Intro to Artificial Intelligence - ENGR3720
Project - Part A

Created by

Daniel Smullen

Jonathan Gillett

Joseph Heron

04/03/2013

Document Revision 1.8

Table of Contents

Equations and Figures List

Equations

Figures

The Eight Queens Puzzle

Background

Our solution for the Eight Queens Puzzle was to implement a genetic algorithm whose chromosomes model the position of each of the queens on the board. The values permuted represent the sequential row positions of each of the queens, encoded into binary values. Thus, the column positions increase by one for each queen and the intersections (referred to as collisions) are calculated per row or on the diagonal to determine fitness.

Solution Architecture

The solution was written in Java, using an object oriented approach. The genetic algorithm component of the solution was divided into 5 classes: $Breed$, $Chromosome$, $Fitness$, and $Selection$. The $EightQueens$ class handles the logic for generating and rendering the solutions using the other classes. External classes were also created and used for rendering output to the screen, and for generating plots. These were unrelated to the solution implementation and are outside the scope of this report.

Chromosome Structure

The chromosome structure is specified in the $Chromosome$ class, which includes methods and attributes for storing and manipulating the data structures which represent the chromosomes for the genetic algorithm. One gene is required for each queen in the problem, and therefore 8 genes make up the chromosome. The column value is used as the index of each gene within the chromosome. Figure 1 shows a small sample of 3 genes. Each gene represents the queen marked by the gene's index. Therefore, queen 1 is located in the first column on the board, and queen 8 on the last column. The board is made up of 8 rows and 8 columns. The class also contains functions for validating the values produced within the genes, in order to prevent erroneous or impossible game boards containing queens outside the range of possible spaces. Each chromosome is a $ArrayList$ data structure (of genes), which store the individual queen positions.

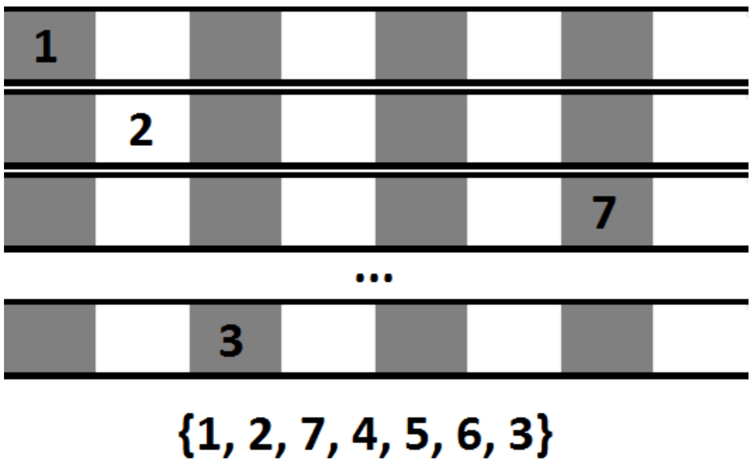


Figure 1 - Sample Chromosome Showing Four Genes

Evaluating Fitness

As with any genetic algorithm, evaluating the fitness of the chromosomes in a generation is required to determine if a solution has been generated. This section discusses the fitness evaluation and fitness function.

Calculating Collisions

When two or more queens share a row or diagonal, a collision occurs for each pair of queens. Within the $Fitness$ class, the function $calcCollisions()$ takes a chromosome and determines how many collisions are occurring. This function compares each gene across each row iteratively. During this process, the slope between queens on the board is tested. If the absolute value of the slope is found to be 1, the queens lie on the same diagonal and a collision occurs.

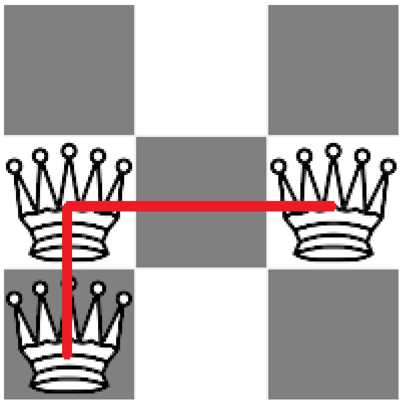


Figure 2 - Sample with Two Collisions

Fitness Function

The fitness function chosen is expressed as:

Equation 1 - Eight Queens Fitness Function

$$f(x) = \max\left(\frac{1}{N}\right)$$

In Equation 1, N represents the number of collisions between queens on the board. This value can be up to a maximum of $(1) + 8$ where eight collisions occur. In the base case there are no collisions and $N = 1$. In this instance, the function is at the absolute maximum and a solution has been found. Figure 3 demonstrates an example solution, where all queens are placed in an arrangement where no collisions have occurred. In theory, 92 unique solutions are possible within the bounds of these rules.

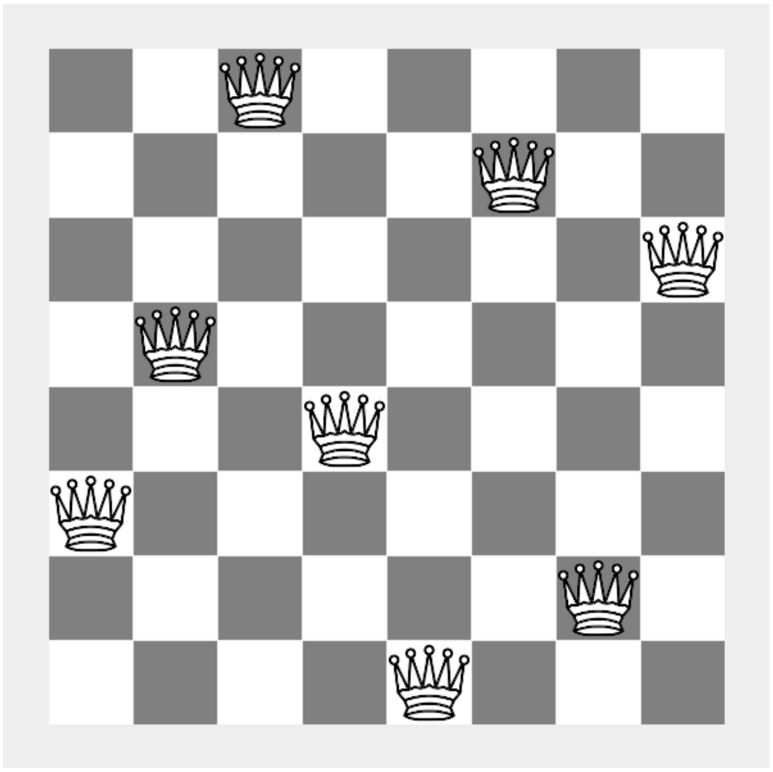


Figure 3 - Sample Solution

Symmetrical Solutions

Once a single solution has been found, the nature of the square chess board allows for a new and unique solution to be generated by rotating the board 90 degrees, 180 degrees, and 270 degrees. In Figure 4, a new solution is generated as a result of rotating the first solution 90 degrees. Since the movements of the queens are all symmetrical, rotating the board will not disturb the arrangement of the queens in any way which will result in a collision where there was none previously. Thus, computation of 1 solution automatically results in 4 solutions.

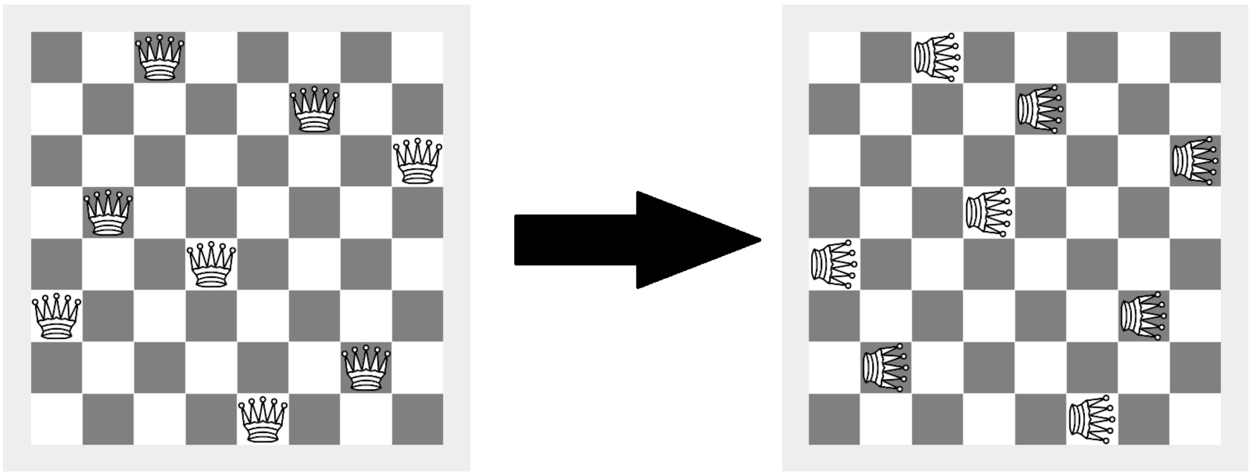


Figure 4 - Symmetrical Solution

Crossover Operator

To perform the crossover function, a random number between 0 and the size of the chromosome is generated. This serves as the point from which the crossover occurs. Genes from the $ArrayList$, which make up each chromosome, are selected by index from the random number up to the size of the chromosome (the end of the $ArrayList$). These genes are swapped between the two, creating a new child which is added to the genome.

Mutation Operator

Mutation occurs randomly for each generation of chromosome. When a random number is generated, a gene is randomly selected for mutation and replaced with a random value within the parameters of the board size. A gene's coded value is changed as a result. Helpful mutation through a process called 'inbreeding' has been added to our implementation, which increases the randomness within the genome, and increases the speed and likelihood of finding a new solution when the fitness evaluation is stuck in a rut of similar chromosomes.

"Inbreeding" and Dynamic Mutation Probability Values

In nature, organisms with very similar chromosomes (such as family members) are increasingly susceptible to genetic mutations when breeding. The result of this genetic expression in biology is chaotic and generally destructive, but implementing this effect in genetic algorithms was theorized to be helpful to increase the efficiency of finding solutions. In our implementation of genetic algorithms, this effect is captured by increasing the mutation likelihood coefficient based on the degree of similarity between chromosomes in a generation. As the similarity increases, the mutation rate increases proportionally. The result increases the speed of the algorithm at finding solutions, and increases the effectiveness of the mutation operator. By using an analogous representation of the effect of inbreeding in nature, our genetic algorithm implementation can indeed traverse the solution space more efficiently. In addition, as the similarity decreases, the mutation rate decreases as well, preventing wildly different solutions from entering the genome, and limiting the amount of variance across generations to a more sensible value. This is supported by graphing and comparing the performance of the dynamic mutation rate versus various static mutation rates. The mathematical difference between the ordinary stochastic search that results from using the original static mutation coefficient and the improved dynamic 'inbreeding' method can be seen when comparing the amount of generations required to attain solutions to the puzzle.

Figure 5 shows the performance statistics for finding all 92 solutions for the puzzle using the 'inbreeding' dynamic mutation rate, with an exponential best fit curve matching the average across 3 separate runs. In general, this solution performs well, finding all the solutions in less than 40,000 generations. Because the algorithm is stochastic, some variance is to be expected in terms of the number of generations required to find these solutions.

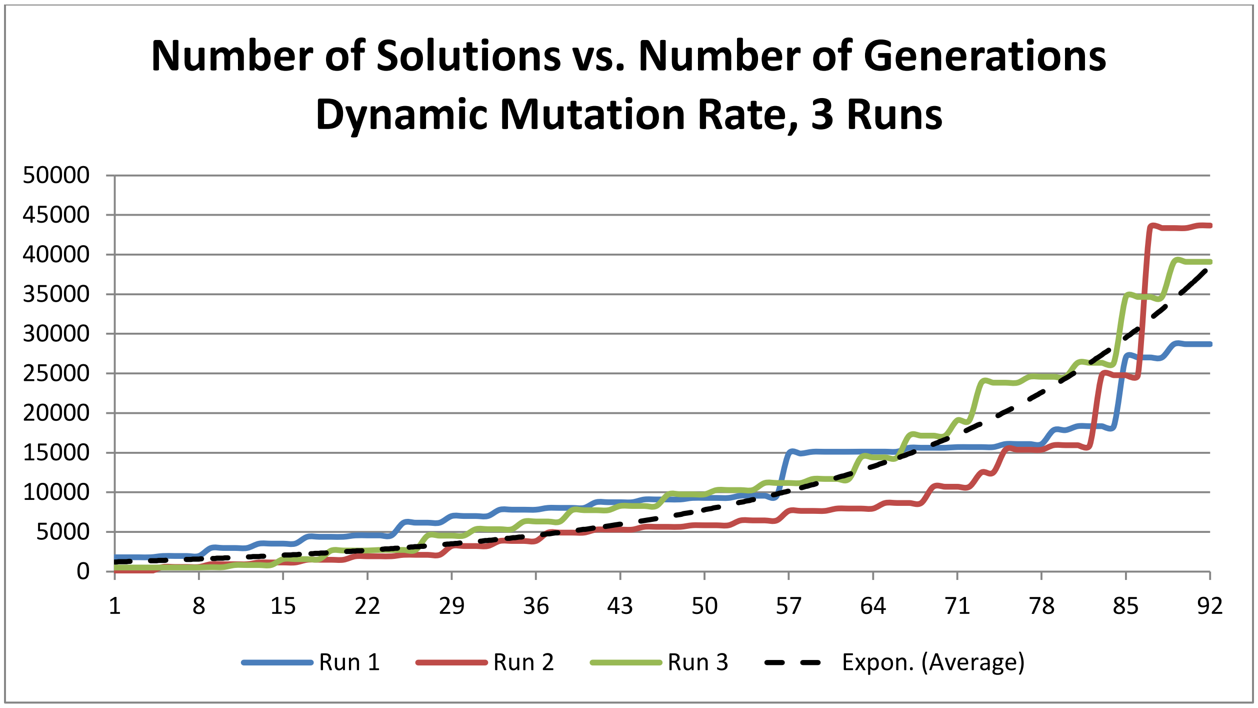
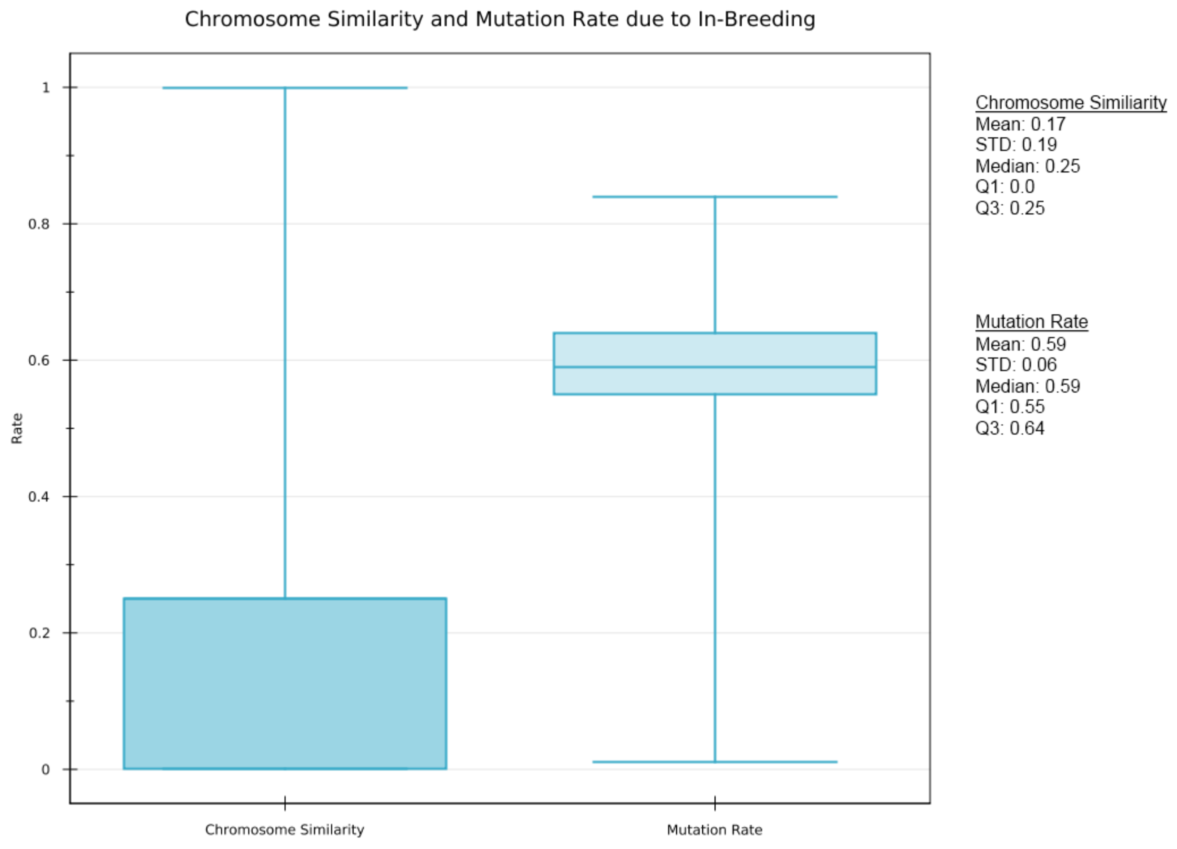


Figure 5 - Dynamic Mutation Rate Plot

Overall, the effect of the 'inbreeding' method is that the randomness of the system is increased, as the similarity of the chromosomes is decreased. This means that the expressed genome is more likely to have a shift in fitness values per generation when the gene pool begins to stagnate at a particular fitness level. Figure 7 demonstrates that the dynamic mutation rate results in far fewer required generations in comparison to various other fixed mutation rates. With a fixed mutation rate of 50%, performance is similar to the dynamic mutation rate albeit slightly poorer. Overall, the convergence of the dynamic mutation rate was toward a median mutation rate of approximately 59%, with standard deviation of 6%. This is seen in Figure 6. Incidentally, the 50% fixed mutation rate is the closest fixed value to the median of 59% - in theory, the dynamic mutation rate should converge upon something close to an optimal value, and the 50% fixed mutation rate is the closest approximate value in the dataset. Further investigation by using finer-tuned fixed mutation rates may yield even more optimal output in this solution space. Regardless, the best computational performance generated all 92 solutions in under 40,000 generations, as seen in Figure 5.

For the required output of 10 solutions (which are depicted in Appendix A), corresponding fitness graphs can be seen found in Figure 8.



Chromosome Similiarity
Mean: 0.17
STD: 0.19
Median: 0.25
Q1: 0.0
Q3: 0.25

Mutation Rate
Mean: 0.59
STD: 0.06
Median: 0.59
Q1: 0.55
Q3: 0.64

Figure 6 - Mutation Rate and Chromosome Similarity

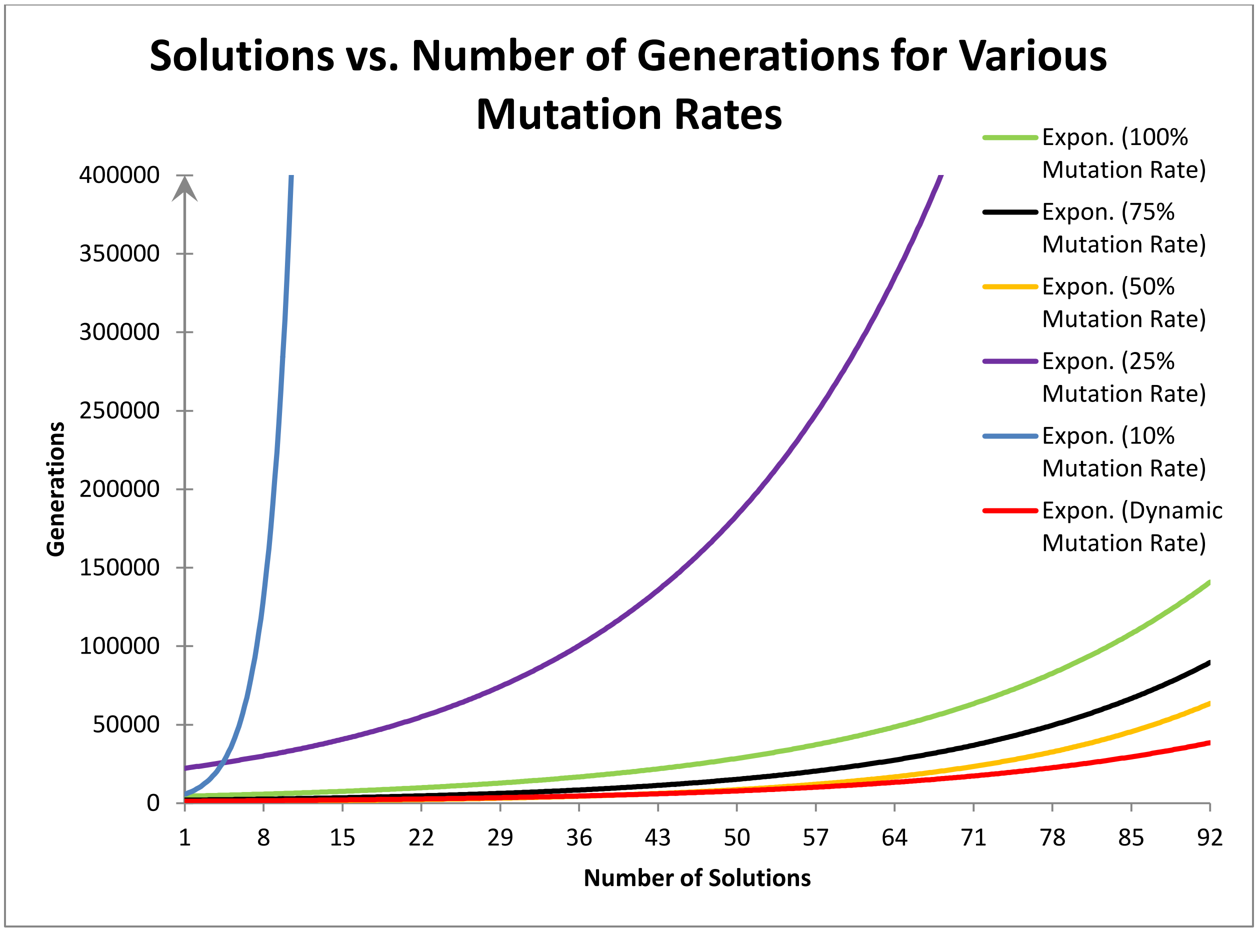


Figure 7 - Plotting Various Mutation Rates

The general trends seen in Figure 7 seems to indicate that while dynamic mutation values perform better overall, correctly selected fixed mutation values can improve performance, but with diminishing returns if they are set too high or too low. This is evidenced by the performance trend. The performance is worse with mutation rates higher than 50%, requiring more generations overall. The poorest performer is the lowest mutation rates, however, with 10% and 25% fixed mutation rates showing the most required generations overall. The indirect result of this is finding solutions with better fitness in fewer generations, because the generations are less similar.

Using Dynamic Mutation Values in Other Applications & Further Investigation

We hypothesize that the average number of generations required per mutation rate is tightly bound to the contour of the solution space. If the solution space is highly chaotic and random with large peaks and many solutions, traversing similar solutions would produce vastly different fitness values, and this approach may not be as effective as a more exhaustive search of similar areas of the solution space with smaller mutation values. We believe that the results show that the solution space for the Eight Queens Puzzle is much smoother, making similar solutions produce relatively similar fitness values as there are few fitness optima in the overall solution space. This is because there are only 92 solutions, compared to $\binom{64}{8} = 4{,}426{,}165{,}368$ possible arrangements of the eight queens. This hypothesis could perhaps be tested by using an exhaustive (brute-force) search of the solution space, and mapping the fitness of each solution generated. This would create a general contour for the solution space which could be compared to similar computationally expensive problems in chess or elsewhere, but that investigation is beyond the scope of this report.

Chromosome Similarity

The dynamic mutation rate works based on the similarity of chromosomes – if 15% or more of the genes within the chromosome are the same, inbreeding occurs and the mutation rate increases by 1% per generation. If fewer than 15% of the genes within the chromosomes are the same, the mutation rate decreases by 1% as inbreeding does not occur. This leads to rapid changes in mutation rates across groups of similar generations, rapidly widening the traversal of the solution space when similarities occur. These values were selected arbitrarily, and further optimization of these values could yield even better results than those in this study.

Selection Method

This implementation of genetic algorithms uses standard roulette wheel selection. As the fitness is evaluated for each chromosome in a generation, so is the fitness ratio in comparison with all other chromosomes in the generation. Chromosomes are randomly selected for usage in the next generation's evaluation, with probability of selection equal to their fitness ratio.

Differential Evolution (DE)

Background

DE works similarly to genetic algorithms, except rather than using an encoded chromosome to represent the expression for evaluation; the chromosome is represented directly by the values used for the expression. This is iteratively improved using cloning, crossover, and mutation of values. In this implementation, five different mathematical expressions were evaluated using differential evolution. Constraints were provided which limit the evaluation of the expressions, and the operation of the differential evolution process.

Evaluation Constraints

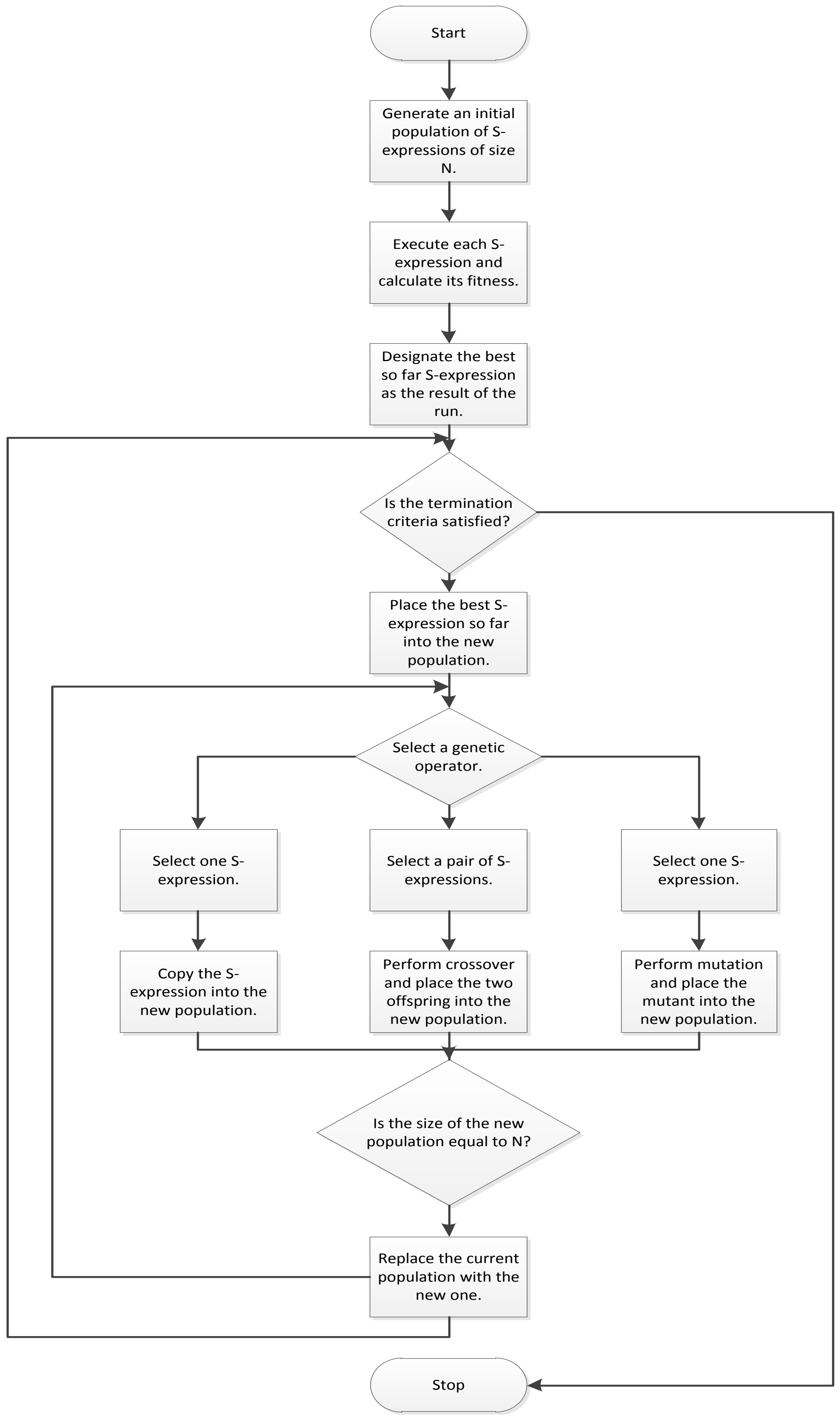
Evaluation and implementation of the differential evolution algorithm was constrained to the following parameters:

Dimensionality	$n = 30$
Population Size	$N_p = 2 \times n$
Mutation Constant	$F = 0.5$
Crossover Rate	$C_r = 0.9$
Maximum Function Calls	$MAX_{NFC} = 1000 \times n$
Runs Per Function	50

Solution Architecture

Each function for optimization was put into a class ($DeJong$, $HyperEllipsoid$, $Schwefel$, $RosenbrocksValley$, $Rastrigin$), which provides methods and operations that interact with the $Vector$ class. The $Vector$ class holds the vector data structure, and the $VectorOperations$ class provides the differential evolution operations on the vectors. Fitness is evaluated within functions provided by the $FitnessFunction$ class, and the differential evolution parameters are stored within the $ControlVariables$ class attributes.

DE Flow Chart


Start
Generate an initial population of S-expressions of size N.
Execute each S-expression and calculate its fitness.
Designate the best so far S-expression as the result of the run.
Is the termination criteria satisfied?
Place the best S-expression so far into the new population.
Select a genetic operator.
Select one S-expression.
Select a pair of S-expressions.
Select one S-expression.
Copy the S-expression into the new population.
Perform crossover and place the two offspring into the new population.
Perform mutation and place the mutant into the new population.
Is the size of the new population equal to N?
Replace the current population with the new one.
Stop

Results and Analysis

Each function was evaluated and a fitness plot was generated, which shows the average best fitness value per generation, plotted over the specified maximum number of generations. The standard deviation was also found.

Function	Average Best Fitness Value	Standard Deviation
De Jong Function	$4.090886027568621 \times 10^{-7}$	$2.8312617965241445 \times 10^{-7}$
Axis Parallel Hyper-Ellipsoid	$6.061343887290475 \times 10^{-6}$	$4.457886494351611 \times 10^{-6}$
Schwefel's Problem	12696.918026780339	4195.459432774405
Rosenbrock's Valley	25.728013835578516	1.2199790548954668
Rastrigin's Function	193.62849833017637	14.36980082622347

De Jong Function

The De Jong function takes the form:

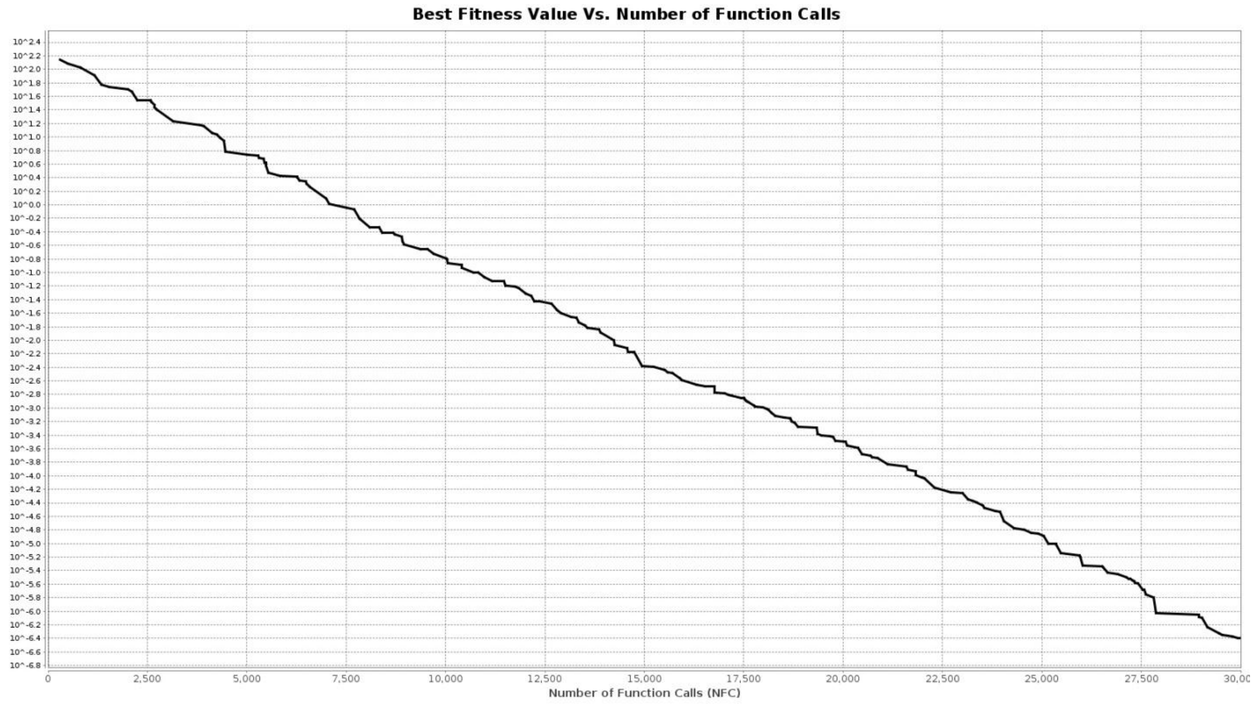
Equation 2 - De Jong Function

$$f_1(X) = \left.\sum_{i=1}^{n} x_i^2 \right\} with - 5.12 \leq x_i \leq 5.12, \min(f_1) = f_1(0, \dots, 0) = 0$$

Output

$Mean$: $4.090886027568621 \times 10^{-7}$

$Standard\ Deviation$: $2.8312617965241445 \times 10^{-7}$



Axis Parallel Hyper-Ellipsoid

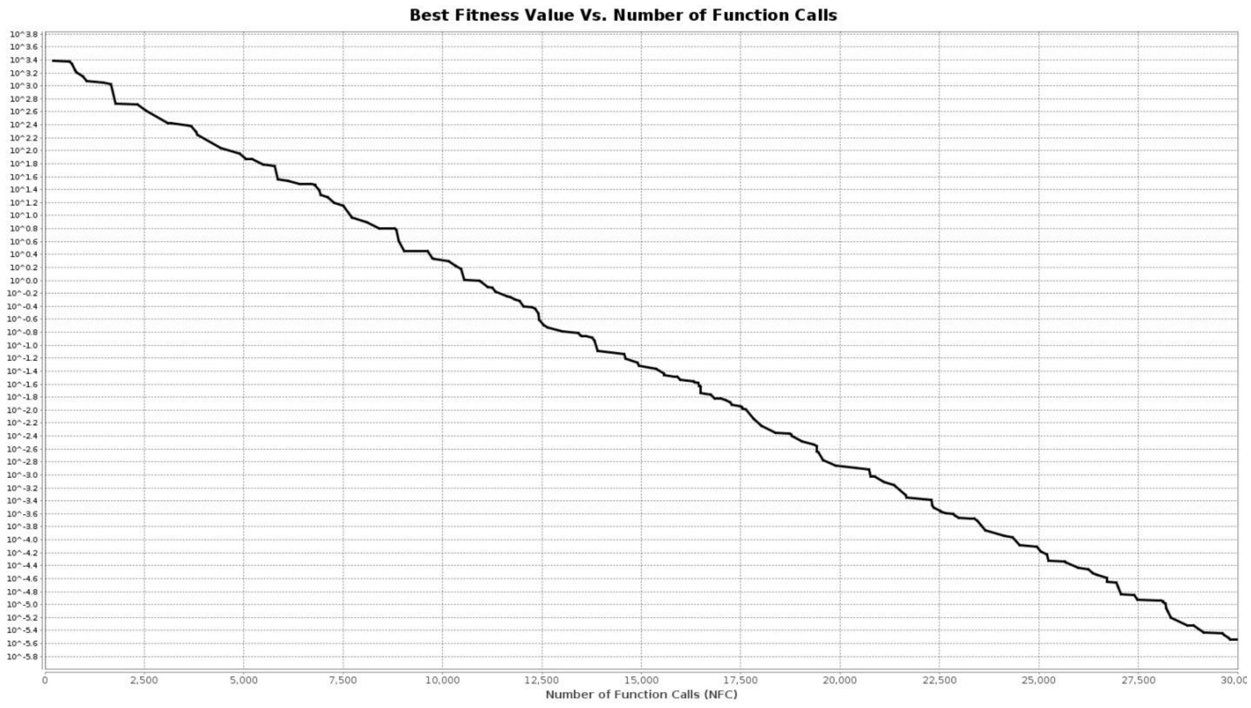
The Axis Parallel Hyper-Ellipsoid takes the form:

Equation 3 - Axis Parallel Hyper-Ellipsoid

$$f_2(X) = \left.\sum_{i=1}^{n} i x_i^2 \right\} with - 5.12 \le x_i \le 5.12, \min(f_2) = f_2(0, \dots, 0) = 0$$

Output

$Mean$: $6.061343887290475 \times 10^{-6}$
$Standard\ Deviation$: $4.457886494351611 \times 10^{-6}$



Schwefel's Problem

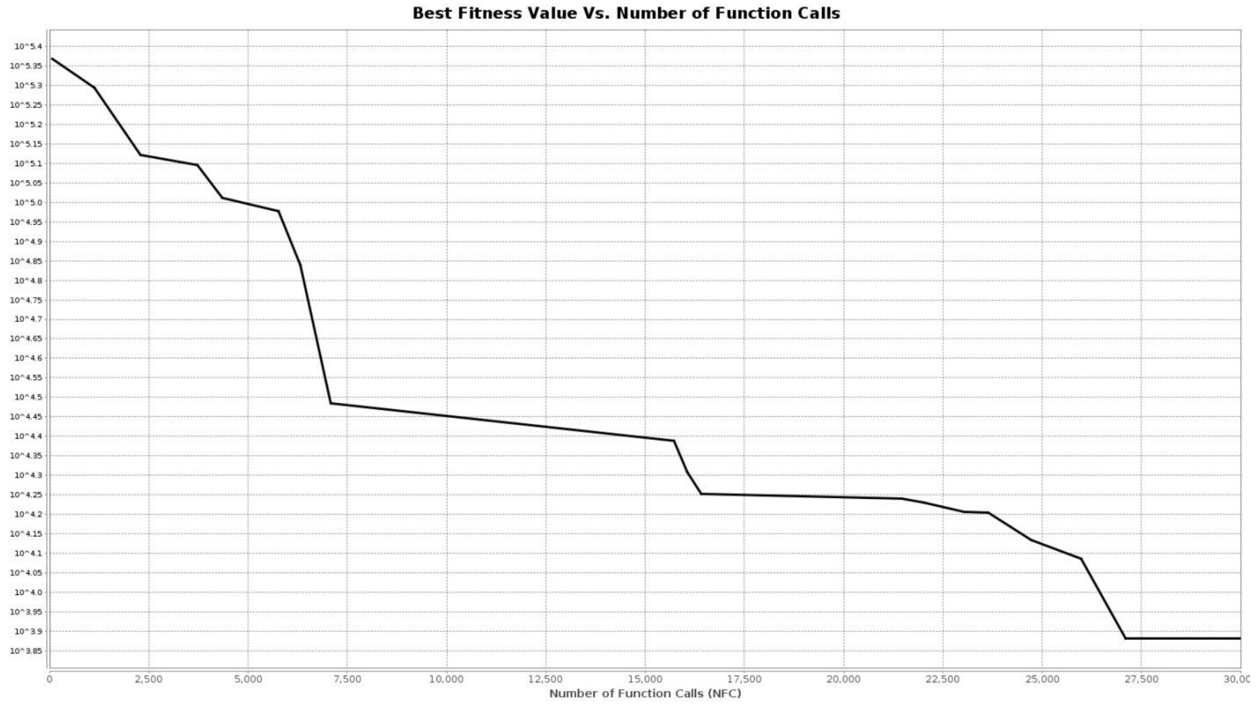
Schwefel's Problem takes the form:

Equation 4 - Schwefel's Problem

$$f_3(X) = \sum_{i=1}^{n}\left(\sum_{j=1}^{i} x_j\right)^2 \Bigg\} \, with - 65 \leq x_i \leq 65, \min(f_3) = f_3(0, \dots, 0) = 0$$

Output

$Mean$: 12696.918026780339
$Standard\ Deviation$: 4195.459432774405



Rosenbrock's Valley

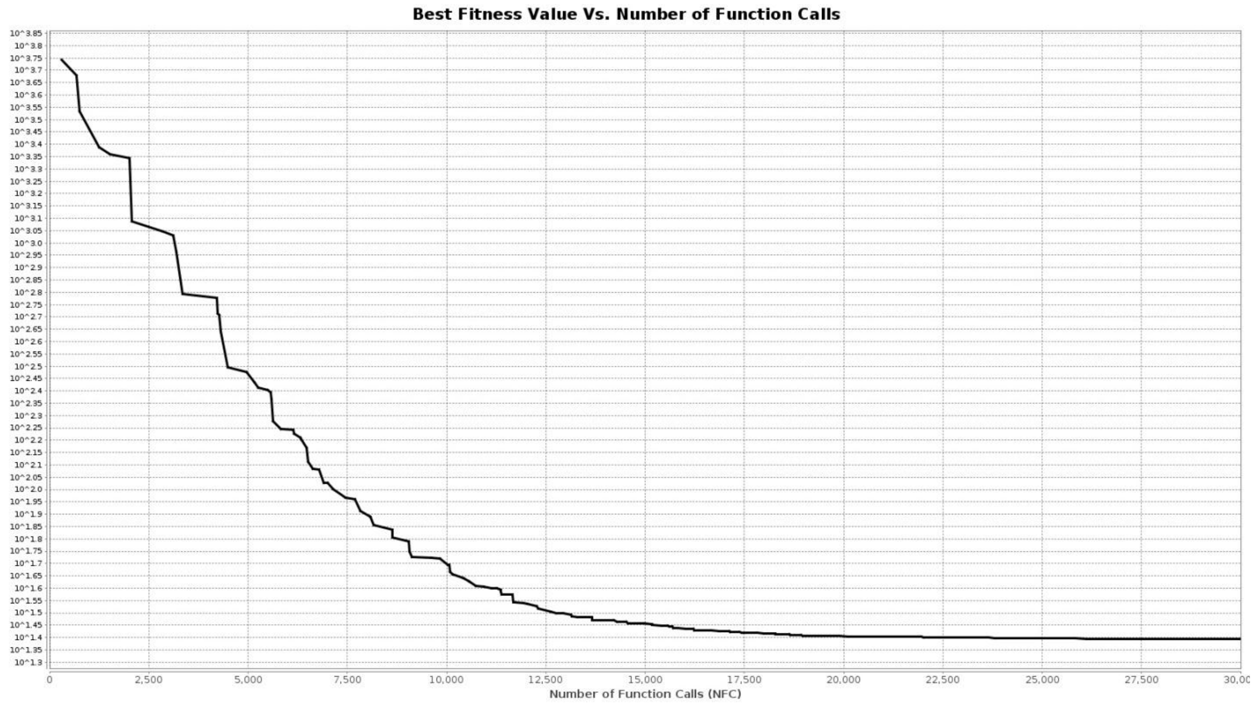
Rosenbrock's Valley takes the form:

Equation 5 - Rosenbrock's Valley

$$f_4(X) = \sum_{i=1}^{n-1} \left[100\left(x_{i+1} - x_i^2\right)^2 + (1 - x_i)^2\right] \Bigg\} \, with - 2 \le x_i \le 2, \min(f_4) = f_4(1, \dots, 1) = 0$$

Output

$Mean$: 25.728013835578516
$Standard\ Deviation$: 1.21997905489546668



Rastrigin’s Function

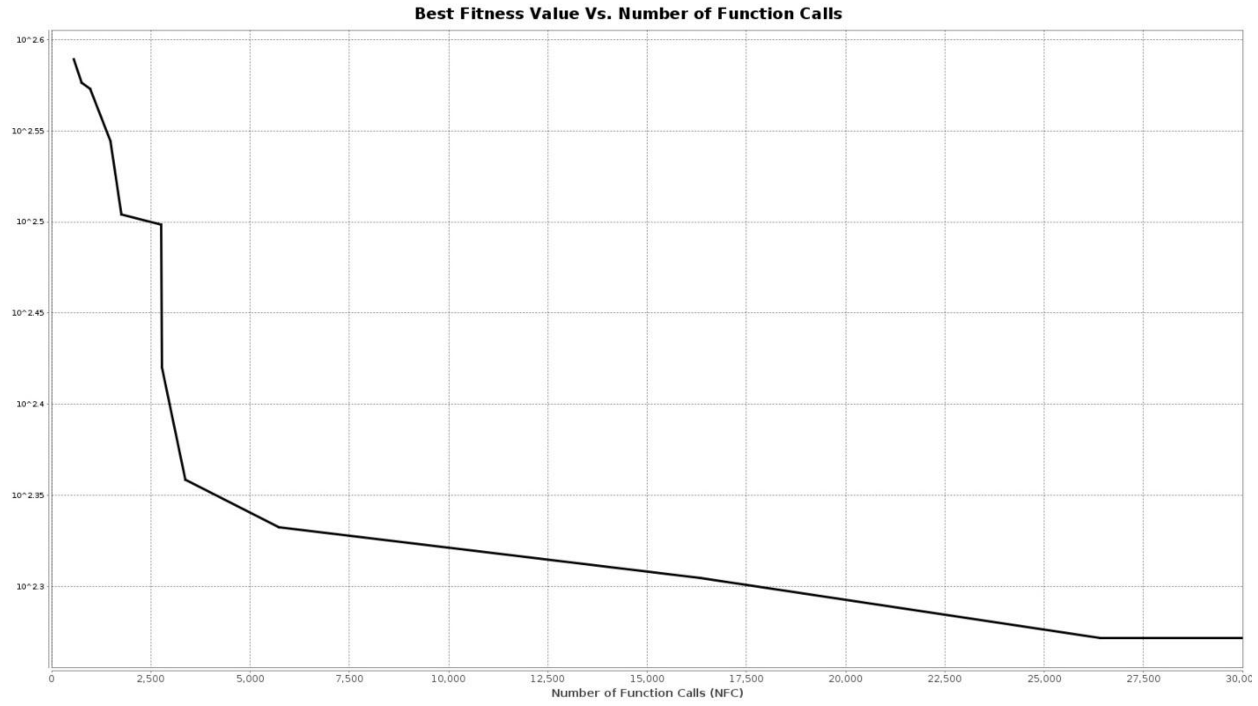
Rastrigin’s Function takes the form:

Equation 6 - Rastrigin's Function

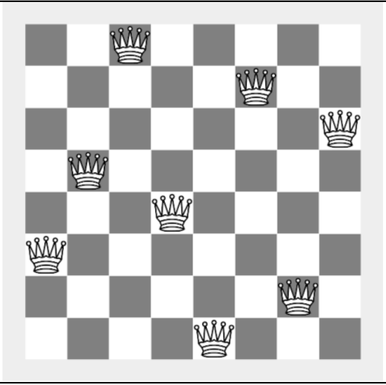
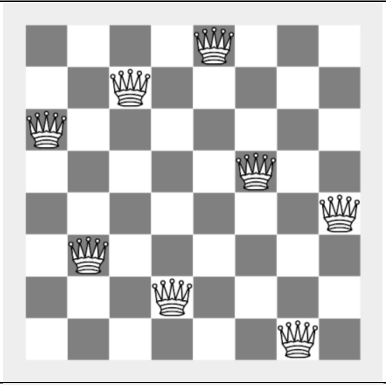
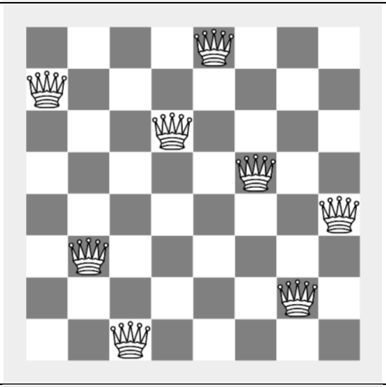
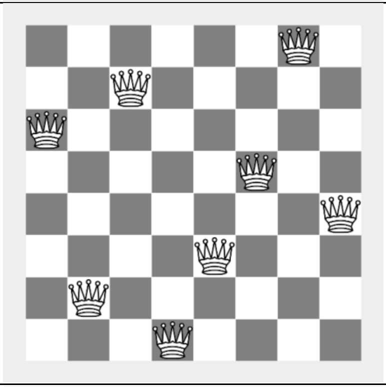
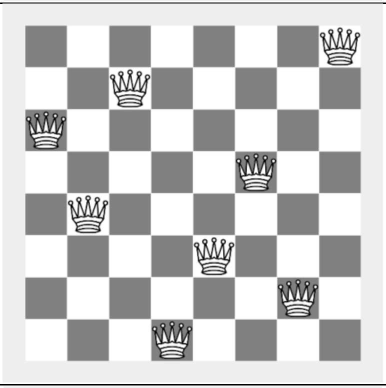
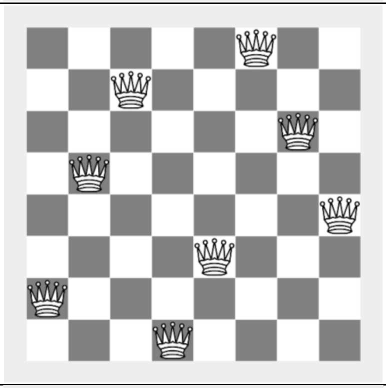
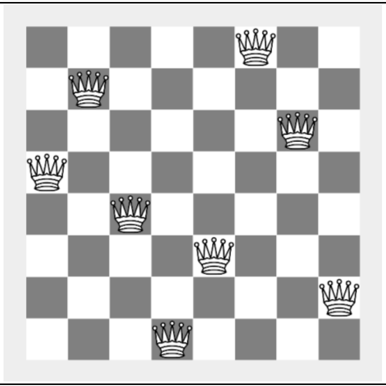
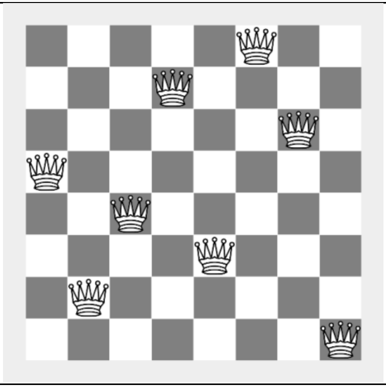
$$f_5(X) = 10n + \sum_{i=1}^{n}\left(x_i^2 - 10\cos(2\pi x_i)\right)\Big\} \; with - 5.12 \leq x_i \leq 5.12, \min(f_5) = f_5(0, \dots, 0) = 0$$

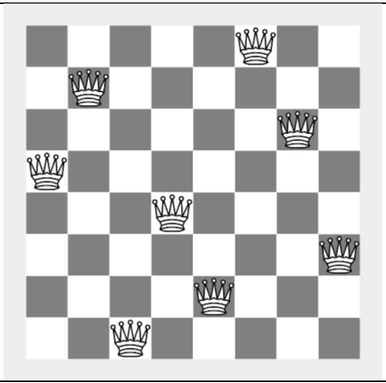
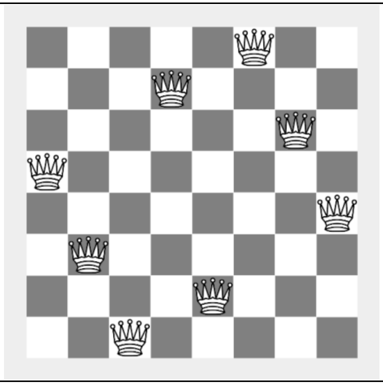
Output

$Mean$: 193.62849833017637
$Standard\ Deviation$: 14.36980082622347



Appendix A – Ten Unique Solutions

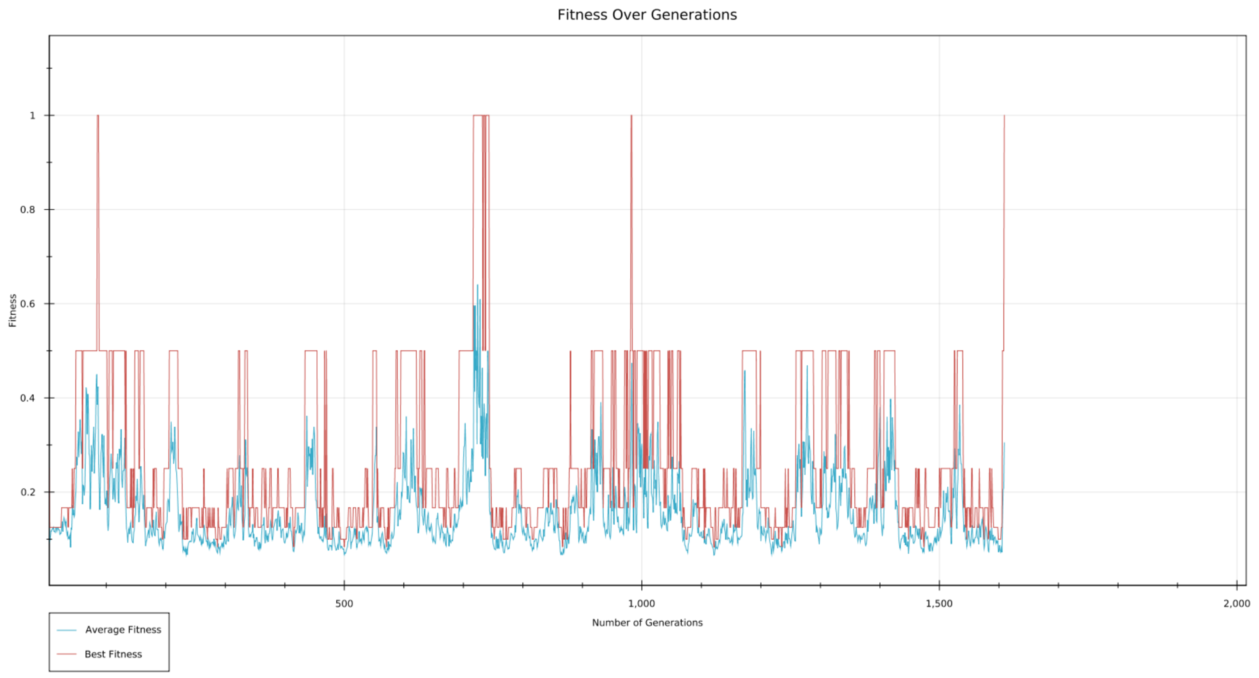


Figure 8 - Fitness over Generations for Ten Unique Solutions