Optimization Bio Computation

Lam Ha - 19035157

1 INTRODUCTION

As mentioned in the title, I chose Optimization as my further work to carry on from worksheet 3 which indicates the implementation of Genetic Algorithm using maximization function such as Counting One's function and two minimization functions as below:

$$f(x) = 10n + \sum_{i=1}^{n} x_i^2 - 10.\cos(2\pi.x_i) \qquad \text{Where } -5.12 \leq x_i \leq 5.12, \text{ and use } n=10, 20$$

Figure 1.0 Rastrigin minimization function

$$f(x) = -20\exp\left(-0.2\sqrt{\frac{1}{D}\sum_{i=1}^{D} x^2}\right) - \exp\left(\frac{1}{D}\sum_{i=1}^{D} cos2\pi x_i\right)$$

Figure 1.1 Ackley minimization function

After completing 3 worksheets, I have been able to develop a system that generates each population based on float genes number and genetic algorithm process. To be concise, in the worksheet 1 and 2, I was working on the maximization function (Counting One's function) to calculate the fitness of each individual person and apply the GAs such as Selection, Crossover, and Mutation process. For worksheet 3, the challenge is to work on real numbers on genes array instead of a binary number like 1 and 0 and apply the same GA on the minimization function instead of the maximization one. And in order to get the GA to work even more efficiently is to work on the optimization which is the task asked and also try it on the new minimization function.

By working on the optimization, my approach is to develop some new operators to compare with the initial algorithm, and from that, I can notice which operator works better in terms of competitive performance. In my code, the initial selection method required in the worksheet is called Tournament selection which is simpler since we only compare the fitness of the two individuals when looping and the individual with higher fitness will be assigned to the population. And compare against to that selection method is Roulette Wheel selection that I have created, basically meaning divide a wheel into a number of slots proportional and from there the individuals with the higher fitness values will have more slots on the roulette wheel and more likely to be picked. I have also executed my code over 10 runs to record the average fitness and plotted images which will be shown in the Experimentation section. Additionally, I have altered the Crossover process to make it work since the Crossover method in the worksheet resulting wrong answers to me.

Besides, I will also research the differences between Tournament selection and Roulette Wheel selection and what changes in the algorithm implementation are going to apply to maximization and minimization functions

and also identify examples where using AI for a task has raised ethical issues and state them, and all of that will be enclosed in the background research section.

After this assignment, I expect to advance and expand my knowledge as well as my understanding of how Genetic Algorithms work and also the performance of the process inside GAs including Selection, Mutation, and Crossover.

2 BACKGROUND RESEARCH

2.1 Introduction to Genetic Algorithms

Genetic algorithms are a type of optimization algorithm, meaning they are used to find the maximum or minimum of a function (Jenna Carr, 2014). According to this assignment, we are going to apply a Genetic Algorithm on Maximization (Counting One's function) and Minimizations which are Rastrigin and Ackley functions.

According to Jadaan, Rajamani, & Rao (2008), I can summarise the Genetic Algorithm's steps as following:

- SELECTION: this first step in GAs is required to select two random parents for reproduction. Depending on the optimization function that selection method will work differrent. For example, the inidividuals with higher fitness are likely to be selected for mating in terms of Maximization and those with lower fitness will be selected when working with minimization.
- RECOMBINATION: this step requires the crossover process to create new better individual's representation of gene from its' parent's representation by using one point swapping.
- MUTATION: once the offspring population is created after crossover, the mutation operator is used to vary the chromosomes inside the individual's representation by bit flipping.
- EVALUATION: build rule-base from genome and try on the problem to calculate individual's fitness from fitness function.
- REPLACEMENT: after one generation finishes, the parent population will be replaced by the offspring population to maintain the next cycle.

The Genetic Algorithm stops when the population converges towards the optimal solution.

The visual process of Genetic Algorithm is displayed below:

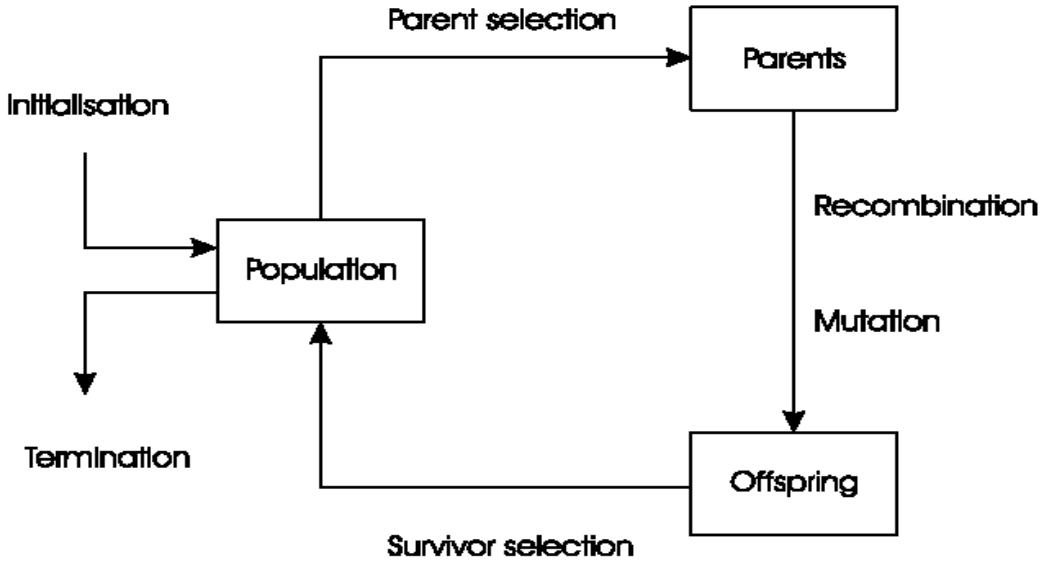


Figure 1.2 Schematic of Genetic Algorithm

```
BEGIN
  INITIALISE population with random candidate solutions;
  EVALUATE each candidate;
  REPEAT UNTIL ( TERMINATION CONDITION is satisfied ) DO
    1 SELECT parents;
    2 RECOMBINE pairs of parents;
    3 MUTATE the resulting offspring;
    4 EVALUATE new candidates;
    5 SELECT individuals for the next generation;
  OD
END
```

Figure 1.3 Genetic Algorithm Pseudo Code

2.2 Selection Methods:

In this assignment, the two commonly used Selection methods that I am talking about are Tournament selection and Roulette Wheel selection.

Tournament selection:

According to Wikipedia, Tournament selection is the method to select an individual from a population of individuals in a Genetic Algorithm. By the involvement of running several “Tournaments” among a few individuals (or "chromosomes") chosen at random from the population. The individual with the best fitness will be selected for the next generation. In other words, the Tournament selection is used to select the best fittest candidate from the current cycle of generation in a Genetic Algorithm. Moreover, Wikipedia also describes how the Tournament algorithm works in steps:

1. choose k (the tournament size) individuals from the population at random
2. choose the best individual from the tournament with probability p
3. choose the second best individual with probability p*(1-p)
4. choose the third best individual with probability $p*((1-p)^2)$ and so on

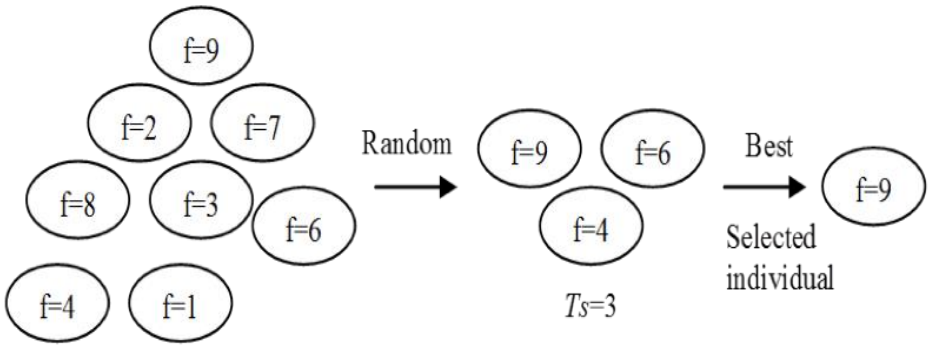


Figure 1.4 Selection strategy with tournament mechanism

Concerning my Tournament Algorithm, k is regarded as 2, which means for each of N generations that I loop through, I will select 2 random parents from the individual's population and compare their fitness, the one with better fitness will be taken to the offspring population.

Roulette Wheel selection:

Noraini Mohd Razali, John Geraghty state that in a proportional roulette wheel, individuals are selected with a probability that is directly proportional to their fitness values i.e. an individual's selection corresponds to a portion of a roulette wheel. That means fitter individuals take more proportional slots and obviously the same happens to those whose fitness is weaker, they take small slots. The N times the wheel spun, the N times the individuals are selected. This process is repeated until the desired number of individuals has been selected (Jadaan, Rajamani, & Rao, 2008).

Noraini Mohd Razali, John Geraghty also state about the selected probability formula as below:

$$p_i = \frac{f_i}{\sum_{j=1}^{n} f_j}$$

Figure 1.5 selection probability, Pi for individual i

Where n is the size of population and f_1, f_2, … f_n are the fitness of individual 1,2,…,n.

Following are the steps for Roulette Wheel selection (A. Shukla, H. M. Pandey and D. Mehrotra, 2015):

1. Calculate the sum of the fitness values of every individual in the population.
2. Calculate the fitness value of each individual and their probability of selection by dividing individual chromosome's fitness by the sum of fitness values of whole population.

3. Partition the roulette wheel into sectors according to the probabilities calculated in the second step.
4. Spin the wheel 'n' number of times. When the roulette stops, the sector on which the pointer points corresponds to the individual being selected.

Ethical Issues:

One of the related GA applications that I find interested in is Artificial Intelligence via Neural Networks Applied to Medical Applications. Artificial Intelligence can improve society's overall health (Ternent, James Thompson, Maximilian 2020). For example, it is used in healthcare for diagnostic purposes such as places as clinical practice and translational research. Ternent, James Thompson, Maximilian state that One of the most successful areas in diagnostics is automated image-based diagnosis. Medical professionals such as those in radiology, ophthalmology, dermatology, and pathology rely on image-based diagnoses. A neural network was trained on 129,450 images to identify skin melanoma. The accuracy of the neural network was greater than the average dermatologists-based comparison between the assessments of the neural network and twenty- one dermatologist. They also describe how useful AI can be when it is applied in the administration of the clinical practice. They state that artificial intelligence can perform repetitive and routine tasks such as patient data entry and automated review of laboratory data and imaging results. If machine learning algorithms are attached to electronic health records, they can assist clinicians and administrators retrieve context-related patient data allowing patients to be better treated (Ternent, James Thompson, Maximilian 2020). Besides all the beneficial sides that AI in medical application bring about, there also raise several ethical issues.

Firstly, what we need to concern about is the impact of Expectations of ML in Medicine according to Ternent, James Thompson, Maximilian. They state that There are unrealistically high hopes for machine learning in medicine, and there are just as many unrealistic fears to complement them. They also state about the statistic of 63% of the adult population [of the United Kingdom] is uncomfortable with allowing personal data to be used to improve healthcare and is unfavorable to artificial intelligence systems replacing doctors and nurses in tasks they usually perform. However, drawing from a German survey, they express that a large portion of medical students believe wholeheartedly that machine learning will improve medicine. Though they are not eager enough to embrace it with a little caution, expressing that they are "skeptical that it will establish conclusive diagnoses" (Ternent, James Thompson, Maximilian 2020). Therefore, there might have a disconnect between an acceptable explanation for a machine learning expert and an acceptable explanation for a patient, which can cause distress and possibly even harm.

The next one is Data and Privacy; they argue that the limitations of data sources (such as electronic health records) can introduce bias because they were only ever intended to be used (in their current state) for clinical care and billing. They also state that bills such as the General Data Protection Regulation (GDPR) bringing well-deserved attention to the storage and processing of personal data people are significantly more skeptical of what personal data is being collected and how it is being used. Therefore, it may create greater pressure on researchers to identify a concrete reason for including a data stream; this has the potential to introduce bias through the choice of which data streams are available to a given machine learning model and which are deemed unnecessary (Ternent, James Thompson, Maximilian 2020).

lastly, they also mention the Lack of Acceptable Explanation of Decision Making. They argue that exposing the facets of a problem considered by a neural network highlights an important ethical consideration: what potential aspects of a decision are suitable for an ethical explanation, and on what level should unethical aspects be removed or recontextualized? For instance, suppose skin tone was an emergent feature in a machine learning model; this would clearly be an unacceptable facet of a patient to consider. Therefore, there will be confusion in making decisions in medical diagnoses.

3 EXPERIMENTATION

From all the 3 worksheets, I have implemented a Genetic Algorithm containing Fitness calculation, Initialisation, Tournament selection, One-point crossover, and Bit-flipping mutation. All processes are in a form of function and are executed when I called those. And in order to compare with the Tournament selection, I also created a new function that illustrates the method of Roulette Wheel selection, so that when I execute 2 GAs with different selection methods within a program, I can evaluate and plot the performance between those via matplotlib.pyplot library and the recorded results will be averages shown over 10 runs. Since we are working with not only maximization but also minimization, I have created 2 types of optimization programs, one for maximization using Counting One's function and two for minimization using respectively Rastrigin function and Ackley function to calculate fitness. From worksheet 3, we are required to change fitness calculation to work with real numbers instead of the binary representation (0 and 1).

This is my fitness calculation function with reals, wherein the fitness of an individual is equal to the number of '1's in its array of genes (genome) is easily extended to the sum of the real-valued genes:

```
# Calculate individual's fitness
# the individual's fitness is equal to the number of '1's in its array of genes (genome)
# instance parameter
def counting_ones(ind):
    fitness = 0
    for i in range(0, N):
        # if(ind.gene[i] == 1): # if gene of an individual at index i equals to 1
            fitness = fitness + ind.gene[i]
    return fitness
```

Figure 1.6 Counting One's fitness calculation function, see the equation in figure 1.0

Two below are my Rastrigin and Ackley fitness calculation functions with real numbers:

```
# Calculate individual's fitness
def mini_function(ind):
    fitness = 0
    squared = 0
    cosin = 0
    for i in range(0, N):
        squared = ind.gene[i] * ind.gene[i] # x^2
        cosin = 10 * math.cos(ind.gene[i] * (2 * math.pi)) # 10.cos(2.pi.x)
        fitness += squared - cosin # x^2 - 10.cos(2.pi.x)
    fitness += 10 * N # 10.N + x^2 - 10.cos(2.pi.x)
    return fitness
```

Figure 1.7 Rastrigin fitness calculation function, see the equation in figure 1.1

```
# Calculate individual's fitness
# first minimisation function
def mini_function(ind):
    fitness = 0
    first_exp = 0
    second_exp = 0
    for i in range(0, N):
        first_exp += math.cos(2 * math.pi * ind.gene[i])
        second_exp += ind.gene[i] * ind.gene[i]
    first_exp = math.exp((1/N) * first_exp)
    second_exp = (-20) * math.exp((-0.2) * math.sqrt((1/N) * second_exp))
    fitness = second_exp - first_exp
    return fitness
```

Figure 1.8 Ackley fitness calculation function

And what I have done to my fitness calculation functions was that I passed the instance of the individual's population (ind) as a parameter, access each genome by looping from i to the size of individual's genes (ind.gene[i]), calculate and return its fitness.

Since Counting One's function changed to work better with real numbers, I also need to alter the range in each process in a form of float numbers, for example:

Maximization program (Counting One's):

```
# Initialise original population
def initialise_population():
tempgene.append(random.uniform(0.0, 1.0)) # a random gene between 0.0 and 1.0(inclusive)

# Roulette Wheel Selection Process
def RW_selection(population):
selection_point = random.uniform(0.0, initial_fits)

#Bit-wise Mutation
def mutation(crosover_offspring, MUTRATE, MUTSTEP):
ALTER = random.uniform(0.0, MUTSTEP)
MUTPROB = random.uniform(0.0, 100.0)
if (MUTPROB < (100*MUTRATE)):
    if(random.randint(0, 1) == 1): # if random num is 1, add ALTER
        gene += ALTER
    else: # if random num is 0, minus ALTER
        gene -= ALTER
    if(gene > 1.0): # if gene value is larger than 1.0, reset it to 1.0
        gene = 1.0
    if(gene < 0.0): # if gene value is smaller than 0.0, reset it to 0.0
        gene = 0.0
```

Counting One’s range is [0.0, 1.0]

Minimization program (Rastrigin function):

```
# Initialise original population
def initialise_population():
tempgene.append(random.uniform(-5.12, 5.12)) # a random gene between -5.12 and 5.12(inclusive)

# Roulette Wheel Selection Process
def RW_selection(population):
selection_point = random.uniform(0.0, total)

#Bit-wise Mutation
def mutation(crossover_offspring, MUTRATE, MUTSTEP):
ALTER = random.uniform(0.0, MUTSTEP)
MUTPROB = random.uniform(0.0, 100.0)
if (MUTPROB < (100*MUTRATE)):
    if(random.randint(0, 1) == 1): # if random num is 1, add ALTER
        gene += ALTER
    else: # if random num is 0, minus ALTER
        gene -= ALTER
    if(gene > 5.12): # if gene value is larger than 5.12, reset it to 5.12
        gene = 5.12
    if(gene < -5.12): # if gene value is smaller than -5.12, reset it to -5.12
        gene = -5.12
```

Rastrigin range is [-5.12, 5.12]

Minimization program (Ackley function):

```
# Initialise original population
def initialise_population():
tempgene.append(random.uniform(-32.0, 32.0)) # a random gene between -5.12 and 5.12(inclusive)

# Roulette Wheel Selection Process
def RW_selection(population):
selection_point = random.uniform(0.0, total)

#Bit-wise Mutation
def mutation(crossover_offspring, MUTRATE, MUTSTEP):
```

```
        ALTER = random.uniform(0.0, MUTSTEP)
        MUTPROB = random.uniform(0.0, 100.0)
        if (MUTPROB < (100*MUTRATE)):
            if(random.randint(0, 1) == 1): # if random num is 1, add ALTER
                gene += ALTER
            else: # if random num is 0, minus ALTER
                gene -= ALTER
            if(gene > 32.0): # if gene value is larger than 32.0, reset it to 32.0
                gene = 32.0
            if(gene < -32.0): # if gene value is smaller than -32.0, reset it to -32.0
                gene = -32.0
```

Ackley range is [-32.0, 32.0]

Additionally, I also made changes to the Crossover algorithm to make it work correctly and better since the old crossover returned wrong individual's gene after swapping and I assume the reason could be the changes of temp when we assign back to offspring[i + 1]. I also tried to use deepcopy but it did not work in ways as I expected.

Old Crossover

```
temp = individual()
for i in range(0, P, 2):
    temp = coopy.deepcopy(offspring[i])
    crosspoint = random.randint(0, N - 1)
    for j in range(crosspoint, N):
        offspring[i].gene[j] = offspring[i + 1].gene[j].copy()
        offspring[i + 1].gene[j] = temp.gene[j].copy()
```

New Crossover

```
# Single-point Crossover process
def crossover(offspring):
    # Recombine pairs of parents from offspring
    crossover_offspring = []
    for i in range(0, P, 2):
        crosspoint = random.randint(0, N - 1) #pick up one random point in the gene length
        # 2 new temporary instances
        temp1 = individual()
        temp2 = individual()
        # 2 heads and 2 tails
        head1 = []
        tail1 = []
        head2 = []
        tail2 = []
        # print(offspring[i].gene, offspring[i+1].gene, crosspoint)
        # 0 to crosspoint adding gene to each head
```

```
    for j in range(0, crosspoint):
        head1.append(offspring[i].gene[j])
        head2.append(offspring[i + 1].gene[j])
    # crosspoint to N adding gene to each tail
    for j in range(crosspoint, N):
        tail2.append(offspring[i + 1].gene[j])
        tail1.append(offspring[i].gene[j])
    # print("head1 + tail2")
    # print(head1, tail2)
    # print("head2 + tail1")
    # print(head2, tail1)
    temp1.gene = head1 + tail2 # add first gene after crossover to temp1
    temp2.gene = head2 + tail1 # add second gene after crossover to temp2
    temp1.fitness = counting_ones(temp1) # call counting_ones to add fitness to temporary indv
    temp2.fitness = counting_ones(temp2)
    # append temp1, temp2 respectively to crosover_offspring_offspring
    crossover_offspring.append(temp1)
    crossover_offspring.append(temp2)
    # print(crosover_offspring_offspring[i].gene, crosover_offspring_offspring[i+1].gene)

return crossover_offspring
```

Figure 1.9 Crossover process

And to even make the program more efficient, I have created an optimization function that illustrates the Elitism. Jadaan, Rajamani, & Rao (2008) state that Elitism involves replacing the worst chromosomes in the children population with the best members of the parent population. This operator has proved to increase the speed of convergence of the GA because it ensures that the best solution found in each generation is retained (Jadaan, Rajamani, & Rao, 2008).

In the Maximization program, I replaced the two worst in the new population with the two best in the old population, the best fitness is the max fitness (best fitness).

```python
# Optimisation
def optimising(population, new_population):
    # more optimising
    # sorting instance with descending fitness
    population = sorting(population)

    # take two instances with the best fitness in the old population at index 0 and index 1
    bestFit_old_1 = population[0]
    bestFit_old_2 = population[1]

    # overwrite the old population with mutate_offspring
    population = copy.deepcopy(new_population)

    # sorting instance with descending fitness
    population = sorting(population)

    # after deepcopy new pop to old pop
    # take the two instance with the worst fitness in the new population at index -1 and index -2
    worstFit_new_1 = population[-1]
    worstFit_new_2 = population[-2]

    # compare the fitness btw the ones in the old pop and the ones in the new pop
    # replace the two worst fitness/gene by the two best fitness/gene at specific index in the new population
    if(bestFit_old_1.fitness > worstFit_new_1.fitness):
        population[-1].fitness = bestFit_old_1.fitness
        population[-1].gene = bestFit_old_1.gene
    if(bestFit_old_2.fitness > worstFit_new_2.fitness):
        population[-2].fitness = bestFit_old_2.fitness
        population[-2].gene = bestFit_old_2.gene

    return population
```

Figure 2.0 Elitism optimization for Maximization

And the reverse in Minimization one, I replaced the two best in the new population with the two worst in the old population, the best fitness is the min fitness (worst fitness).

```python
# Minimisation Optimisation
def optimising(population, new_population):
    # more optimising
    # sorting instance with descending fitness
    population = sorting(population)

    # take the two instance with the worst fitness in the old population at index -1 and index -2
    worstFit_old_1 = population[-1]
    worstFit_old_2 = population[-2]

    # overwrite the old population with mutate_offspring
    population = copy.deepcopy(new_population)

    # sorting instance with descending fitness
    population = sorting(population)

    # after deepcopy new pop to old pop
    # take two instances with the best fitness in the new population at index 0 and index 1
    bestFit_new_1 = population[0]
    bestFit_new_2 = population[1]

    # compare the fitness btw the ones in the old pop and the ones in the new pop
    # replace the two best fitness/gene by the two worst fitness/gene at specific index in the new population
    if(worstFit_old_1.fitness < bestFit_new_1.fitness):
        population[0].gene = worstFit_old_1.gene
        population[0].fitness = worstFit_old_1.fitness
    if(worstFit_old_2.fitness < bestFit_new_2.fitness):
        population[1].gene = worstFit_old_2.gene
        population[1].fitness = worstFit_old_2.fitness
```

Figure 2.1 Elitism optimization Minimization

1. MAXIMIZATION RESULTS:

Two figures below are Tournament selection and Roulette Wheel selection that I have implemented to my maximization program using Counting One's function.

```
# Tourament Selection Process
def touranment_selection(population):
    offspring = []
    # Select two parents and recombine pairs of parents
    for i in range(0, P):
        parent1 = random.randint(0, P - 1)
        off1 = population[parent1]
        parent2 = random.randint(0, P - 1)
        off2 = population[parent2]
        if (off1.fitness > off2.fitness): # if one's fitness higher then add to temp offsptring
            offspring.append(off1)
        else:
            offspring.append(off2)

    return offspring
```

Figure 2.2 Tournament selection Maximization

```
# Roulette Wheel Selection Process
def RW_selection(population):
    # total fitness of initial pop
    initial_fits = total_fitness(population)

    offspring = copy.deepcopy(population)
    # Roulette Wheel Selection Process
    # Select two parents and recombine pairs of parents
    for i in range(0, P):
        selection_point = random.uniform(0.0, initial_fits)
        running_total = 0
        j = 0
        while running_total <= selection_point:
            running_total += population[j].fitness
            j += 1
            if(j == P):
                break
        offspring[i] = population[j-1]

    return offspring
```

Figure 2.3 Roulette Wheel selection Maximization

While the Tournament selection divides the chance of choosing individuals equally over the whole population, the Roulette Wheel selects individuals based on their fitness meaning better individuals (higher fitness) get more chance to be picked up.

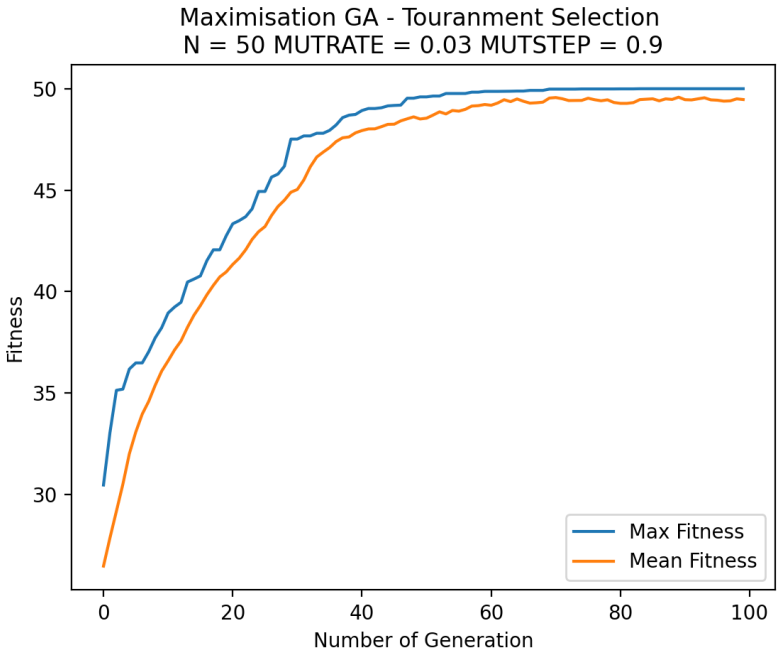


Figure 2.4 a single run of Maximization GA. – Tournament selection

Gene length: N = 50
Individual popultion: P = 50
Mutation rate: MUTRATE = 0.03
Mutation step: MUTSTEP = 0.9
Number of Generation: GENERATIONS = 100

Over 10 runs of my maximization GA with Tournament selection, I found that the max fitness increased and converged into a horizontal line at 50.0, while the mean fitness only reached 49.46629078875493. This plot was conducted over 100 cycles of generations.

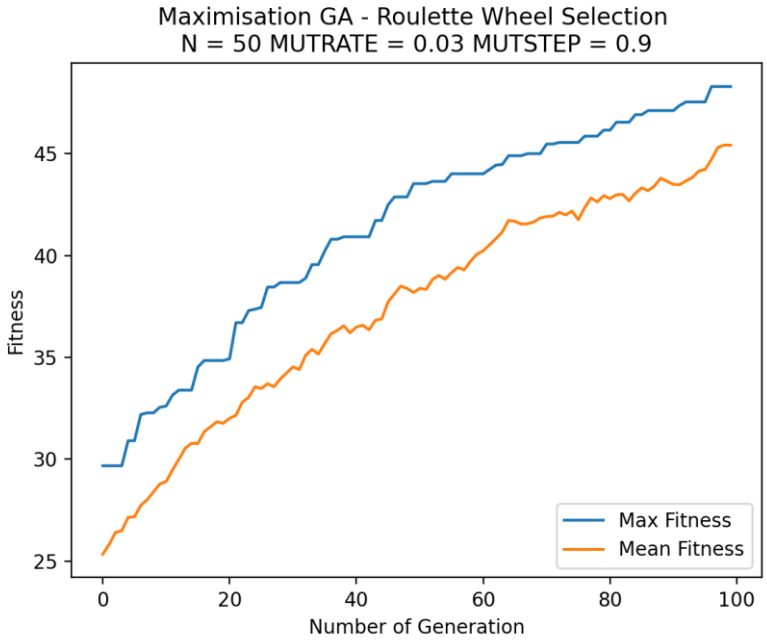


Figure 2.5 a single run of Maximization GA. – Roulette Wheel selection

I plotted the same statistics with my Maximization GA with Roulette Wheel selection over 10 runs, and even though we could see that both Max and Mean fitness increased, but they did not converge at all. The max fitness only reached 48.297451161234456 while the mean one was even lower, about 45.41549856413016.

After altering the Mutation rate and for both maximization GA with Tournament and Roulette Wheel over 4 different mutation rates, I noticed that the fitness reached max when MUTRATE = 0.03 and MUTSTEP = 0.9, which can be explained in the plot below:

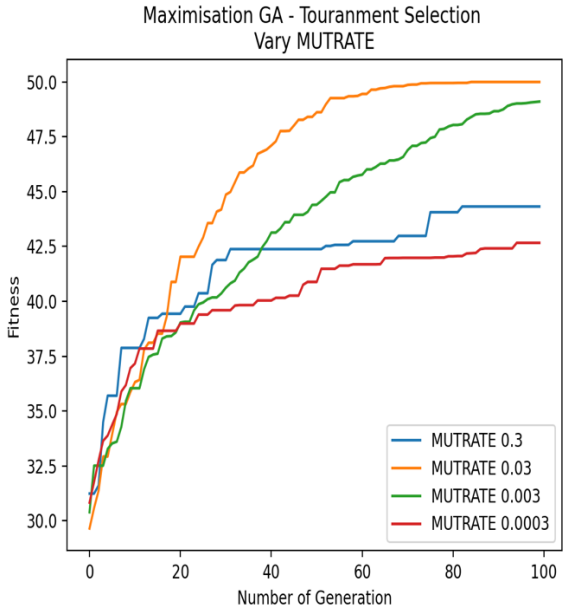


Figure 2.6 Max fitness at 50.0 - MUTRATE 0.03

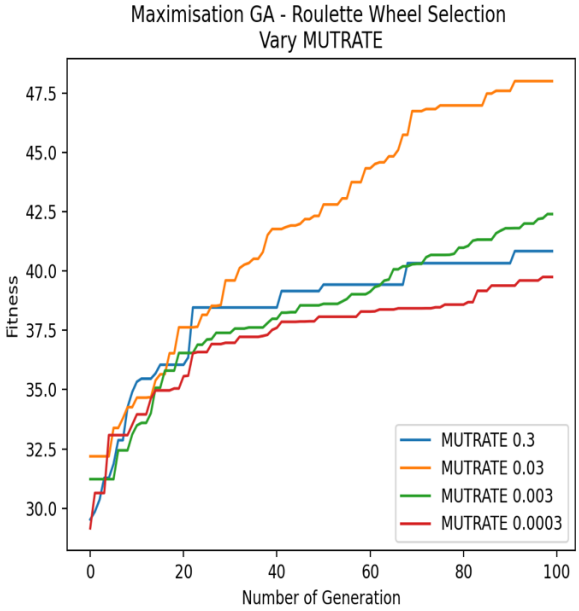


Figure 2.7 Max fitness at 48.001452528286876 – MUTRATE 0.03

I also made a plot to show a clear competitive performance between two selection methods as below:

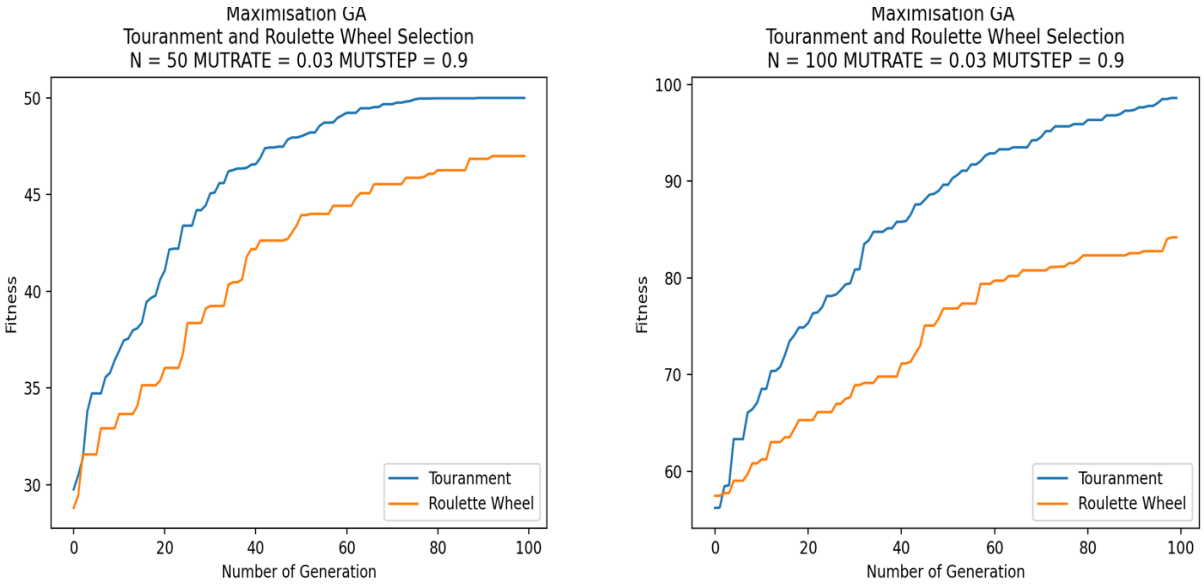


Figure 2.8 2.9 comparisons between Tournament selection and Roulette Wheel selection

The first plot with the gene length of 50 and the second one with the gene length of 100. As you can see, for both gene lengths, the fitness of Tournament selection always reaches its max first compared to those of Roulette Wheel selection. Tournament's fitness reached 50.0 in the first plot and 98.61599778697504 in the second one but never reaches 100.0 as I expect, so I think the problem might depend on the number of Generations, maybe when we increase the cycle, the fitness with 100 genes length hopefully be able to reach its target. In regard to the Roulette Wheel's fitness, it reached max at 46.9937534034408 for the first plot and 84.18984529423331 for the second one.

2. MINIMIZATION RESULTS:

As we are working with minimization program, then the best fitness that we get should be the smallest one and we as well should select the individuals with lower fitness in the population. Therefore, there are going to be some changes in the Selection process while implementing GA.

2.1 Rastrigin minimization function

From the equation in figure 1.0 – Rastrigin minimization function, I have made some changes in Tournament selection and Roulette Wheel selection compared against those in figure 2.2 and 2.3.

```
# Tourament Selection Process
def touranment_selection(population):
    offspring = []
    # Select two parents and recombine pairs of parents
    for i in range(0, P):
        parent1 = random.randint(0, P - 1)
        off1 = population[parent1]
        parent2 = random.randint(0, P - 1)
        off2 = population[parent2]
        if (off1.fitness > off2.fitness): # if one's fitness higher then add the smaller one to temp offsptring
            offspring.append(off2)
        else:
            offspring.append(off1)

    return offspring
```

Figure 3.0 Minimization Tournament selection Rastrigin function

In this minimization Tournament selection, I changed the if condition so that the individual with smaller fitness will be selected for the offspring population.

```
# Roulette Wheel Selection Process
def RW_selection(population):
    # total fitness of initial pop
    total = 0
    for ind in population:
        total += 1/ind.fitness

    offspring = []
    # Roulette Wheel Selection Process
    # Select two parents and recombine pairs of parents
    for i in range(0, P):
        selection_point = random.uniform(0.0, total)
        running_total = 0
        j = 0
        while running_total <= selection_point:
            running_total += 1 / (population[j].fitness)
            j += 1
            if(j == P):
                break

        # print(running_total)
        # print(j)
        offspring.append(copy.deepcopy(population[j-1]))

    return offspring
```

Figure 3.1 Minimization Roulette Wheel selection Rastrigin function

In this minimization Roulette Wheel selection, we wanted to make sure the individuals with small fitness will get more chance to be selected, so what I did was to inverse the fitness of the individuals and took the sum of it, and that will make the algorithm pick up the weaker individuals.

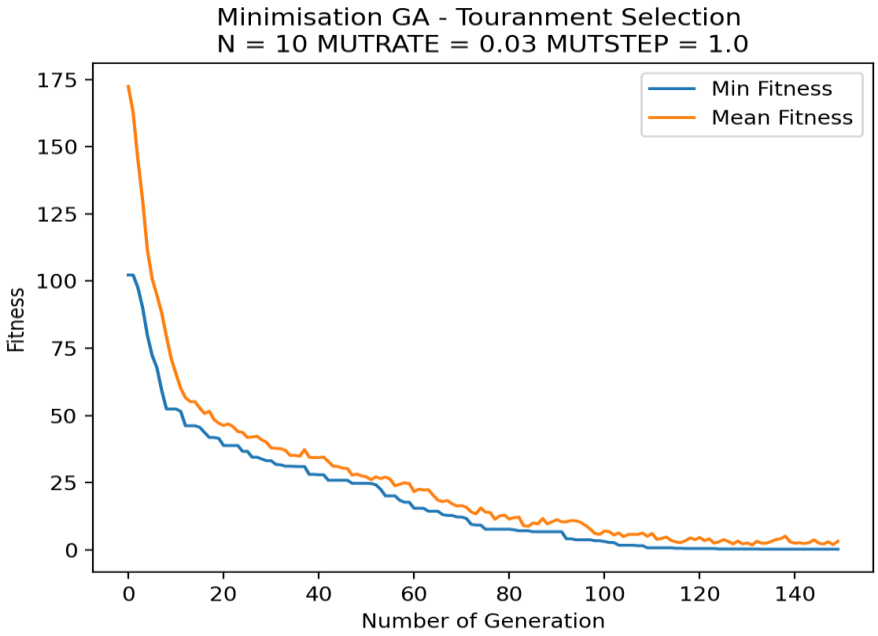


Figure 3.2 a single run of Minimization GA. – Tournament selection

Gene length: N = 10
Individual popultion: P = 50
Mutation rate: MUTRATE = 0.03
Mutation step: MUTSTEP = 1.0
Number of Generation: GENERATIONS = 150

Over 10 runs of my minimization GA with Tournament selection, I found that the min fitness decreased sharply and converged into a horizontal line at 0.2762741152367312 which is the best fitness that I can get over 10 different runs, however, I expect to see it drops to 0.0 but the more cycles of generations I increased, it still stayed at nearly 0.0 and I think it will make the algorithm more realistic to have some approximate error, while the mean fitness only reduced to 3.217860610617649. This plot was conducted over 150 cycles of generations.

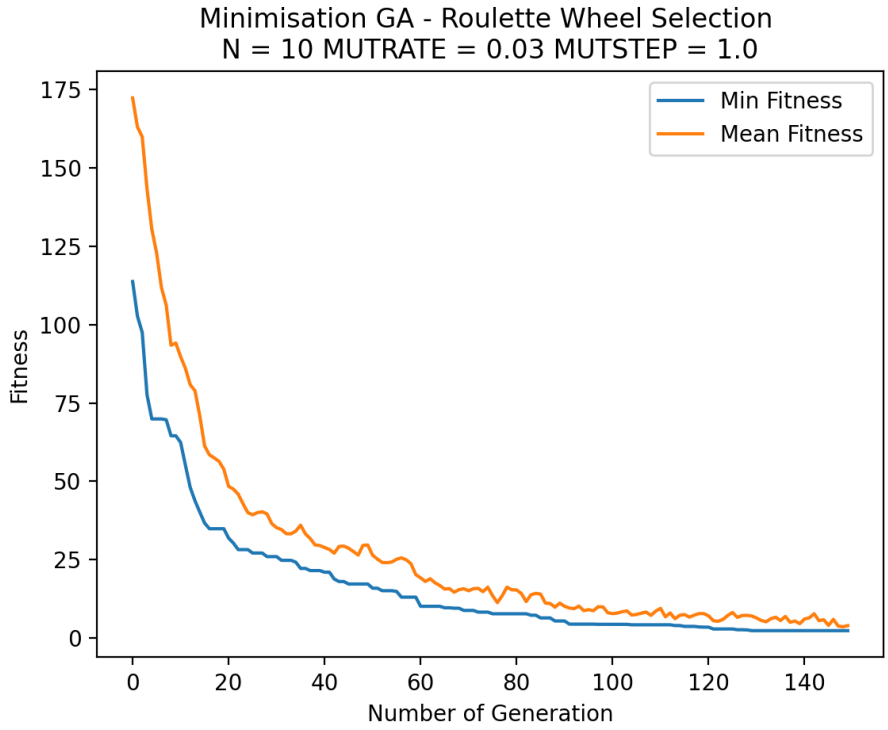
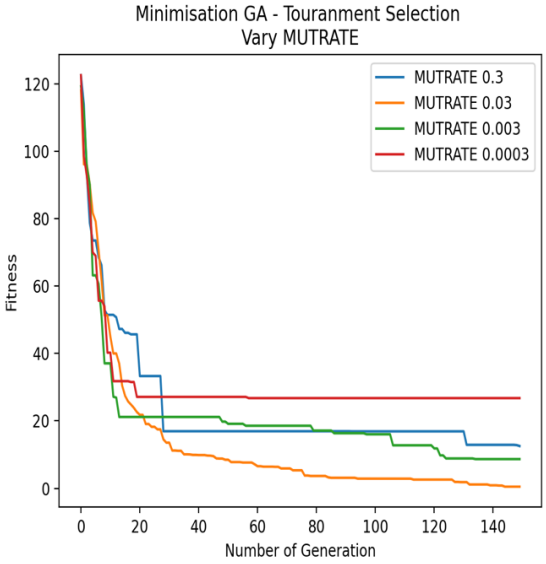


Figure 3.3 a single run of Minimization GA. – Roulette Wheel selection

With the same statistics, I also plotted a minimization GA using Roulette Wheel Selection and the lowest possible fitness achieved over 10 runs was 2.347698494062243 and the mean fitness was 3.9497076377781446

I also made a variation of Mutation rate and Mutation step, and after altering and changing the Mutation step from 0.3 to 0.7 to 1.0, the best one that made the fitness decreased that much is 1.0 and the best possible mutation rate is 0.03 as always.



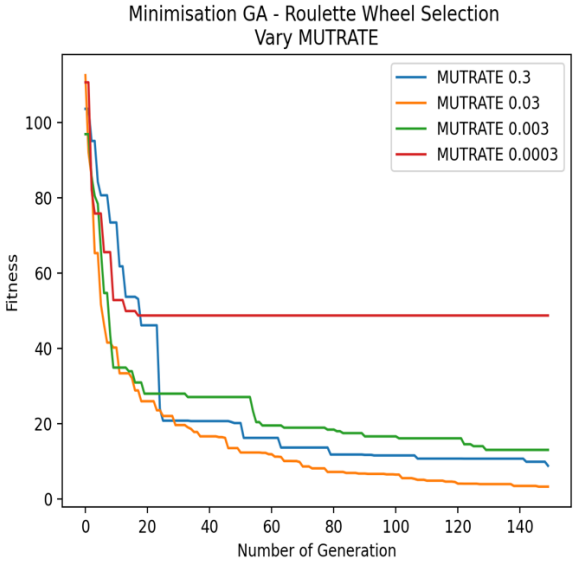


Figure 3.4 Max fitness at 0.4921120534116028 - MUTRATE 0.03 Figure 3.5 Max fitness at 3.3014290163064572 - MUTRATE 0.03

The next plots would be the comparison between two selection methods:

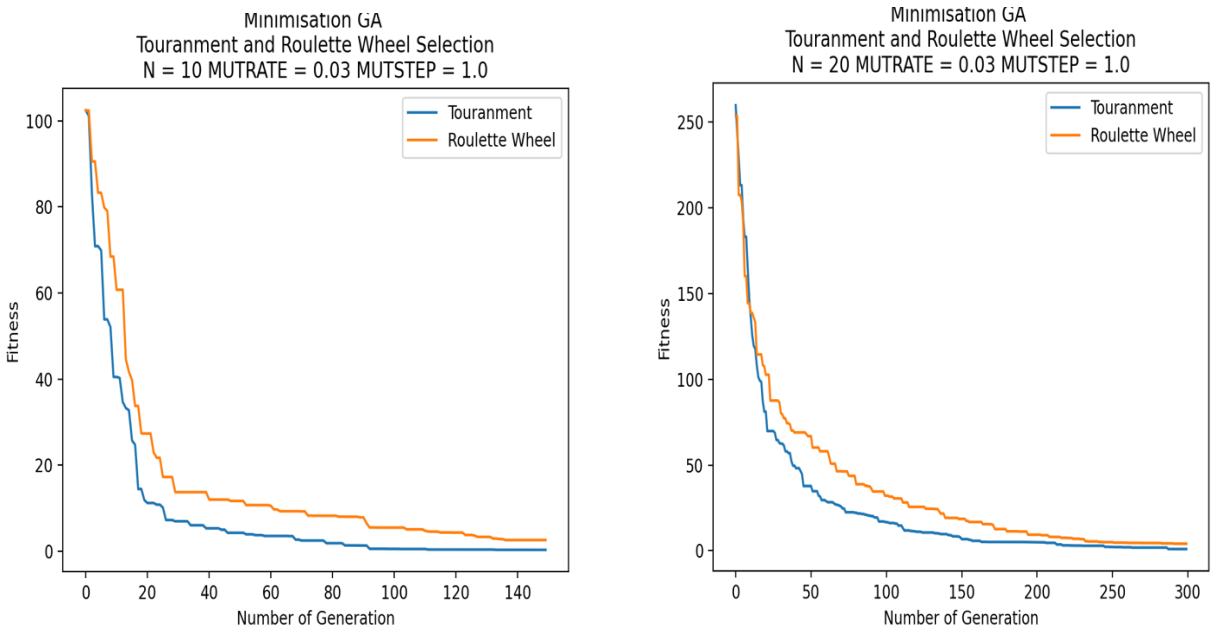


Figure 3.6 3.7 comparisons between Tournament selection and Roulette Wheel selection

As we can see that both selection methods work well for minimization GA (Rastrigin function), but Tournament selection seems likely to be better. The min fitness of Tournament is 0.35614317207772217, while the one of Roulette Wheel is 2.646551764532333. Since I wanted to challenge my GA over a larger gene length which is 20, I noticed that it took a bit longer to process and conducted more numbers of generations (300 generations) to converge horizontal line.

2.2 Ackley minimization function

Another minimization function that I need to show the experimental results is the Ackley function from figure 1.1. The only changes that I needed to make are the Roulette Wheel selection.

```
# Roulette Wheel Selection Process
def RW_selection(population):
    # total fitness of initial pop
    total = 0
    for ind in population:
        total += abs(ind.fitness)

    offspring = []
    # Roulette Wheel Selection Process
    # Select two parents and recombine pairs of parents
    for i in range(0, P):
        selection_point = random.uniform(0.0, total)
        running_total = 0
        j = 0
        while running_total <= selection_point:
            running_total += abs(population[j].fitness)
            j += 1
            if(j == P):
                break

        # print(running_total)
        # print(j)
        offspring.append(copy.deepcopy(population[j-1]))

    return offspring
```

Figure 3.8 Minimization Roulette Wheel selection Ackley function

since Ackley function returns negative fitness always so that I used the abs() in math library to convert all negative numbers into positive ones. As a result, the largest negative number will become the smallest positive one and reverse, which looks like the Roulette Wheel selection for maximization program.

The next plot would be the comparison between Tournament and Roulette Wheel selection over the minimization GA of Ackley function.

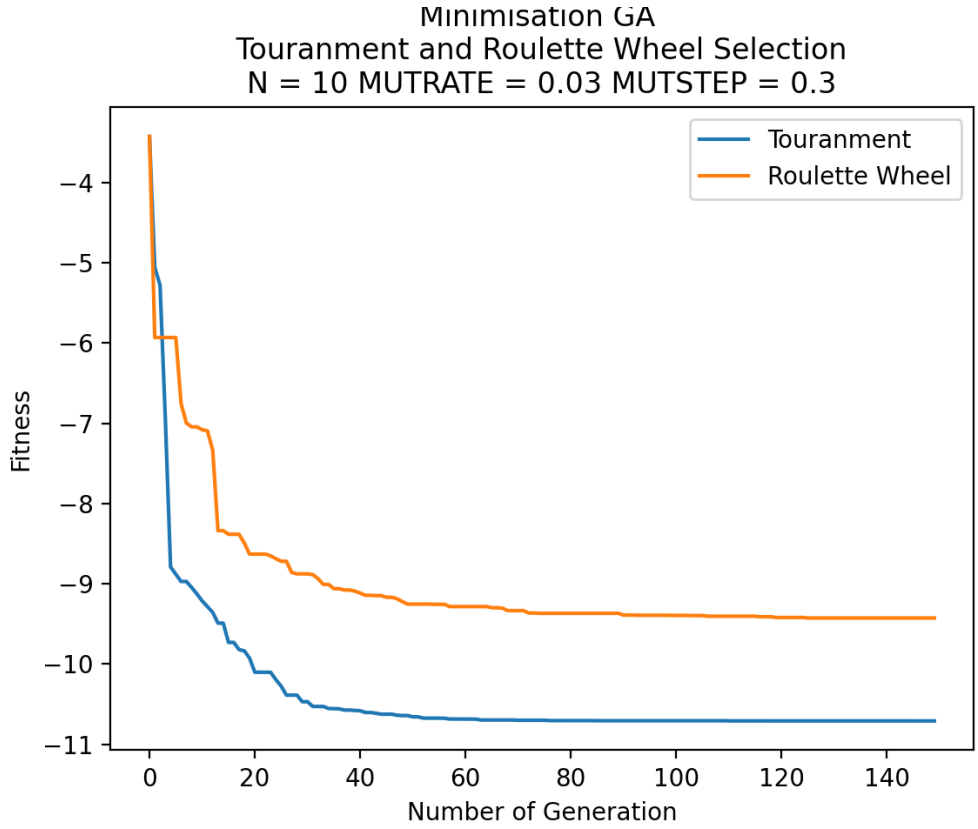


Figure 3.9 comparison between Tournament selection and Roulette Wheel selection

Gene length: N = 10
Individual popultion: P = 50
Mutation rate: MUTRATE = 0.03
Mutation step: MUTSTEP = 0.3
Number of Generation: GENERATIONS = 150

In this plot, each se**lec**tion method converged at its own line. The finesses of using Tournament selection and Roulette Wheel selection converged horizontally respectively at -10.708492338435587 and -9.426192516643226 over 150 numbers of generations. Both functions used mutation rate of 0.03 and mutation step of 0.3. I also found that it took a bit longer for Roulette Wheel selection to print out result than Tournament selection.

4 CONCLUSIONS

From all the comparisons between Tournament and Roulette Wheel selection in Figure 2.8 2.9 3.6 3.7, I noticed that the performance of Tournament always reached the optimum fitness better than those of Roulette Wheel regardless of the same statistics. In the maximization program, with the same statistics (mutation rate of 0.03, mutation step of 0.9, 100 numbers of generations) the Tournament selection works better and reaches its max fitness while the Roulette wheel does not. I also found that with 0.03 MUTRATE and 0.9 MUTSTEP that work both for the two selections, however, I believe that their performances might change if we set the statistics not equally, which is not fair to either of the selection methods. In minimization programs, the mutation rate stays the same for both minimization functions which is 0.03, while the mutation step altered 1.0

for Rastrigin function and 0.3 for Ackley function. The change of mutation for each minimization program did not affect the whole GAs, especially the selection methods since I both compared Tournament and Roulette Wheel selection for each minimization program. And once again, the convergence of Tournament looks better than that of Roulette Wheel in all optimization function, which could be explained that the Roulette Wheel selection is based on the probability, the fitter individuals get more chance to be selected, but there is a possibility that the weaker individuals get chosen as well, so the algorithm cannot always optimize the highest fitness as it can, however, the Tournament selection is possible to do that since it always requires and compares two random individuals in the population, and the one with higher fitness will be selected in the offspring, therefore, the algorithm will be able to optimize the best fitness over numbers of generations. From all those facts, I finally can make a conclusion that Tournament selection is a better algorithm to work within GAs.

REFERENCES

Jenna Carr. (2014). An Introduction to Genetic Algorithms [Accessed 1 Dec 2020].

Jadaan, O.A., Rajamani, L., & Rao, C.R. (2008). IMPROVED SELECTION OPERATOR FOR GA 1 [Accessed 1 Dec 2020].

Wikipedia, Tournament selection. Available URL: https://en.wikipedia.org/wiki/Tournament_selection [Accessed 5 Dec 2020].

Razali, Noraini & Geraghty, John. (2011). Genetic Algorithm Performance with Different Selection Strategies in Solving TSP. 2 [Accessed 10 Dec 2020].

A. Shukla, H. M. Pandey and D. Mehrotra. (2015). "Comparative review of selection techniques in genetic algorithm," [Accessed 13 Dec 2020].

Ternent, J., & Thompson, M. (2020). *Ethical Considerations of Artificial Intelligence via Neural Networks Applied to Medical Applications*. Worcester Polytechnic Institute [Accessed 20 Dec 2020].

Source code as an appendix
access my Github link to easily see the code
https://github.com/Liam1809/Bio_Computation

MAXIMIZATION PROGRAM (COUNTING ONE'S):

```
import random
import copy
import matplotlib.pyplot as plt

# Each individual should be represented by a data structure,
# consisting of an array of binary genes and a fitness value
class individual:
    gene = []
    fitness = 0

    def __repr__(self):
        return "Gene string " +  "".join(str(x) for x in self.gene) + " - fitness: " + str(self.fitness)
```

```
# The initial population array of such individuals,
# and random gene length number of 50 and population of 50
P = 50
N = 50
GENERATIONS = 100 # initialise 100 generations

# random mutation rate and mutation step
MUTRATE = 0.03
MUTSTEP = 0.9

# Calculate individual's fitness
# the individual's fitness is equal to the number of ‘1’s in its array of genes (genome)
# instance parameter
def counting_ones(ind):
    fitness = 0
    for i in range(0, N):
        # if(ind.gene[i] == 1): # if gene of an individual at index i equals to 1
            fitness = fitness + ind.gene[i]
    return fitness

# Calculate population's fitness
# list parameter
def total_fitness(population):
    totalfit = 0
    for ind in population:
        totalfit += ind.fitness
    return totalfit

# Initialise original population
def initialise_population():
    population = []
    # Initialise population with random candidate solutions
    # Generate random genes and append to a temporary gene list
    # Assign fitness and gene to individual and append one to population
    for x in range(0, P):
        tempgene = []
        for x in range(0, N):
            tempgene.append(random.uniform(0.0, 1.0)) # a random gene between 0.0 and 1.0(inclusive)
        # print(tempgene)
        newindi = individual() # initialise new instance
        newindi.gene = tempgene.copy() # copy the gene from tempgene and assign to gene of individual
        newindi.fitness = counting_ones(newindi) # initialise instance's fitness
        population.append(newindi)
    return population

# Tourament Selection Process
def touranment_selection(population):
```

```
    offspring = []
    # Select two parents and recombine pairs of parents
    for i in range(0, P):
        parent1 = random.randint(0, P - 1)
        off1 = population[parent1]
        parent2 = random.randint(0, P - 1)
        off2 = population[parent2]
        if (off1.fitness > off2.fitness): # if one's fitness higher then add to temp offsptring
            offspring.append(off1)
        else:
            offspring.append(off2)

    return offspring

# Roulette Wheel Selection Process
def RW_selection(population):
    # total fitness of initial pop
    initial_fits = total_fitness(population)

    offspring = copy.deepcopy(population)
    # Roulette Wheel Selection Process
    # Select two parents and recombine pairs of parents
    for i in range(0, P):
        selection_point = random.uniform(0.0, initial_fits)
        running_total = 0
        j = 0
        while running_total <= selection_point:
            running_total += population[j].fitness
            j += 1
            if(j == P):
                break
        offspring[i] = population[j-1]

    return offspring

# Single-point Crossover process
def crossover(offspring):
    # Recombine pairs of parents from offspring
    crossover_offspring = []
    for i in range(0, P, 2):
        crosspoint = random.randint(0, N - 1) #pick up one random point in the gene length
        # 2 new temporary instances
        temp1 = individual()
        temp2 = individual()
        # 2 heads and 2 tails
        head1 = []
        tail1 = []
        head2 = []
```

```
        tail2 = []
        # print(offspring[i].gene, offspring[i+1].gene, crosspoint)
        # 0 to crosspoint adding gene to each head
        for j in range(0, crosspoint):
            head1.append(offspring[i].gene[j])
            head2.append(offspring[i + 1].gene[j])
        # crosspoint to N adding gene to each tail
        for j in range(crosspoint, N):
            tail2.append(offspring[i + 1].gene[j])
            tail1.append(offspring[i].gene[j])
        # print("head1 + tail2")
        # print(head1, tail2)
        # print("head2 + tail1")
        # print(head2, tail1)
        temp1.gene = head1 + tail2 # add first gene after crossover to temp1
        temp2.gene = head2 + tail1 # add second gene after crossover to temp2
        temp1.fitness = counting_ones(temp1) # call counting_ones to add fitness to temporary indv
        temp2.fitness = counting_ones(temp2)
        # append temp1, temp2 respectively to crosover_offspring_offspring
        crossover_offspring.append(temp1)
        crossover_offspring.append(temp2)
        # print(crosover_offspring_offspring[i].gene, crosover_offspring_offspring[i+1].gene)

    return crossover_offspring

#Bit-wise Mutation
def mutation(crosover_offspring, MUTRATE, MUTSTEP):
    # Mutate the result of new_offspring
    mutate_offspring = []
    for i in range(0, P):
        new_indi = individual()
        new_indi.gene = []
        for j in range(0, N):
            gene = crosover_offspring[i].gene[j]
            ALTER = random.uniform(0.0, MUTSTEP)
            MUTPROB = random.uniform(0.0, 100.0)
            if (MUTPROB < (100*MUTRATE)):
                if(random.randint(0, 1) == 1): # if random num is 1, add ALTER
                    gene += ALTER
                else: # if random num is 0, minus ALTER
                    gene -= ALTER
                if(gene > 1.0): # if gene value is larger than 1.0, reset it to 1.0
                    gene = 1.0
                if(gene < 0.0): # if gene value is smaller than 0.0, reset it to 0.0
                    gene = 0.0
            new_indi.gene.append(gene) # add gene to instance
        new_indi.fitness = counting_ones(new_indi) # add fitness to instance by calling counting_ones
        mutate_offspring.append(new_indi)
```

```
    return mutate_offspring

# Descending sorting
def sorting(population):
    #  descending sorting based on  individual's fitness
    population.sort(key=lambda individual:individual.fitness, reverse=True)

    return population

# Optimisation
def optimising(population, new_population):
    # more optimising
    # sorting instance with descending fitness
    population = sorting(population)

    # take two instances with the best fitness in the old population at index 0 and index 1
    bestFit_old_1 = population[0]
    bestFit_old_2 = population[1]

    # overwrite the old population with mutate_offspring
    population = copy.deepcopy(new_population)

    # sorting instance with descending fitness
    population = sorting(population)

    # after deepcopy new pop to old pop
    # take the two instance with the worst fitness in the new population at index -1 and index -2
    worstFit_new_1 = population[-1]
    worstFit_new_2 = population[-2]

    # compare the fitness btw the ones in the old pop and the ones in the new pop
    # replace the two worst fitness/gene by the two best fitness/gene at specific index in the new
    population
    if(bestFit_old_1.fitness > worstFit_new_1.fitness):
        population[-1].fitness = bestFit_old_1.fitness
        population[-1].gene = bestFit_old_1.gene
    if(bestFit_old_2.fitness > worstFit_new_2.fitness):
        population[-2].fitness = bestFit_old_2.fitness
        population[-2].gene = bestFit_old_2.gene

    return population

def GA(population, Selection, MUTRATE, MUTSTEP):
    # storing data to plot
    meanFit_values = []
    maxFit_values = []
    # ===========GENETIC ALGORITHM==============
```

```
    for gen in range(0, GENERATIONS):
        # # touranment/ RW selection process
        offspring = Selection(population)

        # crossover process
        crossover_offspring = crossover(offspring)
        # mutation process
        mutate_offspring = mutation(crossover_offspring, MUTRATE, MUTSTEP)
        # optimising
        population = optimising(population, mutate_offspring)

        # calculate Max and Mean Fitness
        # storing fitness in a list
        Fit = []
        for ind in population:
            Fit.append(counting_ones(ind))
        # print(Fit)

        maxFit = max(Fit) # take out the max fitness among fitnesses in Fit
        meanFit = sum(Fit)/ P # sum all the fitness and divide by Population size

        # append maxFit and meanFit respectively to MaxFit_values and MeanFit_values
        maxFit_values.append(maxFit)
        meanFit_values.append(meanFit)

        # display
        # print("GENERATION " + str(gen + 1))
        # print("Mean Fitness: " + str(meanFit))
        # print("Max Fitness: " + str(maxFit) + "\n")
    print("Max Fitness: " + str(maxFit) + "\n")
    print("Mean Fitness: " + str(meanFit) + "\n")

    return maxFit_values, meanFit_values

# plotting
plt.ylabel("Fitness")
plt.xlabel("Number of Generation")

#  Storing
maxFit_data1 = []
maxFit_data2 = []
maxFit_data3 = []
maxFit_data4 = []

meanFit_data1 = []
meanFit_data2 = []
meanFit_data3 = []
```

```
meanFit_data4 = []


# EXPERIMENT

# ============================================================
# TOURANMENT vs ROULETTE WHEEL SELECTION COMPARISON
# ============================================================

# [----------------- UNCOMMENT THIS AND ALTER N TO TEST -----------------]
N = 50
plt.title("Maximisation GA \n Touranment and Roulette Wheel Selection \n"
          + "N = " + str(N) + " MUTRATE = " + str(MUTRATE) + " MUTSTEP = " + str(MUTSTEP))

# initialise original population
population = initialise_population()

maxFit_data1, meanFit_data1 = GA(population, touranment_selection, 0.03, 0.9)
maxFit_data2, meanFit_data2 = GA(population, RW_selection, 0.03, 0.9)

plt.plot(maxFit_data1, label="Touranment")
plt.plot(maxFit_data2, label="Roulette Wheel")
# [----------------- UNCOMMENT THIS AND ALTER N TO TEST -----------------]

# N = 50
# MUTRATE = 0.03
# MUTSTEP = 0.9
# Max Fitness: 50.0 - TS
# Max Fitness: 46.9937534034408 - RW



# N = 100
# MUTRATE = 0.03
# MUTSTEP = 0.9
# Max Fitness: 98.61599778697504 - TS
# Max Fitness: 84.18984529423331 - RW



# ============================================================
# TOURANMENT SELECTION
# ============================================================


# Best Fitness and Mean Fitness of TS
# [----------------- UNCOMMENT THIS TO TEST -----------------]
# plt.title("Maximisation GA - Touranment Selection \n"
```

```
#           + "N = " + str(N) + " MUTRATE = " + str(MUTRATE) + " MUTSTEP = " + str(MUTSTEP))

# # initialise original population
# population = initialise_population()

# maxFit_data1, meanFit_data1 = GA(population, touranment_selection, 0.03, 0.9)

# plt.plot(maxFit_data1, label="Max Fitness")
# plt.plot(meanFit_data1, label="Mean Fitness")
# [----------------- UNCOMMENT THIS TO TEST -----------------]
# N = 50
# MUTRATE = 0.03
# MUTSTEP = 0.9
# Max Fitness: 50.0
# Mean Fitness: 49.46629078875493



# Vary MUTRATE
# [----------------- UNCOMMENT THIS TO TEST -----------------]
# plt.title("Maximisation GA - Touranment Selection \n"
#           + "Vary MUTRATE")

# # initialise original population
# population = initialise_population()

# maxFit_data1, meanFit_data1 = GA(population, touranment_selection, 0.3, 0.9)
# maxFit_data2, meanFit_data2 = GA(population, touranment_selection, 0.03, 0.9)
# maxFit_data3, meanFit_data3 = GA(population, touranment_selection, 0.003, 0.9)
# maxFit_data4, meanFit_data4 = GA(population, touranment_selection, 0.0003, 0.9)

# plt.plot(maxFit_data1, label="MUTRATE 0.3")
# plt.plot(maxFit_data2, label="MUTRATE 0.03")
# plt.plot(maxFit_data3, label="MUTRATE 0.003")
# plt.plot(maxFit_data4, label="MUTRATE 0.0003")
# [----------------- UNCOMMENT THIS TO TEST -----------------]
# N = 50
# MUTSTEP = 0.9
# Max Fitness: 44.322739025242676 - MUTRATE 0.3
# Max Fitness: 50.0 - MUTRATE 0.3
# Max Fitness: 49.10608786326543 - MUTRATE 0.3
# Max Fitness: 42.6666028718545 - MUTRATE 0.3



# ============================================================
# ROULETTE WHEEL SELECTION
```

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


# Best Fitness and Mean Fitness of RW
# [----------------- UNCOMMENT THIS TO TEST -----------------]
# plt.title("Maximisation GA - Roulette Wheel Selection \n"
#           + "N = " + str(N) + " MUTRATE = " + str(MUTRATE) + " MUTSTEP = " + str(MUTSTEP))

# # initialise original population
# population = initialise_population()

# maxFit_data1, meanFit_data1 = GA(population, RW_selection, 0.03, 0.9)

# plt.plot(maxFit_data1, label="Max Fitness")
# plt.plot(meanFit_data1, label="Mean Fitness")
# [----------------- UNCOMMENT THIS TO TEST -----------------]
# N = 50
# MUTRATE = 0.03
# MUTSTEP = 0.9
# Max Fitness: 48.297451161234456
# Mean Fitness: 45.41549856413016


# Vary MUTRATE
# [----------------- UNCOMMENT THIS TO TEST -----------------]
# plt.title("Maximisation GA - Roulette Wheel Selection \n"
#           + "Vary MUTRATE")

# # initialise original population
# population = initialise_population()

# maxFit_data1, meanFit_data1 = GA(population, RW_selection, 0.3, 0.9)
# maxFit_data2, meanFit_data2 = GA(population, RW_selection, 0.03, 0.9)
# maxFit_data3, meanFit_data3 = GA(population, RW_selection, 0.003, 0.9)
# maxFit_data4, meanFit_data4 = GA(population, RW_selection, 0.0003, 0.9)

# plt.plot(maxFit_data1, label="MUTRATE 0.3")
# plt.plot(maxFit_data2, label="MUTRATE 0.03")
# plt.plot(maxFit_data3, label="MUTRATE 0.003")
# plt.plot(maxFit_data4, label="MUTRATE 0.0003")
# [----------------- UNCOMMENT THIS TO TEST -----------------]
# N = 50
# MUTSTEP = 0.9
# Max Fitness: 40.83573375201377 - MUTRATE 0.3
# Max Fitness: 48.001452528286876 - MUTRATE 0.03
# Max Fitness: 42.398036522166805 - MUTRATE 0.003
# Max Fitness: 39.74645609300905 - MUTRATE 0.0003
```

```
# DISPLAY PLOT
plt.legend(loc = "lower right")
plt.show()
```

MINIMIZATION PROGRAM (RASTRIGIN FUNCTION):

```
import random
import copy
import math
import matplotlib.pyplot as plt

# Each individual should be represented by a data structure,
# consisting of an array of binary genes and a fitness value
class individual:
    gene = []
    fitness = 0

    def __repr__(self):
        return "Gene string " +  "".join(str(x) for x in self.gene) + " - fitness: " + str(self.fitness)

# The initial population array of such individuals,
# and random gene length number of 10 and population of 50
P = 50
N = 10
GENERATIONS = 150 # initialise 150 generations

# random mutation rate and mutation step
MUTRATE = 0.03
MUTSTEP = 1.0

# Calculate individual's fitness
def mini_function(ind):
    fitness = 0
    squared = 0
    cosin = 0
    for i in range(0, N):
        squared = ind.gene[i] * ind.gene[i] # x^2
        cosin = 10 * math.cos(ind.gene[i] * (2 * math.pi)) # 10.cos(2.pi.x)
        fitness += squared - cosin # x^2 - 10.cos(2.pi.x)
    fitness += 10 * N # 10.N + x^2 - 10.cos(2.pi.x)
    return fitness

# Calculate population's fitness
# list parameter
```

```
def total_fitness(population):
    totalfit = 0
    for ind in population:
        totalfit += ind.fitness
    return totalfit

# Initialise original population
def initialise_population():
    population = []
    # Initialise population with random candidate solutions
    # Generate random genes and append to a temporary gene list
    # Assign fitness and gene to individual and append one to population
    for x in range(0, P):
        tempgene = []
        for x in range(0, N):
            tempgene.append(random.uniform(-5.12, 5.12)) # a random gene between -5.12 and
    5.12(inclusive)
        # print(tempgene)
        newindi = individual() # initialise new instance
        newindi.gene = tempgene.copy() # copy the gene from tempgene and assign to gene of individual
        newindi.fitness = mini_function(newindi) # initialise instance's fitness
        population.append(newindi)
    return population

# Tourament Selection Process
def touranment_selection(population):
    offspring = []
    # Select two parents and recombine pairs of parents
    for i in range(0, P):
        parent1 = random.randint(0, P - 1)
        off1 = population[parent1]
        parent2 = random.randint(0, P - 1)
        off2 = population[parent2]
        if (off1.fitness > off2.fitness): # if one's fitness higher then add the smaller one to temp offsptring
            offspring.append(off2)
        else:
            offspring.append(off1)

    return offspring

# Roulette Wheel Selection Process
def RW_selection(population):
    # total fitness of initial pop
    total = 0
    for ind in population:
        total += 1/ind.fitness

    offspring = []
```

```
    # Roulette Wheel Selection Process
    # Select two parents and recombine pairs of parents
    for i in range(0, P):
        selection_point = random.uniform(0.0, total)
        running_total = 0
        j = 0
        while running_total <= selection_point:
            running_total += 1 / (population[j].fitness)
            j += 1
            if(j == P):
                break

        # print(running_total)
        # print(j)
        offspring.append(copy.deepcopy(population[j-1]))

    return offspring

# Single-point Crossover process
def crossover(offspring):
    # Recombine pairs of parents from offspring
    crossover_offspring = []
    for i in range(0, P, 2):
        crosspoint = random.randint(1, N - 1) #pick up one random point in the gene length
        # 2 new temporary instances
        temp1 = individual()
        temp2 = individual()
        # 2 heads and 2 tails
        head1 = []
        tail1 = []
        head2 = []
        tail2 = []
        # print(offspring[i].gene, offspring[i+1].gene, crosspoint)
        # 0 to crosspoint adding gene to each head
        for j in range(0, crosspoint):
            head1.append(offspring[i].gene[j])
            head2.append(offspring[i + 1].gene[j])
        # crosspoint to N adding gene to each tail
        for j in range(crosspoint, N):
            tail2.append(offspring[i + 1].gene[j])
            tail1.append(offspring[i].gene[j])
        # print("head1 + tail2")
        # print(head1, tail2)
        # print("head2 + tail1")
        # print(head2, tail1)
        temp1.gene = head1 + tail2 # add first gene after crossover to temp1
        temp2.gene = head2 + tail1 # add second gene after crossover to temp2
        temp1.fitness = mini_function(temp1) # call counting_ones to add fitness to temporary indv
```

```
        temp2.fitness = mini_function(temp2)
        # append temp1, temp2 respectively to crosover_offspring_offspring
        crossover_offspring.append(temp1)
        crossover_offspring.append(temp2)
        # print(crosover_offspring_offspring[i].gene, crosover_offspring_offspring[i+1].gene)

    return crossover_offspring

#Bit-wise Mutation
def mutation(crossover_offspring, MUTRATE, MUTSTEP):
    # Mutate the result of new_offspring
    #Bit-wise Mutation
    mutate_offspring = []
    for i in range(0, P):
        new_indi = individual()
        new_indi.gene = []
        for j in range(0, N):
            gene = crossover_offspring[i].gene[j]
            ALTER = random.uniform(0.0, MUTSTEP)
            MUTPROB = random.uniform(0.0, 100.0)
            if (MUTPROB < (100*MUTRATE)):
                if(random.randint(0, 1) == 1): # if random num is 1, add ALTER
                    gene += ALTER
                else: # if random num is 0, minus ALTER
                    gene -= ALTER
                if(gene > 5.12): # if gene value is larger than 5.12, reset it to 5.12
                    gene = 5.12
                if(gene < -5.12): # if gene value is smaller than -5.12, reset it to -5.12
                    gene = -5.12
            new_indi.gene.append(gene)
        new_indi.fitness = mini_function(new_indi) # add fitness to instance by calling mini_function
        mutate_offspring.append(new_indi)

    return mutate_offspring

# Descending sorting
def sorting(population):
    #  descending sorting based on individual's fitness
    population.sort(key=lambda individual:individual.fitness, reverse=True)

    return population

# Minimisation Optimisation
def optimising(population, new_population):
    # more optimising
    # sorting instance with descending fitness
    population = sorting(population)
```

```
    # take the two instance with the worst fitness in the old population at index -1 and index -2
    worstFit_old_1 = population[-1]
    worstFit_old_2 = population[-2]

    # overwrite the old population with mutate_offspring
    population = copy.deepcopy(new_population)

    # sorting instance with descending fitness
    population = sorting(population)

    # after deepcopy new pop to old pop
    # take two instances with the best fitness in the new population at index 0 and index 1
    bestFit_new_1 = population[0]
    bestFit_new_2 = population[1]

    # compare the fitness btw the ones in the old pop and the ones in the new pop
    # replace the two best fitness/gene by the two worst fitness/gene at specific index in the new
    population
    if(worstFit_old_1.fitness < bestFit_new_1.fitness):
        population[0].gene = worstFit_old_1.gene
        population[0].fitness = worstFit_old_1.fitness
    if(worstFit_old_2.fitness < bestFit_new_2.fitness):
        population[1].gene = worstFit_old_2.gene
        population[1].fitness = worstFit_old_2.fitness


    return population

def GA(population, Selection, MUTRATE, MUTSTEP):
    # ==========GENETIC ALGORITHM==============
    # storing data to plot
    meanFit_values = []
    minFit_values = []


    for gen in range(0, GENERATIONS):
        # touranment selection process / RW selection process
        offspring = Selection(population)
        # crossover process
        crossover_offspring = crossover(offspring)
        # mutation process
        mutate_offspring = mutation(crossover_offspring, MUTRATE, MUTSTEP)
        # optimising
        population = optimising(population, mutate_offspring)

        # calculate Max and Mean Fitness
        # storing fitness in a list
        Fit = []
```

```
        for ind in population:
            Fit.append(mini_function(ind))
        # print(Fit)

        minFit = min(Fit) # take out the min fitness among fitnesses in Fit
        meanFit = sum(Fit)/ P # sum all the fitness and divide by P size

        # append minFit and meanFit respectively to MinFit_values and MeanFit_values
        minFit_values.append(minFit)
        meanFit_values.append(meanFit)

        # display
        # print("GENERATION " + str(gen + 1))
    print("Min Fitness: " + str(minFit) + "\n")
    print("Mean Fitness: " + str(meanFit) + "\n")


    return minFit_values, meanFit_values

# plotting
plt.ylabel("Fitness")
plt.xlabel("Number of Generation")

# Storing
minFit_data1 = []
minFit_data2 = []
minFit_data3 = []
minFit_data4 = []

meanFit_data1 = []
meanFit_data2 = []
meanFit_data3 = []
meanFit_data4 = []


# EXPERIMENT

# ============================================================
# TOURANMENT vs ROULETTE WHEEL SELECTION COMPARISON
# ============================================================

# [----------------- UNCOMMENT THIS AND ALTER N TO TEST -----------------]
N = 10
plt.title("Minimisation GA \n Touranment and Roulette Wheel Selection \n"
          + "N = " + str(N) + " MUTRATE = " + str(MUTRATE) + " MUTSTEP = " + str(MUTSTEP))

# initialise original population
population = initialise_population()
```

```
minFit_data1, meanFit_data1 = GA(population, touranment_selection, 0.03, 1.0)
minFit_data2, meanFit_data2 = GA(population, RW_selection, 0.03, 1.0)

plt.plot(minFit_data1, label="Touranment")
plt.plot(minFit_data2, label="Roulette Wheel")
# [----------------- UNCOMMENT THIS AND ALTER N TO TEST -----------------]

# N = 10
# MUTRATE = 0.03
# MUTSTEP = 1.0
# Min Fitness: 0.35614317207772217 - TS
# Min Fitness: 2.646551764532333 - RW



# N = 20
# MUTRATE = 0.03
# MUTSTEP = 1.0
# Min Fitness: 0.9973430696900323 - TS
# Min Fitness: 4.102001421670991 - RW


# ============================================================
# TOURANMENT SELECTION
# ============================================================


# Best Fitness and Mean Fitness of TS
# [----------------- UNCOMMENT THIS TO TEST -----------------]
# plt.title("Minimisation GA - Touranment Selection \n"
#           + "N = " + str(N) + " MUTRATE = " + str(MUTRATE) + " MUTSTEP = " + str(MUTSTEP))

# # initialise original population
# population = initialise_population()

# minFit_data1, meanFit_data1 = GA(population, touranment_selection, 0.03, 1.0)

# plt.plot(minFit_data1, label="Min Fitness")
# plt.plot(meanFit_data1, label="Mean Fitness")
# [----------------- UNCOMMENT THIS TO TEST -----------------]
# N = 10
# MUTRATE = 0.03
# MUTSTEP = 1.0
# Min Fitness: 0.2762741152367312
# Mean Fitness: 3.217860610617649
```

```
# Vary MUTRATE
# [----------------- UNCOMMENT THIS TO TEST -----------------]
# plt.title("Minimisation GA - Touranment Selection \n"
#           + "Vary MUTRATE")

# # initialise original population
# population = initialise_population()

# minFit_data1, meanFit_data1 = GA(population, touranment_selection, 0.3, 1.0)
# minFit_data2, meanFit_data2 = GA(population, touranment_selection, 0.03, 1.0)
# minFit_data3, meanFit_data3 = GA(population, touranment_selection, 0.003, 1.0)
# minFit_data4, meanFit_data4 = GA(population, touranment_selection, 0.0003, 1.0)

# plt.plot(minFit_data1, label="MUTRATE 0.3")
# plt.plot(minFit_data2, label="MUTRATE 0.03")
# plt.plot(minFit_data3, label="MUTRATE 0.003")
# plt.plot(minFit_data4, label="MUTRATE 0.0003")
# [----------------- UNCOMMENT THIS TO TEST -----------------]
# N = 10
# MUTSTEP = 1.0
# Min Fitness: 12.5460038868992 - MUTRATE 0.3
# Min Fitness: 0.4921120534116028 - MUTRATE 0.03
# Min Fitness: 8.66909400410961 - MUTRATE 0.003
# Min Fitness: 26.745458844423908 - MUTRATE 0.0003



# ============================================================
# ROULETTE WHEEL SELECTION
# ============================================================


# Best Fitness and Mean Fitness of RW
# [----------------- UNCOMMENT THIS TO TEST -----------------]
# plt.title("Minimisation GA - Roulette Wheel Selection \n"
#           + "N = " + str(N) + " MUTRATE = " + str(MUTRATE) + " MUTSTEP = " + str(MUTSTEP))

# # initialise original population
# population = initialise_population()

# minFit_data1, meanFit_data1 = GA(population, RW_selection, 0.03, 1.0)

# plt.plot(minFit_data1, label="Min Fitness")
# plt.plot(meanFit_data1, label="Mean Fitness")
# [----------------- UNCOMMENT THIS TO TEST -----------------]
```

```
# N = 10
# MUTRATE = 0.03
# MUTSTEP = 1.0
# Min Fitness: 2.347698494062243
# Mean Fitness: 3.9497076377781446




# Vary MUTRATE
# [----------------- UNCOMMENT THIS TO TEST -----------------]
# plt.title("Minimisation GA - Roulette Wheel Selection \n"
#           + "Vary MUTRATE")

# # initialise original population
# population = initialise_population()

# minFit_data1, meanFit_data1 = GA(population, RW_selection, 0.3, 1.0)
# minFit_data2, meanFit_data2 = GA(population, RW_selection, 0.03, 1.0)
# minFit_data3, meanFit_data3 = GA(population, RW_selection, 0.003, 1.0)
# minFit_data4, meanFit_data4 = GA(population, RW_selection, 0.0003, 1.0)

# plt.plot(minFit_data1, label="MUTRATE 0.3")
# plt.plot(minFit_data2, label="MUTRATE 0.03")
# plt.plot(minFit_data3, label="MUTRATE 0.003")
# plt.plot(minFit_data4, label="MUTRATE 0.0003")
# [----------------- UNCOMMENT THIS TO TEST -----------------]
# N = 10
# MUTSTEP = 1.0
# Min Fitness: 8.848386018229803 - MUTRATE 0.3
# Min Fitness: 3.3014290163064572 - MUTRATE 0.03
# Min Fitness: 13.071407182572443 - MUTRATE 0.003
# Min Fitness: 48.747183334043555 - MUTRATE 0.0003




# DISPLAY PLOT
plt.legend(loc = "upper right")
plt.show()
```

MINIMIZATION PROGRAM (ACKLEY FUNCTION):

```
import random
import copy
import math
```

```
import matplotlib.pyplot as plt

# Each individual should be represented by a data structure,
# consisting of an array of binary genes and a fitness value
class individual:
    gene = []
    fitness = 0

    def __repr__(self):
        return "Gene string " +  "".join(str(x) for x in self.gene) + " - fitness: " + str(self.fitness)

# The initial population array of such individuals,
# and random gene length number of 10 and population of 50
P = 50
N = 10
GENERATIONS = 150 # initialise 150 generations
 # random mutation rate and mutation step
MUTRATE = 0.03
MUTSTEP = 0.3

# Calculate individual's fitness
# first minimisation function
def mini_function(ind):
    fitness = 0
    first_exp = 0
    second_exp = 0
    for i in range(0, N):
        first_exp += math.cos(2 * math.pi * ind.gene[i])
        second_exp += ind.gene[i] * ind.gene[i]
    first_exp = math.exp((1/N) * first_exp)
    second_exp = (-20) * math.exp((-0.2) * math.sqrt((1/N) * second_exp))
    fitness = second_exp - first_exp
    return fitness

# Calculate population's fitness
# list parameter
def total_fitness(population):
    totalfit = 0
    for ind in population:
        totalfit += ind.fitness
    return totalfit

# Initialise original population
def initialise_population():
    population = []
    # Initialise population with random candidate solutions
    # Generate random genes and append to a temporary gene list
    # Assign fitness and gene to individual and append one to population
```

```
    for x in range(0, P):
        tempgene = []
        for x in range(0, N):
            tempgene.append(random.uniform(-32.0, 32.0)) # a random gene between -5.12 and
    5.12(inclusive)
        # print(tempgene)
        newindi = individual() # initialise new instance
        newindi.gene = tempgene.copy() # copy the gene from tempgene and assign to gene of individual
        newindi.fitness = mini_function(newindi) # initialise instance's fitness
        population.append(newindi)

    return population

# Tourament Selection Process
def touranment_selection(population):
    offspring = []
    # Select two parents and recombine pairs of parents
    for i in range(0, P):
        parent1 = random.randint(0, P - 1)
        off1 = population[parent1]
        parent2 = random.randint(0, P - 1)
        off2 = population[parent2]
        if (off1.fitness > off2.fitness): # if one's fitness higher then add the smaller one to temp offsptring
            offspring.append(off2)
        else:
            offspring.append(off1)

    return offspring

# Roulette Wheel Selection Process
def RW_selection(population):
    # total fitness of initial pop
    total = 0
    for ind in population:
        total += abs(ind.fitness)

    offspring = []
    # Roulette Wheel Selection Process
    # Select two parents and recombine pairs of parents
    for i in range(0, P):
        selection_point = random.uniform(0.0, total)
        running_total = 0
        j = 0
        while running_total <= selection_point:
            running_total += abs(population[j].fitness)
            j += 1
            if(j == P):
                break
```

```
        # print(running_total)
        # print(j)
        offspring.append(copy.deepcopy(population[j-1]))

    return offspring

# Single-point Crossover process
def crossover(offspring):
    # Recombine pairs of parents from offspring
    crossover_offspring = []
    for i in range(0, P, 2):
        crosspoint = random.randint(1, N - 1) #pick up one random point in the gene length
        # 2 new temporary instances
        temp1 = individual()
        temp2 = individual()
        # 2 heads and 2 tails
        head1 = []
        tail1 = []
        head2 = []
        tail2 = []
        # print(offspring[i].gene, offspring[i+1].gene, crosspoint)
        # 0 to crosspoint adding gene to each head
        for j in range(0, crosspoint):
            head1.append(offspring[i].gene[j])
            head2.append(offspring[i + 1].gene[j])
        # crosspoint to N adding gene to each tail
        for j in range(crosspoint, N):
            tail2.append(offspring[i + 1].gene[j])
            tail1.append(offspring[i].gene[j])
        # print("head1 + tail2")
        # print(head1, tail2)
        # print("head2 + tail1")
        # print(head2, tail1)
        temp1.gene = head1 + tail2 # add first gene after crossover to temp1
        temp2.gene = head2 + tail1 # add second gene after crossover to temp2
        temp1.fitness = mini_function(temp1) # call counting_ones to add fitness to temporary indv
        temp2.fitness = mini_function(temp2)
        # append temp1, temp2 respectively to crosover_offspring_offspring
        crossover_offspring.append(temp1)
        crossover_offspring.append(temp2)
        # print(crosover_offspring_offspring[i].gene, crosover_offspring_offspring[i+1].gene)

    return crossover_offspring

#Bit-wise Mutation
def mutation(crossover_offspring, MUTRATE, MUTSTEP):
    # Mutate the result of new_offspring
```

```
    #Bit-wise Mutation
    mutate_offspring = []

    for i in range(0, P):
        new_indi = individual()
        new_indi.gene = []
        for j in range(0, N):
            gene = crossover_offspring[i].gene[j]
            ALTER = random.uniform(0.0, MUTSTEP)
            MUTPROB = random.uniform(0.0, 100.0)
            if (MUTPROB < (100*MUTRATE)):
                if(random.randint(0, 1) == 1): # if random num is 1, add ALTER
                    gene += ALTER
                else: # if random num is 0, minus ALTER
                    gene -= ALTER
                if(gene > 32.0): # if gene value is larger than 32.0, reset it to 32.0
                    gene = 32.0
                if(gene < -32.0): # if gene value is smaller than -32.0, reset it to -32.0
                    gene = -32.0
            new_indi.gene.append(gene)
        new_indi.fitness = mini_function(new_indi) # add fitness to instance by calling mini_function
        mutate_offspring.append(new_indi)

    return mutate_offspring

# Descending sorting
def sorting(population):
    #  descending sorting based on individual's fitness
    population.sort(key=lambda individual:individual.fitness, reverse=True)

    return population

# Minimisation Optimisation
def optimising(population, new_population):
    # more optimising
    # sorting instance with descending fitness
    population = sorting(population)

    # take the two instance with the worst fitness in the old population at index -1 and index -2
    worstFit_old_1 = population[-1]
    worstFit_old_2 = population[-2]

    # overwrite the old population with mutate_offspring
    population = copy.deepcopy(new_population)

    # sorting instance with descending fitness
    population = sorting(population)
```

```
    # after deepcopy new pop to old pop
    # take two instances with the best fitness in the new population at index 0 and index 1
    bestFit_new_1 = population[0]
    bestFit_new_2 = population[1]

    # compare the fitness btw the ones in the old pop and the ones in the new pop
    # replace the two best fitness/gene by the two worst fitness/gene at specific index in the new
    population
    if(worstFit_old_1.fitness < bestFit_new_1.fitness):
        population[0].gene = worstFit_old_1.gene
        population[0].fitness = worstFit_old_1.fitness
    if(worstFit_old_2.fitness < bestFit_new_2.fitness):
        population[1].gene = worstFit_old_2.gene
        population[1].fitness = worstFit_old_2.fitness


    return population

def GA(population, Selection, MUTRATE, MUTSTEP):
    # ===========GENETIC ALGORITHM===============
    # storing data to plot
    meanFit_values = []
    minFit_values = []


    for gen in range(0, GENERATIONS):
        # touranment selection process
        offspring = Selection(population)
        # crossover process
        crossover_offspring = crossover(offspring)
        # mutation process
        mutate_offspring = mutation(crossover_offspring, MUTRATE, MUTSTEP)
        # optimising
        population = optimising(population, mutate_offspring)

        # calculate Max and Mean Fitness
        # storing fitness in a list
        Fit = []
        for ind in population:
            Fit.append(mini_function(ind))
        # print(Fit)

        minFit = min(Fit) # take out the min fitness among fitnesses in Fit
        meanFit = sum(Fit)/ P # sum all the fitness and divide by P size

        # append minFit and meanFit respectively to MinFit_values and MeanFit_values
        minFit_values.append(minFit)
        meanFit_values.append(meanFit)
```

```
        # # display
        # print("GENERATION " + str(gen + 1))
    print("Min Fitness: " + str(minFit) + "\n")
    print("Mean Fitness: " + str(meanFit) + "\n")


    return minFit_values, meanFit_values

# plotting
plt.ylabel("Fitness")
plt.xlabel("Number of Generation")

# Storing
minFit_data1 = []
minFit_data2 = []
minFit_data3 = []
minFit_data4 = []

meanFit_data1 = []
meanFit_data2 = []
meanFit_data3 = []
meanFit_data4 = []


# EXPERIMENT

# ================================================================
# TOURANMENT vs ROULETTE WHEEL SELECTION COMPARISON
# ================================================================

# [----------------- UNCOMMENT THIS AND ALTER N TO TEST -----------------]
plt.title("Minimisation GA \n Touranment and Roulette Wheel Selection \n"
          + "N = " + str(N) + " MUTRATE = " + str(MUTRATE) + " MUTSTEP = " + str(MUTSTEP))

N = 10
# initialise original population
population = initialise_population()

minFit_data1, meanFit_data1 = GA(population, touranment_selection, 0.03, 0.3)
minFit_data2, meanFit_data2 = GA(population, RW_selection, 0.03, 0.3)

plt.plot(minFit_data1, label="Touranment")
plt.plot(minFit_data2, label="Roulette Wheel")
# [----------------- UNCOMMENT THIS AND ALTER N TO TEST -----------------]

# N = 10
# MUTRATE = 0.03
```

```
# MUTSTEP = 0.3
# Min Fitness: -10.708492338435587 - TS
# Min Fitness: -9.426192516643226 - RW



# N = 50
# MUTRATE = 0.03
# MUTSTEP = 0.3
# Min Fitness: -4.225967898881414 - TS
# Min Fitness: -4.0251242091551624 - RW




# ================================================================
# TOURANMENT SELECTION
# ================================================================


# Best Fitness and Mean Fitness of TS
# [----------------- UNCOMMENT THIS TO TEST -----------------]
# plt.title("Minimisation GA - Touranment Selection \n"
#           + "N = " + str(N) + " MUTRATE = " + str(MUTRATE) + " MUTSTEP = " + str(MUTSTEP))

# # initialise original population
# population = initialise_population()

# minFit_data1, meanFit_data1 = GA(population, touranment_selection, 0.03, 0.3)

# plt.plot(minFit_data1, label="Min Fitness")
# plt.plot(meanFit_data1, label="Mean Fitness")
# [----------------- UNCOMMENT THIS TO TEST -----------------]
# N = 10
# MUTRATE = 0.03
# MUTSTEP = 0.3
# Min Fitness: -10.852278293354441
# Mean Fitness: -10.804764892978262



# Vary MUTRATE
# [----------------- UNCOMMENT THIS TO TEST -----------------]
# plt.title("Minimisation GA - Touranment Selection \n"
#           + "Vary MUTRATE")

# # initialise original population
```

```
# population = initialise_population()

# minFit_data1, meanFit_data1 = GA(population, touranment_selection, 0.3, 0.3)
# minFit_data2, meanFit_data2 = GA(population, touranment_selection, 0.03, 0.3)
# minFit_data3, meanFit_data3 = GA(population, touranment_selection, 0.003, 0.3)
# minFit_data4, meanFit_data4 = GA(population, touranment_selection, 0.0003, 0.3)

# plt.plot(minFit_data1, label="MUTRATE 0.3")
# plt.plot(minFit_data2, label="MUTRATE 0.03")
# plt.plot(minFit_data3, label="MUTRATE 0.003")
# plt.plot(minFit_data4, label="MUTRATE 0.0003")
# [----------------- UNCOMMENT THIS TO TEST -----------------]
# N = 10
# MUTSTEP = 0.3
# Min Fitness: -8.610121679863145 - MUTRATE 0.3
# Min Fitness: -11.170587355807932 - MUTRATE 0.03
# Min Fitness: -9.524295765255925 - MUTRATE 0.003
# Min Fitness: -8.867615254560471 - MUTRATE 0.0003




# ================================================================
# ROULETTE WHEEL SELECTION
# ================================================================


# Best Fitness and Mean Fitness of RW
# [----------------- UNCOMMENT THIS TO TEST -----------------]
# plt.title("Minimisation GA - Roulette Wheel Selection \n"
#           + "N = " + str(N) + " MUTRATE = " + str(MUTRATE) + " MUTSTEP = " + str(MUTSTEP))

# # initialise original population
# population = initialise_population()

# minFit_data1, meanFit_data1 = GA(population, RW_selection, 0.03, 0.3)

# plt.plot(minFit_data1, label="Min Fitness")
# plt.plot(meanFit_data1, label="Mean Fitness")
# [----------------- UNCOMMENT THIS TO TEST -----------------]
# N = 10
# MUTRATE = 0.03
# MUTSTEP = 0.3
# Min Fitness: -7.929936862642863
# Mean Fitness: -7.686559831777039
```

```
# Vary MUTRATE
# [----------------- UNCOMMENT THIS TO TEST -----------------]
# plt.title("Minimisation GA - Roulette Wheel Selection \n"
#           + "Vary MUTRATE")
# # initialise original population
# population = initialise_population()

# minFit_data1, meanFit_data1 = GA(population, RW_selection, 0.3, 0.3)
# minFit_data2, meanFit_data2 = GA(population, RW_selection, 0.03, 0.3)
# minFit_data3, meanFit_data3 = GA(population, RW_selection, 0.003, 0.3)
# minFit_data4, meanFit_data4 = GA(population, RW_selection, 0.0003, 0.3)

# plt.plot(minFit_data1, label="MUTRATE 0.3")
# plt.plot(minFit_data2, label="MUTRATE 0.03")
# plt.plot(minFit_data3, label="MUTRATE 0.003")
# plt.plot(minFit_data4, label="MUTRATE 0.0003")
# [----------------- UNCOMMENT THIS TO TEST -----------------]
# N = 10
# MUTSTEP = 0.3
# Min Fitness: -14.254437249778938 - MUTRATE 0.3
# Min Fitness: -10.447878031888726 - MUTRATE 0.03
# Min Fitness: -9.727185493333165 - MUTRATE 0.003
# Min Fitness: -8.94342734801122 - MUTRATE 0.0003



# DISPLAY PLOT
plt.legend(loc = "upper right")
plt.show()
```