




UNIVERSITEIT VAN PRETORIA  
UNIVERSITY OF PRETORIA  
YUNIBESITHI YA PRETORIA

Faculty of Engineering, Built Environment and  
Information Technology

# EAI320

## Intelligent systems

Practical 2: Genetic Algorithms

Name and Surname	Student number	Signature
Liam Burgess	18015001	

# **Table of Contents**

Table of Contents.....	2
Introduction .....	3
Aim .....	3
Implementation (How is each step performed) .....	4
Generation .....	4
Fitness function .....	4
Selection .....	4
Survival of the fittest (Linear selection):.....	4
Tournament: .....	4
Crossover .....	5
Fifty-fifty: .....	5
Checkered: .....	5
Mutation .....	5
Total randomness of the population (Maximize exploration).....	5
Similar to total randomness (Reduced exploration) .....	5
Results.....	6
Discussion of results .....	12
Reflection.....	14
Conclusion .....	14
References .....	15
Appendix A: Code .....	16
Figure 1: A population of eight, five generations without mutation, linear selection and f.....	6
Figure 2: A population of eight, ten generations without mutation .....	7
Figure 3: A population of eight, fifteen generations without mutation.....	7
Figure 4: A population of eight, five generations with mutation.....	8
Figure 5: A population of eight, ten generations with mutation .....	8
Figure 6: A population of eight, fifteen generations with mutation.....	9
Figure 7: A population of eight, five generations without mutation.....	9
Figure 8: A population of eight, ten generations without mutation .....	10
Figure 9: A population of eight, ten generations with mutation .....	10
Figure 10: A population of eight, ten generations with mutation .....	11

## **Introduction**

Genetic algorithms form an important tool for software engineers and problem solvers in the modern computing era. The algorithm can be applied to large scope of problems, adapting to generate solutions from a solution space to the problem when setup correctly. Genetic algorithms are applied to play the game of Rock, Paper, Scissors (RPS).

## **Aim**

To implement the different steps of a Genetic Algorithm to find the goal of a successful sequence of RPS moves against an opposing agent. The steps being: Generate a starting population of agents. Evaluate their fitness through some heuristic. Perform a selection based on the heuristic, value of win-lose-draw. Crossover and eliminate some of the population, generating new agents to replace them in the population. To prevent possible genetic stagnation a mutation will need to be employed to randomly change some data of some (or all) of the agents to ensure the solution space is effectively explored.

# **Implementation (How is each step performed)**

## **Generation**

Each actor in the population is generated with a starting random eighty-one gene sequence. The safest method for starting the process is randomly. Starting a sequence as just rock objects for example may lead the actor to be seen as a good solution but will become stagnant quickly and won't generate much better objects depending on the crossover and mutation process. Randomly generated solutions allow more of the solution space to be observed meaning a better solution than a single object sequence to possibly be found.

## **Fitness function**

The basic fitness function links each gene within the sequence to the sequence found on level four of an RPS tree using BFS, i.e. RRRR is linked to index one. The data given has a second column which is the last move played by an opponent. Comparing the object at the gene index with the opponent's move allows a simple scoring system to be implemented. Win is one point; draw is zero and lose is zero where previously investigated as win at two, draw at one and lose at zero. The reason for the change being in the [discussion of results](#). The data is iterated through for each index sequence to build a score for each position-sequence and then totaled at the end for each agent.

## **Selection**

Selection was implemented in two ways:

### **Survival of the fittest (Linear selection):**

The top half, ordered by fitness, of agents are selected for reproduction. This allows convergence to a solution relatively fast as the better solutions are only used for regeneration. This may however lead to stagnation again as convergence to a solution will occur, but it may only be a weak solution. Performing mutation here is necessary to ensure exploration for an even more optimal solution occurs. The children replace the half of the population removed. The data is observed linearly with bigger meaning fitter.

### **Tournament:**

The agents are randomly placed against one another with the probability of winning based proportionally on the agent's fitness:

$$P(\text{Agent 1}) = \frac{Fitness_{Agent-1}}{Fitness_{Agent-1} + Fitness_{Agent-2}}$$

Playing probabilistically allows the chance for the weaker agent to win against a much better agent. This allows solution space to be explored at the cost of a more fit agent be sacrificed to find a better solution. The probability is proportional to help ensure not all the better agents are potentially lost if a game of fifty-fifty probability were to occur. The agents are ranked based on their progress in the tournament.

## Crossover

Basic non-fitness-based methods:

### **Fifty-fifty:**

Two agents at a time have their eighty-one gene set split in half and swapped with the other agent's. This method creates two new agents who are used in the next generation.

### **Checkerboard:**

Agents are paired. Every odd index of the first agent's gene-sequence is swapped with the next agent.

## Mutation

### **Total randomness of the population (Maximize exploration)**

A random number is generated and compared against the mutation rate chosen. If it is lower, mutation occurs. A random number of agents and their gene-sequences are then modified randomly to help with non-stagnation in solutions. More of the solution is then explorable when randomly changing the agents although at the cost of making the solution pool potentially weaker.

### **Similar to total randomness (Reduced exploration)**

The children are only given the chance to mutate whereby each gene is given a random chance to mutate to the winning object against what it is currently. [1]

## Results

The results obtained were not of absolute optimum standard. This may be a coding issue whereby a bug limited the results.

The method of survival of the fittest mixed with fifty-fifty crossover and no mutation was investigated first with the result showing a stagnation in fitness as expected. The fitness over five, ten and fifteen generations were examined with a population of sixteen members. The sequence was found to flatline around six hundred and twenty thousand at a generation of five which gradually reduced as generations increased.

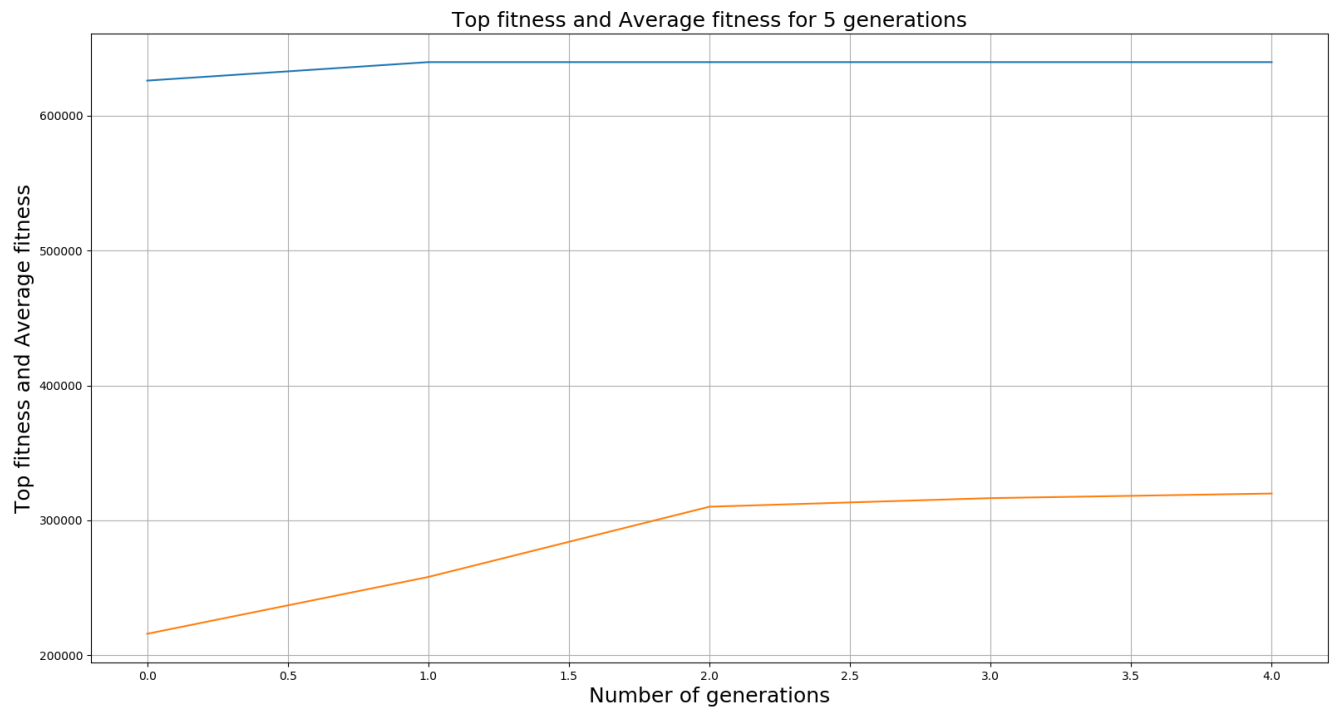


Figure 1: A population of eight, five generations without mutation, linear selection and f

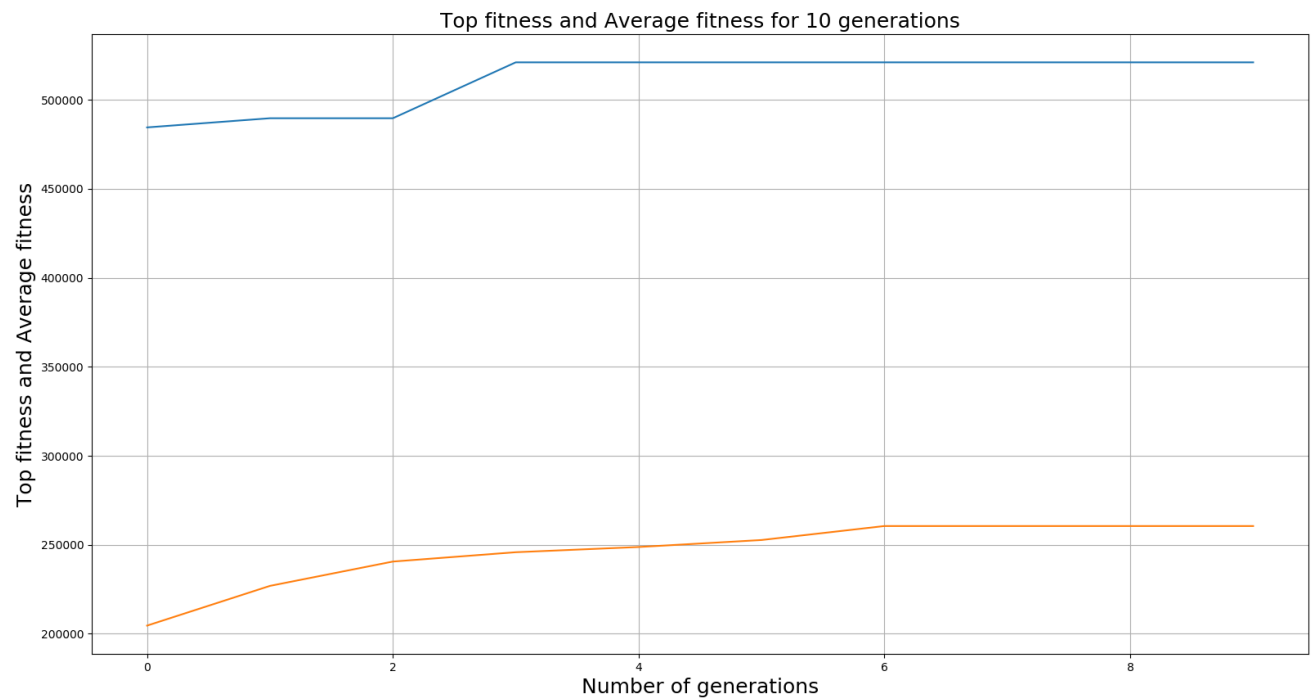


Figure 2: A population of eight, ten generations without mutation

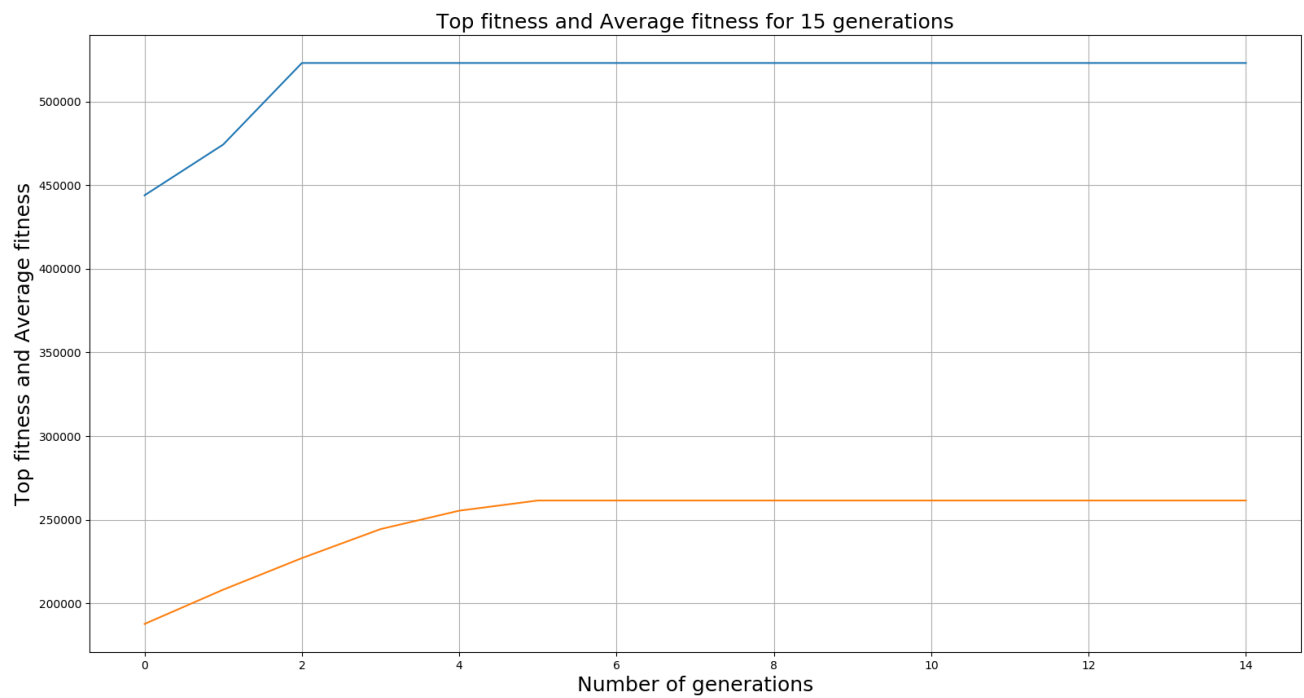


Figure 3: A population of eight, fifteen generations without mutation

Absolute random mutation was then enabled with a gradual increase to a about 720000 within ten generations but a drop off was observed for fifteen generations:

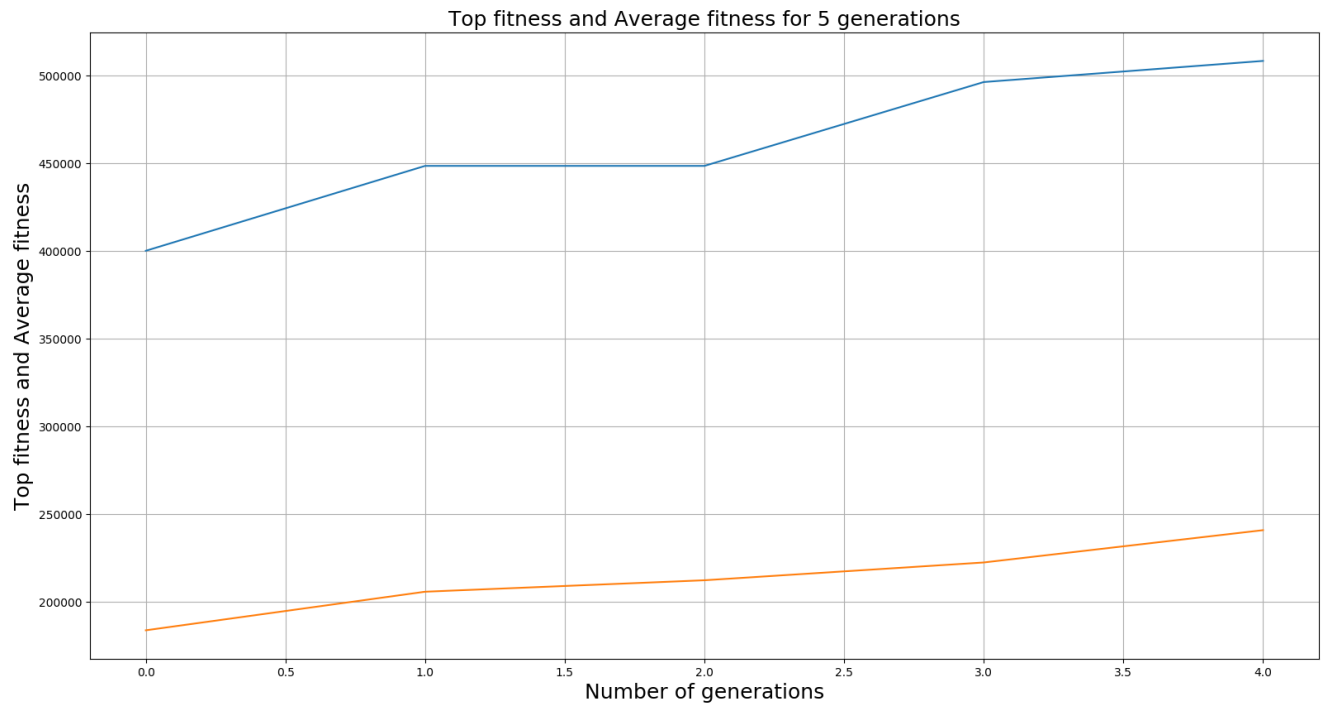


Figure 4: A population of eight, five generations with mutation

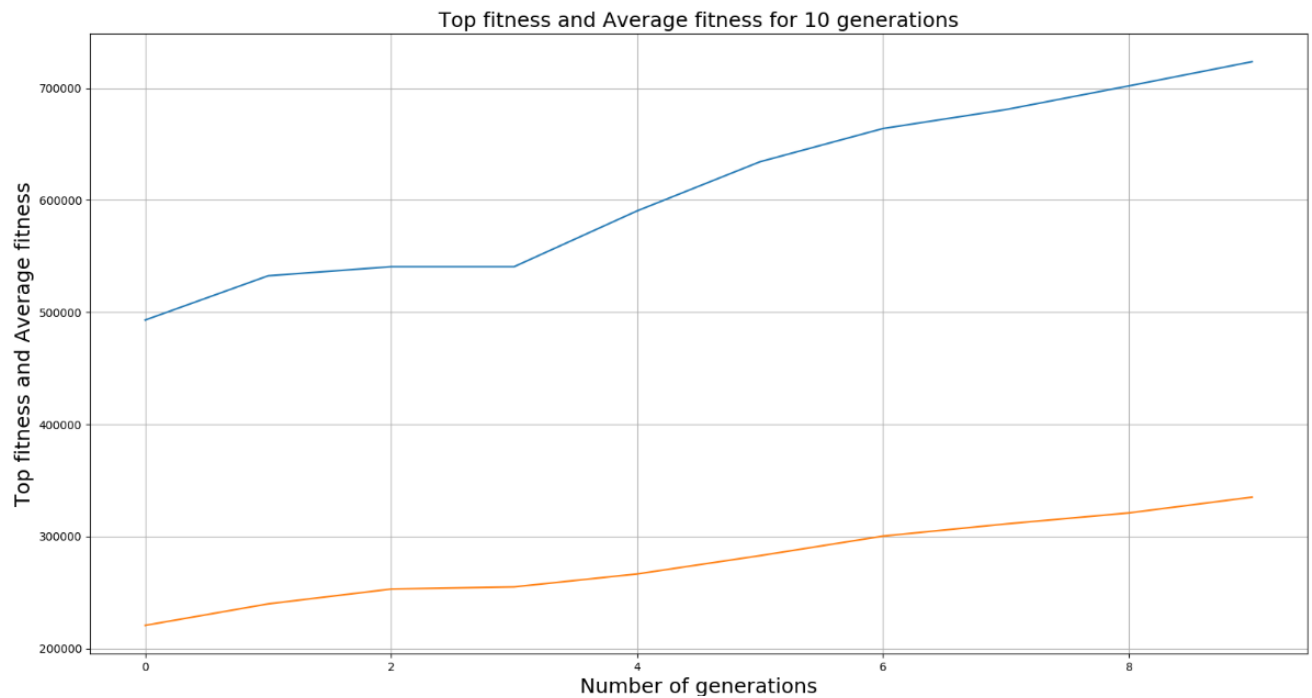


Figure 5: A population of eight, ten generations with mutation



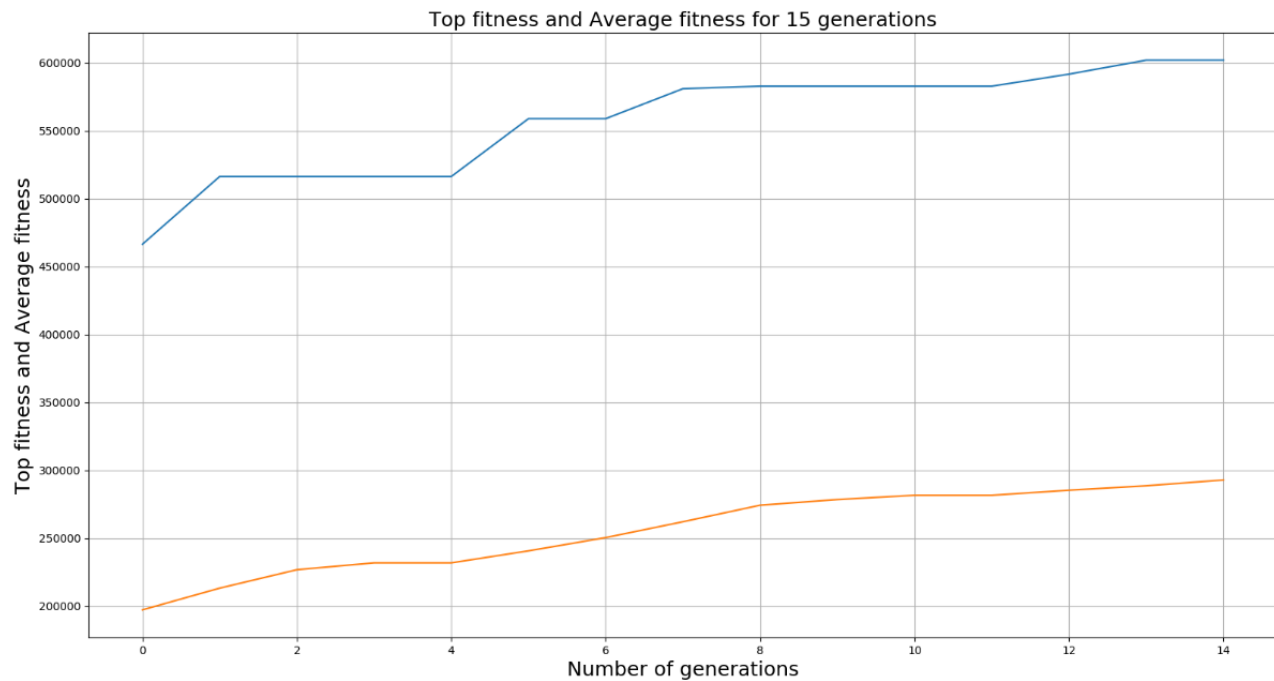


Figure 6: A population of eight, fifteen generations with mutation

Tournament at five and ten generations was also considered:

Without mutation:

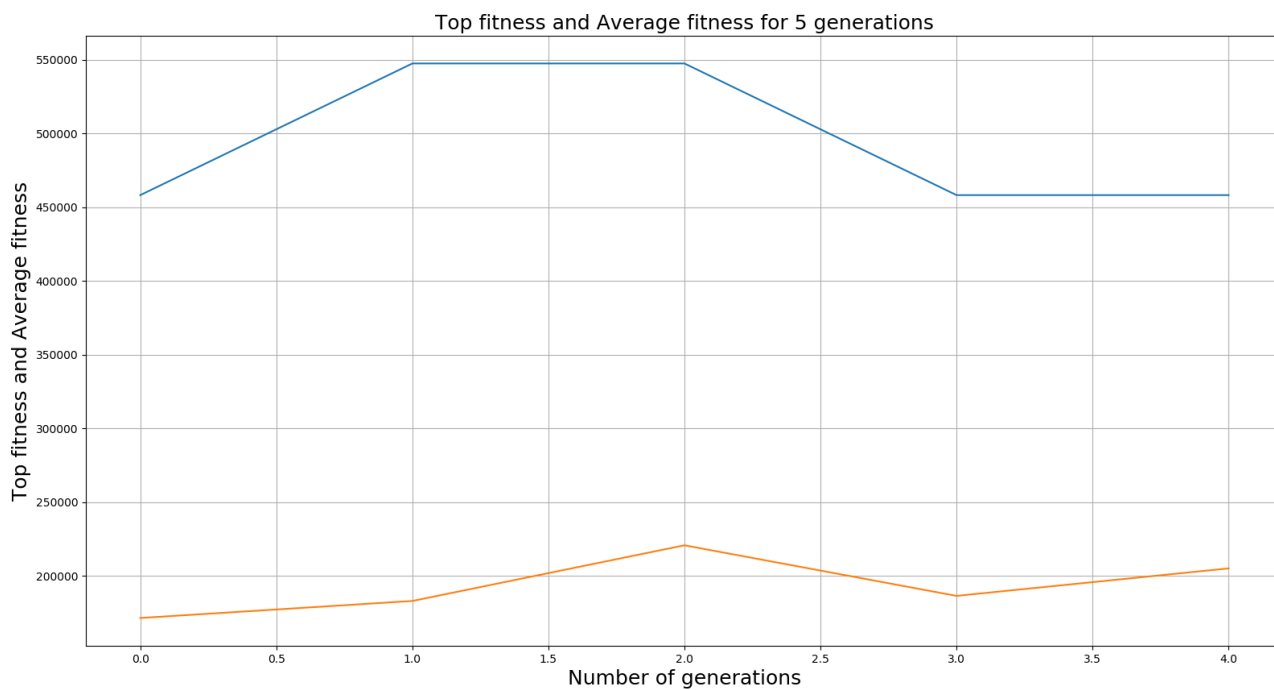


Figure 7: A population of eight, five generations without mutation

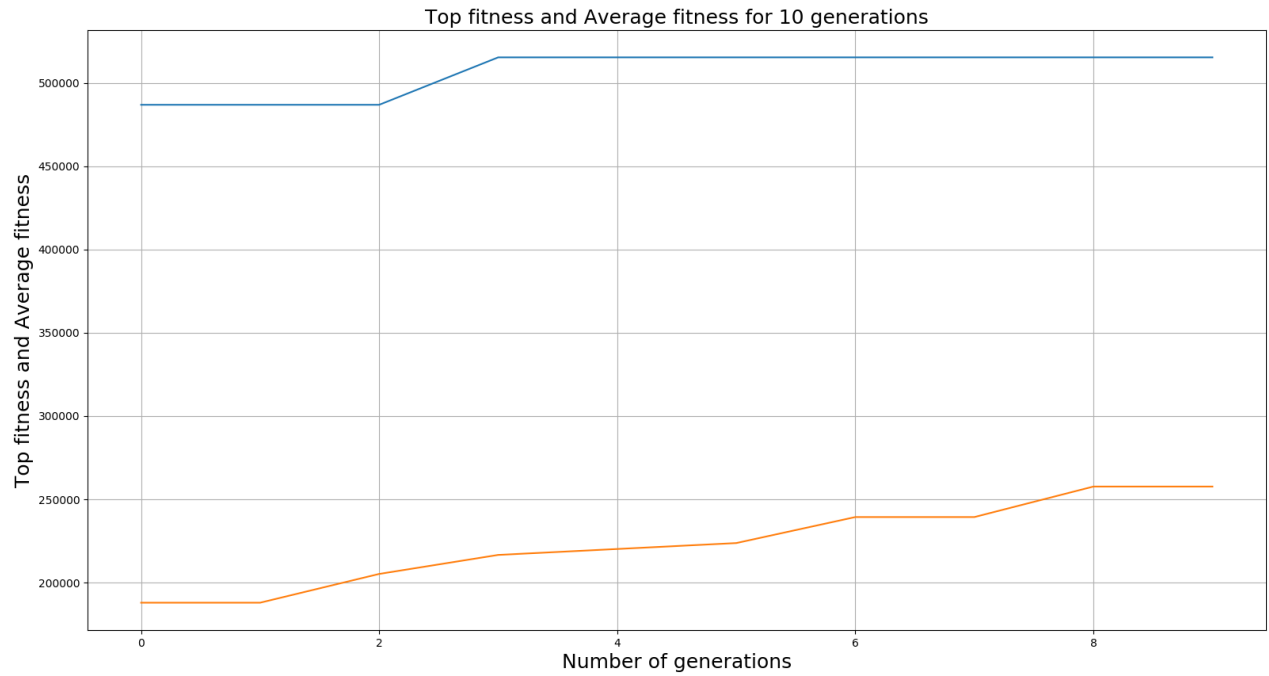


Figure 8: A population of eight, ten generations without mutation

With mutation:

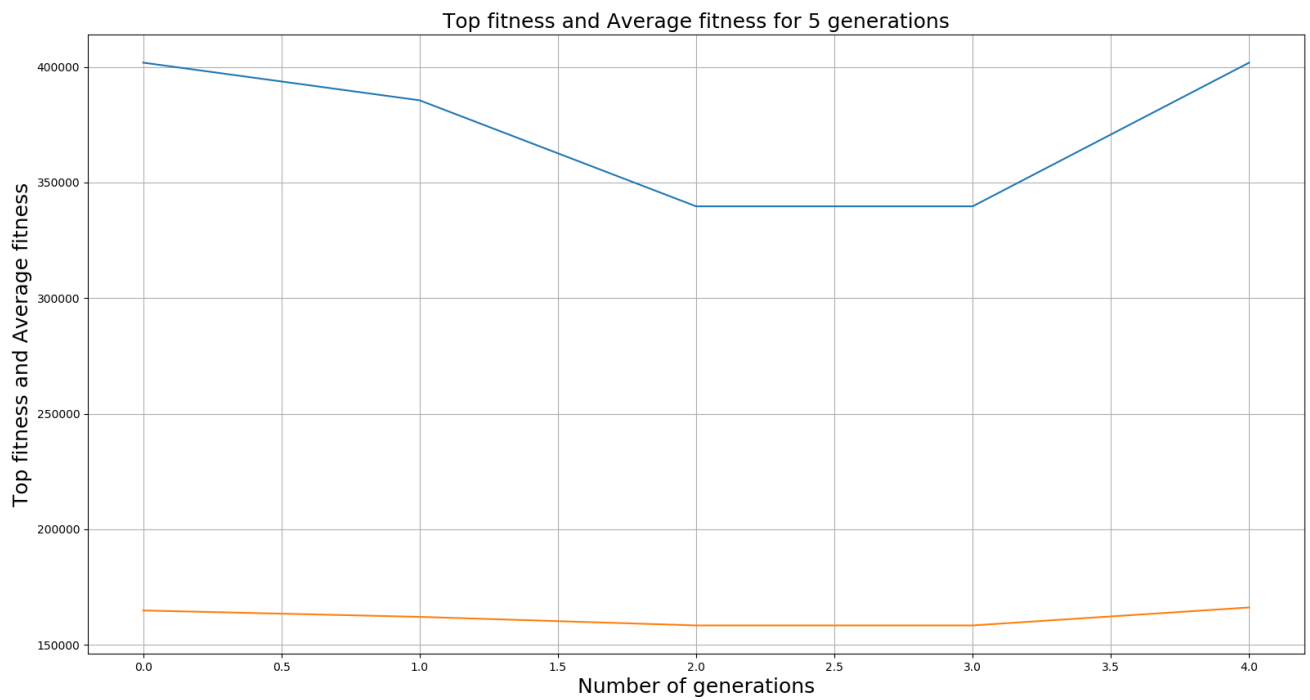


Figure 9: A population of eight, ten generations with mutation

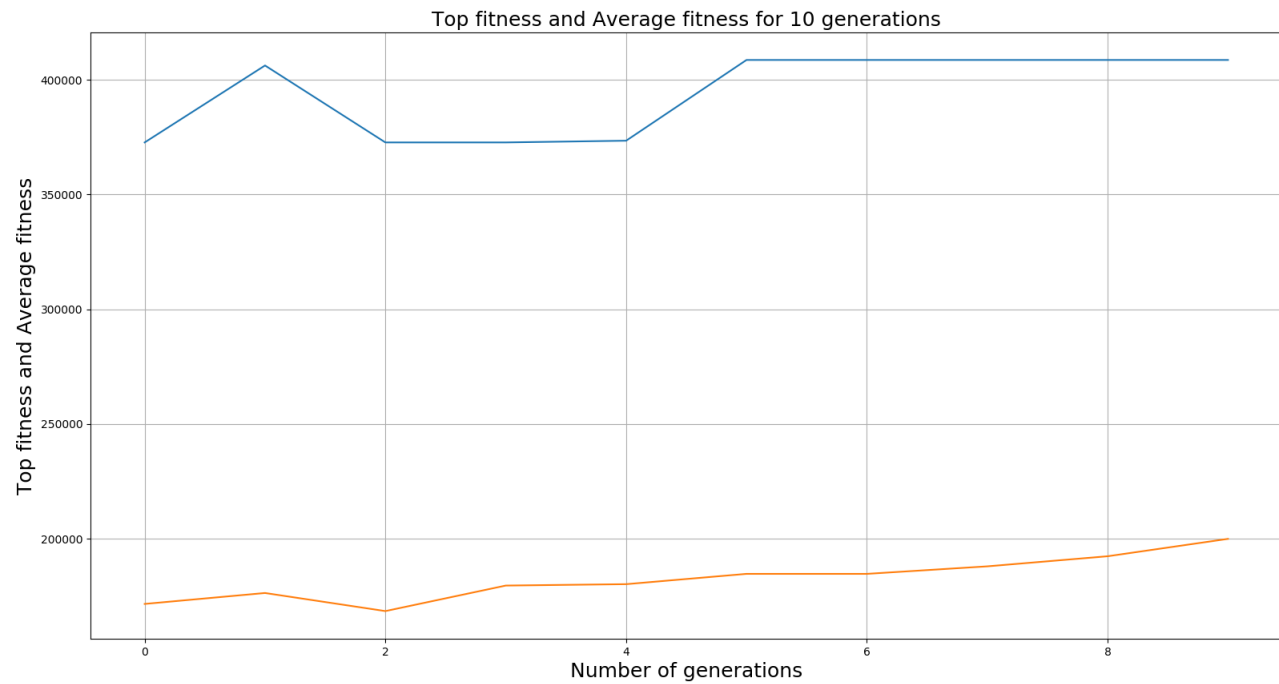


Figure 10: A population of eight, ten generations with mutation

## **Discussion of results**

The observed stagnation in fitness value improving occurs from a lack of exploring the solution space. Mutation is required to allow the solution space to be explored by modifying actor's genes. If the eighty-one gene string can be observed as variables, mutation allows changing of some of these variables to move the actor to a new location in the solution space where other solutions may possibly be found. These solutions may also be more optimal than the ones occupying the space originally where the agent was previously.

The stagnation is also attributed to the nature of genetic algorithms sharing characteristics of a hill climbing algorithms. Once an optimal solution is located the agent will 'struggle' to move to another solution or be willing to explore the solution space for more solutions due to the fitness of the agent being reduced if it were to move. Mutation once again helps by forcing the solution to be modified and thus moving the agent away from the local optima it is currently preceding on.

Mutation however holds caveats which may hinder the genetic algorithm from finding more optimal solutions. Depending on the method for modifying data when a mutation occurs, for example randomly changing objects in an agent, may actually reduce the effectiveness of the algorithm. If the probability associated with the mutation function is too high and too much data is mutated (changed) it may negatively affect the agent's ability to converge to an optimal solution. The agent may explore the solution space too often and is not given a chance to converge to a solution. The solutions may thus become weaker and a mutation probability must be found which maximizes the benefits associated with the solution. Reducing the probability and thus reducing the exploration as iterations continue may allow for better convergence.

The effectiveness of mutation requires to be balanced with both the probability of occurring and the crossover method applied.

Fixed crossover methods like checkered and fifty-fifty used generate a consistency in how children are generated and placed in the solution space. The crossover point can determine how far away from each parent the children are deployed in the solution space. A random crossover point leads to one child more likely closer to one parent and the parent's closest maxima and the other similarly to the other parent. A consistent crossover allows a better distance for the children to begin generating at. Crossover however has the risk of pulling a solution away from converging to its current converging solution (like mutation) when crossing over with a solution which is not anywhere near the other solution. This is why the list of solutions should preferably be organized to ensure relatively close solutions to one another will crossover to further get closer to the solution.

The basic linear selection scheme makes use of characteristics of a hill climbing algorithm. Observing the most fit agents only leads to optima in the solution space being observed only majorly. This is problematic if the mutation rate is not balanced to push agents to explore the solution space as discussed above. This may reduce convergence of the algorithm to more optimal solutions.

Tournament is implemented as an alternative to linear selection. Tournament allows for a weaker solution to be allowed to be accepted in the selection process. This is due to the

probability of the weaker agent being a viable option. The benefit of having a weaker solution selected is that it increases exploration of the solution space where usually tournament will favor converging solutions which have the higher probability. This give a suitable balance to exploration when integrated well with the mutation rate.

The amount of times this was allowed to occur, or the number of generations, allows for more exploration and better convergence to a solution potentially. If the mutation rate is high however: the generation where a very optimal solution is found, may be lost if the generations do not stop and instead a mutation (or multiple) occurs and moves that agent away from the solution.

The number of generations balanced with the probability of the mutation rate can however also allow solutions to better converge to their maxima as the number of generations increase. If the other solutions are also approaching the maxima (and not pulling it away with crossing over to another solution as discussed with crossover above).

The fitness function's heuristic was based on being a win as one and the other options as zero. This allows the fitness function to distinctly represent each agent's fitness. If a two is used for win and a one for draw the results can become vague as the solutions may settle for more of a 'happy medium' between winning and drawing instead of attempting to maximize winning potential. The function should not reward drawing as it is not moving the agent towards a goal state of the maximum optimal solution.

The size of the population affects the processing time of the program while improving the exploration space if they are placed far enough away from each other on initialization. The diversity found at initialization is thus valuable for exploration. The more agents present may allow for better exploration (with a balanced mutation algorithm) to find better solutions. Computer resources should not be considered too much of a caveat for a larger population unless a much, much larger solution space requires to be observed and hardware is limited.

The pseudo optimal strategy of the genetic algorithm was found to be rating the winning genes as a point of one as the heuristic, totaled to make the fitness of the agent. The agents were sorted highest to lowest or played in tournament (similar effectiveness) and then the top half of contenders taken to reproduce. The other half being culled. Crossover of fifty-fifty was used. A medium to low mutation rate was used to generate mutation and the results then observed over a relatively large amount of iterations (around fifteen to twenty for final tests). The generation size a moderate size of about twelve to sixteen.

## **Reflection**

The assignment gave much insight into the design of genetic algorithms. A larger fundamental understanding of understanding the mechanisms of the genetic algorithm were understood. Visually imagining the solution space and applying that to understanding and effectively implementing the genetic algorithm will make of great use whenever genetic algorithms have to be used again. Learning latex, with some issues still present, was also a valuable experience which will be used further in the rest of the course.

## **Conclusion**

Choice and fine tuning are the biggest factors which need to be considered with genetic algorithms. No one method is going to lead to absolutely optimal results. The size of the solution space relative to the employed algorithms affect the ability of the genetic algorithm to perform.

The genetic algorithm implemented above is successful with a minor fault here and there, to generate a sequence which may be used to beat Agents 1 and 2. The effectiveness of balancing different functionalities of the algorithm was investigated with a larger amount of generations requiring to be balanced with the mutation rate. The crossover rate having to be balanced with the mutation rate as well. The fitness function requires a good heuristic in order to provide sufficient data that can be used for performing the genetic algorithm.

## **References**

- [1] R. Stuart, Artificial Intelligence, A Modern Approach, Pearson, 2010.
- [2] "Introduction to Optimization with Genetic Algorithm," 03 July 2018. [Online]. Available: <https://towardsdatascience.com/introduction-to-optimization-with-genetic-algorithm-2f5001d9964b>. [Accessed 2020 March 04].
- [3] Geeks4Geeks, "Genetic Algorithms," 23 August 2018. [Online]. Available: <https://www.geeksforgeeks.org/genetic-algorithms/>. [Accessed 3 March 2020].

## **Appendix A: Code**

**Uploaded as txt as my library wasn't working**

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

#=====

filename = "data1.csv"

#random.seed = 10
history = []
last_play = []

plays = ['R', 'P', 'S']
population = 8
gene_size = 81
generations = 5

global mutation_chance
mutation_chance = 32/81
#(population - 2) / (population)+generations/2
#=====

with open(filename) as CSV:
    reader = csv.reader(CSV, delimiter=',')
    for row in reader:
        history.append(row[0])
```



```

        last_play.append(row[1])

BFS = list()
for i in plays:
    for j in plays:
        for k in plays:
            for l in plays:
                hold = str(i+j+k+l)
                BFS.append(hold)

#=====

def play_value(Opp_move, agent_move):
    if Opp_move == "R":
        if agent_move == "R":
            return 0
        elif agent_move == "P":
            return 1
        else:
            return 0
    elif Opp_move == "P":
        if agent_move == "P":
            return 0
        elif agent_move == "S":
            return 1
        else:
            return 0
    elif Opp_move == "S":
        if agent_move == "S":
            return 0

```

```

        elif agent_move == "R":
            return 1
        else:
            return 0

#=====

#All code above works

class actor():
    def __init__(self, genes = []):
        self.genes = genes
        if len(self.genes) == 0:
            self.gen_genes()
        self.fitness = 0

    def gen_genes(self):
        for x in range(0,81):
            self.genes.append(np.random.choice(['R', 'P', 'S']))
        print(self.genes)

    def return_gene(self, index):
        return self.genes[index]

# returns a deep copy of gene-object sequence
    def return_entire_gene(self):
        copy = list()
        for x in self.genes:
            copy.append(x)
        return copy

```

```

def return_fitness(self):
    return self.fitness

# add to the fitness of the agent
def add_fitness(self, add=0):
    self.fitness += add
    return

def clear_fitness(self):
    self.fitness = 0
    return

def set_genes(self, genes):
    self.genes = genes
    return

#=====
#initiliaze GA class and generate each actor with a random 81 string chromosone list
class genetic_algorithm():
    def __init__(self, population):
        self.actors = list()
        self.fitness_values = list()
        self.population = population
        for x in range(self.population):
            self.actors.append(actor([]))

#Above all works

#Generate acts as a wrapper function calling the different parts of the GA
def generate(self, gener):
    global mutation_chance
    gener

```

```

avg = []
best = []
for _ in range(gener):

    #Call each stage of the GA
    self.fitness()
    #self.Selection()
    self.tournament()
    self.crossover()
    self.mutation()

    avg.append(self.avg_fitness())
    best.append(self.best_fitness())

    for z in range(len(self.actors)):
        self.actors[z].clear_fitness()

    print("Generation: "+str(_))

    #mutation_chance -= 1/27

# Display run results
self.results(avg, best, gener)
self.textfile(self.best_seq())
return

#revision 2/3/2020
def fitness(self):
    gene_count = 0

```

```

#Starting at RRRR (== Gene 0) go through each gene
for seq in BFS:
    count = 0
    #Keeps track in textfile of index location
    count_last_play = 0
    #Seq in current line of textfile
    for hist in history:
        count += 1
        # When the current gene's sequence matches the line in the textfile
        if seq == hist:
            # Iterate through each member
            for act in self.actors:
                #The move the textfile played versus the member is compared and valued via
play value
                act.add_fitness(play_value(last_play[count_last_play],
act.return_gene(gene_count)))
                count_last_play += 1
            if count == -1:
                break
            gene_count += 1
    return

def Selection(self):
    #take top half of population as reproducing candidates
    self.actors.sort(key = lambda actor: actor.fitness, reverse = True)
    elite = []
    self.population = int(self.population / 2)
    for i in range(self.population):
        elite.append(self.actors[i])
    self.actors = elite

```

```

# Code above should all work

# Tournament play with weighted win probability
def tournament(self):
    losers = []
    losers_hold = []
    # The players are removed from winners as they are eliminated
    winners = self.actors
    random.shuffle(self.actors)
    while len(winners) != 1:
        for i in range(0, len(winners), 2):
            p1 = winners[i]
            p2 = winners[i+1]

            # weighted probability
            p_p1 = p1.return_fitness() / (p1.return_fitness() + p2.return_fitness())
            p_p2 = p2.return_fitness() / (p1.return_fitness() + p2.return_fitness())

            prob = [p_p1, p_p2]
            # to not break the for loop, the indices are stored and the
            # items are removed after all the iterations of a round complete
            if random.choices([0,1], prob) == 0:
                losers.append(p2)
            else:
                losers.append(p1)

        for k in losers:
            winners.pop(winners.index(k))
        losers_hold.extend(losers)
        losers.clear()

```

```

# elements are placed into losers in the order they lose in the tournament
losers_hold.extend(winners)

# the winner is then added last and the entire list is flipped to give tournament order
losers_hold.reverse()

self.actors.clear()

```

```

for i in range(int(self.population/2)):
    self.actors.append(losers_hold[i])
self.population = int(self.population/2)

return

```

```

#=====
=====

```

#Simple crossover styles are applied here (50/50, checkered - both children kept- no metrics to choose genes)

```

def crossover(self):
    children = list()

    l = 0

    #Cross over between actor and neighboring actor
    while l in range(self.population):
        #first half from first parent
        genome = self.actors[l].return_entire_gene()[0:41]

        #second half from second parent
        genome.extend(self.actors[l+1].return_entire_gene()[41:82])

        #Create first child
        child = actor(genome)

        #call alternate mutation
        #child.set_genes(self.mutation(child.return_entire_gene()))

```

```

children.append(child)

# first half of second parent
genome = self.actors[l+1].return_entire_gene()[0:41]
# second half of first parent
genome.extend(self.actors[l].return_entire_gene()[41:82])
l+=2

#Create Second child
child = actor(genome)
#call alternate mutation
#child.set_genes(self.mutation(child.return_entire_gene()))
children.append(child)

#print(children[0].return_entire_gene())
#print(children[1].return_entire_gene())

self.actors.extend(children)
self.population = len(self.actors)

#=====

return

#revise because I lack vision 22/02/2020
def mutation(self, seq = []):

# generate a 1 or 0 dependant on the mutation probability defined above
prob = [mutation_chance, 1-mutation_chance]

```



```

options = [1, 0]
if seq == []:
    #A method to generate a random value
    if random.choices(options, prob):
        print("Mutation")
        x = random.randrange(0, self.population/2)
        for i in range(x):
            #Choose a member to edit
            actor_edit = random.randrange(0, self.population)
            #Choose a random amount of genes to edit
            how_many = random.randrange(0, 42)
            seq = self.actors[actor_edit].return_entire_gene()
            for i in range(how_many):
                #Randomize a random gene
                j = random.randrange(0, 81)
                seq[j] = np.random.choice(['R', 'P', 'S'])
                #Overwrite the object at the random chosen spot
                hold = actor(seq)
                hold.add_fitness(self.actors[actor_edit].return_fitness())
                self.actors[actor_edit] = hold
                #print("New object:" +str(hold.return_fitness()))
            return
    else:
        rpsr = ['R', 'P', 'S', 'R']
        for x in range(81):
            opt = random.choices(options, prob)
            if opt==[1]:
                #print("Mutation 2")
                seq[x] = rpsr[rpsr.index(seq[x])+1]
        return seq

```

```
#=====
```

```
#Evaluate the fitness of a generation
```

```
#Average fitness of generation N
```

```
def avg_fitness(self):
```

```
    avg = 0
```

```
    for act in self.actors:
```

```
        avg += act.return_fitness()
```

```
    avg = avg/self.population
```

```
    return avg
```

```
#Best fitness recipient of generation N
```

```
def best_fitness(self):
```

```
    best = 0
```

```
    for act in self.actors:
```

```
        if best<act.return_fitness():
```

```
            best = act.return_fitness()
```

```
    return best
```

```
def best_seq(self):
```

```
    best = 0
```

```
    ret = []
```

```
    for act in self.actors:
```

```
        if best<act.return_fitness():
```

```
            best = act.return_fitness()
```

```
            ret = act.return_entire_gene()
```

```
    return best
```

```

#Display results
def results(self, top = [], avg = [], generations = 0):
    plt.title("Top fitness and Average fitness for "+ str(generations) + " generations", size = 18)
    plt.xlabel("Number of generations", size = 18)
    plt.ylabel("Top fitness and Average fitness", size = 18)
    plt.grid(1)
    generations = np.arange(0, len(avg), 1)
    plt.plot(generations, avg)
    plt.plot(generations, top)
    plt.show()
    return

#Returns best sequence
def textfile(self, data = []):
    with open("sequence.txt", "w", newline="") as file:
        writer = csv.writer(file)
        writer.writerow(data)

#print("Prepare to crash")
#GA = genetic_algorithm(population)
#GA.generate(generations)
#print("No crash")
#print(str(GA.best_fitness()))

```