An investigation into the effects of parameter changes and operator methods on genetic algorithm performance.

Ashley Pearson - 20001030

1 INTRODUCTION

This paper aims to explore the effects of chosen operators such as mutation and how the alteration of the parameters within these methods such as mutation rate can impact both the fitness achieved and the algorithms' ability to find the optimum solution within the search space. In addition is a brief investigation into the role of Artificial Intelligence (AI) in the modern world and the ethical concerns that can and will continue to arise given the rising prevalence of intelligent systems.

2 BACKGROUND RESEARCH

AI has become a buzzword in modern society, being heralded as a revolutionary technology that will free humans from mundane responsibilities that should be embraced wholeheartedly, but also as threat to civilisation as it is known, with fears a general super intelligence could reign supreme, superseding humans.

In truth, neither of these scenarios represent AI's current capabilities and offer a very black or white picture (Schwarz, 2019). However, there are multiple moral and ethical concerns arising as machine learning is ingrained deeper into all facets of modern life. This brief research aims to provide an insight into just two examples of the dilemmas faced both by private institutions and at national level.

Internet usage in China has quickly grown to 854.59 million individual users as of June 2019 (China Internet Network Information Center (CNNIC), 2019) and is widely considered to be at the forefront of becoming a digital society. The state announced in 2014 it would launch a social credit system (Creemers, 2014), similar to a traditional credit score, expanded to monitor all aspects of citizens lives, with "untrustworthiness" potentially resulting in reduced travel, employment and financial prospects, whilst positive behaviour is rewarded (Kostka, 2019).

One of the larger firms, Sesame credit which operates what is soon to be a mandatory credit score service, uses AI to rate individuals on a scale of 350-950 using big data from "a thousand variables across five data sets" (Campbell, n.d.) such as age, gender, "growth potential based on educational and professional history", compliance on social media and the scores of people with which they communicate online (Reis & Press, 2019).

The use of AI and big data to automatically assign scores to individuals allows an alarming degree of automated control over a population. It is believed Facial Recognition Technology (FRT) is already in use in combination with over 200 million surveillance cameras (Donnelly, 2021) to allow the system to also identify and track individuals offline. Given AI's history of making gender and racially biased predictions (Noor, 2020) in scenarios where it was not intentional, giving it direct control of an individual's livelihood where a bias, political or otherwise may be implemented poses alarming questions if used by a regime with an agenda,

FRT specifically is one of the most recognisable and favoured biometric methods, it has for example been used throughout COVID-19 in South Korea to track contacts and has aided global efforts to combat the virus. Its use resulted in lower incidences curves and lower mortality rates than countries without such tracking (Whitelaw et al., 2020).

It can however pose extreme ethical issues. Clearview AI is a relatively new company founded in 2017 which scrapes the web for faces matching that of the input. The program uses a neural net to convert images into vectors and store them in "neighbourhoods" based on similarities, when a photo is uploaded to be searched, vectors that are stored in the same neighbourhood are returned (Hill, 2020). This technology was quickly marketed toward law enforcement worldwide, offering a database of over 3 billion faces, it is used by over 600 law enforcement agencies in the US alone.

Concerningly, a private company is able to offer such data to anyone of their choosing, having already faced security breaches (Anon, 2020). The technology has been used in the US to arrest and charge an individual for committing a crime, typical police software would not have identified them as they were in no government databases, yet due to social media content they were traced within 20 minutes (Hill, 2020).

Twitter and Youtube among others have sent cease-and-desist letters regarding data scraping but the practice is passionately defended by the company due to its ability to combat many crimes, including "terrorism and child exploitation" (Rezende, 2020). Australia and the UK have begun legal proceedings against the practice on the grounds of data protection, despite being used in both countries previously (Anon, 2021).

Despite the controversary, the software is still growing in popularity and raises the question of the need for global discussion on data's use in AI. As the reader of this report there is a considerable chance that your face is accessible through the Clearview AI database, it must be discussed whether this is an acceptable price to pay to a private company's algorithm for the national security it may provide.

3 EXPERIMENTATION

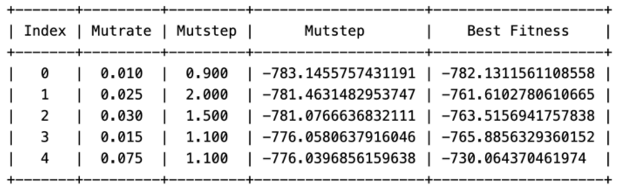
The program created and used during this investigation aims to demonstrate the influence of parameter changes and operator methods on the ability of a Genetic Algorithm (GA) to work towards the global optima of a search space, it is an evolutionary algorithm inspired by Darwin's natural selection built around the principles of selection and reproduction, with the initial population randomly generated.

The GA unless stated otherwise takes a population (P) of 200 individuals with a chromosome length (N) of 20, encoded as floats ran for 200 generations, with results averaged over two complete runs allowing the most consistent parameter combinations to be uninfluenced by outlying results. Results will be rounded to 3 decimal places for clarity and the only stopping criteria is fixed generations reached.

The program initially evaluates the fitness score of all individuals before performing selection on the population, with only superior individuals being chosen. The newly formed population is then subject to crossover, producing a new offspring. The offspring are then altered with the mutation operator, as crossover continually recombines two of the best solutions in the population there is a risk the lack of diversity may result in convergence at a local minimum, the introduction of 'random' values into the chromosome allows individuals the chance to escape the local minima across the search space.

Styblinski-Tang

The Styblinski-Tang function has a global minimum of -39.16599*N (Surjanovic & Bingham, 2013) resulting in an approximate minimum of -783.3198 where N=20 and each value within N (x) = between -5 and 5. Following population generation and evaluation the program runs a wide parameter sweep of both chance of mutation (mutrate) and maximum size of mutation (mutstep), mutrate ranges from 0 to 0.1 (0.1 = 10% chance) in 0.005 increments and mutstep ranges from 0 to 2 in 0.1 increments, all combinations of mutrate and mutstep within the ranges are tested, sorted by fitness and recorded, with the top five performers separated and displayed as shown in figure 1.


```
+-------+---------+---------+--------------------+--------------------+
| Index | Mutrate | Mutstep |      Mutstep       |    Best Fitness    |
+-------+---------+---------+--------------------+--------------------+
|   0   |  0.010  |  0.900  | -783.1455757431191 | -782.1311561108558 |
|   1   |  0.025  |  2.000  | -781.4631482953747 | -761.6102780610665 |
|   2   |  0.030  |  1.500  | -781.0766636832111 | -763.5156941757838 |
|   3   |  0.015  |  1.100  | -776.0580637916046 | -765.8856329360152 |
|   4   |  0.075  |  1.100  | -776.0396856159638 | -730.064370461974  |
+-------+---------+---------+--------------------+--------------------+
```
Figure 1 – Top 5 performing combinations with the best fitness achieved and the average fitness in P.

Tournament selection is used to improve the populations average fitness initially, with two random individuals being drawn from the population P times and having fitness's compared, with the greater being passed forward into the offspring. This is a simple selection method that can be scaled to a larger tournament where required, in this instance the small number of 'competitors' allows genetic diversity to be maintained. If two below average individuals are selected, regardless of their relative weakness to the population one shall be passed along allowing for potentially useful parts of the chromosome to be utilised in crossover. Other selection methods are widely used such as Roulette Wheel Selection however due to the fitness function handling both positive and negative values it would not be suitable for use, in other minimization functions it could be used if data was normalised to allow for correct proportionalities to be applied (Abd Rahman et al., 2016).

Crossover mimics mating in nature, combining two individuals (chosen as promising during selection) to create offspring that hopefully inherit the best parts of both parents to create a superior individual. Using single-point crossover the best achieved fitness score was -783.146, here a random point is chosen between 0 and N-1 to be used as the crosspoint, all values before the crosspoint in an individual are swapped with the corresponding value from the next member of the population. Multi-point crossover is very similar, however an additional crosspoint is selected between the initial crosspoint and N-1, all values beyond the new crosspoint are once again switched, this can take place any number of times, multi-point here achieved -781.673.

When ran with both single-point crossover and multi-point crossover for the full parameter sweep, whilst results are similar amongst the best achieving solutions it can be observed that especially with a higher population size a better individual is likely to be found with multi-point crossover, it is possible this is due to the higher likelihood an individual's values will be altered during crossover, increasing the probability weaker individuals will inherit stronger genes from the other parent. On initial sweep single-point crossover achieves only nine individuals with a score lower than -775, multi-point achieves 26, likewise the average population fitness is generally improved by an average of 12 points across the top 30 performers.

Offspring are then mutated to introduce genetic variation, the effect mutrate has on the GA's ability to search the space and obtain solutions is evident when tweaked to extremes. Setting mutrate to 1% results in early convergence due to the lack of genetic variation resulting in the population becoming trapped in local optimums. Setting mutrate to 50% however mutates so regularly that promising solutions are destroyed instead of being developed, with the average population fitness being pseudo-random akin to random search (see figure 2). Multiple runs of wide parameter sweeps suggest that a mutrate of up to 10% provides the best balance.

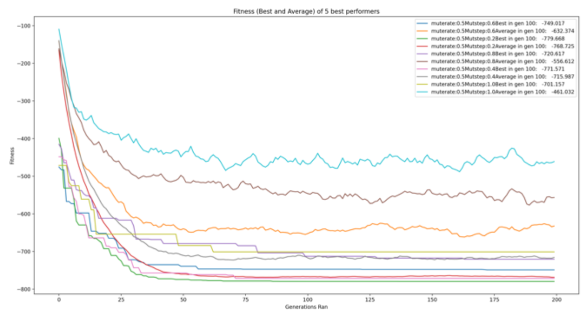


Figure 2 – High mutrate inhibits population from settling, hence causing an unpredictable average.

The program also implements elitism, once mutation is complete the best individual is sought out as well as the worst, the worst individual is then replaced with a duplicate of the best individual eliminating a relatively 'bad' combination of genes with a strong performer, whilst this reduces diversity slightly, it increases the chance that another poor individual will combine with a strong individual to create something useful.

The program then runs the five best mutrate/mutstep combinations discovered through the GA and plots the results to a graph (shown in Figure 3), the non-deterministic nature means that results vary, but from experimentation it is rare for outlying results to be produced by the selected combinations.

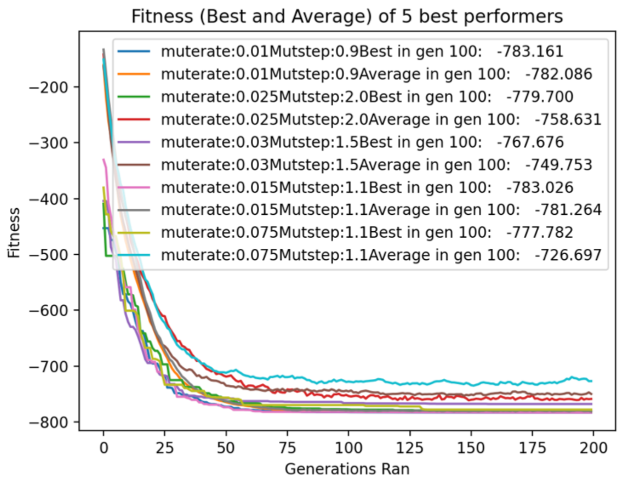


Figure 3 – Best performers graphed across a GA run.

Figure 4 shows the distribution of gene values in random individuals scattered across the graph, as opposed to the black squares representing the best found individual tightly centred around the global minima at -2.903534 (Surjanovic & Bingham, 2013). Figure 5 shows three runs of the best combination, with best fitness, average population fitness and the average of both figures, the graph shows all runs moving in a consistent fashion with scores coming together at approximately 50 generations.

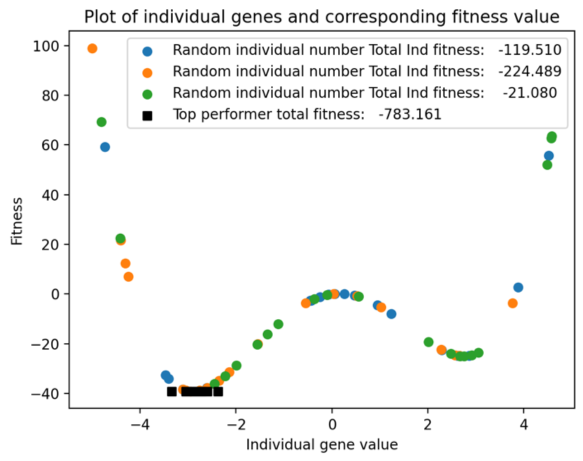


Figure 4 – Scatter showing random individuals vs best performer avoiding the local optimum.

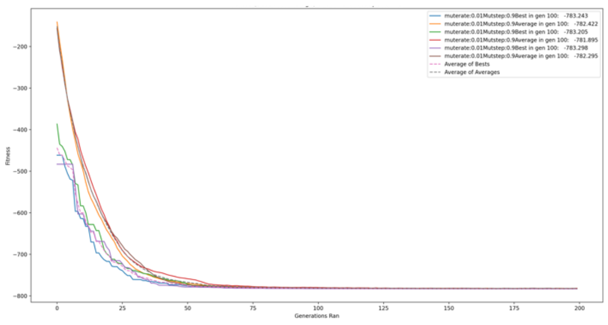


Figure 5 – 3 runs of the best combination plotted.

To investigate the effects of population size and number of generations the program runs the best combination with populations of 200-400 and generation numbers of 100-500. Unsurprisingly the higher the number of both the stronger the results typically, with the highest combination achieving -783.22 with single-point crossover. The greater the population size the higher the probability of finding a good individual at initiation and there are more opportunities for a single individual to mutate into a good solution. More generations ran also allows more chances at a beneficial mutation, likewise selection and crossover take place more times allowing the population to consistently fine tune. Figure 6 below shows the results of the generation/population experiment on the single-point crossover GA.

```
+------+----------+-------------------+-------------------+
| Runs | Pop Size |        Best       |      Average      |
+------+----------+-------------------+-------------------+
| 100  |   200    | -782.0853902718338 | -780.651653352363  |
| 100  |   250    | -768.4838856594257 | -767.1788701120753 |
| 100  |   300    | -782.7429092571437 | -781.1606262015626 |
| 100  |   350    | -782.6409523080247 | -781.1582249192676 |
| 100  |   400    | -783.0275830352413 | -781.739540454251  |
| 200  |   200    | -783.2601864560095 | -782.0786474926846 |
| 200  |   250    | -783.2601792306968 | -782.285712406458  |
| 200  |   300    | -783.2699841776599 | -781.9854123001428 |
| 200  |   350    | -783.2878128286391 | -782.0108138867221 |
| 200  |   400    | -783.2873540634781 | -781.9544726094093 |
| 300  |   200    | -783.3041467642835 | -782.2704128704129 |
| 300  |   250    | -783.3049752610656 | -782.1510853844024 |
| 300  |   300    | -769.1787271100588 | -768.2199646376828 |
| 300  |   350    | -783.3149981955642 | -782.1730244104668 |
| 300  |   400    | -783.3128235826582 | -782.5102379810469 |
| 400  |   200    | -783.3070664060567 | -782.0841858189584 |
| 400  |   250    | -783.3155277699376 | -782.2754669842759 |
| 400  |   300    | -783.3186806172581 | -782.1361531456865 |
| 400  |   350    | -783.3200154172565 | -782.3634434919128 |
| 400  |   400    | -783.3198874945309 | -782.0387991007126 |
| 500  |   200    | -783.3181151811796 | -782.2996814368257 |
| 500  |   250    | -783.3186287906217 | -782.6664937782999 |
| 500  |   300    | -783.3180934965083 | -782.068748887326  |
| 500  |   350    | -783.3223669615063 | -782.622104513928  |
| 500  |   400    | -783.3221906412045 | -782.4722953535374 |
+------+----------+-------------------+-------------------+
```

Figure 6 – Generation and Population experiment

Dixon-Price Function

This section of the report investigates the Dixon-Price test function and will focus on the results achieved with less explanation of the programs functionality; the only changes are the test function itself. Elitism, single point crossover and tournament selection are used unless stated otherwise. The test function is a minimisation function with a global optimum fitness score of zero (Surjanovic & Bingham, 2013).

Running the wide parameter sweep produced scores ranging from 1.019 through to 34,616.730. Of the top five performers in figure 7 three have a mutrate of 3% and two have 2% suggesting this range is desirable. Mutstep is more varied, however the value for four of the top achievers sits between 0.8-1, it is possible that due to slight value changes creating exponential fitness changes, mutrate should be relatively low to allow crossover to hone solutions, opposed to mutation jumping across the search space. However, the 19 worst performers of the sweep all have a mutstep of 0.1 or 0.2 regardless of mutrate, suggesting that when mutation does take place a low mutstep has a negative impact.

```
+-------+---------+---------+--------------------+--------------------+
| Index | Mutrate | Mutstep |    Best Fitness    |    Avg Fitness     |
+-------+---------+---------+--------------------+--------------------+
|   0   |  0.030  |  0.800  | 1.0189757146978886 | 5.953388039123576  |
|   1   |  0.020  |  1.000  | 1.1198782027732785 | 8.066173499583007  |
|   2   |  0.020  |  1.100  | 1.3799829005763153 | 8.417906515705765  |
|   3   |  0.030  |  1.000  | 1.411690155484591  | 11.526437273324559 |
|   4   |  0.030  |  0.600  | 1.4652430698892944 | 3.717949034099425  |
+-------+---------+---------+--------------------+--------------------+
```

Figure 7 – Top performers.

Upon rerunning the above combinations, index 1 was found to be the best achieving 0.884 and was chosen as the combination to investigate. Figure 8 shows 3 further runs, narrowly missing 0 on only 200 generations, whereas poor mutrate/mutstep combinations achieve scores in the 30,000 range.

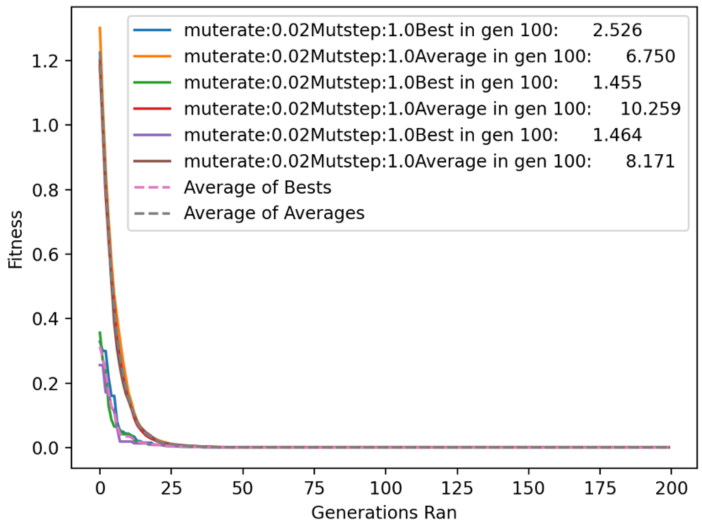
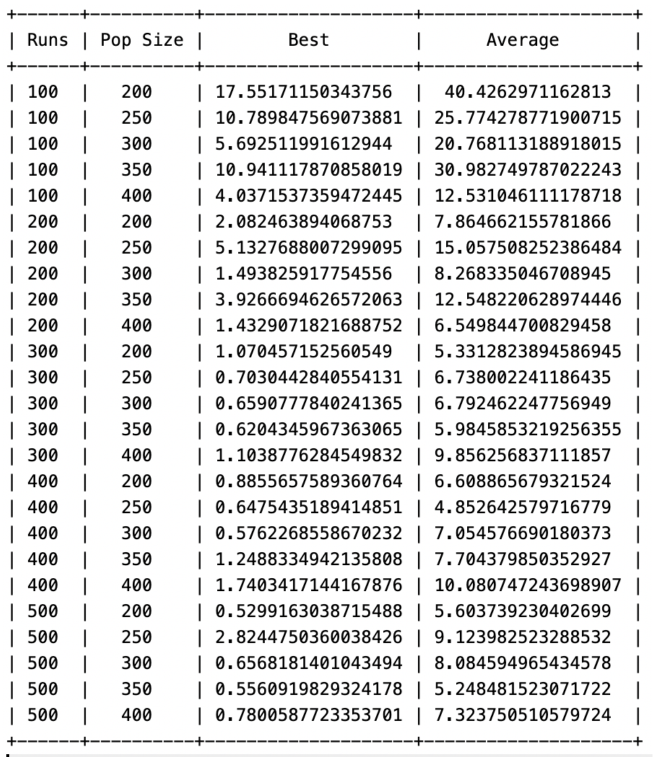


Figure 8 – Multiple runs of best combination.

Single point cross over appears to create better individuals, this may be due to the exponential change in fitness score with only small changes to an individual's values, meaning that extra disruption to the chromosome causes more harm than good once suitable individuals have been found. Multi-point crossover achieves 1.860 as the best score in the wide sweep but fails to achieve a sub-one score even when ran for P=400 and 500 generations, it is worth noting however that the worst result in the sweep was 20274.584, over 14,000 better than the worst in single point.

When population and generations are increased on the single-point individual however 0 is achieved multiple times, unsurprisingly the higher both P and runs, the better outcome. When looking at a population of 200 for example in figure 9, results consistently improve with an increased run size. When looking at all runs of 200 with varying population sizes however there is less consistent improvement with population size, suggesting that for this problem generations ran is paramount to the population size.



```
+------+----------+--------------------+--------------------+
| Runs | Pop Size |        Best        |      Average       |
+------+----------+--------------------+--------------------+
| 100  |   200    | 17.55171150343756  |  40.4262971162813  |
| 100  |   250    | 10.789847569073881 | 25.774278771900715 |
| 100  |   300    | 5.692511991612944  | 20.768113188918015 |
| 100  |   350    | 10.941117870858019 | 30.982749787022243 |
| 100  |   400    | 4.0371537359472445 | 12.531046111178718 |
| 200  |   200    | 2.082463894068753  | 7.864662155781866  |
| 200  |   250    | 5.1327688007299095 | 15.057508252386484 |
| 200  |   300    | 1.493825917754556  | 8.268335046708945  |
| 200  |   350    | 3.9266694626572063 | 12.548220628974446 |
| 200  |   400    | 1.4329071821688752 | 6.549844700829458  |
| 300  |   200    | 1.070457152560549  | 5.3312823894586945 |
| 300  |   250    | 0.7030442840554131 | 6.738002241186435  |
| 300  |   300    | 0.6590777840241365 | 6.792462247756949  |
| 300  |   350    | 0.6204345967363065 | 5.9845853219256355 |
| 300  |   400    | 1.1038776284549832 | 9.856256837111857  |
| 400  |   200    | 0.8855657589360764 | 6.608865679321524  |
| 400  |   250    | 0.6475435189414851 | 4.852642579716779  |
| 400  |   300    | 0.5762268558670232 | 7.054576690180373  |
| 400  |   350    | 1.2488334942135808 | 7.704379850352927  |
| 400  |   400    | 1.7403417144167876 | 10.080747243698907 |
| 500  |   200    | 0.5299163038715488 | 5.603739230402699  |
| 500  |   250    | 2.8244750360038426 | 9.123982523288532  |
| 500  |   300    | 0.6568181401043494 | 8.084594965434578  |
| 500  |   350    | 0.5560919829324178 | 5.248481523071722  |
| 500  |   400    | 0.7800587723353701 | 7.323750510579724  |
+------+----------+--------------------+--------------------+
```

Figure 9 – Generation and population experimentation.

4 CONCLUSIONS

Throughout the multiple experiments it became clear that beyond initial parameter changes, using as large of a population and as many generations as computationally affordable is the most beneficial change in finding the best solution regardless of the problem. If the experiments were to be repeated, it may be beneficial to implement stopping criteria based on fitness for problems with known solutions to report the efficiency of different parameters and operators, as shown in many of the graphs throughout the report many algorithms found the best solution early on and then aimlessly search the space for a considerable period afterwards with very marginal, if any improvements. Though by no means specialised or tailored for any one task, it is evident that the GA used here is robust in its ability to navigate two very different search spaces to find the optimum solution, when given feasible parameters and sufficient time.

REFERENCES

Abd Rahman, R., Ramli, R., Jamari, Z. & Ku-Mahamud, K. (2016) Evolutionary Algorithm with Roulette-Tournament Selection for Solving Aquaculture Diet Formulation. [online]. Hindawi Publishing Corporation. [Accessed 18 December 2021].

Anon (2020) Controversial firm Clearview gets hacked. 2020 (3), pp. 12-12. [Accessed 1 January 2022].

Campbell, C. (n.d.) How China Is Using Big Data to Create a Social Credit Score. Available from: https://time.com/collection/davos-2019/5502592/china-social-credit-score/ [Accessed 1 January 2022].

China Internet Network Information Center (CNNIC) (2019) Statistical Report on Internet Development in China. [online]. pp. 13. Available from: https://www.cnnic.com.cn/IDR/ReportDownloads/201911/P020191112539794960687.pdf [Accessed 16 December 2021].

Creemers, R. (2014) Planning Outline for the Construction of a Social Credit System (2014-2020). Available from: https://chinacopyrightandmedia.wordpress.com/2014/06/14/planning-outline-for-the-construction-of-a-social-credit-system-2014-2020/ [Accessed 20 December 2021].

Donnelly, D. (2021) China Social Credit System [Punishments & Rewards] in 2021. Available from: https://nhglobalpartners.com/china-social-credit-system-explained/ [Accessed 24 December 2021].

Hill, K. (2020) The Secretive Company That Might End Privacy as We Know It (Published 2020). Available from: https://www.nytimes.com/2020/01/18/technology/clearview-privacy-facial-recognition.html [Accessed 1 January 2022].

Houser, K. (2021) Artificial Intelligence and the Struggle Between Good and Evil. Washburn law journal. 60 (3), pp. 475-478. [Accessed 14 December 2021].

Kostka, G. (2019) China's social credit systems and public opinion: Explaining high levels of approval. New Media & Society. 21 (7), pp. 1565-1593. [Accessed 20 December 2021].

Noor, P. (2020) Can we trust AI not to further embed racial bias and prejudice?. BMJ. [Accessed 1 January 2022].

Reis, A. & Press, L. (2019) Sesame Credit and the Social Compliance Gamification in China. [online]. pp. 270-275. Available from: https://www.sbgames.org/sbgames2019/files/papers/ArtesDesignFull/196937.pdf [Accessed 1 January 2022].

Rezende, I. (2020) Facial recognition in police hands: Assessing the 'Clearview case' from a European perspective. New Journal of European Criminal Law. 11 (3), pp. 375-389. [Accessed 27 December 2021].

Schwarz, E. (2019) Günther Anders in Silicon Valley: Artificial intelligence and moral atrophy. Thesis Eleven. 153 (1), pp. 94-112. [Accessed 11 December 2021].

Surjanovic, S. & Bingham, D. (2013) Styblinski-Tang Function. Available from: https://www.sfu.ca/~ssurjano/stybtang.html [Accessed 19 December 2021].

Anon (2021) The UK's Information Commissioner's Office and the Office of the Australian Information Commissioner conclude joint investigation into Clearview AI Inc.. Available from: https://ico.org.uk/about-the-ico/news-and-events/news-and-blogs/2021/11/clearview-statement/ [Accessed 23 December 2021].

Whitelaw, S., Mamas, M., Topol, E. & Van Spall, H. (2020) Applications of digital technology in COVID-19 pandemic planning and response. The Lancet Digital Health. 2 (8), pp. e435-e440. [Accessed 1 January 2022].

Source code as an appendix – No further comment is made on the experiment beyond this point.

This point onward is formatted as one column to allow for clearer readability of code.

Code available as .txt or .py if required due to formatting issues.

The only variance in the below code is the test_func for the two experiments, as such both have been included and highlighted, only one of these may be ran at once

```
from typing import final
import matplotlib
from matplotlib import pyplot as plt
import random
import numpy as np
import copy
import statistics
from numpy.core.fromnumeric import size, sort
from tabulate import *
import json
import time


# Defining individual class


class individual:
    def __init__(self):
        self.gene = [0] * N
        self.fitness = 0


# Declare N(individual gene length) and P(Population size)
N = 20
P = 200

################ ADJUSTABLE MUTRATE/MUTSTEP SETTINGS ################
mutrate = 0.0
noofmutrateincreases = 20
mutrateincrement = 0.005
mutstep = 0.0
noofmutstepincreases = 20
mutstepincrement = 0.1
startingmutrate = mutrate
startingmutstep = mutstep
startingP = P

################ ADJUSTABLE MUTRATE/MUTSTEP SETTINGS ################

# Best of run holds the best score from a full run of the GA (100 generations for example.)
bestofrun = []
# avgofavgs holds a populations average score at the end of a run.
avgofavgs = []
# List of average is a 2-d array that holds the mutstep,mutrate average of best scores and average population scores after
    numerous runs.
listofaverage = []
```

```
# Set  numpy to 5 decimal places and supress scientific notation for readibility in tables.
np.set_printoptions(precision=5, suppress=True)

# Define test function


def test_func(ind, N):
    # Set fitness initially to 0.
    fitness = 0
    # Calculate total fitness of an individual.
    fitnesstotal = 0
    # Score individual fitness scores in an array.
    fitnessarray = []
    # Loop through an inidividuals length.
    for i in range(N):
        # Dixon-Price fitness function
        fitness = (ind.gene[i]**4 - 16 * ind.gene[i]**2 + 5 * ind.gene[i])
        fitness = fitness / 2
        fitnessarray.append(fitness)
    # Append individual fitnesses to an array.
    fitnesstotal = sum(fitnessarray)
    return fitness, fitnessarray, fitnesstotal

def test_func(ind, N):
    # Set fitness initially to 0.
    fitness = 0
    # Calculate total fitness of an individual.
    fitnesstotal = 0
    # Score individual fitness scores in an array.
    fitnessarray = []
    # Loop through an inidividuals length.
    for i in range(1, N):
        # Dixon-Price fitness function
        fitness = (i * ((2 * ((ind.gene[i] ** 2)) - (ind.gene[i - 1])) ** 2))
        fitnessarray.append(fitness)

    # Append individual fitnesses to an array.
    fitnesstotal = sum(fitnessarray) + (ind.gene[0] - 1)**2
    return fitness, fitnessarray, fitnesstotal


# define function that takes the best ten results from the wide parameter sweep, with the settings that produced those
    scores.


def scoreplotter(N, P, test_func, mutrate, mutstep, runs):
    # array to hold the average scores of a population
    avgscorelist = []
    # Array to hold a population of individuals
    population = []
    # Array to hold best score in a generation.
    yaxis = []

# Loop through population
```

```
    for x in range(0, P):
        # Create array to hold individiual gene values
        tempgene = []
        # Loop through genes in an individual
        for y in range(0, N):
            # Add a gene between of a value between -5 and 5 (x)
            tempgene.append(random.uniform(-5, 5))
            # Create an individual called new ind
        newind = individual()
        # Assign newind the genes created in the above loop.
        newind.gene = tempgene.copy()
        # Add the indiviudal to the population, continue until x=P
        population.append(newind)

    xaxis = []
    genz = 0
# Initialise popbestscore to the first individual in the population.
    fitscore = test_func(population[0], N)
    popbestscore = fitscore[2]
# Loop until the specified number of generations.
    for currentrun in range(0, runs):
        popscorelist = []
        genz += 1
# Loop through the population and assign a fitness score to each individual.
        for x in population:
            fitscore = test_func(x, N)
            x.fitness = fitscore[2]
# Create an array to contain offspring chosen from tournament selection.
        offspring = []
        # Loop through population, selecting and copying the
        for i in range(0, P):
            # create a copy of two random individuals
            parent1 = random.randint(0, P - 1)
            off1 = population[parent1]
            parent2 = random.randint(0, P - 1)
            off2 = population[parent2]
            # Check which copy has the better fitness and append to the new offspring array.
            if off1.fitness > off2.fitness:
                offspring.append(off2)
            else:
                offspring.append(off1)
# Loop through population in steps of 2 and select a crosspoint.
        crosspoint = random.randint(0, N - 1)
        crosspoint2 = random.randint(crosspoint, N - 1)
        for i in range(0, P, 2):
            tempgene = offspring[i].gene.copy()
            for k in range(0, crosspoint):
                offspring[i].gene[k] = offspring[i + 1].gene[k]
                offspring[i + 1].gene[k] = tempgene[k]
            for k in range(crosspoint2, N):
                offspring[i].gene[k] = offspring[i + 1].gene[k]
                offspring[i + 1].gene[k] = tempgene[k]

# Create an array to hold mutated genes.
        mutatedgenes = []
        for i in range(0, P):
            # Create an individual with an empty gene array.
            newind = individual()
            newind.gene = []
            # Loop through an indiviudals array of genes
            for j in range(0, N):
                mutprob = random.random()
                gene = offspring[i].gene[j]
                # pick a value between 0 and mutstep to alter the gene by.
                alter = abs(random.uniform(0, mutstep))
                # If random number between 0-1 is smaller than mutrate there is a 50% chance for the gene to have the alter
    rate added.
                if (mutprob) < (mutrate):
                    if random.random() < 0.5:
                        gene += alter
                        if gene > 5:
                            gene = 5
                    else:
                        # If random number between 0-1 is larger than mutrate there is a 50% chance for the gene to have the alter
    rate subtracted.
                        gene -= alter
                        if gene < -5:
                            gene = -5
                # Append the gene, altered or not the the new individual and then to the mutated genes array.
                newind.gene.append(gene)
            mutatedgenes.append(newind)
# Loop through mutated genes and assign fitness scores.
        for x in mutatedgenes:
            fitscore = test_func(x, N)
            x.fitness = fitscore[2]
# Set both the worst individual and best individual to individual 0 so a value is assigned.
        worsebaby = mutatedgenes[0]
        bestbaby = mutatedgenes[0]
# Loop through mutated genes until both the best and worst individuals have been found
        for x in mutatedgenes:
            if x.fitness < bestbaby.fitness:
                bestbaby = x
            if x.fitness > worsebaby.fitness:
                worsebaby = x
        # Get the index number of the worst individual in the array.
        indexno = mutatedgenes.index(worsebaby)
        # Replace the worst individual in the population
        mutatedgenes[indexno] = bestbaby
        # Make population a copy of the newly created mutated genes array.
        population = copy.deepcopy(mutatedgenes)
# Loop through population setting indiivdual fitness scores.
        for x in population:
            fitscore = test_func(x, N)
            x.fitness = fitscore[2]
            # Add fitness scores to popscorelist array.
            popscorelist.append(x.fitness)
            # Loop through the array finding the best score.
            if x.fitness <= popbestscore:
                popbestscore = x.fitness
                # Set a copy of the best individual to bestinrun.
                bestinrun = x
# Append popbestscore to array
        yaxis.append(popbestscore)
# Calculate the average fitness in a population.
        avgpopscore = statistics.mean(popscorelist)
# Append average score to an array containing average scores of each generation
        avgscorelist.append(avgpopscore)
        # Append the current run to the xaxis.
        xaxis.append(currentrun)
# If the run == the last run of the GA:
        if currentrun == runs - 1:
            # Store final best score and avg of population of the run.
            bestrunfinal = popbestscore
            avgrunfinal = avgpopscore
            # Plot best scores and average scores to the graph of best performers.
            plt.plot(yaxis, label="muterate:" + str(mutrate) +
                    "Mutstep:" + str(mutstep) + "Best in gen 100: " +
                    "{:10.3f}".format(popbestscore))
            plt.plot(avgscorelist, label="muterate:" + str(mutrate) +
                    "Mutstep:" + str(mutstep) + "Average in gen 100: " +
                    "{:10.3f}".format(avgpopscore))

    return xaxis, yaxis, avgscorelist, bestrunfinal, avgrunfinal, bestinrun


# Get the number of complete runs to average over
numbertoavgover = int(input("Number of generation runs to average over: "))

# Get the number of generations per run to complete.
runs = int(input("Enter the number of generations to run for: "))
startingruns = runs
# Store the settings of the run in a dictionary to be printed to file later.
GAsettings = {"Starting mutrate: ": startingmutrate, "Starting mutstep": startingmutstep, "Number of runs to average
    over": numbertoavgover,
            "No of mutstep increases": noofmutstepincreases, "No of mutrate increases": noofmutrateincreases, "Size of
    mutrate increases": mutrateincrement, "Size of mutstep": mutstepincrement, "Number of generations": runs}

# Start a timer
start = time.time()
# Set a progresser counter to 0 to be incremented on each generation.
progresscounter = 0
# Set total number of runs to allow for visual progress check on larger runs.
totalruns = (noofmutrateincreases * noofmutstepincreases *
            numbertoavgover * runs)

# Loop through number of mutstep increases
for a in range(noofmutstepincreases):
    # increment mutstep value
    mutstep += mutstepincrement
    # each time inner loop returns to here, reset mutrate to starting mutrate.
    mutrate = startingmutrate
    # Loop through mutrate increases
    for z in range(noofmutrateincreases):
        # increment mutrate value
        mutrate += mutrateincrement
```

```
# Complete each of the parameter combinations 'numbertoavgover' times.
for x in range(numbertoavgover):
    population = []
    # Population initialisation.
    for gen in range(0, P):
        tempgene = []
        for y in range(0, N):
            tempgene.append(random.uniform(-5, 5))
        newind = individual()
        newind.gene = tempgene.copy()
        population.append(newind)
    # Set popbestscore.
    fitscore = test_func(population[0], N)
    popbestscore = fitscore[2]
    # Loop through runs
    for currentrun in range(0, runs):
        popscorelist = []
        avgpopscore = []
        for x in population:
            fitscore = test_func(x, N)
            x.fitness = fitscore[2]

        # Selection
        offspring = []
        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:
                offspring.append(off2)
            else:
                offspring.append(off1)

        # Crossover
        crosspoint = random.randint(0, N - 1)
        # The below line is removed for single point crossover.
        crosspoint2 = random.randint(crosspoint, N - 1)
        for i in range(0, P, 2):
            tempgene = offspring[i].gene.copy()
            for k in range(0, crosspoint):
                offspring[i].gene[k] = offspring[i + 1].gene[k]
                offspring[i + 1].gene[k] = tempgene[k]
        # The below lines are removed for single point crossover.
            for k in range(crosspoint2, N):
                offspring[i].gene[k] = offspring[i + 1].gene[k]
                offspring[i + 1].gene[k] = tempgene[k]
        # Mutation
        mutatedgenes = []
        for i in range(0, P):
            newind = individual()
            newind.gene = []
            for j in range(0, N):
                mutprob = random.random()
                gene = offspring[i].gene[j]
```

```
                        alter = abs(random.uniform(0, mutstep))
                        if (mutprob) < (mutrate):
                            if random.random() < 0.5:
                                gene += alter
                                if gene > 5:
                                    gene = 5
                            else:
                                gene -= alter
                                if gene < -5:
                                    gene = -5
                        newind.gene.append(gene)
                    mutatedgenes.append(newind)
                # Assign fitness values
                for x in mutatedgenes:
                    fitscore = test_func(x, N)
                    x.fitness = fitscore[2]

                # Elitism
                worsebaby = mutatedgenes[0]
                bestbaby = mutatedgenes[0]
                for x in mutatedgenes:
                    if x.fitness < bestbaby.fitness:
                        bestbaby = x
                    if x.fitness > worsebaby.fitness:
                        worsebaby = x
                indexno = mutatedgenes.index(worsebaby)
                mutatedgenes[indexno] = bestbaby
                population = copy.deepcopy(mutatedgenes)

                # Assign fitness scores
                for x in population:
                    fitscore = test_func(x, N)
                    x.fitness = fitscore[2]
                    popscorelist.append(x.fitness)
                    if x.fitness <= popbestscore:
                        popbestscore = x.fitness
                # Calculate average population score
                avgpopscore = statistics.mean(popscorelist)
                # Print progress.
                progresscounter += 1
                print(progresscounter, "/", totalruns)
            # Append best score to an array.
            bestofrun.append(popbestscore)
            avgofavgs = statistics.mean(popscorelist)
            # If number of final popbestscores recorded == number to avg over:
            if len(bestofrun) == numbertoavgover:
                # Calculate an average of best final scores
                averageofruns = statistics.mean(bestofrun)
                # Clear bestofrun for next parameter run
                bestofrun.clear()
                # Append mutrate, mutstep, average of best scores and
                listofaverage.append(
                    ["{:10.3f}".format(mutrate), "{:10.3f}".format(mutstep), float(averageofruns), float(avgofavgs)])
# End timer
end = time.time()
```

```
elapsed = end - start

# created sorted list sorted on index 2 of list of average(averageofruns)
sortedlist = sorted(listofaverage, key=lambda x: x[2])

# Set current best to the first in loop and then loop through bests to find the true best
currentbest = sortedlist[0]
for x in sortedlist:
    if x[2] < sortedlist[0][2]:
        currentbest = x
# Create a table of the bests
table = tabulate(listofaverage, headers=["Index",
                "Mutrate", "Mutstep", "Best Fitness", "Avg Fitness of gen 100"], showindex="always", tablefmt="pretty")
# Create a table of the list sorted based on best score achieved.
tablesorted = tabulate(sortedlist, headers=["Index",
                       "Mutrate", "Mutstep", "Best Fitness", "Avg Fitness of gen 100"], showindex="always", tablefmt="pretty")
# Create an array to hold the best 5 performers from the sorted list.
topperformers = []
for i in range(5):
    topperformers.append(sortedlist[i])
# Create a table of the best 5 performers.
toptable = tabulate(topperformers, headers=[
    "Index", "Mutrate", "Mutstep", "Best Fitness", "Avg Fitness"], showindex="always", tablefmt="pretty")

# Create an array to hold the results of running the top 5 performers.
finalresults = []
p = 0
for x in topperformers:
    # Pass in N,P, test function and the mutrate and number of runs to run.
    toptenrun = scoreplotter(N, P, test_func, float(x[0]), float(x[1]), runs)
    if p == 0:
        best = toptenrun[5]
    # Record the avg population score.
    avgs = toptenrun[4]
    # Record the best recorded score.
    bests = toptenrun[3]
    # Append the results and mutrate, mutstep to an array.
    finalresults.append([x[0], x[1], bests, avgs])
    p += 1
# Sort the final results array and create a table.
finalresults = sorted(finalresults, key=lambda x: x[2])
finalresultstable = tabulate(finalresults, headers=[
    "Index", "Mutrate", "Mutstep", "best fitness", "avg fitness"], showindex="always", tablefmt="pretty")


# Open/Create a text file and print GA settings, all 4 tables, time to complete, and the best final performer before closing
    the file.
f = open(
    "/Users/ashleypearson/Documents/UWE/Year Two/AI2/Assignment/Testresults/runresults.txt", "w")
f.write("-----------------GA Results-----------------\nSettings as follows:\n")
f.write(json.dumps(GAsettings) + "\n")
f.write("Unsorted data:\n" + table + "\n")
f.write("Sorted table: \n" + tablesorted + "\n")
f.write("\nTop 10 performers to further investigate: \n" + toptable + "\n")
f.write("\nFinal Results of running the 5 best discovered combinations" +
        str(runs) + "\n" + finalresultstable)
f.write("\nTime to run: " +
        str(elapsed / 60) + " minutes")
f.write("\nBest performer is:" + "\nMutrate:" +
        str(finalresults[0][0]) + "\nMutstep:" + str(finalresults[0][1]) + "\nBest score: " + str(finalresults[0][2]) + "\nAverage
    score: " + str(finalresults[0][3]))
# Close the file.


# Plot the 5 best performers on a graph, and show it.
plt.legend()
plt.xlabel('Generations Ran')
plt.ylabel("Fitness")
plt.title("Fitness (Best and Average) of 5 best performers")
plt.show()

# Create a scatter graph
fig = plt.figure()
# Create 3 random individuals with random genes
for x in range(3):
    randomind = individual()
    population = []
    tempgene = []
    for y in range(0, N):
        rando = random.uniform(-5, 5)
        tempgene.append(rando)
    randomind.gene = tempgene.copy()
    # Add the individual genes to a new Xaxis array.
    xaxis = []
    for x in range(0, 20):
        xaxis.append(randomind.gene[x])
    # Get the fitness scores for individual genes
    runrandom = test_func(randomind, N)
    # Plot the individiual gene values, and fitness scores on a scatter graph.
    plt.scatter(xaxis, runrandom[1], label=(
        "Random individual number: " + str(x) + "Total Ind fitness: " + "{:10.3f}".format(runrandom[2])))
# Create a new x acis array
xaxis = []
# add the individual gene values of the best discovered individual to an array.
for x in range(0, 20):
    xaxis.append(toptenrun[5].gene[x])
# get the fitness values of best individual genes.
run = test_func(best, N)

# Plot the best performer on the above scatter with Black Squares as the icon to make for consistent, easy comparison.
plt.scatter(xaxis, run[1], marker='s', c='black',
            label="Top performer total fitness: " + "{:10.3f}".format(run[2]))
# Format scatter as desired with correct labels and styles.
plt.ticklabel_format(style="plain")
plt.legend()
plt.ylabel("Fitness")
plt.xlabel("Individual gene value")
plt.title("Plot of individual genes and corresponding fitness value")
plt.show()
```

```
best0 = []
average0 = []
# Run the best individual 3 times with a fixed population and generation number and plot to a graph to show multiple
    runs
for x in range(3):
    run = scoreplotter(N, P, test_func, float(
        finalresults[0][0]), float(finalresults[0][1]), runs)
    # Save the best scores and averages scores from each run to an array.
    best0.append([run[1]])
    average0.append([run[2]])

# Calculate and plot averages of bests and averages to the same graph.
# (Method of calculating mean taken from StackOverflow user - Saullo G. P. Castro, Initial code available below:)
# https://stackoverflow.com/questions/18461623/average-values-in-two-numpy-arrays/18461943
bests = np.mean(np.array([best0[0], best0[1], best0[2]]), axis=0)
plt.plot(bests[0], '--', label="Average of Bests")
avgs = np.mean(np.array([average0[0], average0[1], average0[2]]), axis=0)
plt.plot(avgs[0], '--', label="Average of Averages")

plt.legend()
plt.xlabel('Generations Ran')
plt.ylabel("Fitness")
plt.title("Fitness (Best and Average) of 5 runs of best performer")
plt.show()

scorelist = []
runs = 0
# Create a nested loop that runs th best performer found from the above searches with an incrementing number of
    population members and generations.
counter = 0
for x in range(5):
    runs += 100
    P = startingP
    for j in range(5):
        counter += 1
        print(str(counter) + "/" + str((5 * 5)))
        score = scoreplotter(N, P, test_func, float(
            finalresults[0][0]), float(finalresults[0][1]), runs)
        scorelist.append([runs, P, score[3], score[4]])
        P += 50
scorelist = tabulate(scorelist, headers=[
    "Runs", "Pop Size", "Best", "Average"], tablefmt="pretty")
print(scorelist)
f.write("\n" + scorelist)

f.close()
```