Optimisation Algorithms and Evolutionary Intelligence

Conran Pearce 17027351

1 INTRODUCTION

This report expresses the implementation of a Genetic Algorithm (GA) to solve optimisation problems. Competitive performance on numerical functions is expressed throughout the experimentation. This investigation primarily shows the results of changing the varying parameters in a GA. Selection, crossover and mutation are key components of GA, which are altered changing the output and performance of the algorithm. The solutions determined by these alterations are discussed along with further parameter changes.

A broad general definition of optimisation is explored in this report. Along with discussing optimisation algorithms that have made the news, considering the ethics that result from these algorithms. Along with the comparative performance of Particle Swarm Optimisation (PSO) against GA being carried out in this report.

2 BACKGROUND RESEARCH

2.1 Optimisation

The functionality of an optimisation algorithm is to create a fitness function, evaluating the performance of a function (Tang et Wu, 2011). The algorithm explores various solutions until an optimum or satisfactory solution is achieved. Optimisation algorithms can also be defined as deterministic or stochastic (IITM, 2020). Deterministic algorithms work mechanically without random nature, whereas Stochastic algorithms have some randomness involved. Stochastic algorithms will typically reach a different endpoint every run, despite having the same initial starting point. Where GAs are an example of a stochastic optimisation algorithm (Weile et Michielssen, 1997).

2.2 Using GAs to produce police sketches

GAs are used by police around the world to construct facial compositions of offenders (EvoFit, 2020). EvoFIT produces random plausible facial characteristics, by a GA. Which are then breed together to form facial constructions (Frowd et Hancock, 2008). EvoFIT aim is to create a practical and ethical composition (Frowd et al, 2019). Figure 1 shows the solutions from an EvoFIT facial construction.

Figure 1 (Frowd et al, 2004).

EvoFIT'S GA works by defining the shape and texture of each face, which are held as real number variables in the algorithm (Frowd et al, 2004). Population size and mutation rate are parameters in this algorithm, where Figure 2 shows the parameters in full (Frowd et Hancock, 2008). Typically, a witness would then manually select similar faces, where the GA replaces this with selection, producing a solution with "likeness" to the target face.

Parameter	Setting
Population size	10-32
Generations	5-12 (depending on population size)
Mutation rate	0.1 (probability per parameter)
Breeding opportunities	2:1 (Best face : other selected faces)
Elitism	Enabled
No. of selected faces	Fewer is better (lower limit 3 or 4)
Selection of shapes and textures	Separate

Figure 2 (Frowd et Hancock, 2008).

EvoFIT's algorithm has been in the media exposing its problems. The algorithm had created a construction appearing to be "wearing a lettuce on his head" (Kelly and Winterman, 2010). Causing ethical concern about the reliability and suitability of using an algorithm to help in police investigations. However, EvoFIT's GA has enabled a suspect identification rate of 60%, in comparison to the significantly smaller 5% which occurs from the police's 'feature' system (EvoFIT, 2020). However, comparing EvoFIT facial construction results against

more typical composite systems showed it is not always able to outperform other systems (Rhodes et Haxby, 2011).

2.3 GAs to evade internet censorship

The University of Maryland has made the news for being able to evade internet censorship, using a GA (SciTechDaily, 2019). This GA produces censorship evasion strategies through evolution (Bock et al, 2019). Censorship is typically put in place by governments, prohibiting individuals access to open and free information (SciTechDaily, 2019).

China has the largest censorship firewall in the world (Bloomberg, 2018), known as the "Great Firewall of China" (WhatIs, 2020). This is a national effort to protect Chinese corporate, state secrets and infrastructure from cyberattacks. But this firewall is being used to control what information Chinese citizens can access, resulting in human rights campaigns against Chinese speech censorship due to the ethical concerns (Kim et Douai, 2012). Researchers performed testing in-lab and against real censors in China, India and Kazakhstan (Bock et al, 2019). The GA has proved that it can evade internet censorship by automatically deriving censorship evasion strategies through the evolution stage in the GA. Mutation and crossover occur producing new strategies at each generation of the population. The algorithm successfully discovered all previously published schemes to evade the Great Firewall of China, along with deriving new evasion strategies, that were not previously possible (during in lab testing) (Bock et al, 2019). It could be argued that it was unethical of the researchers to have tested against the Great Firewall of China because there is a justified expectation that China, as a sovereign nation, should be able to have its own policies regarding internet censorship without any interference from outside individuals (D'jaen, 2007).

It has been suggested that the best way to fight against internet censorship would be to develop a global, industry-wide code of conduct, which takes into account the diversity of national values (set by governments) (D'jaen, 2007). However, internet censorship affects those that are in pursuit of free exchange of ideas and the freedom of expression (Stevenson, 2007). Freedom of expression is a fundamental human right act in Article 19 of the Universal Declaration of Human Rights (United Nations, 2020). Therefore, the GA is able to help human rights by evading internet censorship.

2.4 Evolutionary algorithms to create a living robot

Scientists have used an Evolutionary Algorithm (EA) to form the first living robots, known as "Xenobots" (ScienceDaily, 2020), from frog embryo stem cells (Sample, 2020). These robots are living programmable organisms, with the potential to destroy cancer cells and deliver drugs (Coghlan et Leins, 2020). To create Xenobots; stem cells are separated into singular cells and are incubated, cut and joined together through designs specific from an algorithm. The EA forms the cells into new body forms (candidate designs), where the cells can work together.

Candidate designs are created using a behavioural goal and structural building blocks (Kriegman et al, 2020). The EA returns the best candidate design that was found. The EA is run 100 times, starting with a different random population, producing a diverse number of designs. Results are analysed and returned to the EA, influencing further designs. Figure 3 shows designs evolving from the EA.

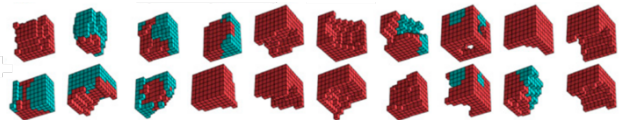

Figure 3 (Kriegman et al, 2020).

Scientists behind Xenobots are aware that there are ethical issues (Coghlan et Leins, 2020). Concerns include if Xenobots should be treated as robots or living creatures (Heaven, 2020). Xenobots may evolve to have nervous systems and sensory cells, giving cognitive abilities (Heaven, 2020). Xenobots current inability to reproduce gives an argument for calling them "life forms" and not organisms. But future development may result in a more intelligent and complex Xenobots, capable of reproduction.

3 EXPERIMENTATION

3.1 Overview of experimentation

This section of the report shows the effects of changing the parameters in a GA algorithm when solving optimisation problems in relation to

behaviour and performance. GA should be able to search and find the best solution for a problem, with a good computational efficiency during search (Wang et al, 2012). Optimisation deal with the problem of maximisation or minimisation for a function (Sivanandam et Deepa, 2008). This report will focus on minimisation.

Parameters for the GA are stated below, along with the fitness calculation seen in Figure 4, which is used in this GA. The population of the GA is initialised of 50 randomly assigned real values within the upper and lower bounds of 32.0 and -32.0 respectively. The number of chromosomes an individual has was set to 20, other than in section 3.6 where a chromosome length of 200 has been explored. The mutation rate was set between 1/Population size and 1/Chromosome length. Except from in section 3.5 where the mutation rate is explored. The mutation step was set between a random float between 0 and the upper bound. 300 generations are carried out in this algorithm, where in each generation, 10 of the best solutions generated are swapped with the worst solutions in the population. The standard used is; tournament selection, single-point crossover and random mutation, except when these parameters are explored in section 3.2, 3.3 and 3.4 respectively. To ensure the reliability of the outputs, all tests were run over 10 iterations, where the average result has been taken.

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

Figure 4, Fitness calculation used in GA.

The coding during this research was carried out using Python as the programming language. This interpretation language allows algorithms to be produced, with libraries allowing easy production of graphs to display results (Plotly, 2020). High readability and simple syntax allow other developers to be able to understand and reproduce results (Desmond et MacKenzie, 2020).

3.2 Selection process in GAs

Selection strategies are used in a GA to select individuals, usually of high fitness (Jebari et Madiafi, 2013), from a population which are then used as a parent to generate the next population (Jebari et Madiafi, 2013). This section of the report involves exploring the outcomes of varying the selection process in the parametrical GA.

Firstly, tournament selection is explored. This involves selecting k-individuals and run a tournament among them. Where only the fittest individuals can move on to the next generation (GeeksforGeeks, 2018). Typically, the larger the tournament size, the smaller the chance the individuals with a less optimal fitness value have to be passed on to the next generation.

Secondly, roulette-wheel selection is then applied to the GA. This selection strategy consists of all chromosomes in the population being placed in a roulette-wheel based on their fitness value (Kumar, 2012). Where the more optimal the fitness value is the larger the segment on the roulette wheel. However, there is no guarantee that good individuals will be passed on to the next generation. This wheel is then spun and the segment it lands on represents the individual selected.

Rank selection is a strategy which involves ranking the population based on the individual's fitness values in order from 1 (worst fitness value) to the size of the population (Basak, 2018). All the chromosomes then have a chance to be selected, but individuals with optimal fitness values have a greater chance to be selected.

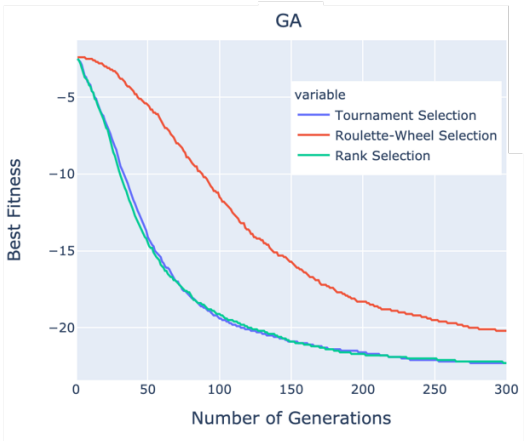

Figure 5, Best solutions found for minimisation by the GA focusing on selection strategies.

Figure 5 shows the selection methods all producing similar convergence to the optimal solution, except

for roulette-wheel selection. Figure 5 expresses a well-known drawback of roulette-wheel selection, where there is a risk of premature convergence of the GA to a local optimum. Roulette-wheel selection would take more computational time to find the optimal solution, compared to the other selection methods. This is due to the selection strategy not guaranteeing to select the individuals with the best fitness Where the global optima was discovered by all of the other selection strategies in a sufficient time.

3.3 Crossover of chromosomes in GAs

Crossover involves swapping parts of solutions with other chromosomes from the same population to provide a mix of solutions (Yang, 2014). Therefore, providing a variety of solutions in the population.

Single-point crossover involves choosing a random crossover point on two individuals and swapping the alternating heads and tails of these individuals, to form new offsprings (Tutorialspoint, 2020).

Multi-point crossover expands on single-point crossover, where multiple segments in individuals swapped, producing offspring (Tutorialspoint, 2020).

Uniform crossover is another method used to create new offspring. This process involves randomly selecting if a gene is selected from the parent chromosomes to see if it will be in the offspring (Tutorialspoint, 2020). The use of tossing a coin is an example technique for determining if the chromosome is passed selected.

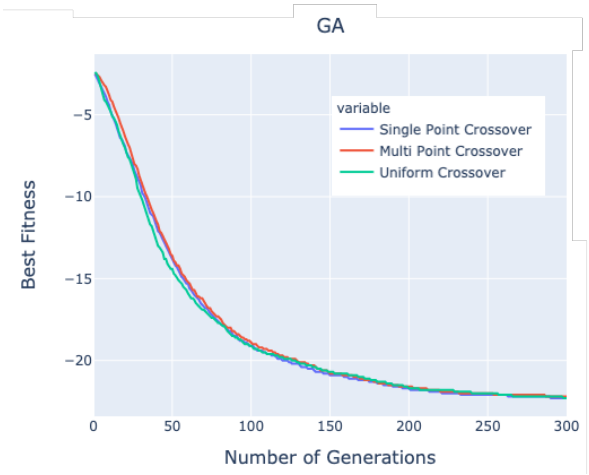

Figure 6, Best solutions found for minimisation by the GA focusing on crossover.

Figure 6 expresses that uniform crossover has a marginally faster convergence to the global optima compared to single-point and multi-point crossover. Single-point and multi-point crossover had very similar convergence to the global optima. None of the crossover methods results showed convergence to a local optima.

3.4 Mutation in GAs

Mutation involves the changing of a part of a solution randomly, to increase the diversity of a population. Which then allows a mechanism to escape from a trapped local optimum (Yang, 2014). Convergence to the local optima causes a loss in genetic diversity in the whole population (Rocha et Neves, 1999).

Random mutation is carried out during this experimentation, which involves setting a random value from a set of permission values (upper and lower bound) to be assigned to a randomly chosen gene (Tutorialspoint, 2020). Firstly, a random mutation probability (a random number between 0 and 100) is created and compare this against the mutation rate, to see if a mutation should be carried out in the chromosome. If the probability is successful, then we randomly select if the value is going to be added or subtracted. We then carry out this mathematical equation, updating the offspring with a new solution.

Gaussian mutation is the process of adding a random value from a Gaussian distribution to an individual when carrying out mutation, producing a new offspring (University of Malaga, 2020). Gaussian mutation results will vary depending on the standard deviation (SD) of the distribution, as seen in Figure 7. Gaussian mutation has a disadvantage that its local search ability is stronger than its global search (Wang et al, 2019). Figure 7 shows that the algorithm falls into local optimums at the varying standard deviations. SD of 1 had the worst performance when compared to the other deviations. SD of 5 found the optimal solution over the 300 generations run. When comparing Gaussian mutation to the other mutation methods, there was a SD of 5. As seen in the results of Figure 8.

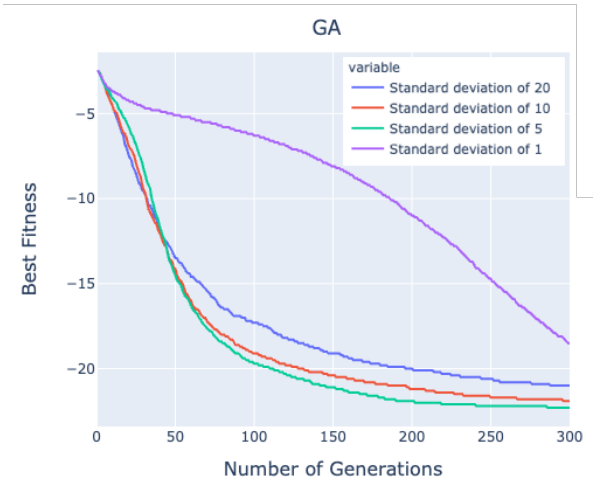

Figure 7, Best solutions found for minimisation by the GA focusing on varying deviations in Gaussian mutation.

Scramble mutation involves randomly shuffling a random subset of genes in a chromosome (TutorialsPoint, 2020). This mutation method maintains diversity by producing a new random sequence permutation (Adrian et al, 2015). During this experimentation, the number of genes being mutated had to be greater than 1 and less than the whole chromosome length.

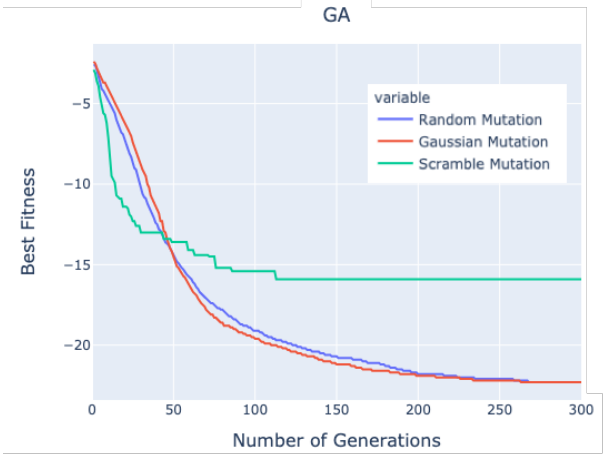

Figure 8, Best solutions found for minimisation by the GA focusing on mutation strategies.

Figure 8 shows that both random and Gaussian mutation reach the global optimal solution after the 300 generations carried out. Random mutation has a greater initial convergence to the optimal solution, however, after around 50 generations Gaussian mutation has a greater convergence towards the global optima. Also, after around 50 generations scramble mutation starts to plateau, where it is not able to find the global optima. Scramble mutation is a mutation method which yields a connected fitness landscape (Prokopec et Golub, 2016). Which results in not having a local optima. However, it is outperformed by the other mutation methods explored.

3.5 Other parameters influencing GAs

There are several other parameters other than selection, crossover and mutation which affects the output and performance of the GA.

Mutation rate was a parameter which significantly affected the solutions produced by the GA. Figure 9 displays the varying mutation rate when using random mutation in the GA. A larger mutation rate of 0.2 shows an initial faster convergence to the global optima, however, the algorithm started to become outperformed by a mutation rate of 0.02 at around 75 generations, with the other mutation rates converging closer to the global optima from around 140 generations. A mutation rate of 0.02 was able to find the global optima solution after 300 generations. Whereas a mutation rate of 0.002 and 0.0002 needed further generations to be carried out to reach the optimal (minimum value) of -22.9.

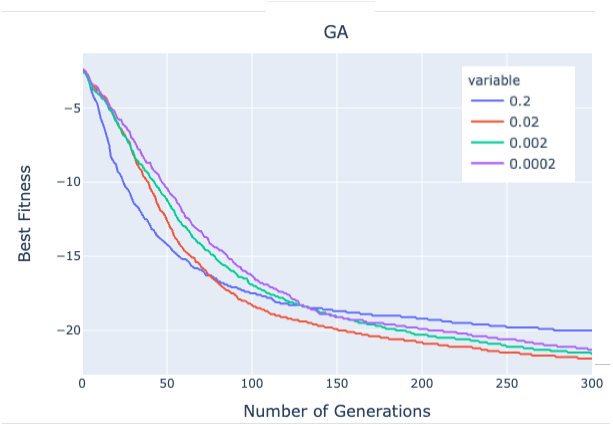

Figure 9, Best solutions found for minimisation by the GA focusing on mutation rates.

Repeating the GA with more than one generation enables the genetic pool to be completed with new solutions, where the genetic pool can be evaluated and restarted to create further solutions (Camara, 2015). 300 generations were chosen during the experimentation of this report. After each generation,

some of the best candidate solutions that are produced in the offspring, are selected and replace the worst candidates in the population. This is repeated through several generations, to arrive at a global optima solution. The number of candidate solutions being swapped at each generation determines the performance of the GA. Figure 10 shows that the greater the number of individuals selected and swapped after each generation, the faster the GA converges to the global optimal solution. As seen in Figure 10, only swapping 1 individual per generation has a slower convergence, where it is not able to reach the global optima after 300 generations. However, swapping 50 individuals (the size of the whole population) in this experimentation, the convergence to the global optima was great, where this number of swaps enabled to find the global optima the fastest.

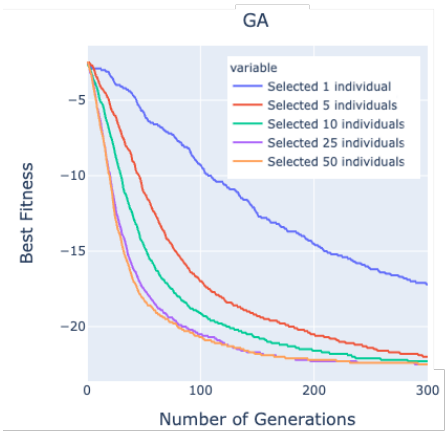

Figure 10, Best solutions found for minimisation by the GA focusing on varying about of individuals selected per generation.

The methods which produced the most efficient optimum results during this report have been implemented together, with a larger chromosome length than previously tested against. A chromosome length has now been set to 200, prior it was set to 20. Larger chromosome length causes a greater computational cost (Kim et Weck, 2005), therefore putting a greater strain on the GA to find an optimum result efficiently. After producing the most efficient results when testing the parameters during this report, tournament selection, uniform crossover, Gaussian mutation and a mutation rate of 0.02 have been chosen to explore the effects of changing the chromosome length to 200. SD of 10 was used in Gaussian mutation. This has been compared against lower-performing methods where roulette-wheel selection, single-point crossover, random mutation and a mutation rate of 0.2. Figure 11 shows both of the GAs have a slower performance of producing optimum solutions, compared to the previous fitness values produced with a chromosome size of 20. The parameters which individually had a better performance implemented together in a GA outperforms the individually lower-performing methods in the GA. The GA consisting of tournament selection, uniform crossover, Gaussian mutation and a mutation rate of 0.02 had a greater convergence to the global optima in a more efficient time, due to being able to create a greater diversity of individuals in the population. Reaching an average minimum fitness value of -10.3 over 300 generations and 10 iterations. Whereas the GA consisting of roulette-wheel selection, single-point crossover, random mutation and mutation rate of 0.2 produced less diversity in the population, therefore lacking optimal solutions being obtained. Where a worse minimum fitness value of -4.3 was produced. Also, the computation time for the algorithm to run when a chromosome length is 200 is significantly longer than when the length is set to 20.

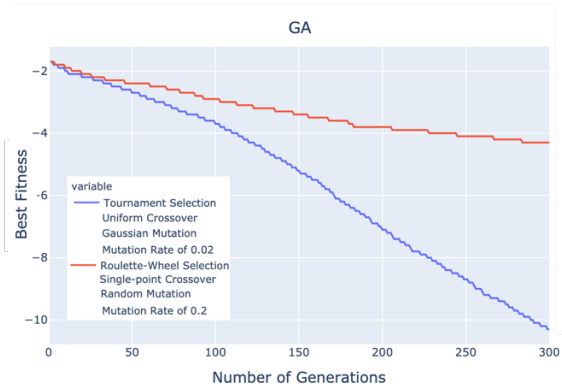

Figure 11, Best solutions found for minimisation by the GA when the chromosome length is 200.

3.6 Comparison of GA against PSO

PSO is a swarm intelligence-based optimisation algorithm (Jatana et Suri, 2020), developed using the analogy of social behaviour in a flock of birds. Both GA and PSO are used for their ability to obtain a global optima (Wang et al, 2012). Where the fitness of an objective is evaluated for each individual. However, PSO differs from GA from that each particle in PSO is influenced by three factors: it's existing momentum, the best position since the first iteration and the best position obtained by any

individual in the swarm.

Competitive comparison has been carried out between PSO algorithm (implemented by Peer et al) (Peer et al, 2003) and the GA proposed in this report, to solve the optimisation problem of Schwefel's function. This function is an optimisation problem to find the global minimum, where the fitness calculation can be seen in Figure 12. With upper and lower bounds of 500.0 and -500.0. Parameters used by Peer et al in the PSO can be seen in Figure 13. A population of 40 individuals was used in the GA for comparison against the PSO. The GA population was double the swarm size to take into account the particles of the swarm remembering their optimal solution attained, along with the current solution. The average results were taken from 100 runs for the GA and PSO. Peer et al implemented 6 variations of the PSO, with results being seen in Figure 14.

$$f(\mathbf{x}) = 418.9829d - \sum_{i=1}^{d} x_i \sin(\sqrt{|x_i|})$$

Figure 12 (Simon Fraser University, 2020)

Parameter	Value
#particles	20
$c_1 = c_2$	1.49618
w	0.729844
$s_c = f_c$	5
k (*lbest*)	2

Figure 13 (Peer et al, 2003)

	$\bar{x} \pm s$	Median	Range
G_g	4539 ± 706.06	4550.2	[2882.2:6534.1]
G_l	4762.1 ± 509.36	4797.8	[3395.4:5863.1]
G_v	4496.9 ± 707.83	4451.5	[2862.3:6356.5]
P_g	4535.6 ± 722.33	4510.8	[2803.1:6179.5]
P_l	4634 ± 642.22	4609.5	[2329.4:6219.5]
P_v	4273.4 ± 565.86	4333.1	[2664.9:5449.1]

Figure 14 (Peer et al, 2003)

Results in Figure 15 show the GA outperformed 5 out of 6 PSO algorithms implemented by Peer et al. The minimum value was 2463.0. A more optimal result was able to be obtained by the GA implemented with single-point crossover, tournament selection and random mutation. The PSO algorithm P1 had a minimum value of 2329.4, a marginally better optima value than the GA's output. Future studies other fitness functions should be carried out, to see if there is a significant difference in solutions outputted between the two algorithms.

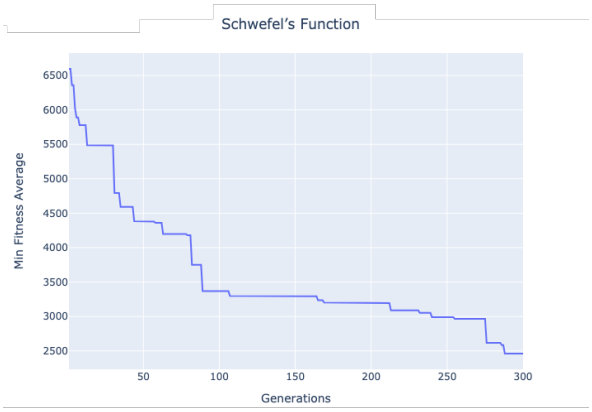

Figure 15, Best solutions found for minimisation by the GA when solving for Schwefel's function.

4 CONCLUSIONS

This report has explored the parameters that affect the solutions produced by GAs. These GAs can be applied successfully to real-life optimisation problems. However, ethics must be considered, no matter the benefits they result in.

Mutation (methods and rates) is the parameter with the greatest effect on GAs performance. Mutation provides diversity within the population, stopping the algorithm prematurely converge towards a local optima. Gaussian mutation was the method which performed the best when finding the global optima.

If this study was repeated again other mutation methods should be explored, along with changing the mutation step and further exploration of larger chromosome lengths. To be able to see how the GA is able to perform with these parameter changes and the output the GA produces. C should be used instead of Python next time. Comparisons of programming languages for bioinformatics showed that C had better performance, suited to solving complex problems like optimisation (Fourment et Gillings, 2008). This would increase the performance of the GA, especially when exploring larger values of the chromosome length.

Comparison between GA and PSO showed that the GA produced more optimal results against 5 out of 6 PSO algorithms when solving for Schwefel's function. Other fitness functions should be implemented, to see if there are other significant comparisons when solving optimisation problems.

REFERENCES

Adrian, A., Utamima, A. and Wang, K. (2015) A comparative study of GA, PSO and ACO for solving construction site layout optimization. *KSCE Journal of Civil Engineering* [online]. 19(3), pp. 520-527. [Accessed 13 December 2020].

Basak, S. (2018) *How to perform Roulette wheel and Rank based selection in a genetic algorithm?* Available from: https://medium.com/@setu677/how-to-perform-roulette-wheel-and-rank-based-selection-in-a-genetic-algorithm-d0829a37a189 [Accessed 09 December 2020].

Bloomberg. (2018) *The Great Firewall of China*. Available from: https://www.bloomberg.com/quicktake/great-firewall-of-china [Accessed 27 November 2020].

Bock, K., Hughey, G., Qiang, X. and Levin, D. (2019) Geneva: Evolving censorship evasion strategies. *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security* [online]. pp. 2199-2214. [Accessed 28 November 2020].

Camara, D. (2015) Evolution and Evolutionary Algorithms. *Bio-inspired Networking* [online]. pp. 1-30. [Accessed 09 December 2020].

Coghlan, S. and Leins, K. (2020) "Living Robots": Ethical Questions About Xenobots. *The American Journal of Bioethics* [online]. 20(5), pp. 1-3. [Accessed 30 November 2020].

Desmondm, K. and MacKenzie, C. (2020) *Why learn Python? 6 reasons why it's so hot right now*. Available from: https://codingnomads.co/blog/business/why-learn-python/ [Accessed 08 December 2020].

D'jaen, M. (2007) *Breaching the Great Firewall of China: Congress Overreaches in Attacking Chinese Internet Censorship*. Seattle [online]. 31, pp. 327. [Accessed 29 November 2020].

EvoFit (2020). *How It Works?* Available from: https://evofit.co.uk/how-it-works/ [Accessed 28 November 2020].

Fourment, M. and Gillings, R. (2008) A comparison of common programming languages used in bioinformatics. *BMC bioinformatics* [online]. 9(1), p. 82. [Accessed 13 December 2020].

Frowd, C. and Hancock, P. (2008) *Evolving human faces. In The Art of Artificial Evolution*. Springer, Berlin, Heidelberg [online]. pp. 189-210. [Accessed 27 Novemeber 2020].

Frowd, C., Hancock, P. and Carson, D. (2004) EvoFIT: A holistic, evolutionary facial imaging technique for creating composites. *ACM Transactions on applied perception (TAP)*, 1(1), pp. 19-39. [Accessed 28 November 2020].

Frowd, C., Portch, E., Killeen, A., Mullen, L., Martin, A. and Hancock, P. (2019) EvoFIT facial composite images: a detailed assessment of impact on forensic practitioners, police investigators, victims, witnesses, offenders and the media. *2019 Eighth International Conference on Emerging Security Technologies,* IEEE [online]. pp. 1-7. [Accessed 27 November 2020].

GeeksforGeeks. (2018) *Tournament Selection (GA)*. Available from: https://www.geeksforgeeks.org/tournament-selection-ga/ [Accessed 09 December 2020].

Heaven, W. (2020) *Predictive policing algorithms are racist. They need to be dismantled*. Available from: https://www.technologyreview.com/2020/07/17/1005396/predictive-policing-algorithms-racist-dismantled-machine-learning-bias-criminal-justice/. [Accessed 30 November 2020].

Indian Institute of Technology Madras (2020) *Optimization Methods*. Available from: https://mech.iitm.ac.in/nspch52.pdf [Accessed 27 November 2020].

Jatana, N. and Suri, B. (2020) Particle swarm and genetic algorithm applied to mutation testing for test data generation: a comparative evaluation. *Journal of King Saud University-Computer and Information Sciences* [online]. 32(4), pp. 514-521. [Accessed 09 December 2020].

Jebari, K. and Madiafi, M. (2013) Selection methods for genetic algorithms. *International Journal of Emerging Sciences* [online]. 3(4), pp. 333-344. [Accessed 09 December 2020].

Kelly, J. and Winterman, D. (2010) *The problem with e-fits*. BBC. [Accessed 28 November 2020].

Kim, I. and De Weck, O. (2005) Variable chromosome length genetic algorithm for progressive refinement in topology optimization. *Structural and Multidisciplinary Optimization* [online]. 29(6), pp. 445-456. [Accessed 07 January 2021].

Kim, S.W. and Douai, A. (2012) Google vs. China's "Great Firewall": Ethical implications for free speech and sovereignty. *Technology in Society* [online]. 34(2), pp.174-181. [Accessed 29 November].

Kriegman, S., Blackiston, D., Levin, M. and Bongard, J. (2020) A scalable pipeline for designing reconfigurable organisms. *Proceedings of the National Academy of Sciences* [online]. 117(4), pp. 1853-1859. [Accessed 30 November 2020].

Kumar, R. (2012) Blending Roulette Wheel Selection & Rank Selection in Genetic Algorithms. *International Journal of Machine Learning and Computing* [online]. 2(4), pp. 365-370. [Accessed 09 December 2020].

Peer, E, Van Den Bergh, F. and Engelbrecht, A. (2003) Using neighbourhoods with the guaranteed convergence PSO. *Proceedings of the 2003 IEEE Swarm Intelligence Symposium* [online]. pp. 235-242. [Accessed 13 December 2020].

Prokopec, A. and Golub, M. (2009) Adaptive mutation operator cycling. *Second International Conference on the Applications of Digital Information and Web Technologies* [online]. pp. 634-639. [Accessed 07 January 2021].

Plotly. (2020) Plotly Express in Python. Available from: https://plotly.com/python/plotly-express/ [Accessed 08 December 2020].

Rhodes, G. and Haxby, J. (2011). *Oxford handbook of face perception*. Oxford University Press.

Rocha, M. and Neves, J. (1999) Preventing premature convergence to local optima in genetic algorithms via random offspring generation. *International Conference on Industrial, Engineering and Other Applications of Applied Intelligent Systems* [online]. pp. 127-136. [Accessed 07 January 2021].

Sample, I. (2020) *Scientists use stem cells from frogs to build first living robots*. Available from: https://www.theguardian.com/science/2020/jan/13/scientists-use-stem-cells-from-frogs-to-build-first-living-robots. [Accessed 29 November 2020].

ScienceDaily. (2020) *Living robots built using frog cells*. Available from: https://www.sciencedaily.com/releases/2020/01/200113175653.htm. [Accessed 29 November 2020].

SciTechDaily. (2019) *New Artificial Intelligence Genetic Algorithm Automatically Evolves to Evade Internet Censorship*. [Accessed 28 November 2020].

Simon Fraser University. (2020) Optimization Test Problems. Schwefel Function. Available from: https://www.sfu.ca/~ssurjano/schwef.html. [Accessed 13 December 2020].

Sivanandam, S. and Deepa, S. (2008) Genetic algorithm optimization problems. *Introduction to genetic algorithms* [online]. pp. 165-209. [Accessed 13 December 2020].

Stevenson, C. (2007) Breaching the great firewall: China's internet censorship and the quest for freedom of expression in a connected world. *BC Int'l & Comp*. 30 [online]. pp. 531. [Accessed 29 November 2020].

Tang, W. and Wu, Q. (2011) Evolutionary Computation. *Condition Monitoring and Assessment of Power Transformers Using Computational Intelligence* [online] pp. 15-36. [Accessed 27 November 2020].

Tutorialspoint. (2020) Genetic Algorithms – Crossover. Available from: https://www.tutorialspoint.com/genetic_algorithms/genetic_algorithms_crossover.htm [Accessed 09 December 2020].

Tutorialspoint. (2020) Genetic Algorithms – Mutation. Available from: https://www.tutorialspoint.com/genetic_algorithms/genetic_algorithms_mutation.htm [Accessed 09 December 2020].

Tutorialspoint. (2020) Genetic Algorithms – Parent Selection. Available from: https://www.tutorialspoint.com/genetic_algorithms/genetic_algorithms_parent_selection.htm [Accessed 09 December 2020].

United Nations. (2020) *Freedom of Expression, a Fundamental Human Right*. Available from: https://www.un.org/en/chronicle/article/freedom-expression-fundamental-human-right. [Accessed 29 November 2020].

University of Malaga. (2020) *Gaussian Mutation*. Available from: https://neo.lcc.uma.es/cEA-web/GMut.htm [Accessed 09 December 2020].

Wang, Z., McCarthy, T. and Sheikh, M. (2012) A comparison of genetic algorithm and particle swarm optimisation for theoretical and structural applications. *Proceedings of the Eleventh International Conference on Computational Structures Technology* [online]. pp. 1-18. [Accessed 09 December 2020].

Wang, J., Zhang, M., Ersoy, O., Sun, K. and Bi, Y. (2019) An Improved Real-Coded Genetic Algorithm Using the Heuristical Normal Distribution and Direction-Based Crossover. *Computational Intelligence and Neuroscience* [online]. [Accessed 13 December 2020].

Weile, D and Michielssen, E (1997). Genetic algorithm optimization applied to electromagnetics: A review. *IEEE Transactions on Antennas and Propagation* [online]. 45(3), pp. 343-353. [Accessed 07 January 2021].

WhatIs. (2020). *Great Firewall of China*. Available from: https://whatis.techtarget.com/definition/Great-Firewall-of-China [Accessed 28 November 2020].

Yang, X. (2014) Genetic Algorithms. *Nature-Inspired Optimization Algorithms* [online]. pp. 77-87. [Accessed 08 December 2020].

Source code as an appendix

```
1.  import random
2.  import math
3.  import plotly.express as px
4.  import pandas as pd
5.  from numpy import random as rand
6.
7.  # Global Variables
8.  N = 20 # Chromosome length
9.  P = 50 # Population
10. GENERATIONS = 300
11. LOWER_BOUND = -32.0
12. UPPER_BOUND = 32.0
13. ITERATIONS = 10
14. SWAP_AMOUNT = int(P / 5)  # How many individuals are replaced with better solutions from th
    e offspring
15. MUTATION_RATE = round(((1/P) + (1/N)) / 2, 2) # Mutation rate is between 1/population and 1
    /chromosome length
16. FITNESS_FUNCTION = "minimisationFitnessFunction2"
17.
18. population, offspring = [], []
19. meanList, generationArr, bestArr, meanArr, minArr, minArrAverage, minAverageCalculated = []
    , [], [], [], [], [], []
20.
21. # Setting the min average array values to 0 so that summation is able to take place
22. for i in range(0, GENERATIONS):
23.     minArrAverage.append(0)
24.
25. class individual:
26.     gene = []
27.     fitness = 0
28.
29. # Outputting a graph to display the average minimum value across the iterations of the GA
30. def setGraphMinimumAverage():
31.     d = {'Min Fitness Average': minAverageCalculated, 'Generations': generationArr }
32.     df = pd.DataFrame(data=d)
33.     fig = px.line(df, x='Generations', y='Min Fitness Average', title="GA")
34.     fig.show()
35.
36. # Outputting a graph to visibly see the change in the fitness of the population after selec
    tion, crossover and mutation
37. def setGraph():
38.     d = {'Min Fitness': minArr, 'Mean Fitness': meanArr, 'Generations': generationArr } # T
    his graph displays mean and minimum function
39.     df = pd.DataFrame(data=d)
40.     fig = px.line(df, x='Generations', y=['Min Fitness', 'Mean Fitness'], title="GA")
41.     fig.show()
42.
43. # Setting and outputting the end fitness after tournament selection, crossover and mutation

44. def calcFitnesses(currentGeneration):
45.     endFitness, meanFitness = 0, 0
46.     minFitness = 10000
47.     # Calculate the minimum fitness by looping through the individuals in the population
48.     for i in population:
49.         endFitness = round(endFitness + i.fitness, 1)
50.         # Work out best fitness of the population (lowest fitness value)
51.         if (i.fitness < minFitness):
52.             minFitness = round(i.fitness,1)
```

```
53.
54.     meanFitness = round(endFitness / P, 1) # Rounded mean fitness to nearest integer
55.     meanList.append(meanFitness)
56.     generationArr.append(currentGeneration)
57.     minArr.append(minFitness)
58.     meanArr.append(meanFitness)
59.
60.     # Output the minimum fitness value and mean fitness value
61.     print("Min fitness is: ", minFitness)
62.     print("Mean fitness is: ", meanFitness)
63.
64. # Setting individual genes to random values between the lower and upper bound
65. def setPopulation():
66.     for x in range (0, P):
67.         tempgene=[]
68.         for x in range (0, N):
69.             tempgene.append(round(random.uniform(LOWER_BOUND, UPPER_BOUND), 1))
70.
71.         newIndividual = individual()
72.         newIndividual.gene = tempgene[:]
73.         population.append(newIndividual)
74.
75.     # Setting fitness value in population
76.     for i in population:
77.         i.fitness = fitnessFunction(i.gene)
78.
79. # Depending on which fitness function the user has chosen carry out that fitness calculatio
    n
80. def fitnessFunction(ind):
81.     if FITNESS_FUNCTION == "minimisationFitnessFunction":
82.         fitness = minimisationFitnessFunction(ind)
83.         return fitness
84.     elif FITNESS_FUNCTION == "minimisationFitnessFunction2":
85.         fitness = minimisationFitnessFunction2(ind)
86.         return fitness
87.     elif FITNESS_FUNCTION == "schwefelFitnessFunction":
88.         fitness = schwefelFitnessFunction(ind)
89.         return fitness
90.
91. # Setting the fitness of an individual with a minimisation function. Lower and upper bounds
     are -5.12 and 5.12 respectively
92. def minimisationFitnessFunction(ind):
93.     fitness = N*len(ind)
94.     for i in range(0, len(ind)):
95.         fitness = fitness + (ind[i] * ind[i] - 10*math.cos(2*math.pi*ind[i]))
96.     return fitness
97.
98. # Setting the fitness of an individual with a minimisation function. Lower and upper bounds
     are -32.0 and 32.0 respectively
99. def minimisationFitnessFunction2(ind):
100.            fitnessSumFirst = 0
101.            fitnessSumSecond = 0
102.            for i in range(0, len(ind)):
103.                fitnessSumFirst += (ind[i] ** 2)
104.                fitnessSumSecond +=  math.cos(2*math.pi*ind[i])
105.            fitness = -20 * math.exp(-0.2 * math.sqrt((1/N) * fitnessSumFirst)) -
     math.exp((1/N) * fitnessSumSecond)
106.
107.            return fitness
108.
```

```
109.        # Fitness function calculation for Schwefel's Function. Lower and upper bounds are
    -500.0 and 500.0 respectively
110.        def schwefelFitnessFunction(ind):
111.            fitnessSum = 0
112.            for i in range(0, len(ind)):
113.                fitnessSum += ind[i] * math.sin(math.sqrt(math.fabs(ind[i])))
114.            fitness = (418.9829 * N) - fitnessSum
115.
116.            return fitness
117.
118.        # Check if the fitness of the worst index in the original population is less than t
    he fitness of the best index in the new population and swap the gene
119.        def replaceChromosomes(lowestIndex, largestIndex, tempOffspring):
120.            if population[largestIndex].fitness > fitnessFunction(tempOffspring[lowestIndex
    ]):
121.                population[largestIndex].gene = tempOffspring[lowestIndex]
122.                population[largestIndex].fitness = fitnessFunction(tempOffspring[lowestInde
    x])
123.
124.        # Find the worst solution in the population (largest fitness value) and return the
    index of this solution
125.        def largestFitnessSolution():
126.            largest = -1000
127.            largestIndex = 0
128.            for i in range(0, P):
129.                tempFitness = fitnessFunction(population[i].gene)
130.                if tempFitness > largest:
131.                    largest = tempFitness
132.                    largestIndex = i
133.
134.            return largestIndex
135.
136.        # Find the best solution in the new offspring population (lowest fitness value) and
     return the index of this solution
137.        def lowestFitnessSolution(tempOffspring):
138.            lowest = 1000
139.            lowestIndex = 0
140.            for i in range(0, P):
141.                if (fitnessFunction(tempOffspring[i]) < lowest):
142.                    lowest = fitnessFunction(tempOffspring[i])
143.                    lowestIndex = i
144.
145.            return lowestIndex
146.
147.        # Using tournament selection to set the offspring population
148.        def tournamentSelection(population):
149.            for i in range(0, P):
150.                parent1 = random.randint(0, P-1)
151.                off1 = population[parent1]
152.                parent2 = random.randint(0, P-1)
153.                off2 = population[parent2]
154.                if off1.fitness < off2.fitness:
155.                    offspring.append(off1)
156.                else:
157.                    offspring.append(off2)
158.
159.        # Using roulette-Wheel selection to set the offspring population
160.        def rouletteWheelSelection(population):
161.            # Calculating the total fitness of the population
162.            totalFitnessPopulation = 0
```

```
163.            for i in population:
164.                totalFitnessPopulation += i.fitness
165.
166.            for i in range(0, P):
167.                # Select a random point from 0 to the total fitness value of the original p
    opulation
168.                selectionPoint = random.randint(math.floor(totalFitnessPopulation), 0)
169.                runningTotal = math.floor(totalFitnessPopulation)
170.                j = 0
171.                # While the running total is not less than the selection point append the f
    itness of value of an individual in the population to the running total
172.                while (runningTotal >= selectionPoint) and (j < P):
173.                    runningTotal -= population[j].fitness
174.                    j = j + 1
175.                # When the running total is less than the selection point, append the last
    individual from the population which fitness what added to the running total
176.                offspring.append(population[j - 1])
177.
178.        # Using rank selection to set the offspring population
179.        def rankSelection(population):
180.            # Sort the individuals in the population in accessinding order based on the fit
    ness value of the individuals
181.            for i in range(0, P):
182.                    for j in range (0, P - i - 1):
183.                        # Swap the individuals in the population position based on if the f
    itness is greater than another
184.                        if (population[j].fitness > population[j+1].fitness):
185.                            temp = population[j]
186.                            population[j] = population[j+1]
187.                            population[j+1] = temp
188.
189.            # Give a ranking from 0 to the size of the population to the individuals
190.            rankSum = 0
191.            for i in range(0, P):
192.                # Setting the rank
193.                population[i].rank = P - i
194.                # Append to the rank sum value
195.                rankSum += population[i].rank
196.
197.            for i in range(0, P):
198.                # Setting the selection point based on a random integer between 0 and the s
    um of the ranked population
199.                selectionPoint = random.randint(0, rankSum)
200.                runningTotal = 0
201.                j = 0
202.                # While the running total is not greater than the selection point append th
    e ranking of value of an individual in the population to the running total
203.                while runningTotal <= selectionPoint and (j < P):
204.                    runningTotal += population[j].rank
205.                    j = j + 1
206.                # When the running total is greater than the selection point, append the la
    st individual from the population which fitness what added to the running total
207.                offspring.append(population[j - 1])
208.
209.        # Single point crossover
210.        def singlePointCrossover(tempOffspring):
211.            # Iterate in 2 for pairs
212.            for i in range(0, P, 2):
213.                # Carry out crossover from a random point from the second position in the c
    hromosome (array index 1)
```

```
214.                crossoverPoint = random.randint(1, N-1)
215.                # Setting the children equal to the original gene in the array before the c
    rossover plus the alternative crossover
216.                tempA = offspring[i].gene[:crossoverPoint] + offspring[i+1].gene[crossoverP
    oint:]
217.                tempB = offspring[i+1].gene[:crossoverPoint] + offspring[i].gene[crossoverP
    oint:]
218.                # Append the new solutions to the new array
219.                tempOffspring.append(tempA)
220.                tempOffspring.append(tempB)
221.
222.        # Multi Point Crossover
223.        def multiPointCrossover(tempOffspring):
224.            # Finding the two crossover points
225.            crossoverPoint1 = 0
226.            crossoverPoint2 = 0
227.            # If N mod 3 returns 0 then split the chromosome into three equal parts, using
    two crossover points
228.            if N % 3 == 0:
229.                crossoverPoint1 = (N / 3)
230.                crossoverPoint2 = (N / 3) * 2
231.            # If the chromosome does not split into three equal parts then work out where t
    o put the crossover points
232.            else:
233.                crossoverPoint1 = round(N / 3)
234.                crossoverPoint2 = round(N / 3) * 2
235.
236.            # Iterate in 2 for pairs
237.            for i in range(0, P, 2):
238.                # Carry out crossover for two crossover points (multi-point crossover)
239.                tempA = offspring[i].gene[:crossoverPoint1] + offspring[i+1].gene[crossover
    Point1:crossoverPoint2] + offspring[i].gene[crossoverPoint2:]
240.                tempB = offspring[i+1].gene[:crossoverPoint1] + offspring[i].gene[crossover
    Point1:crossoverPoint2] + offspring[i+1].gene[crossoverPoint2:]
241.                # Append the new solutions to the new array
242.                tempOffspring.append(tempA)
243.                tempOffspring.append(tempB)
244.
245.        # Uniform Crossover
246.        def uniformCrossover(tempOffspring):
247.            # Iterate in 2 for pairs
248.            for i in range(0, P, 2):
249.                tempA = []
250.                tempB = []
251.                # Flip a coin (random integer of 0 or 1) to decide if each chromosome will
    be included in the off-spring (crossed over)
252.                # for j in  range(0, len(offspring[i].gene)):
253.                for j in  range(0, N):
254.                    # Coin flip - random integer of 0 or 1 is produced
255.                    if random.randint(0, 1) == 0:
256.                        tempA.append(offspring[i+1].gene[j])
257.                        tempB.append(offspring[i].gene[j])
258.                    else:
259.                        tempA.append(offspring[i].gene[j])
260.                        tempB.append(offspring[i+1].gene[j])
261.                tempOffspring.append(tempA)
262.                tempOffspring.append(tempB)
263.
264.        # Random mutation within a range of bounds
265.        def randomMutation(tempOffspring):
```

```
            for i in range(0, P):
                for j in range(0, N):
                    mutationProbability = random.randint(0,100) # Randomly generate a numbe
    r between 0 and 100
                    # If the number generated is less than the mutation rate * 100 then fli
    p the gene in the chromosome
                    if mutationProbability < (100 * MUTATION_RATE):
                        # Carry out mutation of randomly adding or minusing a number in ran
    ge from 0.0 to the mutation step
                        addOrMinus = random.randint(0,1) # Set variable to randomly select
    minus or plus
                        # Create a random integer between 0 and the upper bound for mutatio
    n step, then alter the genes value by a random integer between 0.0 and the mutation step
                        mutationStep = round(random.uniform(0.0, UPPER_BOUND),1)
                        alter = round(random.uniform(0.0, mutationStep),1)

                        # If variable equals 0 then minus a random integer in range 0.0 to
    the mutation step
                        if (addOrMinus == 0):
                            if ((tempOffspring[i][j] - alter) >= LOWER_BOUND):
                                tempOffspring[i][j] = round((tempOffspring[i][j] -
     alter), 1)
                            # If the value goes below the lower bound after the minus then
    set to the lower bound as the minimum value it can be
                            else:
                                tempOffspring[i][j] = LOWER_BOUND
                        # If variable does not equal 0 then plus a random integer in range
    0.0 to the mutation step
                        else:
                            if ((tempOffspring[i][j] + alter) <= UPPER_BOUND):
                                tempOffspring[i][j] = round((tempOffspring[i][j] + alter),
    1)
                            # If the value goes above 1.0 after the addition then set to th
    e upper bound as the maximum value it can be
                            else:
                                tempOffspring[i][j] = UPPER_BOUND

        # Gaussian mutation, mutation within a range of a normal distribution
        def gaussianMutation(tempOffspring):
            # Carry out mutation on every individual in population
            for i in range(0, P):
                for j in range(0, N):
                    mutationProbability = random.randint(0,100)
                    if mutationProbability < (100 * MUTATION_RATE):
                        # Loc indicates the center of the distribution and Scale indicates
    the spread of the distribution
                        alter = round(float(rand.normal(loc=0, scale=5, size=(1))),1)
                        if ((tempOffspring[i][j] + alter) >= LOWER_BOUND) and ((tempOffspri
    ng[i][j] + alter) <= UPPER_BOUND):
                            tempOffspring[i][j] = tempOffspring[i][j] + alter
                        elif ((tempOffspring[i][j] + alter) < LOWER_BOUND):
                            tempOffspring[i][j] = LOWER_BOUND
                        elif ((tempOffspring[i][j] + alter) > UPPER_BOUND):
                            tempOffspring[i][j] = UPPER_BOUND

        # Creating a random end point for scrambled mutation
        def calculateEndPoint(startPoint):
            endPoint = random.randint(startPoint, N)
            # Checking that not the whole gene is scrambled
            if endPoint == N:
```

```
313.                 calculateEndPoint(startPoint)
314.             return endPoint
315.
316.         # Scrambled Mutation
317.         def scrambleMutation(tempOffspring):
318.             # Carry out mutation on every individual in population
319.             for i in range(0, P):
320.                 mutationProbability = random.randint(0,100)
321.                 if mutationProbability < (100 * MUTATION_RATE):
322.                     # Create a starting and end point of where the scrambled mutation on th
    e individuals should take place
323.                     startingPoint = random.randint(0, N-
    1) # Making sure that more than one gene is mutated
324.                     endPoint = calculateEndPoint(startingPoint)
325.                     # Shuffle the genes in the chromosome between the start and end point
326.                     shuffledArray = []
327.                     for j in range(startingPoint, endPoint):
328.                         shuffledArray.append(tempOffspring[i][j])
329.                     rand.shuffle(shuffledArray)
330.                     # Put the scrambled array back into the individual
331.                     for x in range(startingPoint, endPoint):
332.                         for y in range(0, len(shuffledArray)):
333.                             tempOffspring[i][x] = shuffledArray[y]
334.
335.         # Clear the array values so that the GA is able to iterate again
336.         def clearArrays():
337.             meanList.clear()
338.             generationArr.clear()
339.             bestArr.clear()
340.             meanArr.clear()
341.             minArr.clear()
342.             population.clear()
343.             offspring.clear()
344.
345.         # Main function to start code from
346.         def main():
347.             # Carry out iterations of the GA so that we can plot the average of the results

348.             for x in range(0,ITERATIONS):
349.                 setPopulation() # Setting population of individuals
350.                 # Carry out crossover and mutation for as many generations set, this is the
     termination condition of the algorithm
351.                 for i in range(1, GENERATIONS + 1):
352.                     print("\nGeneration ", i)
353.                     tempOffspring = []
354.
355.                     # Selection
356.                     tournamentSelection(population)
357.                     # Crossover
358.                     singlePointCrossover(tempOffspring)
359.                     # Mutation
360.                     randomMutation(tempOffspring)
361.
362.                     # This range determines how many solutions are selected and swapped per
     run
363.                     for j in range(0, SWAP_AMOUNT):
364.                         largestIndex = largestFitnessSolution() # Finding worst solution in
     the population (individual with the largest fitness value)
365.                         lowestIndex = lowestFitnessSolution(tempOffspring) # Finding best s
    olution in the temporary offspring (lowest fitness value)
```

```
366.                         replaceChromosomes(lowestIndex, largestIndex, tempOffspring) # Set
    the worst case of the original population to equal the best case of the temp offspring popu
    lation
367.
368.                     # Calculate and print the fitness of the population after selection, mu
    tation and crossover
369.                     calcFitnesses(i)
370.
371.                 # setGraph() # Setting individual iteration graph using plotly
372.
373.                 for i in range(0, GENERATIONS):
374.                     minArrAverage[i] += minArr[i]
375.
376.                 clearArrays() # Re set the values of the arrays so that other iterations of
     the GA can occur
377.
378.             # Appending the average minimum fitness results from the GA run
379.             for i in range(0, len(minArrAverage)):
380.                 minAverageCalculated.append(round(float(minArrAverage[i] / ITERATIONS),1))

381.                 generationArr.append(i+1) # Set the generation array equal to how many gene
    rations occur per run
382.
383.             setGraphMinimumAverage() # Output the average results on the graph
384.
385.         if __name__ == "__main__":
386.             main()
```