Student name: Luqmaan Abdullahi

AI2 Assignment

Student no. 22013740

1 INTRODUCTION

Evolutionary algorithms leverage key parameters for optimization. The mutation rate drives diversity, while mutation size balances fine tuning and exploration. Tournament size influences selection pressure, shaping the competition among individuals. The number of crossover points governs trait recombination, impacting solution quality and diversity. Generations dictate the algorithm's temporal scope, affecting convergence speed. Population size, the number of coexisting individuals, strikes a balance between exploration and computational efficiency. These parameters collectively direct the algorithm's behaviour, crucial for designing it to specific optimization challenges in complex search spaces.

2 EXPERIMENTATION

My basic EA is composed of 6 functions.

I start by instantiating a number of objects in the individual class controlled by the variable P. This gives each individual a random set of genes, between the max and min value, and adds that into the population. All the individuals in the population are tested after this.

The algorithm selects parents from the population by picking 2 random individuals and choosing the one with the better fitness to be a parent in the offspring list.

I then crossover the genes of 2 individuals and make copies of the newly made individuals into the offspring list.

I have a mutation function which has a chance to mutate a gene by rate called MUTRATE and by an amount called MUTSTEP.

The algorithm then tests all the offspring individuals and then overwrites and copies the list of offspring into the population list.

Finally I do this a number of times (my program does it 10 times) and use the final results to get an average.

Function 1

Mutation rate and step

My initial results are in this table that shows the average and the best fitness of each of the mutation rate and mutation steps. I used a population of 50 and 50 generations. I first tried to do a grid search for my values of mutation rate and mutation step. At the beginning, I chose the values from 0 up to 1.5 for mutation rate and from 1 to 5 for the mutation step.

		mutation steps		
		1	3	5
mutation rate	0.5	average: 3,341 best: 1,456	average: 84,627 best: 32,341	average: 578,460 best: 201,196
	1	average: 7,521 best: 2,483	average: 413,279 best: 162,924	average: 1,360,373 best: 593,837
	1.5	average: 10,915 best: 5,490	average: 482,764 best: 297,623	average: 1,661,035 best: 912,496

fig 1

		mutation steps		
		0.5	1	1.5
mutation rate	0.25	average: 6,807 best: 4,525	average: 3000 best: 1,184	average: 2113 best: 876
	0.5	average: 3,867 best: 2,582	average: 3,152 best: 782	average: 10,795 best: 3,080
	0.75	average: 3,278 best: 1,623	average: 4,116 best: 1,165	average: best: 10,904

fig 2

		mutation steps		
		1	1.5	2
mutation rate	0.1	average: 8,566 best: 6,098	average: 2,136 best: 1,096	average: 2,086 best: 846
	0.25	average: 2,876 best: 881	average: 5,253 best: 1,620	average: 7,435 best: 3,182
	0.4	average: 2,478 best: 421	average: 8,092 best: 2,843	average: 27,806 best: 5,211

fig 3

Student name: Luqmaan Abdullahi

		mutation steps		
		1.75	2	
mutation rate	0.05	average: 10,747 best: 5,643	average: 3,108 best: 1,393	average: 5,439 best: 3,186
	0.1	average: 2,001 best: 696	average: 3,214 best: 1,768	average: 6,505 best: 1,980
	0.15	average: 1,930 best: 955	average: 6,457 best: 2,188	average: 5,230 best: 1,916

fig 4

As you can see in figure 1, a lot of the numbers I am finding for fitness are really large. This makes sense as the larger the mutation rate, the more mutations that are going to happen throughout all the generations. This then results in the algorithm exploring the search space too aggressively. This can lead to a rapid and chaotic movement across the search space, without focusing on promising regions. This may hinder convergence to optimal solutions.

After 3 search grids, I found my optimal values for mutation rate and mutation step (figure 4). My optimal values for the first function are: for mutation rate is 0.15 and for mutation step it is 1.75.

Elitism

Without Elitism	With Elitism
Average: 3836 Best: 1315	Average: 2452 Best: 1381

fig 5

```
4840.284950
3715.765429
1858.622397
4110.844767
15124.23381
3715.765429
2845.488905
2310.870792
```

fig 6. 15,124 is really high compared to the other fitnesses in the population

Elitism is a technique commonly used in evolutionary algorithms (figure 5). It allows the algorithm to preserve the best individuals from one generation to the next. This is useful because it ensures that the best-performing solutions discovered so far are kept in the population, preventing them from being made worse with the other functions that are used such as crossover and mutation. It also allows your algorithm to never get a worse “best individual”, which allows the algorithm to explore the search space without losing its strong foundation.

There is also an interesting thing being shown in figure 6 that shows another reason why elitism is useful. It allows you to get rid of your worst value which in many cases could be affecting your average quite a bit.

Tournament size

Tournament size	Fitness
2	Average: 2688 Best: 1533
3	Average: 1327 Best: 493
4	Average: 249 Best: 108
5	Average: 162 Best: 78
6	Average: 91 Best: 37
7	Average: 116 Best: 73
8	Average: 94 Best: 35

fig 7

Tournament selection is a type of selection which chooses individuals based on their utility compared to others in the population. It works by randomly selecting a group of individuals from the population and then choosing the best individual from that group as a parent. The tournament is a competition among the selected individuals, and the winner is the one with the highest fitness.

Larger tournament sizes provide a better balance between exploring the search space and finding an optimal local solution (figure 7). Smaller tournaments may result in too much randomness, because less fit individuals have a higher chance of being selected. On the other hand, larger tournaments provide a more competitive environment, favouring the selection of stronger individuals.

In my understanding of how tournament size is affecting the first function, it seems as though the algorithm is way more suited to exploring the searching the solution space than it should be when tournament size is less than 6 (figure 7). The probability that the best individuals will be picked is quite low as the chances they get picked is 1/50 every time. Whereas, when tournament size is over 6, the probability that the best solution will be picked to make offspring is 6/50 or 3/25.

However the tournament size being increased more than 6 does not yield as much benefit because the algorithm gets stuck with strong individuals that have a high chance of getting picked but do not help search the space for a better solution.

Number of Crossover Points

Number of Crossover Points	Fitness
0	Average: 235 Best: 136
1	Average: 144 Best: 92
2	Average: 98 Best: 58
3	Average: 105 Best: 77
4	Average: 116 Best: 69

fig 8

There is another vital function which is used in my evolutionary algorithm called Crossover. It is also known as recombination and is a technique which allows genes to be swapped over at a certain point in the gene sequence. It takes place between two or more parent individuals to create new offspring. The purpose of crossover is to explore the search space by mixing and exchanging genetic material, potentially generating individuals with improved characteristics.

Recombination of genes allows you to explore the solution space in an effective and quick way (figure 8). This is done by using the information in the current population to explore regions of the search space that are near promising areas of current individuals. This can be used to traverse the space in larger parts than just mutation alone (figure 4).

However, there are diminishing returns when it comes to increasing the number of crossover points (figure 8). One reason might be because crossing over multiple times leads to a random subset of genetic material to end up in the gene of the offspring when it was only useful as part of that group. This means that while allowing for more diversity in the population, it also prevents useful groups of genes from being crossed over together.

Number of Generations

Generations	Fitness	Time
50	Average: 115 Best: 75	4.5s
100	Average: 37 Best: 9	9s
150	Average: 43 Best: 13	13.6s
200	Average: 53 Best: 32	18.3s

fig 9

The number of generations is a parameter that can be increased to get better utility values. The reason for this is that the evolutionary algorithm is given more opportunities to explore and find the best solutions in the search space.

However, this also has an impact on the efficiency of the algorithm and also the time taken (figure 9), because for every new generation, there are 50 new calls to the test function and this is what takes up the longest amount of time while running. This increases the time taken by quite a bit whereas the fitness stagnates because all of the individuals have converged in a local minima and don't have enough variety and diversity to change significantly.

Population Size

Population Size	Fitness	Time
25	Average: 76 Best: 53	4.3s
50	Average: 56 Best: 21	9s
100	Average: 51 Best: 22	18.1s
150	Average: 48 Best: 23	27.4s

fig 11

Increasing the size of the population in an evolutionary algorithm can have both positive and negative impacts on the optimization process. It affects both the fitness of the best individual and the time the program takes to run.

A greater number of individuals can explore a larger space faster and a broader search of the space and it also increases the chance the algorithm finds a good solution as soon as possible (figure 11).

Function 2

I repeated all of the same parameter changes to function 2.

Mutation rate and step

		mutation steps		
		1	3	5
mutation rate	0.5	average: 198,468 best: 266	average: 3,513,184 best: 391	average: 14,478,279 best: 363
	1	average: 186,325 best: 165	average: 3,203,929 best: 276	average: 39,535,305 best: 672
	1.5	average: 684,952 best: 249	average: 3,319,041 best: 249	average: 18,350,590 best: 336

fig 12

		mutation steps		
		0.05	0.1	0.15
mutation rate	0.58	average: 291 best: 278	average: 172 best: 138	average: 283 best: 191
	0.6	average: 160 best: 156	average: 266 best: 207	average: 292 best: 146
	0.62	average: 208 best: 204	average: 120 best: 109	average: 396 best: 175

fig 13

I have just shown the first and last grid tables for the grid search from here on as to limit the number of tables used.

I did a grid search to find the optimal mutation rate and mutation step just like I did for the first function (figure 12, 13). The results were quite similar to the first function however I did find another reason as to why a large mutation rate and mutation size works less well than an optimal mutation rate and mutation size.

Extremely large mutation steps and rates can make the algorithm computationally inefficient. It may require a large number of evaluations and generations, more than the optimal solution, to converge to good enough solution, making the algorithm time consuming and resource intensive.

Elitism

Without Elitism	With Elitism
Average: 227 Best: 182	Average: 246 Best: 219

fig 14

I used the elitism technique in function 2 and received similar results as function 1 (figure 14). Just like in function 1, This is due to the consistent high-level of robustness and stability that elitism offers by keeping the best individual from the previous generation and it helps to strike a better balance between searching the problem space and refining a good solution.

Tournament size

Tournament size	Fitness
2	Average: 262 Best: 223

3	Average: 303 Best: 288
4	Average: 160 Best: 154
5	Average: 127 Best: 121
6	Average: 166 Best: 161
7	Average: 197 Best: 192

fig 15

Tournament size is also a very important and useful parameter that I changed in the second function (figure 15). As the tournament size increases, the competition becomes more likely to choose refined solutions and is less likely to choose options that would allow it to search the problem space more efficiently. By selecting individuals with higher fitness values, this change allows the algorithm to focus on refining promising solutions.

Nevertheless, beyond a certain tournament size, the benefits of increased competition may decrease. This is because it leads to a selection which picks the same individuals every time and therefore reduces diversity. A well-chosen tournament size strikes a balance between these 2 extremes.

Number of Crossover Points

Number of Crossover Point	Fitness
1	Average: 192 Best: 186
2	Average: 170 Best: 163
3	Average: 142 Best: 136
4	Average: 217 Best: 208

fig 16

Changing the number of crossover points is very advantageous in function 2, just like in function 1 (figure 16). A moderate number of crossover points is advantageous as it facilitates the combining of beneficial “traits” and enhances the algorithm's ability to explore hard to reach areas of the search space.

However, as the number of crossover points increases beyond a certain threshold, the algorithm may impede the making of essential building blocks within the genes of individuals and the ability of the algorithm to converge to an optimal solution. Beyond this point, the benefits of increased exploration may be outweighed by the negative impact on the preservation of key groups of genes.

Number of Generations

Generations	Fitness	Time
50	Average: 190 Best: 181	2.5s
100	Average: 149 Best: 139	5.1s
150	Average: 99 Best: 93	7.6s
200	Average: 45 Best: 38	10.1s

fig 17

Unlike the first function, the second function has a clear benefit from increasing the generation size (figure 17). As the generation size increases, the average and best fitnessess decrease, due to the higher chance of finding a better solution with more time and more searches. I have decided that more than 10 seconds does not fit into my description of efficiency and so I feel as though 200 generations is my optimal number of generations.

Population Size

Population Size	Fitness	Time
25	Average: 41 Best: 36	4.3s

Student name: Luqmaan Abdullahi

50	Average: 60 Best: 57	10s
100	Average: 43 Best: 40	20.2s
150	Average: 19 Best: 16	30.7s
200	Average: 15 Best: 13	41.1s

fig 18

It is pretty clear as to the reason why population size increasing, aids the algorithm in exploring the search space for optimal solutions. While the benefits of increased diversity and exploration are notable, it's essential to strike a balance, as excessively large populations (figure) may lead to higher computational costs without the same proportion of improvement in solution quality (figure 18).

Function 3

The third function I have decided to use is the Trid function. The Trid function is known for its bowl shape and rugged artificial landscape, featuring multiple local minima and is often employed as a benchmark problem to evaluate the performance of optimization algorithms.

I have used the same parameters with a third function that I tried to solve. Only worthy changes and interesting points are noted down below.

Mutation rate and step

		mutation steps		
		1	3	5
mutation rate	0.5	average: 99,610 best: 98,307	average: 150,755 best: 148,628	average: 51,292 best: 44,711
	1	average: 1,055,548 best: 1,045,506	average: 136,902 best: 131,120	average: 72,636 best: 67,981
	1.5	average: 81,585 best: 80,246	average: 64,770 best: 61,400	average: 77,551 best: 73, 077

		mutation steps		
		13	15	17
mutation rate	0.65	average: 29,580 best: 25,891	average: 30,554 best: 26,295	average: 34,642 best: 28,725
	0.7	average: 16,636 best: 12,075	average: 31,415 best: 25,359	average: 24,913 best: 20,411
	0.75	average: 39,804 best: 33,791	average: 36,621 best: 29,992	average: 13,152 best: 9,137

Elitism

Without Elitism	With Elitism
Average: 26,758 Best: 22,005	Average: 17,796 Best: 11,243

Tournament size

Tournament size	Fitness
2	Average: 26,074 Best: 19,105
3	Average: 16,003 Best: 13,765
4	Average: 10,619 Best: 8,568
5	Average: 7,860 Best: 6,735
6	Average: 3,196 Best: 1,448
7	Average: 6,699 Best: 5,557

Number of Crossover Points

Number of Crossover Point	Fitness
1	Average: 3,723 Best: 2,796
2	Average: 5,058 Best: 3,118
3	Average: 9,694 Best: 7,760
4	Average: 10,468 Best: 9,037

Number of Generations

Generations	Fitness	Time

50	Average: 2,972 Best: 1,656	14.7s
100	Average: 1113 Best: -149	14.6s
150	Average: 959 Best: 15	22.1s
200	Average: 1498 Best: -346	29.4s

Population Size

Population Size	Fitness	Time
25	Average: 41 Best: 36	4.3s
50	Average: 1755 Best: 879	14.8s
100	Average: 2605 Best: 1,496	29.2s
150	Average: 1244 Best: -550	44.1s

3 COMPARISON

Show comparative performance of other well-known approaches to the optimisation problems provided. You don't need to implement other algorithms, use of open source, etc. is fine. For the very keen, other benchmark functions can be explored as well.

The 2 comparisons I have chosen for this report are the simple hill climber and the more nuanced simulated annealing. Here are their results compared to the evolutionary algorithm for all 3 functions.

Function	EA	HillClimb	SAnn
1	Average: 56 Best: 21	85	48
2	Average: 60 Best: 57	72	50
3	Average: 1755 Best: 879	12,333	2,013

fig 19

In this comparison, the Hill Climber algorithm demonstrated a tendency to get stuck in local optima due to its reliance on incremental improvements, limiting its exploration of the solution space (figure 19).

Simulated Annealing, with its probabilistic acceptance of worse solutions, allows for some escape from local optima and is comparable with the evolutionary algorithm. I was not expecting this because I thought that the algorithm might not be able to explore as far as the evolutionary algorithm can, while still keeping those good solutions (figure 19). However, because the temperature was being dynamically changed, this allowed for the SAnn algorithm to achieve a good balance between exploring and finding and refining those optimal solutions.

The Evolutionary Algorithm, benefitted from an optimal tournament size and crossover points, employs a population based approach that facilitates the individuals interacting with each other through the making of offspring and the crossing over of the genes. This diversity enables the algorithm to escape local optima more effectively and converge toward better solutions (figure 19). The tournament selection process and recombination through crossover contribute to maintaining a balance between preserving good solutions and exploring new possibilities, enhancing the algorithm's overall performance in finding optimal solutions for the given optimization problem.

The Trid function (function 3 on figure 19), shows some interesting information. This meant that while the hill climber got stuck in the local optima of the Trid functions rugged landscape, the evolutionary algorithm and the simulated annealing were able to escape due to high enough temperature and the EAs population based approach and the ability to maintain diversity in the search space.

4 CONCLUSIONS

In conclusion, the subtle manipulation of evolutionary algorithm parameters, such as the mutation rate, mutation size, tournament size, number of crossover points, number of generations, and population size, is pivotal for optimization.

I found a lot of improvement when I implemented a larger tournament size and an increased number of crossover points, whereas population size did not garner as large of an improvement as I was hoping for.

Future changes could involve experimenting with alternative parameters for each of the parameters, techniques from other optimisation functions and also graphs that show the change in fitness over time.

Source code as an appendix

```
# Imports
import numpy as np
import matplotlib as mat
import matplotlib.pyplot as plt
from mpl_toolkits import mplot3d
import random as ran
import copy
import math

# Constants
N = 20
P = 50
MUTRATE = 0.15
MUTSTEP = 1.75
MAXVALUE = 10
MINVALUE = -10
GENEREATIONS = 50
SUPERGENEREATIONS = 10
TOURNAMENTSIZE = 2

# Lists
population = []
offspring = []
sum = []
average = []
max = []
avgMax = []
avgMin = []
avgAvg = []
endIndividuals = []

# The individual class allows you to make an individual object which will be the individuals in the evolutionary algorithms
class individual:
    def __init__(self):
        self.gene = []
        self.fitness = 0

    def __str__(self) -> str:
            return (f"Fitness: {self.fitness}, Gene: {self.gene}")

# I instantiate the best individual after the class is made
BESTINDIVIDUAL = individual()

# This is the test function for function 1. It updates their fitness rather than returning it
def test(ind: individual):
    fitness = 0
    fitness = pow((ind.gene[0] - 1), 2)

    for i in range(1, N):
        fitness += i * ( pow((2 * pow(ind.gene[i], 2) - ind.gene[i-1]), 2))

    ind.fitness = fitness

# This function tests all the individuals in a list and calculates the average
def testAll(population: list, isTested: bool = True):
    if (isTested):
        total = 0

    for ind in population:
        test(ind)

        if (isTested):
            total = total + ind.fitness

    if (isTested):
        avg = total/P
```

```
        average.append(avg)

    # This function creates a new population of the indiviual objects
def newPopulation():
    for x in range(0, P):
        tempGene = []

        for y in range(0, N):
tempGene.append(ran.uniform(MINVALUE, MAXVALUE))

        newind = individual()
        newind.gene = tempGene.copy()

        population.append(newind)

# This function saves the best individual in the BESTINDIVIDUAL constant
def getBest(pop):
    global BESTINDIVIDUAL
    fit = []

    for ind in pop:
        fit.append(ind.fitness)

    fit.sort()

    BESTINDIVIDUAL = copy.deepcopy(pop[0])

# This function crosses over the genes of 2 or more individuals
def crossover():
    toff1 = individual()
    toff2 = individual()
    temp = individual()

    for i in range(0, P, 2):
        toff1 = copy.deepcopy(offspring[i])
        toff2 = copy.deepcopy(offspring[i+1])
        temp  = copy.deepcopy(offspring[i])

        crosspoint = ran.randint(1, N)
        crosspoint2 = ran.randint(1, N)
        crosspoint3 = ran.randint(1, N)
        crosspoint4 = ran.randint(1, N)

        while (crosspoint2 == crosspoint):
            crosspoint2 = ran.randint(1, N)

            while (crosspoint3 == crosspoint2 and crosspoint3 == crosspoint):
            crosspoint3 = ran.randint(1, N)

            while (crosspoint4 == crosspoint3 and crosspoint4 == crosspoint2 and crosspoint4 == crosspoint):
            crosspoint4 = ran.randint(1, N)

        for j in range (crosspoint, N):
            toff1.gene[j] = toff2.gene[j]
            toff2.gene[j] = temp.gene[j]

        for j in range (crosspoint2, N):
            toff1.gene[j] = toff2.gene[j]
            toff2.gene[j] = temp.gene[j]

        for j in range (crosspoint3, N):
            toff1.gene[j] = toff2.gene[j]
            toff2.gene[j] = temp.gene[j]

        for j in range (crosspoint4, N):
            toff1.gene[j] = toff2.gene[j]
            toff2.gene[j] = temp.gene[j]

        offspring[i] = copy.deepcopy(toff1)
        offspring[i+1] = copy.deepcopy(toff2)

# This function has a chance of introducing a change in each gene
def mutation():
    for i in range(0, P):
        newind = individual()
        newind.gene = []

        for j in range(0, N):
            curGene = offspring[i].gene[j]
            mutprob = ran.random()

            if mutprob < MUTRATE:
                    alter = ran.uniform(-MUTSTEP, MUTSTEP)
                curGene += alter
                if curGene > MAXVALUE:
                    curGene = MAXVALUE
```

Student name: Luqmaan Abdullahi

```
            elif curGene < MINVALUE:
                curGene = MINVALUE
        newind.gene.append(curGene)

    offspring[i].gene = newind.gene

# This function replaces the worst individual in a population with the one held in BESTINDIVIDUAL
def replaceWorst(pop):
    global BESTINDIVIDUAL
    fit = []

    for ind in pop:
        fit.append(ind.fitness)

    fit.sort()

    fit.pop(-1)
    pop.append(BESTINDIVIDUAL)


# This function overwrites the population with the offspring list
def offspringPopulation():
    population.clear()

    for i in range(len(offspring)):

population.append(copy.deepcopy(offspring[i]))

# This function returns the fitness
def getFitness(self):
    return self.fitness

# This function chooses a parent from the population using a tournament
def selectParent():
    offspring.clear()

    for i in range(0, P):
        parent1 = ran.randint(0, P-1)
        off1 = copy.deepcopy(population[parent1])
        parent2 = ran.randint(0, P-1)
        off2 = copy.deepcopy(population[parent2])
        parent3 = ran.randint(0, P-1)
        off3 = copy.deepcopy(population[parent3])
        parent4 = ran.randint(0, P-1)
        off4 = copy.deepcopy(population[parent4])
        parent5 = ran.randint(0, P-1)
        off5 = copy.deepcopy(population[parent5])
        parent6 = ran.randint(0, P-1)
        off6 = copy.deepcopy(population[parent6])

        test(off1)
        test(off2)
        test(off3)
        test(off4)
        test(off5)
        test(off6)

        offList = [off1, off2, off3, off4, off5, off6]
        offList.sort(key=getFitness)

        offspring.append(offList[0])

# This is the function that calls the functions in order to do evolution
def simpleGeneticAlgorithm():
    avgAvg.clear()
    average.clear()
    endIndividuals.clear()

     # This for loop repeats the algorithm for the same parameters the number of SUPERGENEREATIONS and saves the final population in a list
    for y in range(SUPERGENEREATIONS):
        population.clear()
        offspring.clear()
        newPopulation()
        testAll(population, False)

         # This is the main loop for the algorithm and it happens for the number of GENEREATIONS
        for x in range(GENEREATIONS):
            getBest(population)
            selectParent()
            #crossover()
            mutation()
            testAll(population)
            offspringPopulation()
            replaceWorst(population)

        avgAvg.append(average[-1])
        average.clear()
```

Student name: Luqmaan Abdullahi

```
endIndividuals.append(population)
```