Project 3

Bryce Harmsen, Nisser Aldossary, Eric Engman, & Luis Torres
CS 457 CI & ML

Introduction

Project 3 offered two problems, searching for a magic square and maximizing a sinusoidal-logarithmic function. Our team implemented both. Both problems were solved using genetic algorithms. The magic square focused on minimizing the average deviation from the magic sum for the given matrix size, while the sinusoidal-logarithmic function output was maximized within a bounded space.

Definitions of Fitness

Magic Square

For the magic square problem, fitness for an individual was defined through standard deviation, taking the square root of variance, where variance is the average of the squared differences from the mean.

Sinusoidal Logarithmic

For the sinusoidal-logarithmic function, called function *f*, the fitness was defined as follows:

$$f(x,y) = sin\left(10\pi x + \frac{10}{1+y^2}\right) + ln(x^2+y^2) \text{ where } 3 \leq x \leq 10 \text{ and } 4 \leq y \leq 8 \text{ such that } x, y \in \mathbf{R}$$

Below are the solution details, including the implementation, results and analysis for both problems.

Implementation

Magic Square

The genetic algorithm, or GA, for constructing a magic square was written in C++. It was written using two classes, a class for making an individual (square) and a main class. The construction of the initial squares, the crossover, and the calculation of the fitness scores were done in the Individual class. The genetic operators used were

selection and crossover. For selection, the population of each generation was sorted in ascending order inside of a vector, and the first half (50%) of that population was selected to mate. In the selection loop, a random between 1 and 50 was used to randomly select each parent from the fittest part of the population. The crossover function simply iterated over each parent's matrix, rolling a random number with a 50% chance for either one to insert their gene at the corresponding square in the child.

Sinusoidal Logarithmic

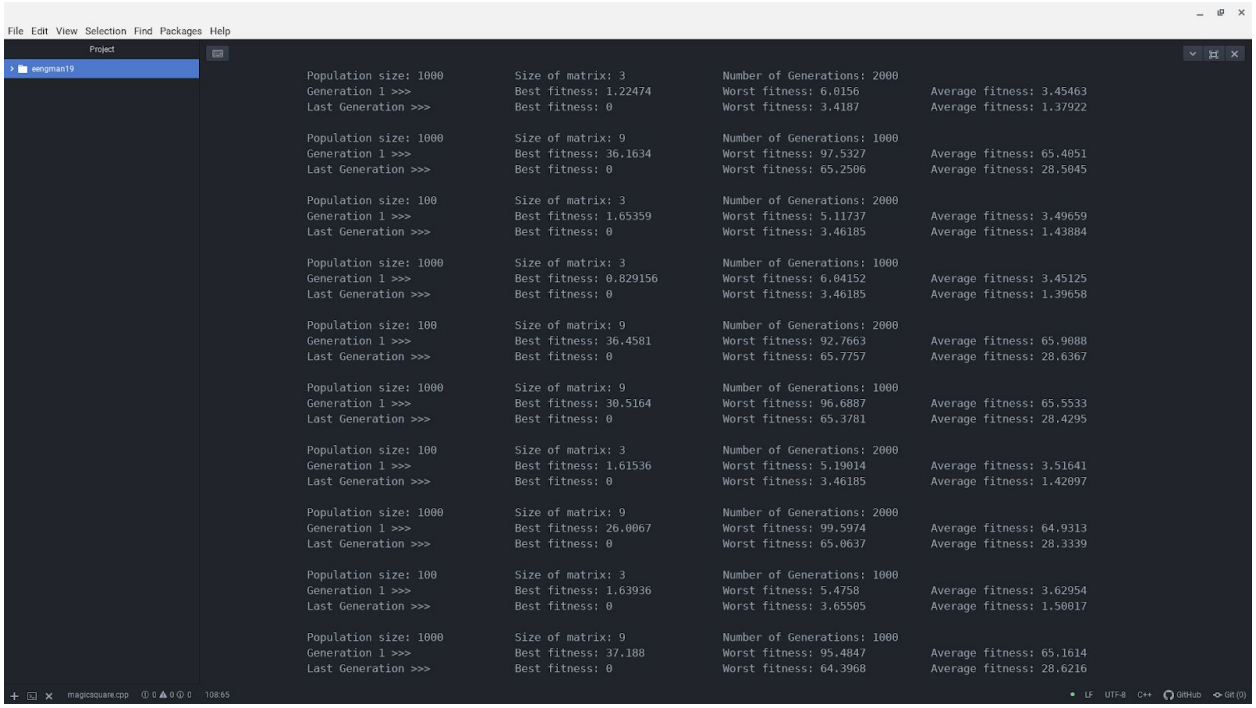
The genetic algorithm, or GA, for maximizing *f* was written in Python. The code was built with future implementations in mind. The general shape of the GA is defined in a high-level class named *GA*. This class has enough definition to iteratively select, crossover, and mutate some population of chromosomes, but is abstract for the selection, crossover, mutation, and fitness methods. The next class down, *SinLn*, defines roulette selection, single-point crossover, mutation based on a user-defined mutation rate, and a fitness function matching the above function *f*.

A number of utility functions are included through a custom importable file named *plotter*. These plotting functions allow for a 3-dimensional approximation plot of the fitness function, a 2-dimensional time series plot of the average and best fitnesses of the chromosome population over the given number of generations, and a 2-dimensional scatter plot of the (x,y) position of each generation’s most fit chromosome.

Results

Magic Square

In the magic square program, the results regarding the average fitness score from the first population to the last were mostly consistent across the different configurations of population size, size of the matrix, and number of generations. On average the fitness of the last generation would be around half when compared to the first. The average fitness score in the final generation decreased in the tests proportional to the size of the matrix. For a 3x3 matrix, the average fitness score would be around 1. For a 4x4, the average fitness score would be around 3. As you increased in size, the average score would get worse, mostly being unaffected by the number of generations or the population size. Below are the results obtained from a set of trials.

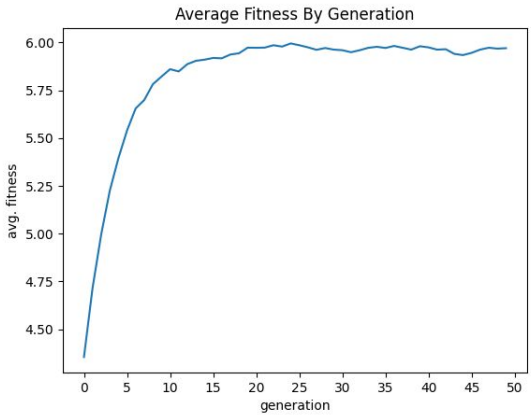


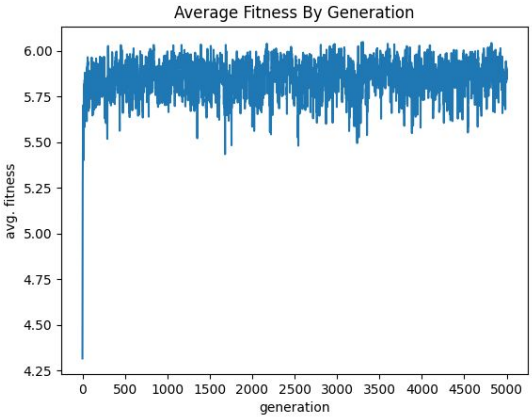
Sinusoidal Logarithmic

The maximum *f*-values obtained by the GA provided are listed below in the *Parameters & Results* table. Two examples are given in the table, each with varying parameter choices and results. The parameters listed can be manipulated from the *params.yaml* file. Because the population is stochastic and random in nature, results vary by execution even when the parameters are held constant. The figures below match with a column in the table. The leftmost figures are connected to the first example, the rightmost connected to the second.

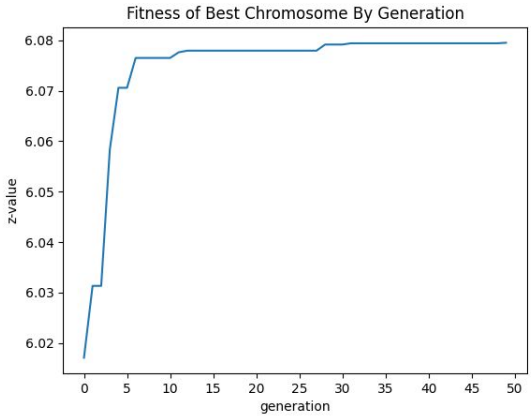
Parameters & Results		
Attributes	**Values for First Example**	**Values for Second Example**
Population Size	1200	100
Generations	50	5000
Mutation Rate	0.1	0.2
Optimal Coordinate	(9.844901697984566, 7.997861381341422)	(9.84549768803308, 7.99910178977885)

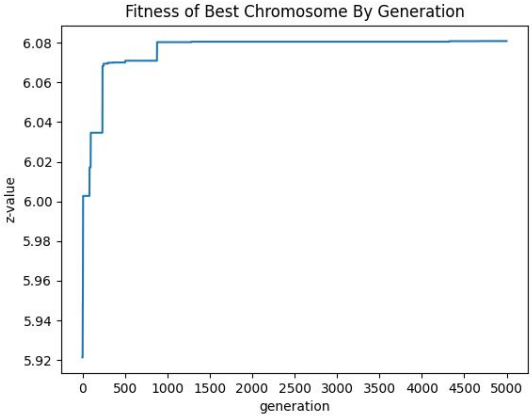
Maximum *f*-value	6.0806882275962915	6.080826630665241





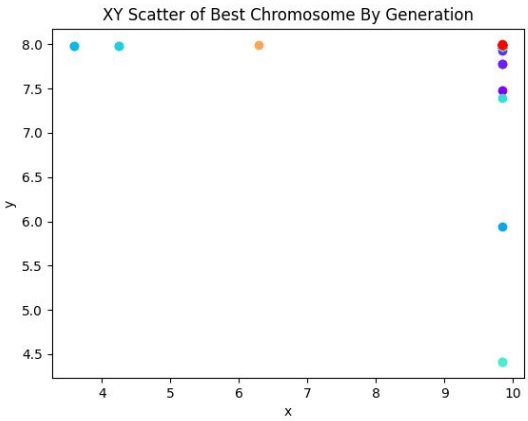
From the results stored from all the GA runs, we calculated the average fitness generation populations collected. We then plotted these results in a graph of average fitness per population versus the appropriate generation number. The leftmost figure shows a plot of average fitness per population from populations selected from generation throughout 50 generations. The rightmost shows a plot of average fitness per population from populations selected from generation throughout 5000 generations. The average of the trials is shown in blue.

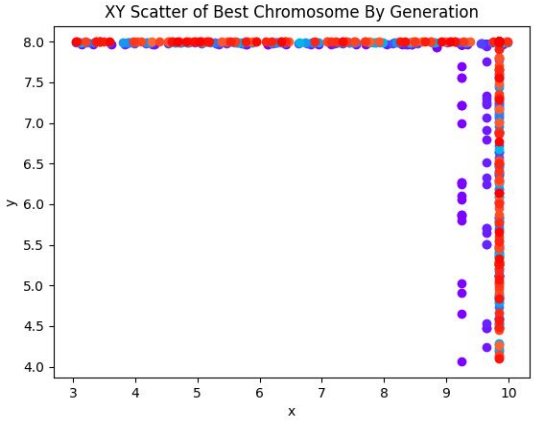




From the results stored from all the GA runs, we calculated the fitness of the best chromosome generation populations collected. We then plotted these results in a graph of chromosome generation per population(z-value) versus the appropriate generation number. The leftmost figure shows a plot of the most fitness chromosome per population from populations selected from generation throughout 50 generations. The rightmost shows a plot of the most fitness chromosomes per population from populations selected from generation throughout 5000 generations. The mutation rate,

and the number of iterations were increased greatly. The fitness of the best chromosome of the trials is shown in blue.





From the results stored from all the GA runs, we calculated the XY scatter of the position of the best fit chromosomes generation populations collected. We then plotted these results in graphs between the XY scatter plots. On the graph the blue spectrum plots representing the early generations and the red spectrum plots representing the later generations.

Analysis

Magic Square

The magic square program failed to fall below an average fitness score of 1 for all tests. Additionally, the ratio of the scores between the first generation and the final generation stayed held true, largely unaffected by the number of generations. This may be due to the selection algorithm that was implemented. In the selection phase, there was the possibility that the same parents would be selected repeatedly in a generation to reproduce. This may lead to less variance in the population, leading to an eventual plateau in improvement over the generations. Implementing a mutation function may have helped remedy this issue.

Sinusoidal Logarithmic

The sinusoidal logarithmic function *f* as described above was maximized through roulette selection, single point crossover, and mutation. The three adjustable parameters were population size, number of generations, and mutation rate. This GA's

performance is assessed through the optimal result obtained at the terminus of the algorithm as well as through the visuals displayed in the *Results* section above.

For the first example, the population size was set to 1200 to reduce average fitness fluctuation in the later generations. This amount of uniformly random data for the given solution space may be overpowering some of the genetic algorithm effects, contributing the benefits of a random walk algorithm, but in this case the large population still allowed for timely results and added a large amount of diversity to the gene pool for selection and crossover. Also, notice that the gains made successive generations tapered off by 20 to 25 generations. Also, because of either the lower mutation rate or the large population, fluctuation in the average fitness was not nearly as lively as in the second example.

In the second example, a smaller population was maintained, only 100 chromosomes. The number of iterations was increased greatly and the mutation rate was also increased. The mixture of a small population and high mutation rate allowed for more volatility throughout the generations. Even with the volatility shown in the average fitness plot, the best fitness did not make noticeable gains after about 1000 to 1500 generations. If this algorithm was used in production, it would be wise to reduce this parameter to around 1500 generations for sake of time efficiency.

The last comparison to make is between the XY scatter plots. These plots show the position of each of the most fit chromosomes by generation. The coloring is in the order of the rainbow, with the more blue spectrum plots representing the early generations and the more red spectrum plots representing the later generations. Notice that even within the few generations of the first example, the best chromosomes are on the upper and rightmost edges of the domain plane. While the second example explores the upper and rightmost edges of the domain more thoroughly, the most fit chromosomes eventually converge to the same upper right corner of the domain space.