Expensive Java

“To what extent can Machine Learning decrease the expensive nature of Genetic Algorithms when solving atypical optimization problems?”

Subject: Computer Science

Word Count: 3,942

May 2021

Table of Contents

Introduction

“The world runs on coffee”, although an exaggeration of the world’s dependency on the caffeinated beverage, data from different countries’ consumption of coffee would suggest an ever increasing demand for this drink; but does the coffee we drink truly taste good? This paper attempts to examine whether Genetic Algorithms, a subset of optimization algorithms known as global search algorithms, can be used to optimize qualitative and objective features such as taste? Moreover due to the nature of Genetic Algorithms (GA), large quantities of data are required to mimic the same evolutionary dynamics that are seen in nature [1]. The GA used in this paper attempts to utilize Machine Learning in the form of polynomial regression to mitigate the ‘expensiveness’ of this algorithm. Thus this paper asks the question, “To what extent can Machine Learning decrease the expensive nature of Genetic Algorithms when solving atypical optimization problems?”. This paper provides a brief overview of the theory surrounding Genetic Algorithms, and contextualizes the Genetic Algorithm in relation to “The Coffee Problem”.

Genetic Algorithms are most notably applied in scenarios where the fitness of an individual’s strength can be described through mathematical formulas. This paper strives to investigate the extent to which Genetic Algorithms can be applied to solve atypical optimization scenarios where the fitness of an individual can not be determined through mathematical means. Furthermore, the paper seeks to provide evidence in support of or against the versatility of Genetic Algorithms and evaluate its effectiveness in reaching the optimized solution while considering time constraints and efficiency.

As has been stated, this paper will focus on optimizing the desirability of a specific coffee recipe. More importantly, the scope of the experiment has been limited to the use of the following three ingredients: Water, Coffee, and Sugar; with the specific quantity of each substance being the controlled parameters. Furthermore, due to the underlying conditions which came as an effect of COVID-19, large scale experimentation could not occur and as a result under the suggestion of my mentor, these tests were done in small data sets. Additionally, while the results demonstrated by the Genetic Algorithm are demonstrative of its ability to solve complex problems, there are limits that present themselves which must be discussed.

Genetic Algorithms: A competition for survival

There is a beauty in the way that nature handles failure. Darwin's model of evolution known as natural selection suggests that adaptation is fueled by the success *and* failure of specific traits in an environment. Natural Selection as proposed by Darwin, brought forth an idea that individuals in a population evolve as a result of the demands of their environment. Darwin observed that finches whose diet was composed of small nuts and seeds had evolved in such a way that their beaks could eat those specific foods with much more ease. Naturally one must ask, what fuels evolution? Is it the failure of those finches who didn't possess smaller, more agile beaks? Or rather, the success of the finches who did? Understanding the fundamental relation between the individual and the environment is paramount to creating an efficient GA (Genetic Algorithm) that takes into account the success and failures of each solution. As stated by [1], "In most cases… genetic algorithms are nothing less than the probabilistic optimization methods which are based on the principles of evolution".

The core of a Genetic Algorithm, lies in its ability to manipulate an individual's genome. A genome encodes the parameters and or variables that make the solution unique, commonly genomes are also referred to as a combination of phenotypes and genotypes, but their purpose and effect on the GA are the same: they allow the GA to interact with problem specific attributes [2]. In a population, each individual follows the same blueprint for it's genome, but the specific values that are assigned to each gene are unique to the individual [1]. In the case of Darwin's finch, the genome blueprint is what made the finch a finch, but the specific genes in the genome is what creates finches with different sized beaks, different claws, etc. The GA then applies a process known as reproduction, where two genomes combine to make a child. The result of the reproduction process, the child, allows for the inheritance of genes from its two parents which provides a way for the algorithm to explore new data points in the search space [3]. Thus, by favoring individuals with a higher fitness for the parents, the algorithm has a higher probability to pass on the genetic material that makes the solution 'good'.

Reproduction, also commonly referred to as crossover, can be done through various ways such as through Uniform Crossover, One Point Crossover, Multi Point Crossover, etc [1], [4]. The process of selecting a parent can be done in many ways such as Stochastic Universal Sampling, Fitness Proportionate Selection, Random Selection, etc. These different techniques tackle the issue of approaching the optimum solution while maintaining diversity. Diversity in a population is key as it has the effect of "preserving genetic diversity and, as an effect, that local maxima can be avoided" [1]. Unfortunately, this same convergence behavior can occur before it reaches the solution - this is known as premature convergence [5]. That is why GA's implement mutations to randomly incorporate diversity in populations [1], [5]. This can be done in many ways such as, Bit Flip Mutations, Random Resetting, Swap Mutations, etc.

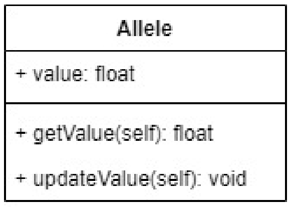
Each of the procedures described above allow the GA to mimic the way that species evolve over time, but what allows these methods to work is the concept of fitness. In Darwin's finches, their fitness was connected to how well they were able to feed themselves as a result of the shapes of their beaks. GA's must find a way to describe how strong a specific genome can perform in said environment. Oftentimes, this is done by creating a mathematical model that describes variable interactions. This allows for the GA to run 'unsupervised' and is in most cases the accepted method as it is both time and resource efficient. When this can't be done fitness is assigned in a supervised manner, leading to an increase in the time required to iterate through one generation of the GA. In optimization algorithms, when the time or resources required to compute fitness or cost is very high, these algorithms are known as expensive optimization problems. They are expensive due to the fact that they require more manpower or money to collect the same amount of data as an unsupervised method [4].

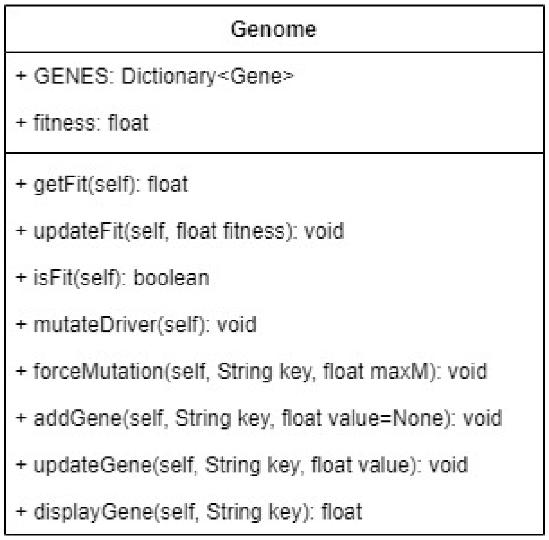
The Coffee Problem: Implementing the GA

The challenge in optimizing ‘taste’ is that there is no existing paradigm that describes how the different quantities of each ingredient interact with our taste buds to create a flavor that is enjoyable. This challenge is compounded by the individuality of our sense of ‘taste’ (i.e, one person may like their coffee without sugar and someone else may like it with a lot of sugar). The unpredictable relationship between how much of an ingredient there is in a recipe and its effect on taste means that finding the ‘ideal’ recipe would mean traversing a large search space. GA’s by design, are created to find the global optimum in large search spaces [1]. With that being said because of the challenge in mathematically predicting taste, each recipe needs to be tested individually in order to assign it a fitness. The GA used in this experiment is designed to actively search for the optimum and employs a form of multivariate polynomial regression to decrease the number of real trials needed to assign fitness. The following passages describe how this GA was designed to fit the problem.

GA Design: Genome

The Coffee Genome is composed of two variables qC and qS; which represent the quantity of Coffee and quantity of Sugar in grams respectively. qC and qS represent Genes, which have Alleles. The UML Diagram for each of the respective classes is provided below.




Genome
+ GENES: Dictionary<Gene>
+ fitness: float
+ getFit(self): float
+ updateFit(self, float fitness): void
+ isFit(self): boolean
+ mutateDriver(self): void
+ forceMutation(self, String key, float maxM): void
+ addGene(self, String key, float value=None): void
+ updateGene(self, String key, float value): void
+ displayGene(self, String key): float

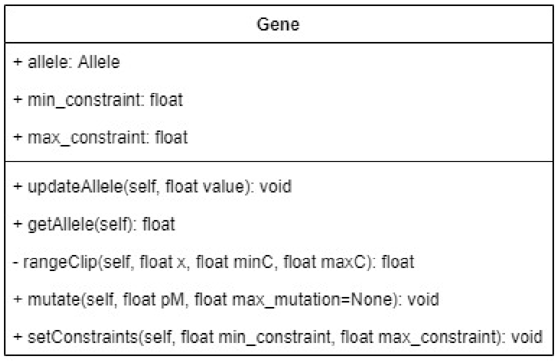

Gene
+ allele: Allele
+ min_constraint: float
+ max_constraint: float
+ updateAllele(self, float value): void
+ getAllele(self): float
- rangeClip(self, float x, float minC, float maxC): float
+ mutate(self, float pM, float max_mutation=None): void
+ setConstraints(self, float min_constraint, float max_constraint): void

Figure 1.A (UML of Allele), 1.B (UML of Genome), 1.C (UML of Gene)

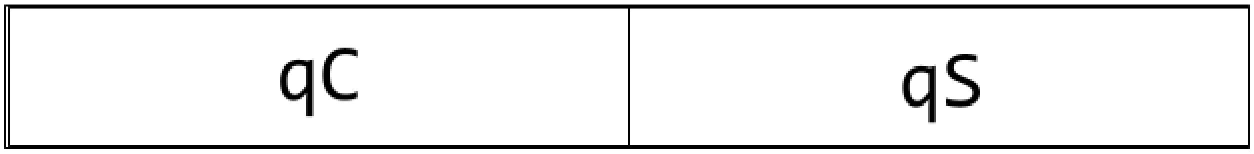
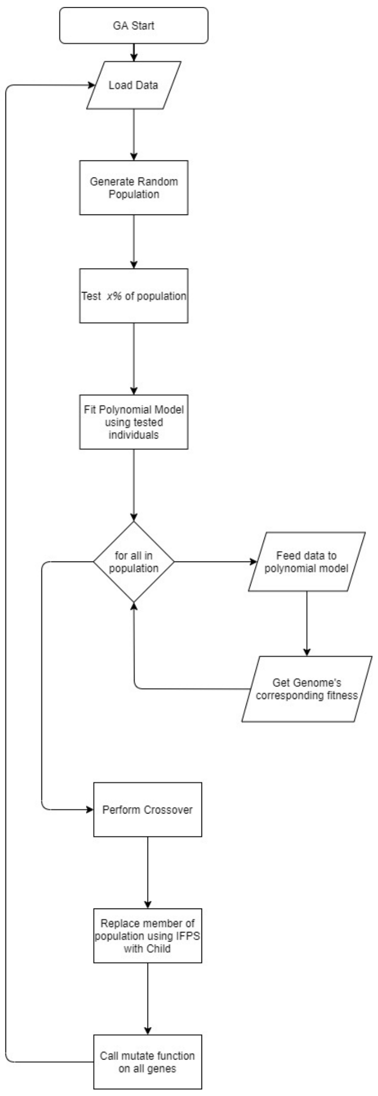


Figure 1.D (Visual Representation of Coffee Genome)

As is denoted by the UML Diagram, the individual Gene's handle the mutation while the Genome tells the gene when to mutate (as shown by the method mutateDriver in Genome class). Furthermore, the individual genes have an allele object that holds the 'value'. Although not necessary, the allele object is created in order to maintain organization in the code and to allow flexibility in the design of the GA. The last point of interest in the design of this Genome is the *GENES* dictionary. This dictionary maps a key to the individual genes. Because the Gene class itself has no identifier, the key in the dictionary is used to access the specific genes. In the case of the Coffee Genome, qC and qS are the keys which reference the respective Genes. This is done, once again, for flexibility. The GA is designed such that the code may be reused regardless of the specific variables needed for the genome.

GA Design: Population

The GA used in this paper initializes n random individuals in a steady state population. This is a type of population model that differs from a generational model, whose focus is on replacing every individual in a population. The steady state model works by creating a child and then swapping it for a member in the population. Below is a flowchart demonstrating the steps that the GA undergoes. The specific steps are discussed in detail thereafter.

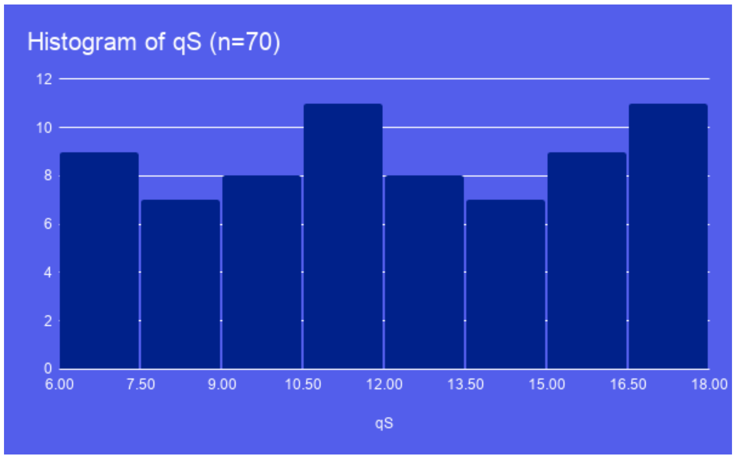

GA Start
Load Data
Generate Random Population
Test x% of population
Fit Polynomial Model using tested individuals
for all in population
Feed data to polynomial model
Get Genome's corresponding fitness
Perform Crossover
Replace member of population using IFPS with Child
Call mutate function on all genes

1) GA Start
2) Generate Random Population
3) Test a percentage of the population
4) Fit the polynomial model using the tested individuals
5) Assign a fitness to all members of the population:
    a) Feed qC and qS to their polynomial model
    b) Average the resulting fitness according to both models
6) Perform Crossover
7) Replace Members using IFPS
8) Apply mutations where valid
9) Repeat from step 3 until termination condition

Figure 1.D (GA Flowchart)

Population Initialization

Random Initialization is done by copying the rootGenome, and in a sense forcing an unconstrained mutation. The rootGeneome is provided by the experimenter as the blueprint for the rest of the population. By starting from an already existing coffee recipe, the GA is able to employ a heuristic approach to local search which maximizes computational cost. The graphs below demonstrate the frequency distribution created from the random initialization of a population.



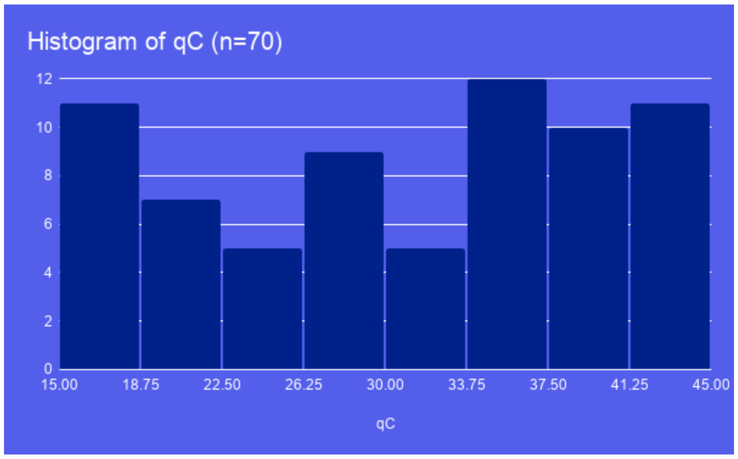


Figure 2.A qS Histogram (Right), 2.B qC Histogram (Left)

The histograms for the qC and qS values demonstrate a ragged plateau distribution which is indicative of no clear pattern in the dataset. This is ideal when creating the random initialization of the population to avoid any bias at the start of a GA run [150].

Crossover

When crossover is called, the GA selects two parents using Fitness Proportionate Selection. FPS (Fitness Proportionate Selection) works by calculating the fitness of each individual relative to the entire population [1]. This is done by dividing the fitness of the current individual in a population by the total fitness of the entire population:

$$P(n) = \frac{fit}{\sum_{i=1}^{N} n.getFit()}$$

Figure 2.C (Equation for FPS)

Where n represents an individual genome in the population P, fit represents the fitness of the current individual, and $n.getFit()$ references the method found in the genome that is used to return the fitness. The following is a sample representation of how FPS can be used to select the fittest individuals in a population. As is noted, 9 (light blue) and 7.4 (light red) are the individuals with the highest fitness and largest slice of pie.

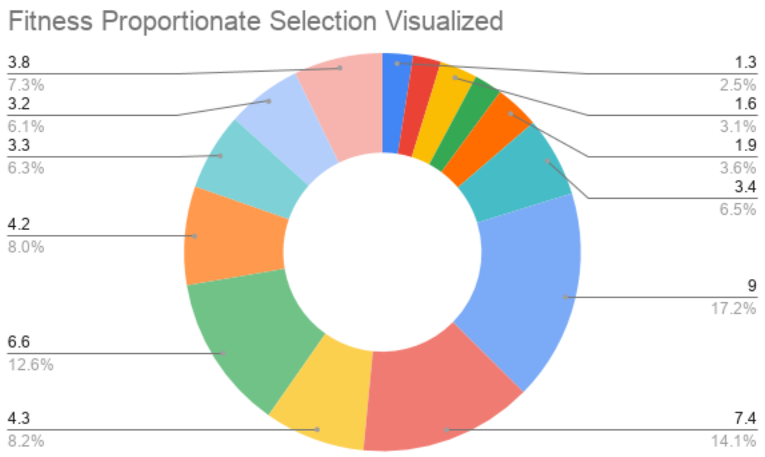


Figure 2.D (FPS Visualized)

Once the parents have been selected, the GA goes on to perform the actual crossover method. In the case for this GA, Uniform Crossover was employed to maximize the chances of each gene to be passed along to the generation [1]. Uniform Crossover gives each gene from each parent an equal chance to be passed onto the child. The GA essentially assigns each gene "heads" or "tails" (1 or 0). It then generates a random float using the inbuilt random.random() method in Python. When the random float is between 0.0 and 0.50, it considers it as tails. Likewise, when it is between 0.50 and 1.0, it considers it as heads. The chosen gene is then passed on to the child, and this process iterates through each gene in the genome until a child is created. The crossover operation, as stated by [1], is the reason why asexual reproduction, whose diversity is dependent solely on mutation, fail to adapt at the same rate as sexually reproducing species.

Replacement

Upon creating the child because the GA is modeled after a steady state algorithm the number of individuals in the population must stay the same. As a result, the child can not be put into the population, it must replace another genome. There are different routes that can be chosen, but after doing some trial runs, it was experimentally determined that Inverse Fitness Selection often led to earlier convergence of the population on the optimum. IFPS (Inverse Fitness Proportionate Selection) involves a similar process as standard FPS, but instead it continues the process by raising the relative fitness to the power of -1, and then dividing it by the sum of all inverse fitnesses. IFPS favors those individuals with a lower relative fitness, and thus allows the algorithm to replace those individuals whose performance is seen as less than optimal. Furthermore, a generation gap is provided to control the amount of the population that will be replaced. The closer the generation gap is to 100%, the more similar it becomes to a standard generational population model and less of a steady state model.

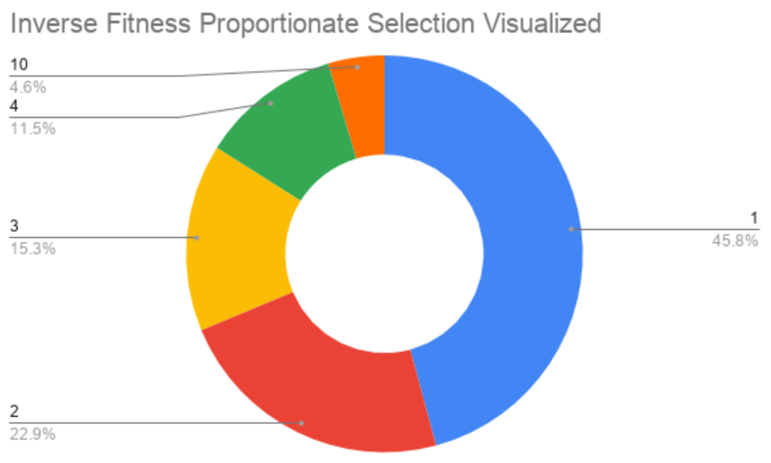


Figure 2.E (IFPS Visualized)

GA Design: Fitness Calculation

GA's require a lot of data, and due to the complexity and challenges of mathematically modeling the interactions between ingredients, solving "The Coffee Problem" ideally would be done under supervised conditions where experimenters rate the coffee recipe provided by the computer. The problem is thus considered an expensive optimization problem, due to the time and resources needed to perform all the tests. The solution proposed in this paper is then to select a small percentage of the population to perform supervised testing. The data is then fed to a 4 degree multivariate polynomial regression model. Once the model is trained, the model is used to fit the entire population.

How Fitness is Calculated

Originally, the fitness was going to be calculated through supervised large scale testing by providing small samples of the coffee drinks to students at my school and allowing them to rank them. The ranking system would work on a scale of 1 to 10 where 1 represents a non desirable coffee and 10 represents a very desirable coffee. This method was already a very expensive method, but with the introduction of COVID it created a challenge. In order to attempt to simulate the same conditions, the problem was simplified to first principles. Having this in mind, the fitness is calculated through the following formula:

$$\frac{10x_p}{1.667} = y_{fit}$$

Figure 2.F (Fitness Calculation Formula)

Where x_p represents the following ratio $\frac{qC}{qS}$, 1.667 represents the ideal ratio, 10 represents the ideal fitness, and y_{fit} represents the calculated fitness. This formula is used to simulate a human drinking a coffee and rating it in the testing stage of the GA. This data is then fed into the regression model. Furthermore, the GA attempts to simulate human error by variating the final fitness, y_{fit}, by up to 10%. This allows us to establish whether the model is able to correctly identify the optimum. This method does have inherent issues and will be discussed in later sections.

The Polynomial Model

Due to the changing complexity of the code, the GA opts for a small adjustment to the multivariate polynomial model. Essentially, both the qC and qS genes each have their own model which they train independent of each other. When a new genome is introduced, the individual values of qC and qS are plugged into the model and the resulting fitnesses are then averaged together. This allows for the model to take into account both how genes independently change the fitness, while also modelling their interactions. Additionally, the polynomial model is allowed 4 degrees of freedom to prevent over and underfitting on sample datasets.

Experiment

The GA was initialized with a population of 80 individuals and a maximum number of generations set to 50. Normally GA's use much more than 80 individuals, but setting the population to such a number allows rapid testing as well as keeps inline with the purpose of our paper: testing the efficiency of ML imbued GA's on abnormal expensive optimization problems. Secondly, the amount of generations was set to 50 in order to provide a benchmark standard for each trial run of the GA, but are normally much higher.

In order to find out which combination of parameters worked best, each run of the GA utilized different parameters as outlined by the excerpt below (full datasheet in appendix):

Test Parameters				Results
% of Population Tested	% of Population Replaced	Fitness Calculation Method	Replacement Method	Final Fitness
10.00%	90.00%	Raw Fitness	Inverse Fitness Proportionate Selection	6.769794497
		Variation	Inverse Fitness Proportionate Selection	6.783838243
		Variation	Random Replacement	3.514705011
20.00%	80.00%	Raw Fitness	Inverse Fitness Proportionate Selection	8.316485027
		Variation	Inverse Fitness Proportionate Selection	7.117840022
		Variation	Random Replacement	4.598995184
30.00%	70.00%	Raw Fitness	Inverse Fitness Proportionate Selection	8.132100881
		Variation	Inverse Fitness Proportionate Selection	8.498495062
		Variation	Random Replacement	6.36830118

Figure 3.A (Test Parameters Table)

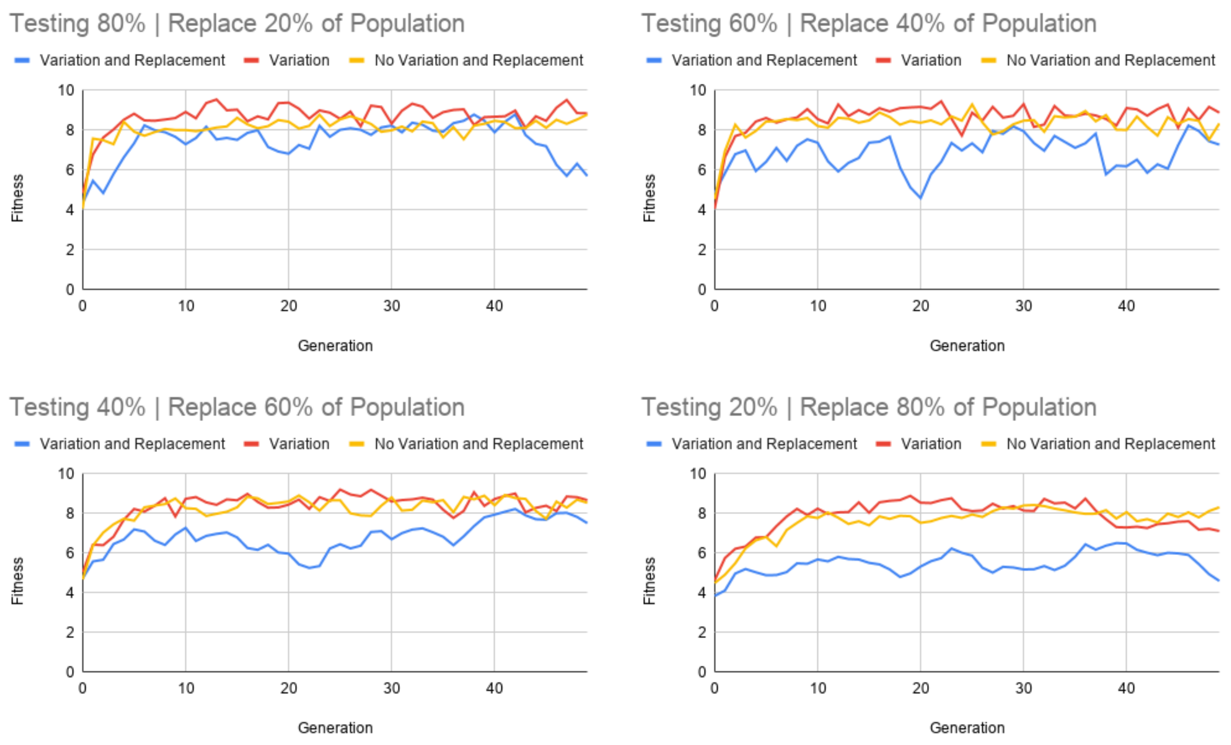
As demonstrated by Figure 3.A, four main parameters were explored. The first parameter as demonstrated by the first column of the table, focused on changing the percentage of the population that was tested. This parameter was altered in order to test the effectiveness of the polynomial regression model with a changing training set size. The second parameter focused on changing the generation gap, that is to say, the number of individuals replaced by children in each generation. Fitness Calculation was also changed using Variation and Raw Fitness. Although fitness calculation has already been explored in an earlier section, a brief summary will be provided. Fitness is calculated through a formula where the proportion of coffee and sugar in an individual recipe is measured, multiplied by ten (the ideal fitness), and later divided by 1.667.

$$\frac{10x_p}{1.667} = y_{fit}$$

The resulting fitness y_{fit}, then undergoes a variation. The variation is done to simulate human guess and provides random error into the calculation of the fitness. This is an attempt to introduce uncertainty into the polynomial regression model in order to analyze its performance. As shown by table above, certain GA runs remove this variation in order to see differences in performance between the two. The final parameter, the replacement method, is used to test the effectiveness of IFPS versus Random Replacement.

Results

Variated fitness with IFPS provided the best results for 89% of the population. Applying variation seems counterintuitive as it trains the model on inaccurate data, but the data suggests that applying 5% variation to the fitness allows the GA to explore a larger search space during the initial generations while also maintaining genetic integrity. Furthermore, adding variation not only has the benefit of increased search spaces, but also stopping early convergence while also mimicking human error associated with supervised fitness models. Graphs demonstrating the evolution of average fitness among each generation are shown below:



Figures 3.B, C, D, E (Generation vs Fitness Graphs)

The only place where variation failed to outperform raw fitness was in trial runs where the percent tested was less than 20% of the population. This is most likely the case due to the lack of information the model has to train with, compounded by the high replacement rate; which further propagates the faulty information.

All 27 GA runs were able to reach convergence by the10th generation or the 800th individual; with the GA's whose testing percentage was above 30% demonstrating a very rapid convergence rate and maintaining said average throughout the entire 50 generations. The ability to reach such high levels of convergence can be attributed to the increased accuracy of the model as demonstrated by the following comparison:

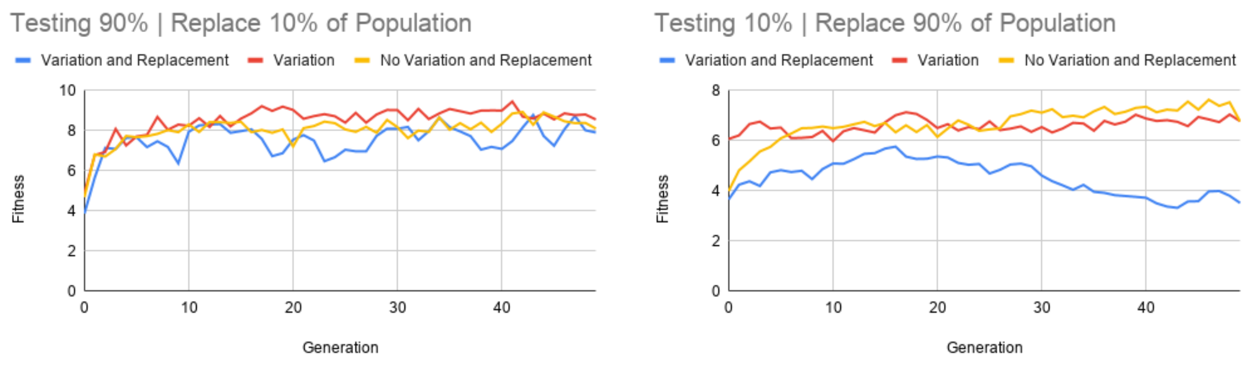
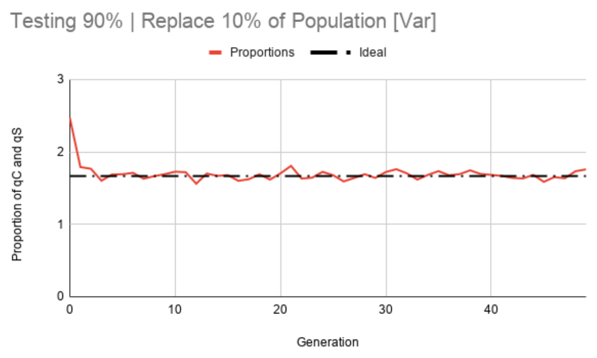


Figure 3.F, G (Generation vs Fitness Graphs)

On the topic of convergence, it can be noted that the GA's that utilized variation as well as replacement were unable to converge and in the worse scenarios, as demonstrated by Figure G (10% - 90%), led to a net loss in population fitness. Making an inference from the data gathered suggests that random replacement of children from crossover leads to the elimination of members in the population that contribute above average genes. Likewise, too much diversity in a population can cause the GA to "behave chaotically like a random search" [1]. While this is still possible in IFPS, it has a far lower chance of occurring. A possible way to circumvent early convergence or a lack thereof when using Random Replacement is through implementing Elitism. Elitism ensures the survival of the most elite members of the population, at the cost of being more prone to early convergence. As [6] suggests, low Elitism values combined with Random Replacement can be beneficial to reaching the optimum at a faster pace. Furthermore, Elitism has the effect of slightly combatting the increasing number of iterations naturally encountered when using purely probabilistic search techniques [7].

Finally, because the GA was designed to attempt and find the best proportion of Coffee and Sugar it would make sense to analyze how the proportions changed over time through each generation.



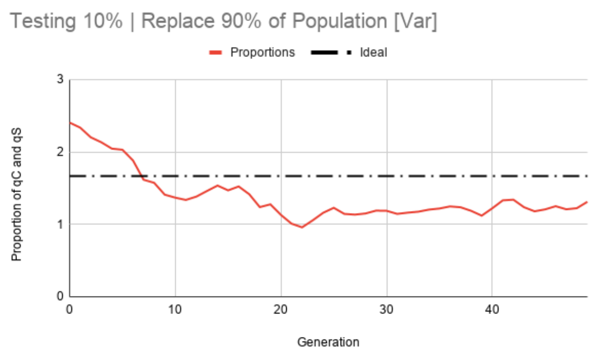


Figure 3.H, I (Evolution of Proportions between qC and qS each Generation)

The graphs above would suggest that when using a more accurate model, the GA's population is able to converge on the ideal solution with quick speed while the least fit model fails to converge on the ideal optimum. Furthermore, while mostly all GA runs were able to converge on the same proportion, each population had differing values for their qC and qS genes. Although it is assumed that the amount of water stays the same, the GA’s ability to find different manifestations of the same solution highlights the GA’s flexible nature.

Conclusion

"The enormous potential of GA's lies elsewhere - in optimization of non differentiable or even discontinuous functions, discrete optimization, and program induction"

- U. Bodenhofer

There is a beauty in the way that nature handles failure. Evolution it may seem, favors the strong, but failure is the currency for success and thus, the natural push and pull between strong and weak that fuels nature's evolutionary process shows us that there can be no success without failure. Understanding this relationship is paramount to designing an ideal Genetic Algorithm. Firstly, it's the environment that dictates who fails and who succeeds. Secondly, it's important to understand that failure and success are relative to each other. An optimal solution may work, but as the environment's demand changes so does the solution.

Analyzing the results of our experiment it would appear that GA's can be used to find optimums *given their environment*. The GA as it was used in this paper, succeeded in converging on our assumed proportion of 1.667. It further demonstrated that the use of polynomial regression for fitness calculation can aid in decreasing the computation cost of assigning fitnesses to population's with many members while avoiding overfitting and underfitting. What this GA fails to prove however, is whether GA's can be used to solve atypical and expensive optimizations problems whose fitness calculation can't be determined through mathematical means. Due to the circumstances, I was not able to do the testing that I would've preferred, but there is hope in the use of machine learning techniques to decrease the computational cost. This paper also determined that an increase in the data used to train the model as well as a generational gap of around 40% allows the GA to maintain a healthy balance between genetic diversity and rate of convergence. Moreover, the importance of diversity in a population is key in the beginning phase to ensure that there is no bias in the starting population.

GA's are very useful when analyzing complex problems, but it is important to understand that the GA performs just as well as the environment it's subjected to. When GA's are designed thoughtfully and the fitness function through to the specific parameters are chosen to fit the problem; GA's are able to traverse search spaces to find the relative optimums. It must be noted once more that GA's do not inherently learn, they mimic a system designed to improve upon relative failures. With the implementation of machine learning, however, GA's are provided with the ability to possess an intuitive like comprehension of what is fit and unfit. The combination of instinct (Genetic Algorithms) and intelligence (Machine Learning) prove to be efficient at decreasing computational cost of GA's while also showing promising results for solving expensive atypical optimization problems.

Bibliography

[1] U. Bodenhofer, “Genetic Algorithms: Theory and Applications” 2002

[2] D. Jong, and A. Kenneth, “Evolutionary Computation: A unified approach” 2006

[3] H. Randy, and S. Haupt, “Practical Genetic Algorithms” vol. 2, 2006

[4] J. Koza, “Genetic Programming: On the Programming of Computer by means of Natural Selection, 1992.

[5] D. Whitley, “A genetic algorithm tutorial,” Stat Comput, vol. 4, no. 2, pp. 65–85, Jun. 1994.

[6] R. Pursehouse, P. Fleming, “Why use Elitism and Sharing in a Multi-Objective Genetic Algorithm,” *University of Sheffield*, Jul. 2002

[6] B. Chakraborty, “On The Use of Genetic Algorithms with Elitism in Robust and Nonparametric Multivariate Analysis,” *Australian Journal of Statistics*, vol. 32, no. 1 & 2, pp. 13 - 27, 2003.

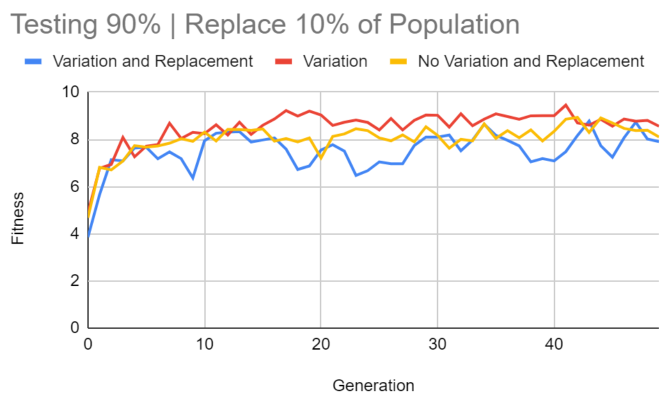
Appendix

GA Runs

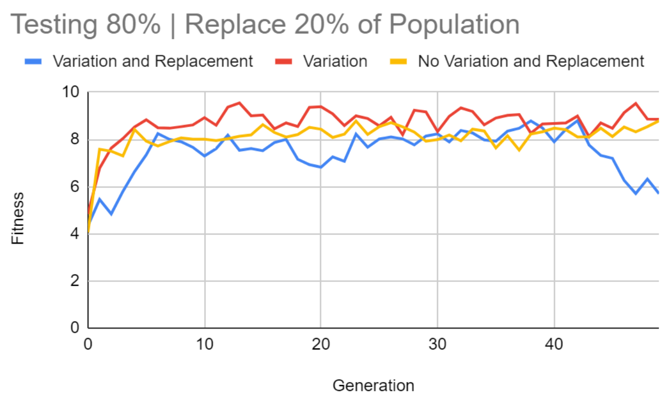
Test Parameters				Results
% of Population Tested	% of Population Replaced	Fitness Calculation Method	Replacement Method	Final Fitness
10.00%	90.00%	Raw Fitness	Inverse Fitness Proportionate Selection	6.769794497
		Variation	Inverse Fitness Proportionate Selection	6.783838243
		Variation	Random Replacement	3.514705011
20.00%	80.00%	Raw Fitness	Inverse Fitness Proportionate Selection	8.316485027
		Variation	Inverse Fitness Proportionate Selection	7.117840022
		Variation	Random Replacement	4.598995184
30.00%	70.00%	Raw Fitness	Inverse Fitness Proportionate Selection	8.132100881
		Variation	Inverse Fitness Proportionate Selection	8.498495062
		Variation	Random Replacement	6.36830118
40.00%	60.00%	Raw Fitness	Inverse Fitness Proportionate Selection	8.543198166
		Variation	Inverse Fitness Proportionate Selection	8.678782956
		Variation	Random Replacement	7.514350123
50.00%	50.00%	Raw Fitness	Inverse Fitness Proportionate Selection	8.367762684
		Variation	Inverse Fitness Proportionate Selection	8.771017873
		Variation	Random Replacement	6.697637303

60.00%	40.00%	Raw Fitness	Inverse Fitness Proportionate Selection	8.3441877
		Variation	Inverse Fitness Proportionate Selection	8.886959035
		Variation	Random Replacement	7.280078897
70.00%	30.00%	Raw Fitness	Inverse Fitness Proportionate Selection	8.085179135
		Variation	Inverse Fitness Proportionate Selection	8.453228255
		Variation	Random Replacement	6.872273834
80.00%	20.00%	Raw Fitness	Inverse Fitness Proportionate Selection	8.793699696
		Variation	Inverse Fitness Proportionate Selection	8.84999268
		Variation	Random Replacement	5.703019224
90.00%	10.00%	Raw Fitness	Inverse Fitness Proportionate Selection	8.107028248
		Variation	Inverse Fitness Proportionate Selection	8.559849021
		Variation	Random Replacement	7.908203991

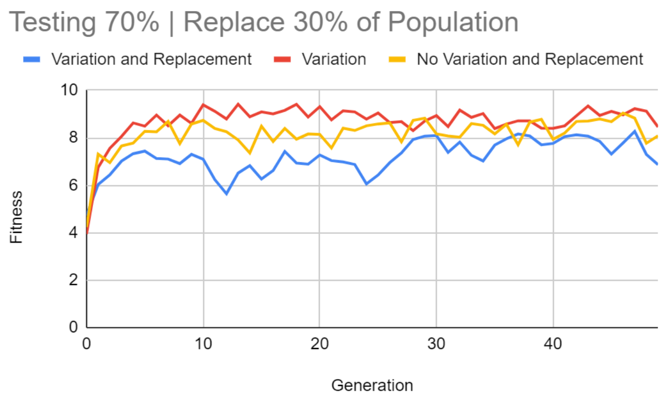
***Testing 90%* | Replace 10%**



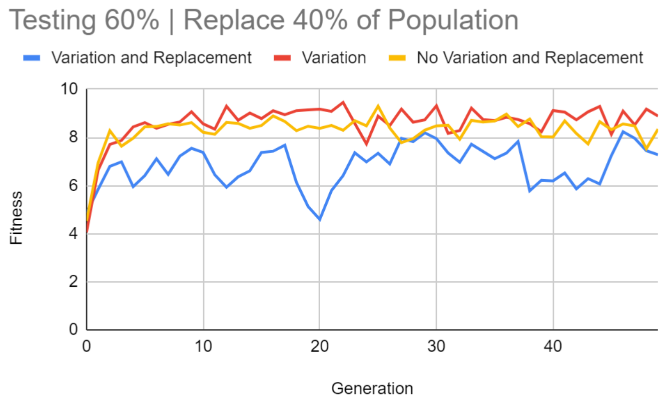
***Testing 80%* | Replace 20%**



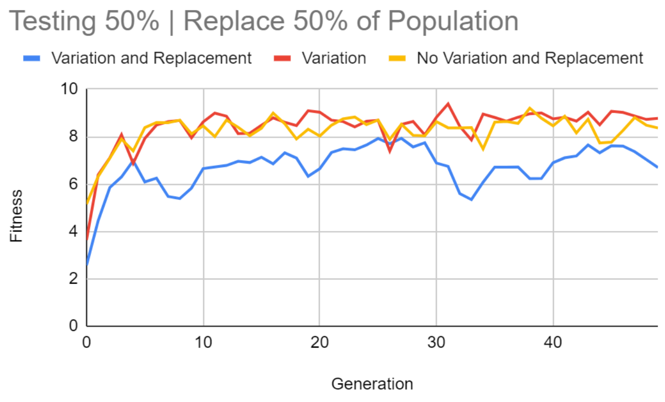
***Testing 70%* | Replace 30%**



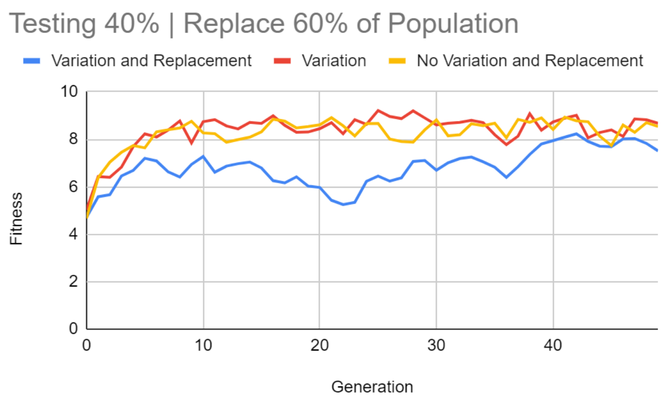
***Testing 60%* | Replace 40%**



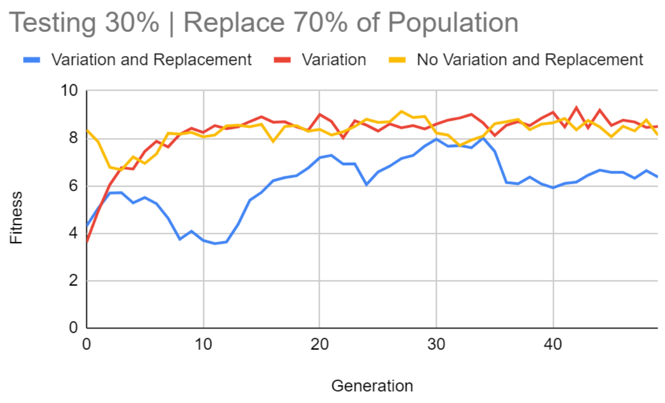
***Testing 50%* | Replace 50%**



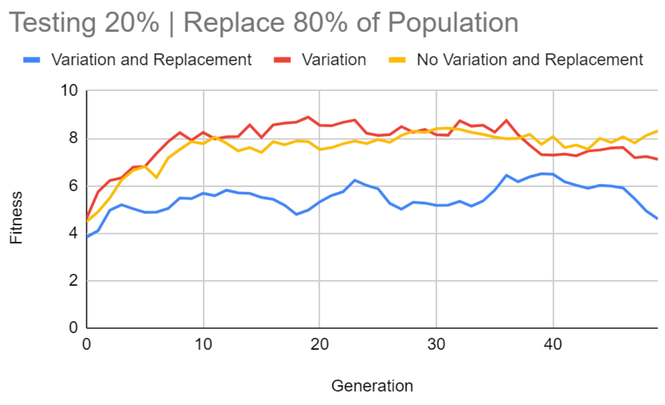
***Testing 40%* | Replace 60%**



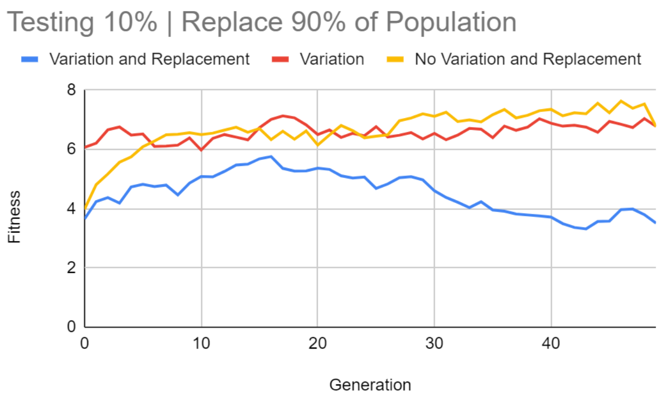
***Testing 30%* | Replace 70%**



***Testing 20%* | Replace 80%**

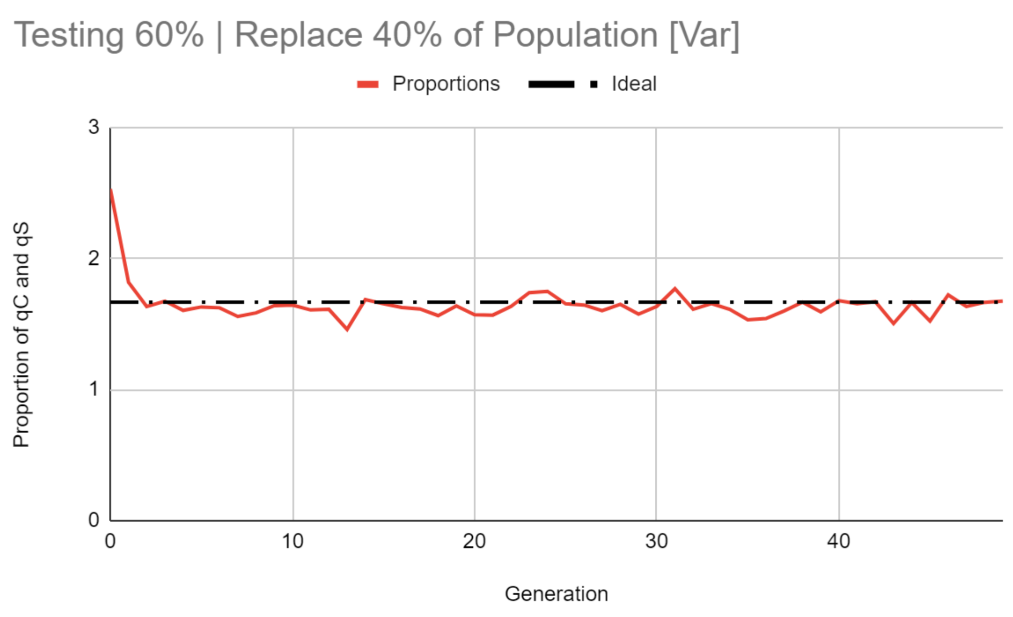


***Testing 10%* | Replace 90%**

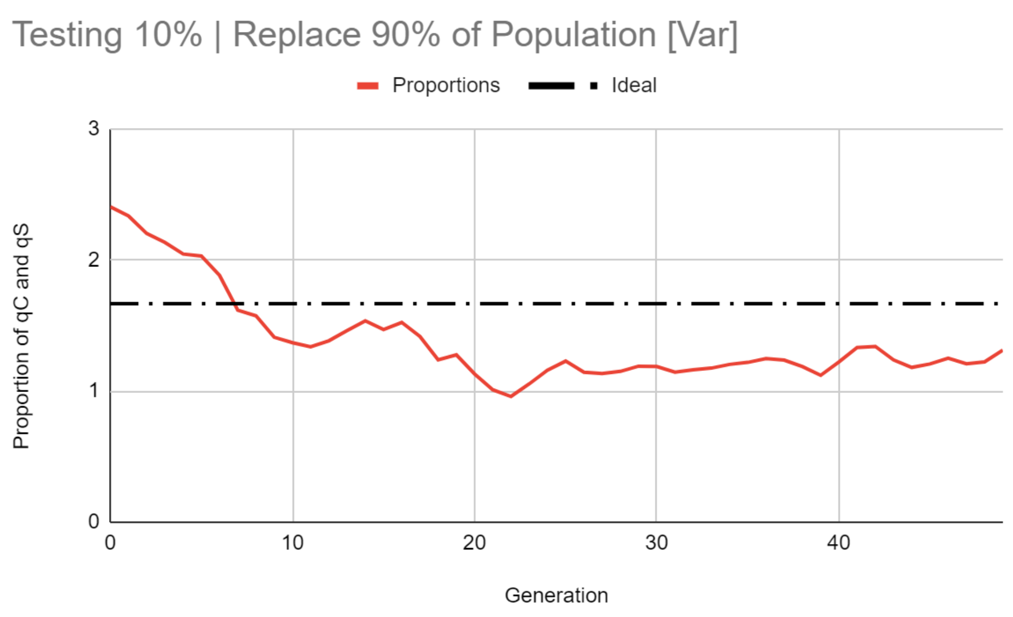

Testing 10% | Replace 90% of Population
Variation and Replacement
Variation
No Variation and Replacement
Fitness
Generation
0
2
4
6
8
0
10
20
30
40

Evolution of proportions between qC and qS

Best Case

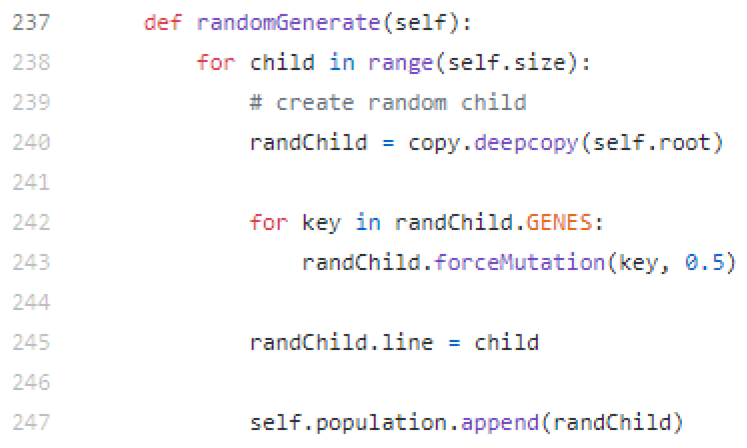


Worst Case



Code Extracts

Random Member Creation



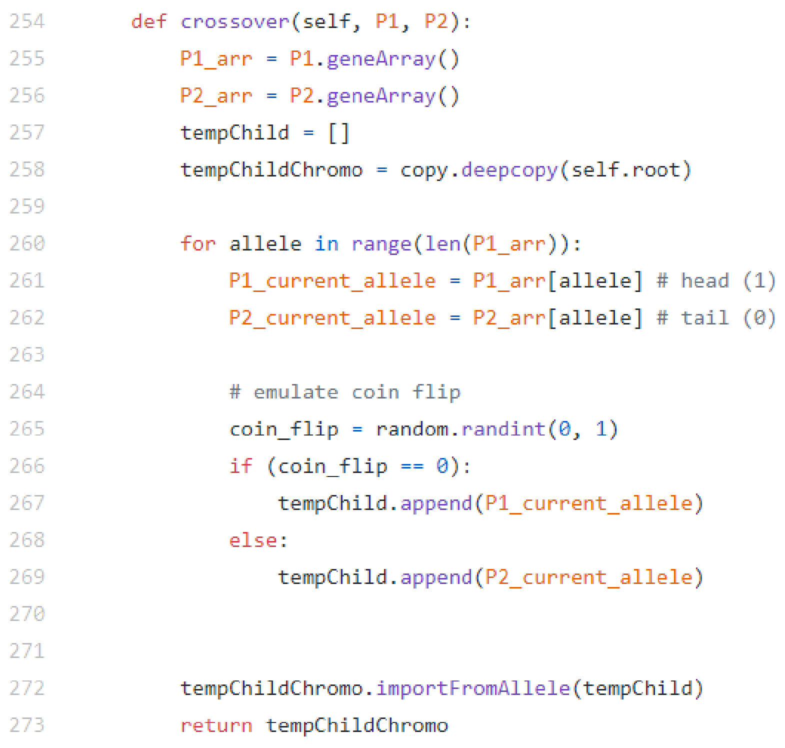
```
    def randomGenerate(self):
        for child in range(self.size):
            # create random child
            randChild = copy.deepcopy(self.root)

            for key in randChild.GENES:
                randChild.forceMutation(key, 0.5)

            randChild.line = child

            self.population.append(randChild)
```

Crossover Method



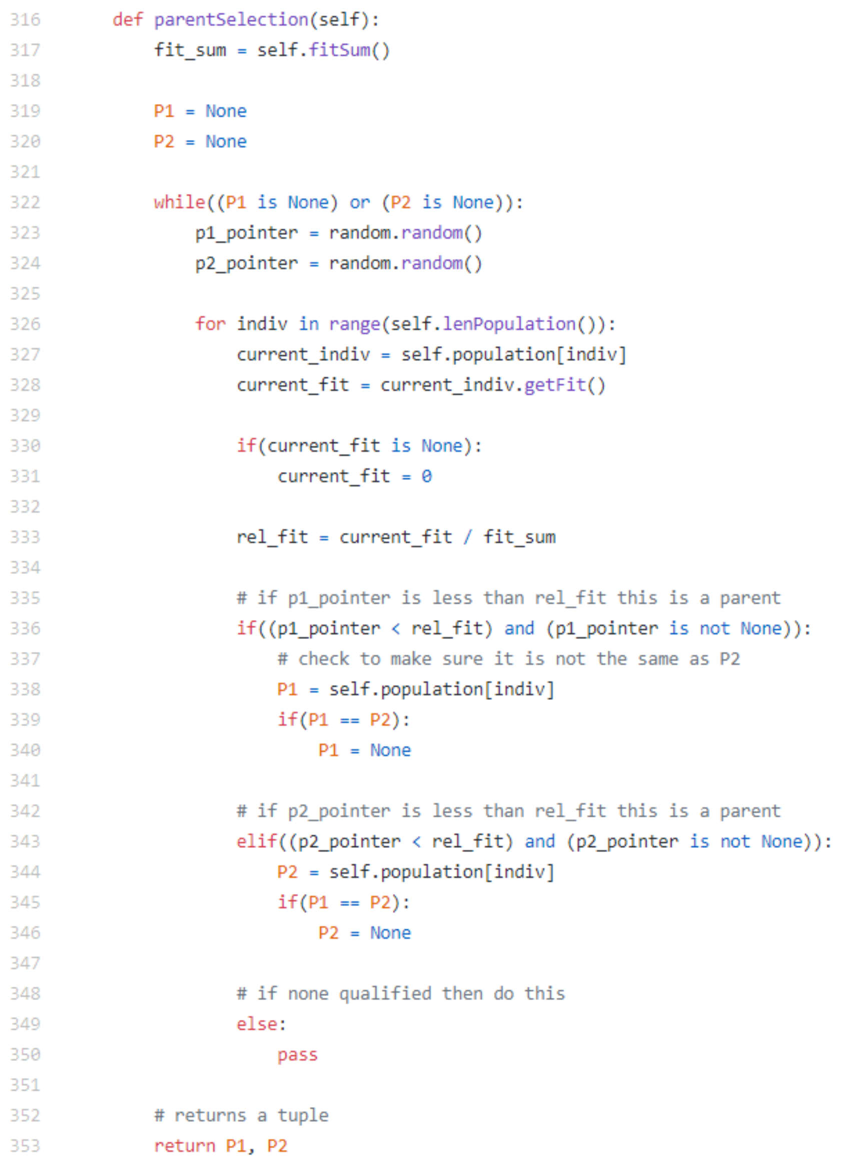
```
    def crossover(self, P1, P2):
        P1_arr = P1.geneArray()
        P2_arr = P2.geneArray()
        tempChild = []
        tempChildChromo = copy.deepcopy(self.root)

        for allele in range(len(P1_arr)):
            P1_current_allele = P1_arr[allele] # head (1)
            P2_current_allele = P2_arr[allele] # tail (0)

            # emulate coin flip
            coin_flip = random.randint(0, 1)
            if (coin_flip == 0):
                tempChild.append(P1_current_allele)
            else:
                tempChild.append(P2_current_allele)


        tempChildChromo.importFromAllele(tempChild)
        return tempChildChromo
```

Parent Selection



```python
    def parentSelection(self):
        fit_sum = self.fitSum()

        P1 = None
        P2 = None

        while((P1 is None) or (P2 is None)):
            p1_pointer = random.random()
            p2_pointer = random.random()

            for indiv in range(self.lenPopulation()):
                current_indiv = self.population[indiv]
                current_fit = current_indiv.getFit()

                if(current_fit is None):
                    current_fit = 0

                rel_fit = current_fit / fit_sum

                # if p1_pointer is less than rel_fit this is a parent
                if((p1_pointer < rel_fit) and (p1_pointer is not None)):
                    # check to make sure it is not the same as P2
                    P1 = self.population[indiv]
                    if(P1 == P2):
                        P1 = None

                # if p2_pointer is less than rel_fit this is a parent
                elif((p2_pointer < rel_fit) and (p2_pointer is not None)):
                    P2 = self.population[indiv]
                    if(P1 == P2):
                        P2 = None

                # if none qualified then do this
                else:
                    pass

        # returns a tuple
        return P1, P2
```

Replace for Death

```
    def replaceForDeath(self):
        fit_sum = self.fitSum()

        iFit = 0

        for indiv in range(self.lenPopulation()):
            current_indiv = self.population[indiv]
            current_indiv_fitness = current_indiv.getFit()


            if(current_indiv_fitness is None):
                current_indiv_fitness = 0
            else:
                rel_fit = current_indiv_fitness / fit_sum

                try:
                    inverse_rel_fit = 1 / rel_fit
                except RuntimeWarning:
                    inverse_rel_fit = 0
                iFit += inverse_rel_fit

        pointer_index = None
        while(pointer_index is None):
            pointer = random.random()
            for itm in range(self.lenPopulation()):
                current_indiv = self.population[itm]
                current_indiv_fitness = current_indiv.getFit()

                if(current_indiv_fitness is None):
                    current_indiv_fitness = 0
                else:
                    rel_fit = current_indiv_fitness / fit_sum
                    inverse_rel_fit = 1 / rel_fit

                    if(isinf(inverse_rel_fit)):
                        inverse_rel_fit = 1.0
                    adjusted_inverse_rel_fit = inverse_rel_fit / iFit

                    if ( pointer < adjusted_inverse_rel_fit ):
                        if (current_indiv.isFit()):
                            pointer_index = itm
                    else:
                        pass

        return pointer_index
```

Select For Testing

```python
    def fitEntirePopluation(self):
        for i in range(self.lenPopulation()):
            current_i = self.population[i]
            # if not fit apply fitness
            if(not current_i.isFit()):
                geneInfo = current_i.geneArray()

                coffee_val = np.array([geneInfo[0]])
                sugar_val = np.array([geneInfo[1]])

                coffee_val = coffee_val.reshape(-1, 1)
                sugar_val = sugar_val.reshape(-1, 1)

                coffee_val = self.qCPoly.fit_transform(coffee_val)
                sugar_val = self.qSPoly.fit_transform(sugar_val)

                qCFit = self.qCModel.predict(coffee_val)
                qSFit = self.qSModel.predict(sugar_val)
                avg = ( (qCFit[0][0]) + (qSFit[0][0]) ) / 2.0
                avg_ = self.rangeClip(avg, 0.1, 10.0)
                current_i.updateFitness(avg_)
```