
            if random.random() < mutation_rate:
                tempChild.gene_sequence[x] = random.choice(['R', 'P', 'S'])
            countPosition += 1
        tempPopulation.append(tempChild)
    # The remaining population is then a product of pure randomness, they are basically reinitialised. This ensures
    # that it will never get stuck in local minimum and will always converge to global maximum even if it take
    # very long.
    for i in range(pop_size - popAccounted):
        tempChild = child()
```

```python
        tempChild.gene_sequence = ["X"] * 81
        for x in range(81):
            tempChild.gene_sequence[x] = random.choice(['R', 'P', 'S'])
        tempPopulation.append(tempChild)
    population = tempPopulation

# The program starts here. (entry point)
if __name__ == "__main__":
    compressData()  # The CSV data set is compressed.
    initialisePopulation()  # The population is initialised
    calculateFitness()  # The 1st generation is ranked and indexed. (There fitness is gathered)
    tick = 1 # Keeps track of the generation.
    generationArray = [tick]  # Helps with plotting fitness vs generation.
    # The program runs until there is a individual with a fitness equal to or greater than the one desired and
    # specified in the while loop.
    while population[0].fitness <= 80:
        tick += 1
        crossover_mutation()  # Each generation mutation takes place
        calculateFitness()  # Each generation the individuals are ranked.
        generationArray.append(tick)  # increase the generation.
    file = open("Data1.txt", "w+")
    for i in range(0, 81):
        file.write(population[0].gene_sequence[i])
        if i != 80:
            file.write(",")
    plt.plot(generationArray, bestChildArray, label="Fittest individual")
    plt.plot(generationArray, averageChildArray, label="Average population fitness")
    plt.xlabel('Generation')
    plt.ylabel('Fittest')
    plt.legend()
    plt.grid(color='black', linewidth=0.5)
    plt.savefig('Graph_Constant10.png')
    plt.show()
```