Biocomp Assignment - 18024097 - Francis Denton

Introduction

This report will discuss the effects changing a number of parameters has on the output of an evolutionary algorithm over a number of generations. It will compare the results reached by both tournament selection and roulette wheel selection, and how these results are affected by changes to the algorithms values. The tests will be conducted in a manner where the fitness of each individual is calculated by a minimization function.

Background Research

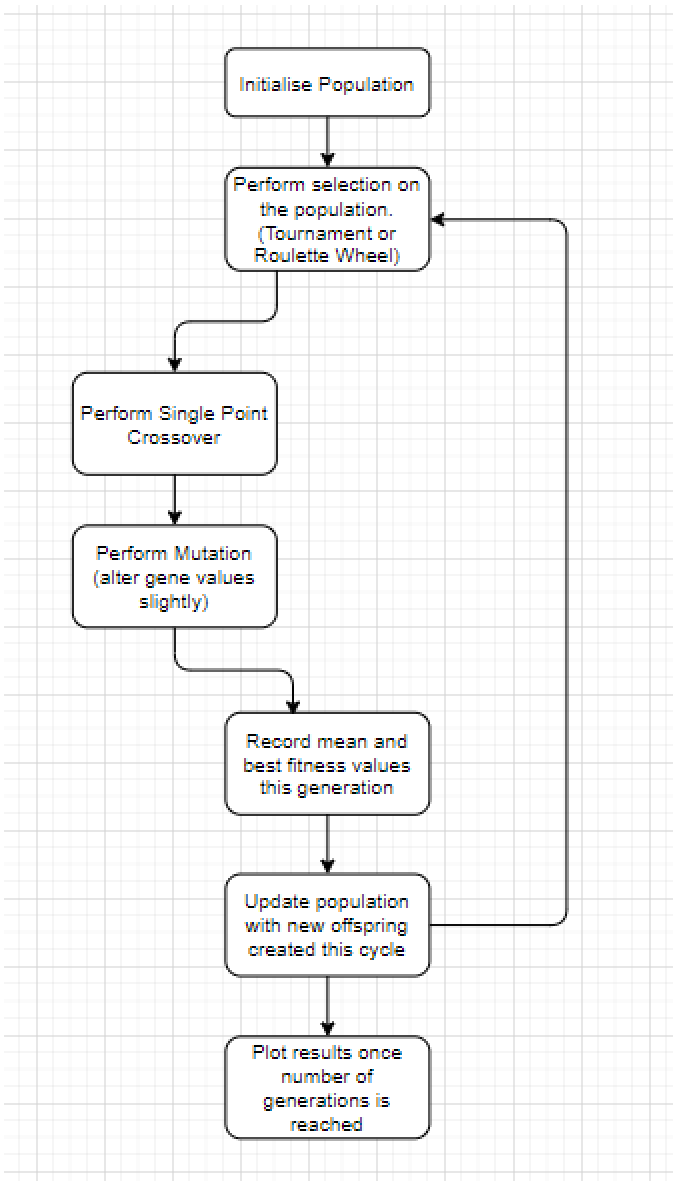
Optimization means finding the best solution among many feasible solutions that are available to us [1] . By tweaking parameters, the aim is to find a set of parameters that can get is to the optimal values in the shortest period of time or lowest number of cycles. In a real world scenario, an optimization problem might be attempting to minimize the cost of a process or attempting to perform a task with maximum efficiency. e.g. TSP (Travelling sales problem).

Genetic algorithms are randomized search algorithms that have been developed in an effort to imitate the mechanics of natural selection and natural genetics.[2] They can be used to solve a variety of different problems. An ideal genetic algorithms performance will tend to improve as it runs, constantly learning and taking the best genes from parents and passing them down to children each generation, the intention being to mix and match pairs of good genes in order to create even better genes. The process of crossover is responsible for exchanging genes between a number of parent individuals in the population. Whilst the process of mutation is responsible for adding random genes to the pool in order to combat the problem of stagnation in the gene pool over a long period of time.

Since artificial intelligence do not contain a moral sense, it is unable to distinguish between ethical and unethical conditions. When looking at an example of an insurance company using AI to decide on premiums for potential customers[3]. The artificial intelligence is given a choice from a multitude of strategies, some potentially could end up using customer data in an unethical manner leading to fines for the organizations should this ever be discovered. With more artificial intelligence being brought into mainstream commercial areas, the more these decisions are potentially made without humans being involved in the decision making process. Even if the chance of an AI choosing an unethical decision is at a low point, that chance is still existing, and cannot be ignored. Therefore it should become a priority to find ways in which these unethical scenarios can be avoided.

Experimentation

The genetic algorithm implement follows a cycle of steps over a large number of generations and tracks the outcomes of each generation to see how it has improved over time. The cycle follows the diagram below:


Initialise Population
Perform selection on the population. (Tournament or Roulette Wheel)
Perform Single Point Crossover
Perform Mutation (alter gene values slightly)
Record mean and best fitness values this generation
Update population with new offspring created this cycle
Plot results once number of generations is reached

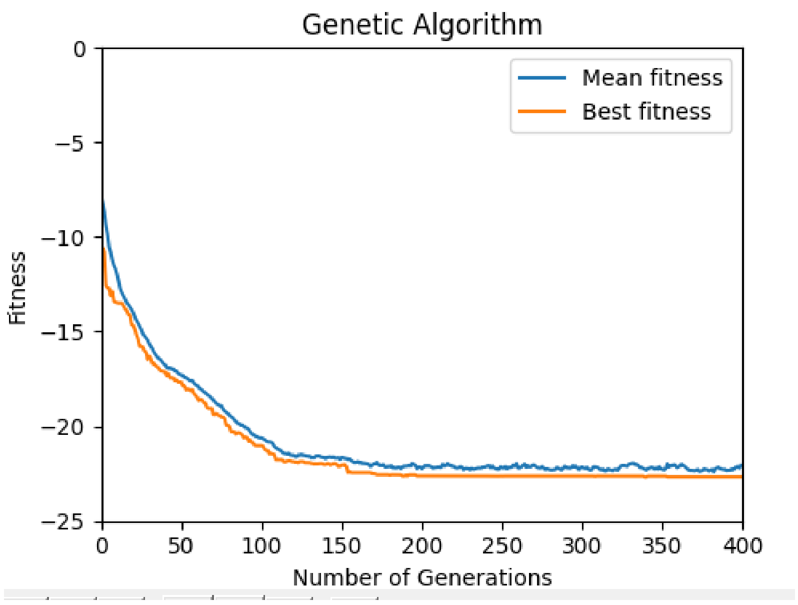
The Default Parameters

- Population Size - 50
- Gene Length - 10
- Gene value range - (-32, 32)
- Generations - 400
- Mutation Rate - 0.05
- Mutation Step (-2,2) - Mutation size between these values.
- Minimization Function: $f(x) = -20 \exp\left(-0.2\sqrt{\frac{1}{D}\sum_{i=1}^{D} x^2}\right) - \exp\left(\frac{1}{D}\sum_{i=1}^{D} \cos 2\pi x_i\right)$

Initial Run

Tournament Selection

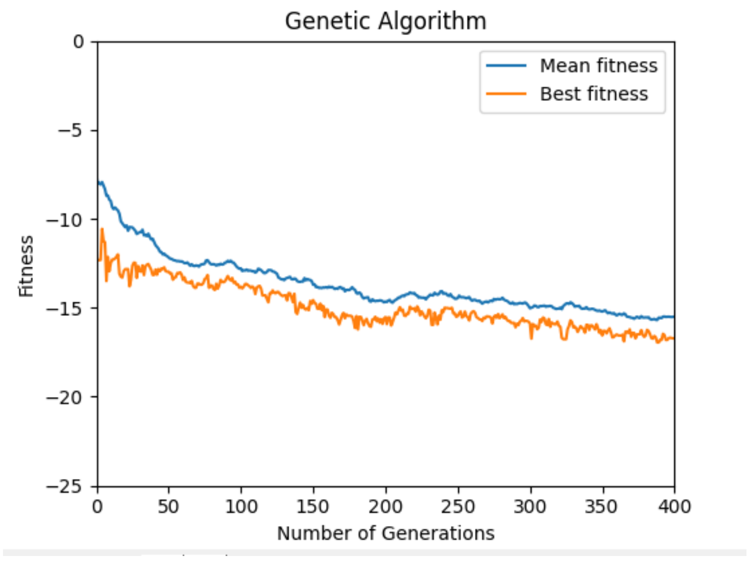
The code for tournament selection can be found in the appendix at the bottom of the report:



On the initial run of our algorithm over 400 generations, the fitness decreases sharply over the first 100 generations, this is due to the algorithm removing the majority of the bad individuals from our population of individuals early on. Since tournament selection picks the better individuals to produce offspring with, the worst individuals in the populations are quickly rooted out. After 100 generations, the rate of fitness decrease appeared to plateau due to only having a majority of good individuals left in the pool and getting close to the lowest fitness value possible for this minimization function.

Roulette Wheel Selection

The code for roulette wheel selection can be found in the appendix at the bottom of the report:

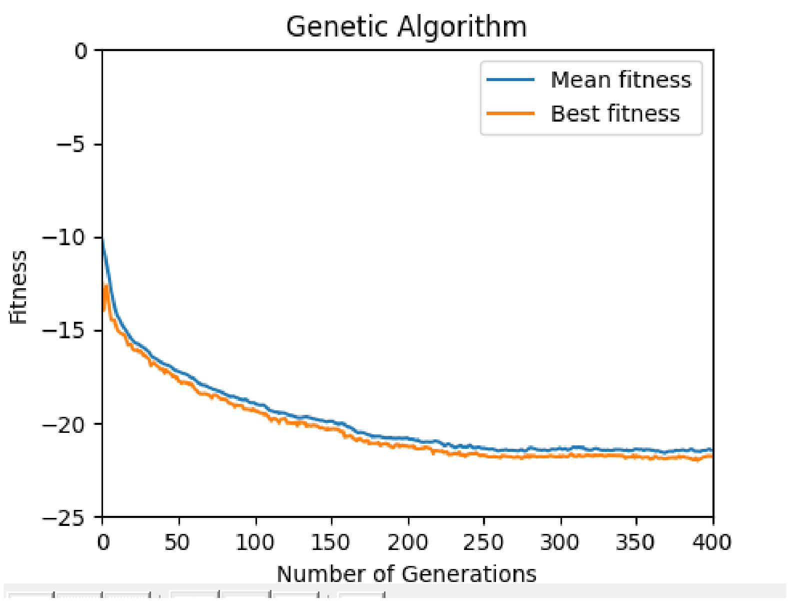


On the initial run of our algorithm using roulette wheel selection, it appears that the rate of fitness decreases is far more gradually then its tournament selection counterpart. This is due to the ability for worse individuals to become part of the offspring due to the probability mechanism of roulette wheel selection. Over 400 generations, it appears that the best fitness value achieved is also not as good as the outcome via tournament selection, this may be due to worse individuals being selected for the initial offspring through early generations.

Changing Gene Length

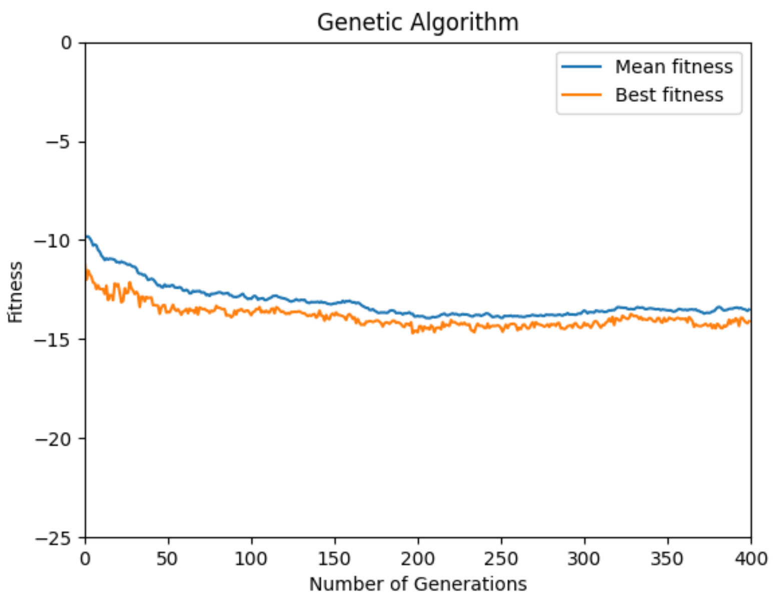
The first experiment conducted will be to increase the gene size and if there is an effect on the outcome. The Initial run was conducted with a gene length of 10, now we will increase this to 20 to see if there is a change in results.

Tournament Selection



When compared to the initial run, the initial decline of fitness comes at a more gradual decline whereas the initial test started out as a steep decline, this is likely due to mutation having a larger effect on a small number of genes than a larger number of genes. e.g. 1/10 gene's mutated = 10% of the individuals' gene pool has changed, vs, 1/20 gene's mutated = 5% of the individuals' gene pool has changed

Roulette Wheel Selection

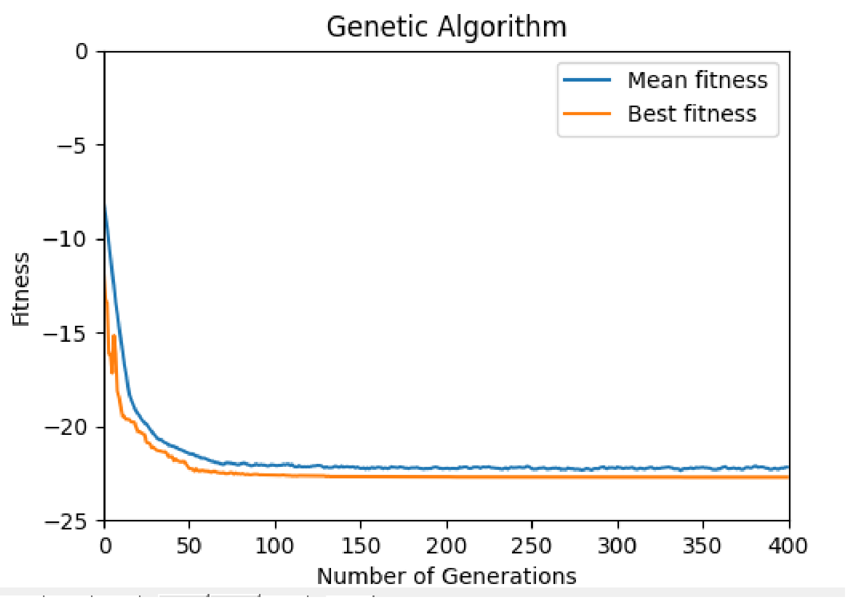


While the initial fitness throughout the early generations seems to be lower then when the gene size was 10, the rate of change appears to be less, with it decreasing at a faster rate until around 50 generations when it begins to gradually decline and eventually plateau and remain at a stable level in minimum change after 160 generations.

Changing Population Size

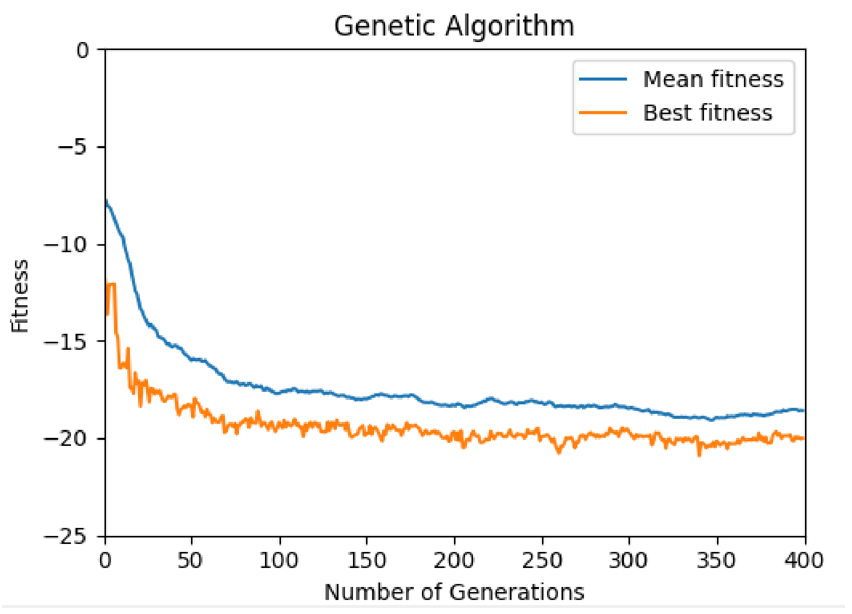
The second experiment conducted will be to increase the population size and if there is an effect on the outcome. The Initial run was conducted with a population size of 50, now we will increase this to 200 to see if there is a change in results.

Tournament Selection



When compared to the initial run, increasing the population size appears to have a positive effect on the performance of the algorithm. The graph clearly displays a fast descend to the desired optimal fitness levels after just 50 generations. Beyond the initial 50 generations, there is a small increase in fitness until generation 70, after that the results stay stable and minimal occur for the rest of the generations. This is the expected outcome as the more higher the population size, the higher the chance of good individuals appearing and knocking out worse individuals via tournament selection.

Roulette Wheel Selection



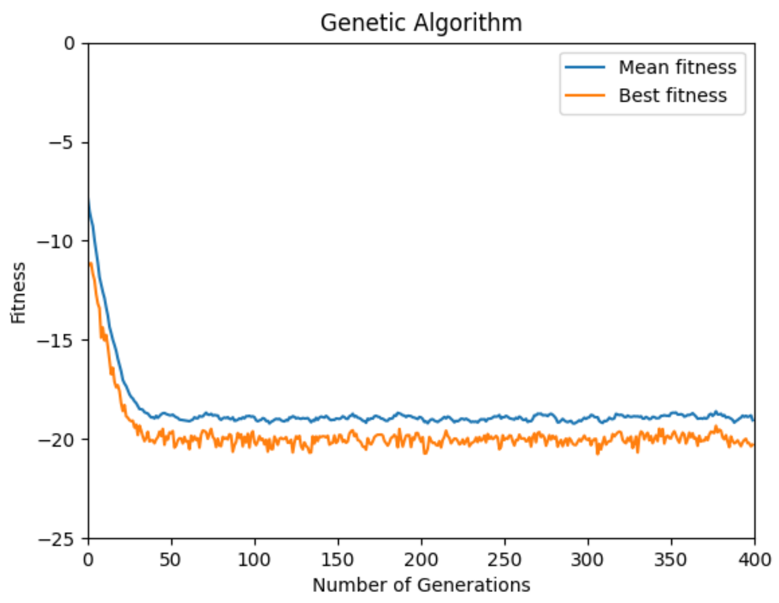
Upon increasing the population size we can see a drastic change in the performance in when compared to the other experiments conducted on roulette wheel selection method thus far. Compared to the initial run, this is a vast improvement, the fitness declines at a fast rate to begin with. This rate of decline slowing down after around 100 generations, where it remains around the same level for the rest of the cycle of generations.

Changing Mutation Rate

Increasing the mutation rate results in a more random pool of genes then was initially generated in the first generation, as the mutations do not account for any other factors besides in this case making sure that the value is kept within the upper and lower bound of possible gene values. Simply altering the genes of an individual by a random value.

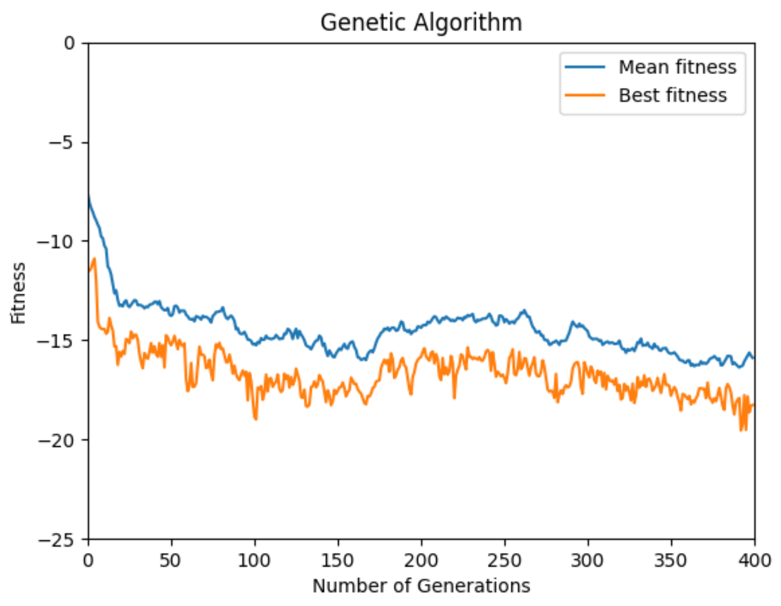
In this experiment the mutation rate will be increased from 0.05 to 0.5. With the size of the mutation staying in the bounds of (-2,2).

Tournament Selection



With this increased mutation rate, the graph above shows that the performance of the tournament selection method improves at a good level. The optimal fitness value is found after around 35-40 generations. This is due to the mutations that result in a positive effect on the offspring are kept in the offspring and are part of the next generation, whilst mutations that result in a negative effect are gotten rid of as they are knocked out of the tournament and do not proceed to the next generation.

Roulette Wheel Selection

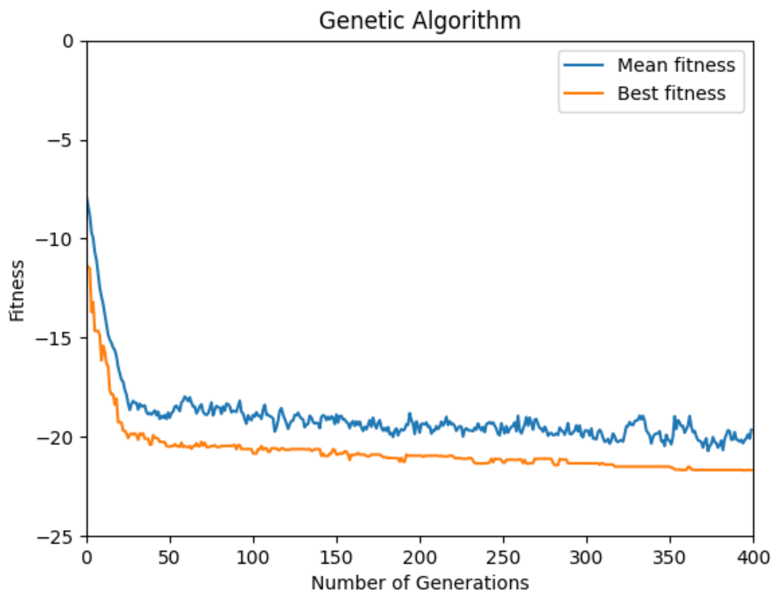


Whilst the improvement of the roulette wheel selection is clear to see when compared to the initial run, we can still see a large variance between mean fitness and best fitness values in each generation after the initial steep decline in the first 20 generations. A large disparity between the mean and best fitness is also clear too, this is a cause of the pool of genes consisting of a wide range of values after so much mutation.

Changing Mutation Size

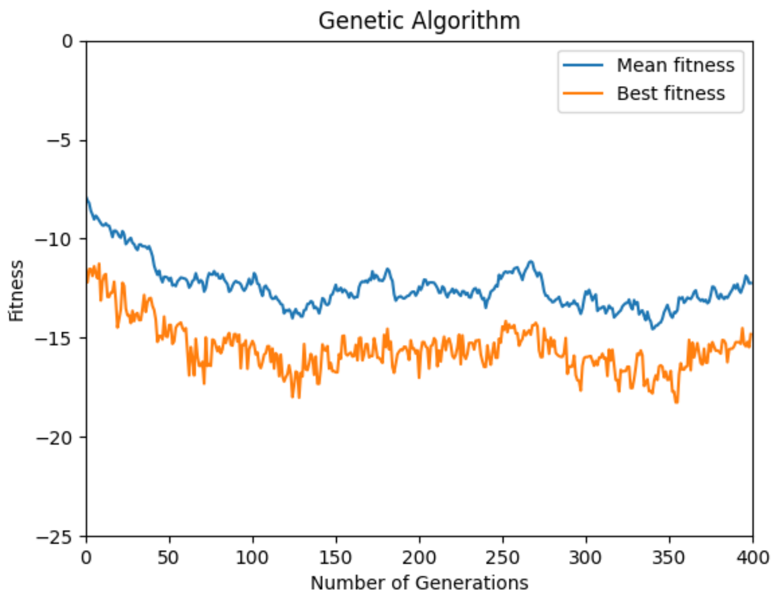
In this experiment we will be reverting the rate of mutation back to 0.05, but increasing the size of the mutation drastically from a range of (-2,2) to (-16, 16).

Tournament Selection



Similarly to the prior experiment, the increase in the mutation size improves the performance of the tournament selection method throughout the early generations, the rate of decline being much steeper. However when compared to its initial run, the mean fitness has a higher variance due to the mutations, and performs worse on average due to the more varied gene pool then our initial experiment.

Roulette Wheel Selection

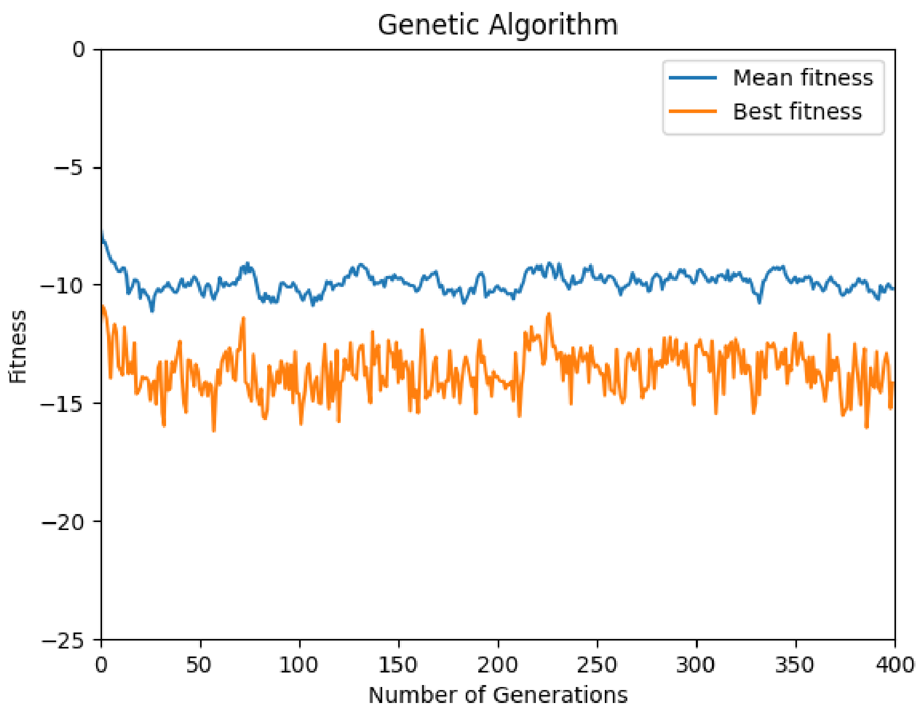


The increase in mutation size results in a large variation of results when applying the roulette wheel selection method, after the initial generations, we no long see a clear trend of improvement, instead the fitness values move up and down due to the mutation of the gene pool having a more significant impact with a larger rate.

Changing Mutation Rate and Mutation Size

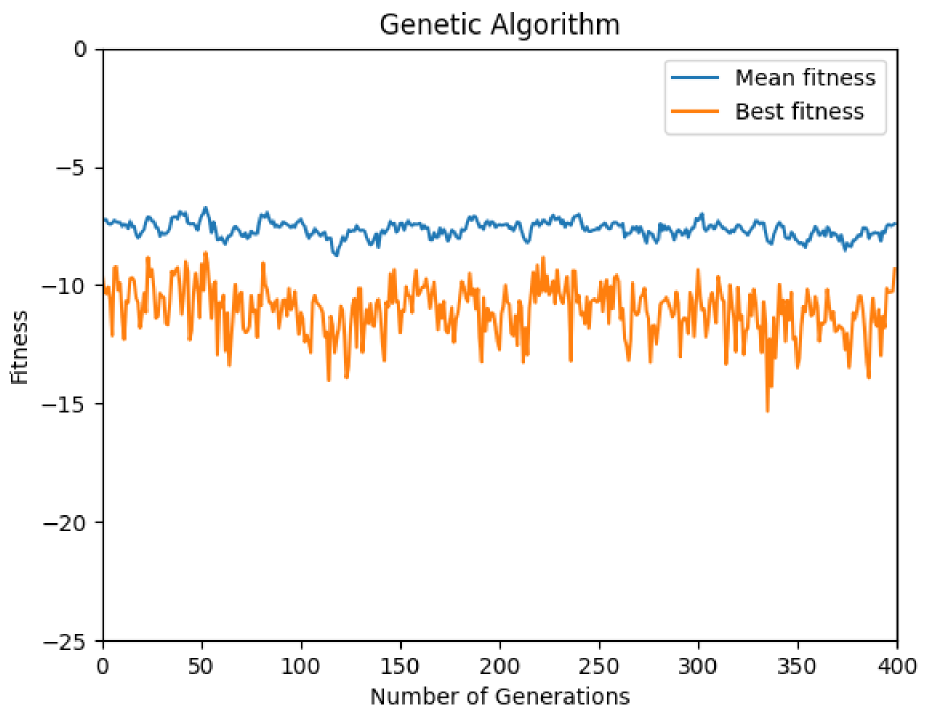
The next experiment that will be conducted will be changing both the mutation rate and mutation size at the same time. Combining both of the values of our past 2 experiments. A mutation rate of 0.5 and a mutation size in the range (-16,16). It is expected that since the mutation rate and mutation size are so large, there gene pool should undergo constant changes and it should result in no clear pattern of improvement.

Tournament Selection



The increase in mutation rate and mutation size has the expected output. There is no improvement or pattern to the fitness values, each generations best individual differs in fitness value from the prior, with the mean moving up and down slightly but remaining roughly around the same point. This is due to the fact that since so many genes are changed on each cycle, the starting population do not have a large impact on the resulting offspring at the end of the cycle.

Roulette Wheel Selection



Similarly to how it affects tournament selection, the increase in these two parameters results in a non-recognizable pattern, the mean staying around the same value throughout the 400 generations, whilst the best increases and decreases at a seemingly large rate each time.

Conclusion

In conclusion of the experiments, in regards to tournament selection, the increase in population size seemed to have the most positive effect on the performance of the selection method. The results of that experiment consisted of the best fitness level across all tests and all generations in the experiments conducted. In regards to roulette wheel selection, the same change in parameter had the greatest positive effect. In comparison to the initial run, the increase in population size caused the roulette wheel selection to achieve results in a similar manner as the tournament selection algorithm, with the curve being a similar pattern albeit to a lower level of results.

References

[1] Arora, R.K. (2015) *Optimization*. doi:10.1201/b18469 [Accessed 14 January 2021].

[2] Roetzel, W., Luo, X. and Chen, D. (2020) Optimal design of heat exchanger networks. *Design and Operation of Heat Exchangers and their Networks*. pp. 231–317. doi:10.1016/b978-0-12-817894-2.00006-6 [Accessed 9 January 2021].

[3] Beale, N., Battey, H., Davison, A.C. and MacKay, R.S. (2020) An unethical optimization principle. *Royal Society Open Science*. 7 (7), pp. 200462. doi:10.1098/rsos.200462 [Accessed 9 January 2021].

Code References

- Worksheet 1
- Worksheet 2
- Worksheet 3
- Assignment Sheet

Appendix

```
'''
Biocomputation Assignment
Francis Denton
```

```
Student Number - 18024097
'''
#Imports
import math
import random
import matplotlib.pyplot as plt
import numpy as np
import copy

# Parameters changed for experiments
P = 50
N = 10
MUTRATE = 0.05
genesize = 32
population = []
offspring = []

random.seed(1)
# Class of an Individual
class individual():
    gene = []
    fitness = 0

# Function to initialise the first population
def generate_population(P, N):
    for x in range(0, P):
        tempgene = []
        for i in range(0, N):
            # Creates a random 10 gene string between two values
            random1 = random.uniform(-genesize, genesize)
            tempgene.append(random1)
        #Calculates the fitness on the created gene
        fitness = minimization_function(tempgene)
        newind = individual()
        newind.gene = tempgene.copy()
        newind.fitness = fitness
        #Adds the set of created individual to create the first populatioin
        population.append(newind)

# Fitness Function for floats from worksheets
def fitness_function(pop):
    for i in range(0, P):
        pop[i].fitness = 0
        for x in range(0, N):
            pop[i].fitness = pop[i].fitness + pop[i].gene[x]

# Coursework Minimilisation function
def minimization_function(gene):
    # Minimisation function, optimal value expected to be around -20
    geneSquared = 0
    cosSum = 0
    for i in gene:
        geneSquared += i * i
        cosSum += math.cos(2.0 * math.pi * i)
    fitness = -20.0 * math.exp(-0.2 * math.sqrt(geneSquared) / N) -
math.exp(cosSum / N)
    return fitness
```

```python
def tournament_selection(population):
    offspring = []
    for x in range(0, P):
        #Select 2 random parents in the population and compare the fitnesses.
        parent1 = random.randint(0, P-1)
        off1 = population[parent1]
        parent2 = random.randint(0, P-1)
        off2 = population[parent2]
        #Add which individual had the better fitness to the new offspring
        if off1.fitness < off2.fitness:
            offspring.append(off1)
        else:
            offspring.append(off2)

    return offspring

# Function to Calculate the total fitness of the population
def total_fitness(pop):
    total_fitness = 0
    for x in range(0, len(pop)):
        total_fitness += pop[x].fitness
    return total_fitness

def roulette_wheelselection(population):  # Roulette wheel selection
    offspring = []
    # Call function to grab total fitness value
    fitnessSum = total_fitness(population)
    for i in range(0, len(population)):
        #Generate a random point between the total fitness and 0 since we are
using negative numbers
        selection_point = np.random.uniform(fitnessSum, 0)
        running_total = 0
        j = 0
        #At the fitness values of each individiual together until it is greater
then the selection point
        #Add that value to the offspring population
        while running_total >= selection_point:
            running_total += population[j].fitness
            j += 1
        offspring.append(population[j-1])

    return offspringpopulation

#Function to calculate mean fitness
def mean_fitness(population):
    total_fitness = 0
    mean_fitness = 0
    #Sum all the fitness values together and divide it by the size of the
population
    for x in range(0, len(population)):
        total_fitness += population[x].fitness
    mean_fitness = total_fitness / P
    return mean_fitness

#Function to calculate best fitness
def best_fitness(population):
    gene_fit = 0.0
    best_fit = 0.0
    for i in range(0,len(population)):
        gene_fit = population[i].fitness
        if gene_fit < best_fit:
            best_fit = gene_fit
    return best_fit

def single_point_crossover(offspring):
    children_after_crossover = []
    for i in range(0, len(offspring), 2):
        #Create two children which will be born for a set of parents
        child1,child2 = individual(), individual()

        # Pick a point in the string of genes.
        Xpoint = random.randint(0, N)
        child1head, child1tail = [], []
        child2head, child2tail = [], []

        #Find the midpoints of the genes, and adds the heads and tails to the
respective child lists.
        for h in range(0, Xpoint):
            child1head.append(offspring[i].gene[h])
            child2head.append(offspring[i + 1].gene[h])
        for j in range(Xpoint, len(offspring[0].gene)):
            child2head.append(offspring[i].gene[j])
            child2tail.append(offspring[i + 1].gene[j])

        #Swap genes between the two parents and call the fitness function to
calculcate fitness value of new child.
        child1.gene = child1head + child2tail
        child2.gene = child2head + child1tail
        child1.fitness = minimization_function(child1.gene)
        child2.fitness = minimization_function(child2.gene)  # 4th dec
        children_after_crossover.append(child1)
        children_after_crossover.append(child2)

    #Return list of updated children after crossover
    return children_after_crossover

#Mutation Function
def mutation(offspring, MUTRATE):
    offspring_after_mutation = []
    for i in range(0, P):
        newind = individual()
        newind.gene = []
        for j in range(0, N):
            gene = offspring[i].gene[j]
            mutprob = random.randint(0, 100)
            if mutprob < (100 * MUTRATE):
                #Generate a random value between these two points which will be
how large the mutation is.
                alter = random.uniform(-2,2)
                #Make sure the gene values stay within the upper and lower bounds
set at the beginning.
                if gene + alter > 32:
                    gene = 32
                elif gene + alter < -32:
                    gene = -32
                else:
                    gene = gene + alter
            newind.gene.append(gene)
            newind.fitness = minimization_function(newind.gene)
        #Add the updated offsrping after mutation to the list.
        offspring_after_mutation.append(newind)
    #Return the list of offspring after mutation.
    return offspring_after_mutation

#Function to update the initial population with the newly created offspring each
cycle
def update_pop(population, offspring):
    population = copy.deepcopy(offspring)

    return population

#Call generate population function
generate_population(P, N)
#Create lists to store best and mean fitness for each generation and the number
of generations
mean = []
best = []
generation = []
for i in range(0,400):
    '''
    Initial population is created outside of the loop of generations
    Step 1. - Call Selection Function (tournament or roulette wheel)
    Step 2. - Call Crossover Function
    Step 3. - Call Mutation Function
    Step 4. - Calculate Mean and Best fitness of generation and append it to the
list
    Step 5. - Call Update population to update intial population with created
offspring for next generation
    '''
    offspring = tournament_selection(population)
    offspring = single_point_crossover(offspring)
    offspring = mutation(offspring, MUTRATE)
    mean1 = mean_fitness(offspring)
    best1 = best_fitness(offspring)
    mean.append(mean1)
    best.append(best1)
    population = update_pop(population, offspring)
    generation.append(i)

#Plot Graph of Mean and Best Fitnesses
plt.xlabel("Number of Generations")
plt.ylabel("Fitness")
plt.title("Genetic Algorithm")
plt.ylim([-25, 0])
plt.xlim([0, len(generation)])
plt.plot(mean, label = "Mean fitness")
plt.plot(best, label = "Best fitness")
plt.legend()
plt.show()
```