ENGR – 3720U

Intro to Artificial Intelligence
Project Part A - Report

Submitted: March 22, 2013

By: Devan Shah 100428864
Parth Patel 100392782
Ravikumar Patel 100423830

Submitted to: Dr. Shahryar Rahnamayan

Table of Contents

Problem 1 – The Eight Queen Puzzle

Comprehensive explanation of Genetic Algorithm

Chromosome Structure

0	1	0	1	0	0	0	0	1	1	1	1	1	0	1	0	1	1	1	1	0	0	0	0

The chromosome to represent the 8-queens problem domain consists of 24 binary bits. Each gene is represented by 3 indivisible bits, such as the one highlighted above. Each gene represents a decimal number between $0(000_2)$ and $7(111_2)$ which is a column location in an 8 by 8 chess board with a starting location of 0. The location of the gene in the chromosome is the row location of the queen. For example the highlighted gene in the chromosome above represents a queen on the 3^{rd} row and 2^{nd} column on the chessboard with a starting location of 1.

In the implementation, we used integers instead of binary to represent our genes. The numbers are from 0 to 7 which still hold the 3 bit structure in binary. The advantage of using integers instead of binary in this project was so that the encoding and decoding do not complicate the project, since the three bit genes are indivisible the mutation and crossover operators can be performed on integers as described in this report. An array of 8 integers represents a chromosome's genes.

Fitness Calculation

To calculate the fitness of each chromosome, the following fitness function was used:

f(x) = 28 – x
Where 28 is the maximum number of distinct collisions.
And x is the number of distinct collisions in a chromosome.

The maximum fitness of a chromosome would be 28, which means there are zero collisions. Two queens are not colliding if they are not in the same row, column, or diagonal to each other. Each chromosome is assigned a fitness value based on the function above.

Roulette Wheel

The roulette wheel uses the fitness ratios of the chromosome calculated by:

$\text{Fitness ratio}_1 = \text{Fitness}_1 / \text{Fitness}_{\text{total}}$

Where fitness total is the sum of the fitness values of all the chromosomes in the current population. The fitness ratios give the chromosomes with high fitness more chance to be selected.

Crossover Operator

The crossover operator is performed if the random number chosen between 0 and 1 exclusive is greater than or equal to the mutation probability. In this case the optimal probability value is 0.7. The crossover operator randomly chooses a break point between the two parent chromosomes and then exchanges the genes after that point. If the random number is lower than the crossover probability then the two parents are cloned as the offspring. The two parents are chosen using the roulette wheel method. This gives the parents with higher fitness ratios more chances to be selected. The implementation of crossover is as follows:

Chosen by roulette wheel:
Parent 1 = [0, 4, 5, 7, 2, 1, 6, 3]
Parent 2 = [6, 4, 3, 1, 7, 2, 0, 5]

If (p_c>=random): crossover

Offspring 1 = [0, 4, 5, 7, 7, 2, 0, 5]
Offspring 2 = [6, 4, 3, 1, 2, 1, 6, 3]

Else: clone

Offspring 1 = [0, 4, 5, 7, 2, 1, 6, 3]
Offspring 2 = [6, 4, 3, 1, 7, 2, 0, 5]

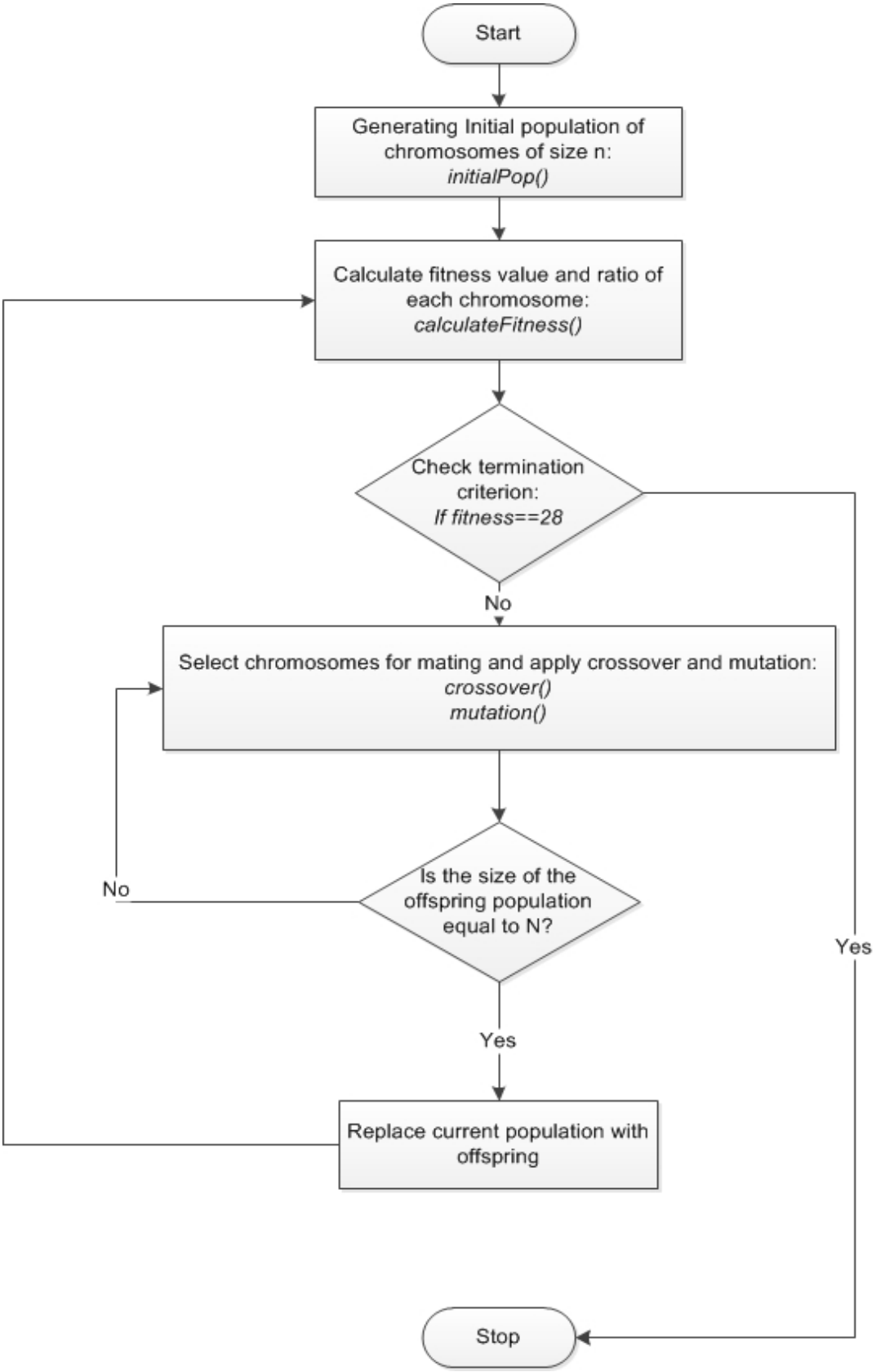
Mutation Operator

The mutation operator selects a random gene in a chromosome and flips it. The role of the mutation operator is to provide a guarantee that the search algorithm is not trapped on a local optimum. In this case it selects a random gene and changes it to a random gene value ranging from 0-7. Mutation probability is the chances of the mutation taking place. An example of the implementation of mutation operator is as follows:

Chromosome = [0, 4, 5, 7, 2, 1, 6, 3]
Random gene = 6;
Random chromosome index = 1

Mutated [0, 6, 5, 7, 2, 1, 6, 3]

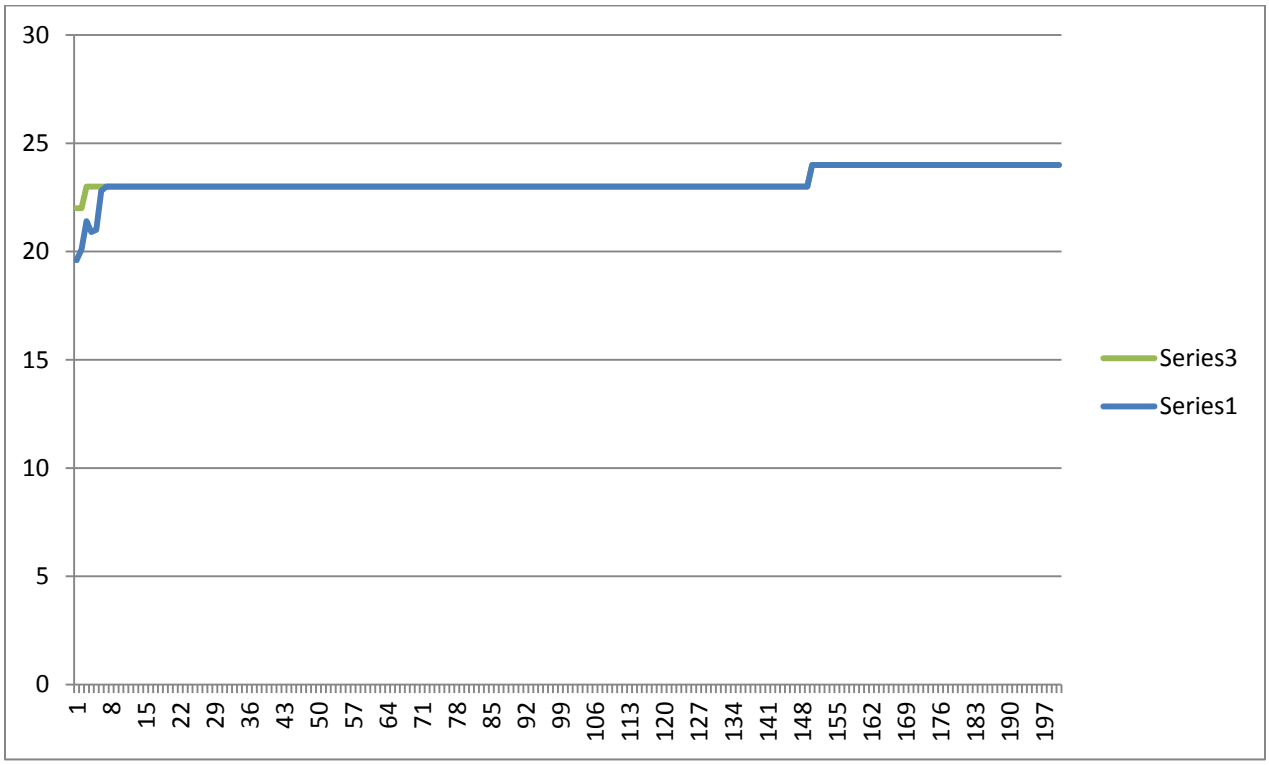
Implementation

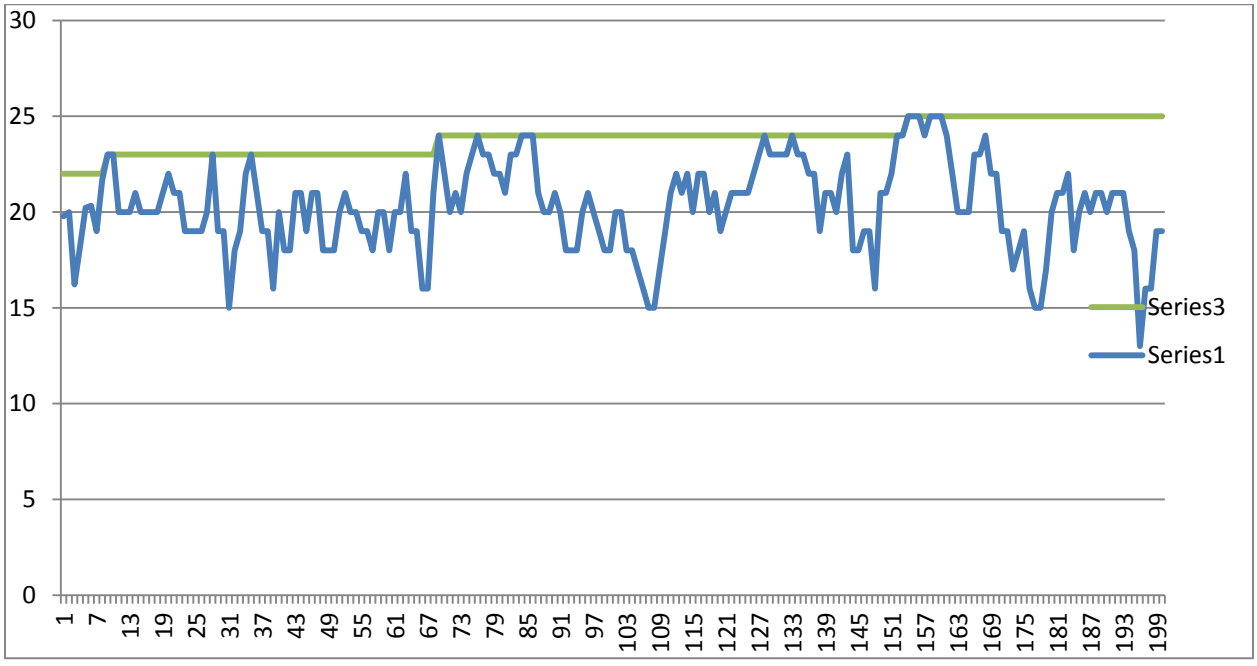


Results

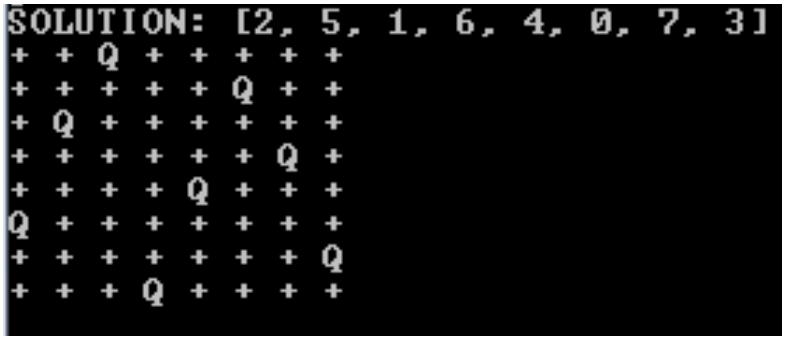
Performance Graphs

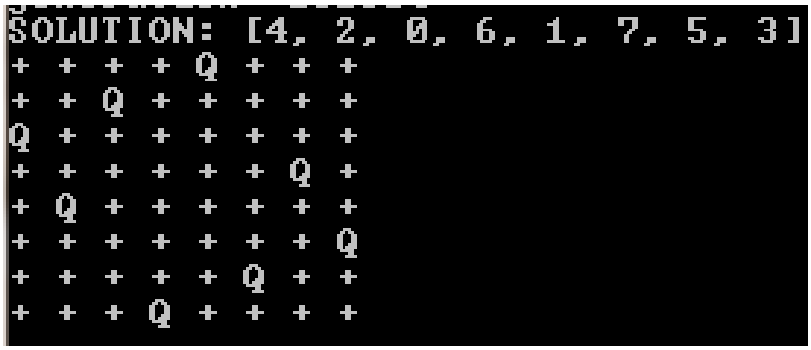
The following graphs show the difference in performance when varying mutation. The first graph has a low mutation and you can see the chromosome trapped in a local optimum. The second graph has a high mutation probability. The green line is the best fitness, and the blue is the average fitness of the chromosomes. The graphs are plotted for 200 generations.

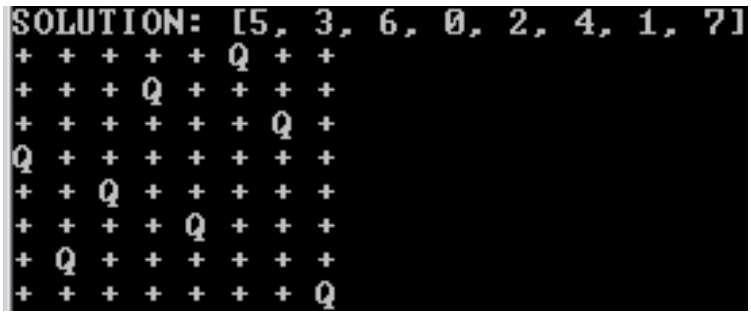


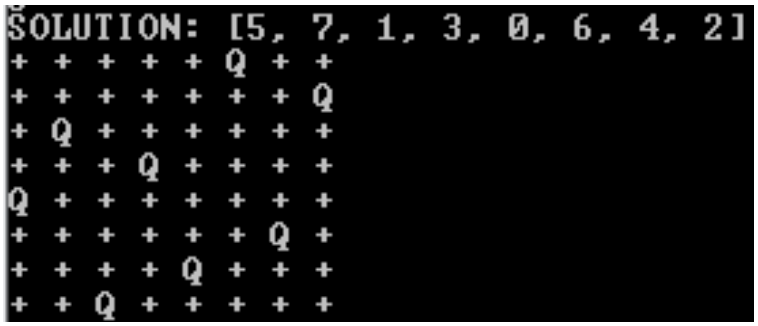


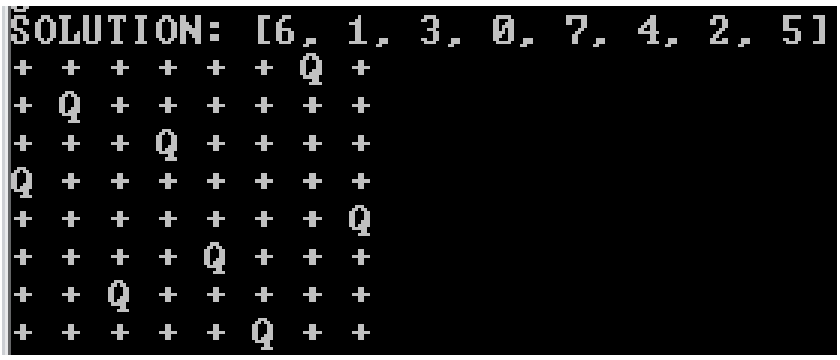
10 Solutions

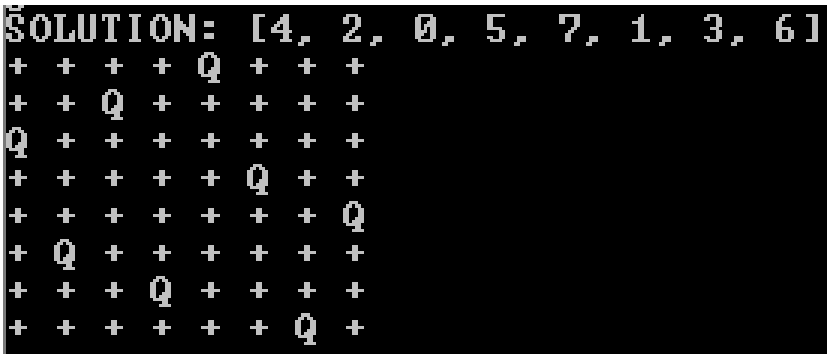


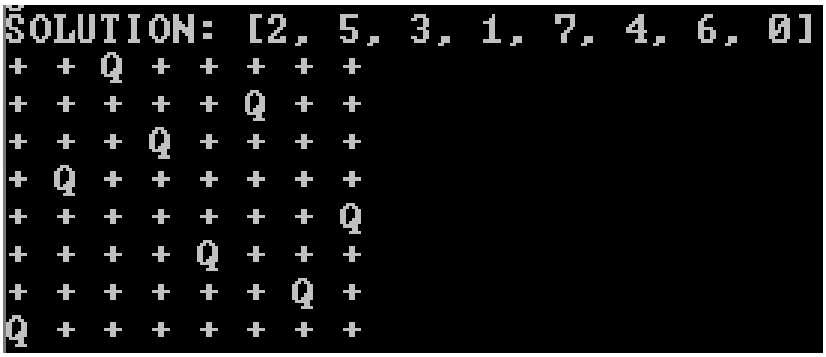


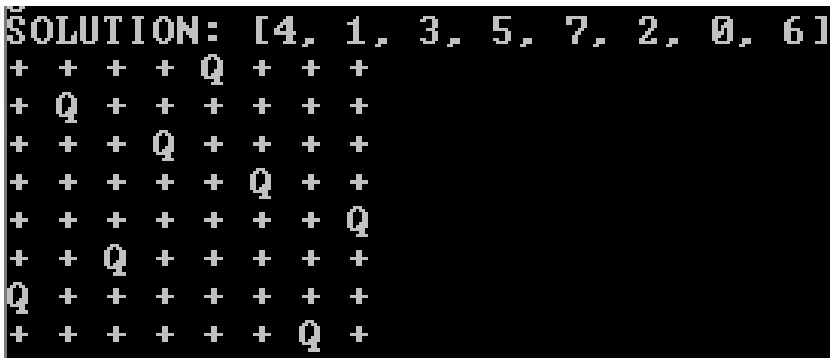


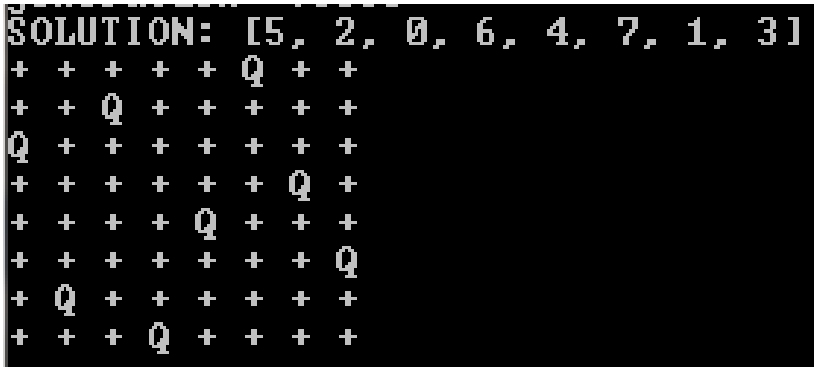


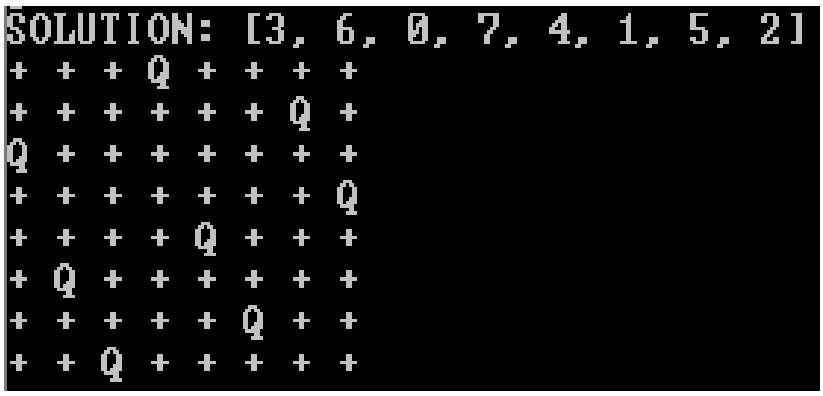

SOLUTION: [6, 1, 3, 0, 7, 4, 2, 5]
+ + + + + + Q +
+ Q + + + + + +
+ + + Q + + + +
Q + + + + + + +
+ + + + + + + Q
+ + + + Q + + +
+ + Q + + + + +
+ + + + + Q + +


SOLUTION: [4, 2, 0, 5, 7, 1, 3, 6]
+ + + + Q + + +
+ + Q + + + + +
Q + + + + + + +
+ + + + + Q + +
+ + + + + + + Q
+ Q + + + + + +
+ + + Q + + + +
+ + + + + + Q +


SOLUTION: [2, 5, 3, 1, 7, 4, 6, 0]
+ + Q + + + + +
+ + + + + Q + +
+ + + Q + + + +
+ Q + + + + + +
+ + + + + + + Q
+ + + + Q + + +
+ + + + + + Q +
Q + + + + + + +


SOLUTION: [4, 1, 3, 5, 7, 2, 0, 6]
+ + + + Q + + +
+ Q + + + + + +
+ + + Q + + + +
+ + + + + Q + +
+ + + + + + + Q
+ + Q + + + + +
Q + + + + + + +
+ + + + + + Q +





Conclusion

Using the genetic algorithm approach we were successful in finding solutions to the 8-queens problem. Each execution of the program yields a different solution. Isolating each GA parameter yielded different reactions as to how the algorithm would respond. An example is changing the mutation probability as discussed above. The optimal GA parameters after tuning were population size of 30, crossover probability of 0.7, and a mutation probability of 0.12. The number of generations/iterations before a solution is found can range approximately from 100k-400k on average. Although the iterations could be very high, the time of completion is often under 60 seconds.

Problem 2 – Differential Evolution

Comprehensive explanation of Differential Evolution

The differential evolution consists of certain steps that need to be followed to achieve the correct solution. The steps include initial population generation, fitness calculation, selecting 3 parents, applying mutation, applying crossover and selection between initial population and mutated population.

Step 1: Generating the initial population depending on the problem dimensionality and also the problem size.

Step 2: Calculate the fitness value of each of the individuals that are in the initial population.

Step 3: Select the first individual in the initial population, which is going to run through the differential evolution algorithm.

Step 4: Select three random parent individuals from the initial population that are not the same and also not the individual that that the differential evolution algorithm is being applied on.

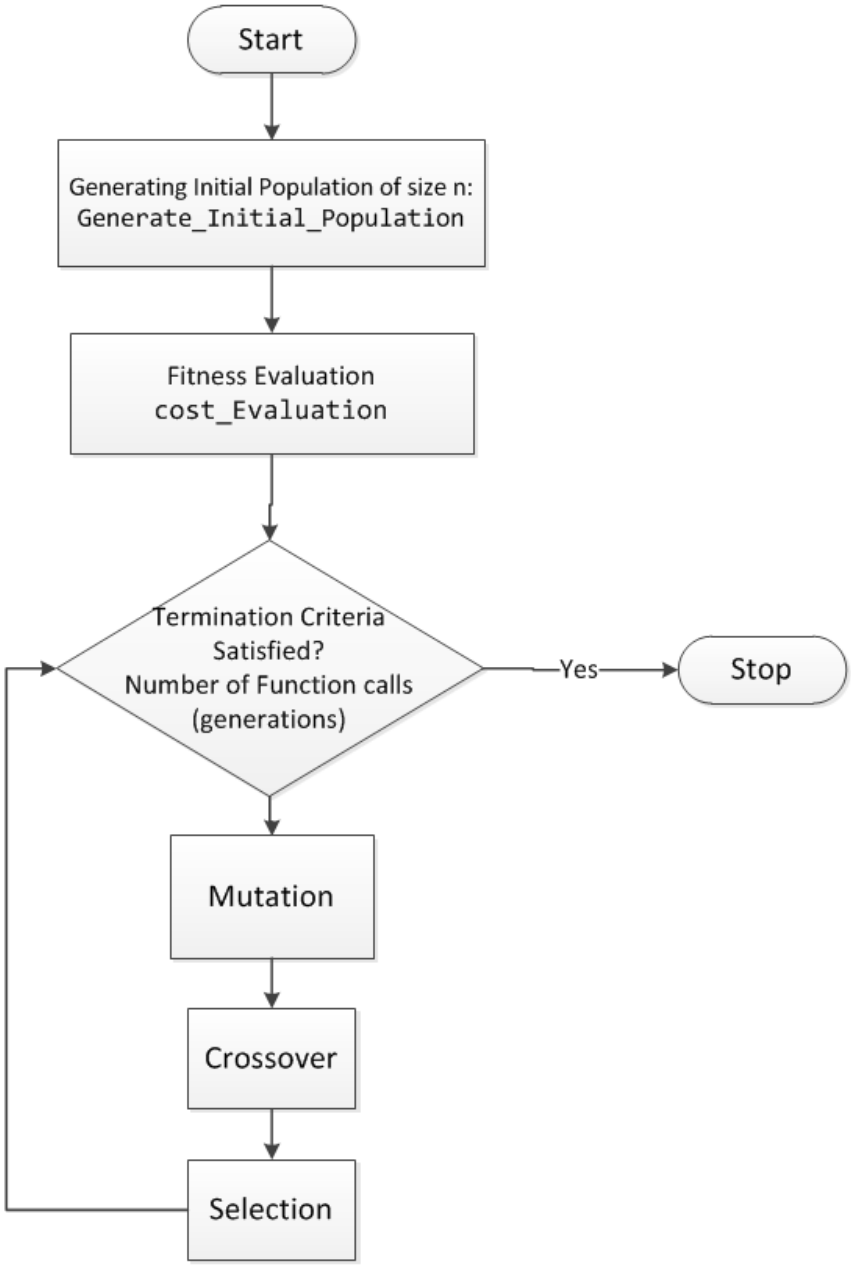
Step 5: Apply mutation on the 3 selected parents from step 4 with the use of the mutation equation of vi <- Xa + F * (Xc - Xb) where vi is the noise vector, F is the mutation constant and Xa,Xb and Xc are the randomly generated parents individuals. Store the mutated population in the noise vector for crossover to be applied on to it.

Step 6: Apply crossover on the noise vector that was retrieved from step 5. Crossover is calculated using a comparison between a randomly generated number between 0 and 1 to the crossover rate; crossover is applied to every parameter in the individual. Also since crossover has to occur at least once there is a second condition for crossover which is generating another number between the 1 and the problem dimensionality and setting it equal to a variable j. So if any of those 2 conditions are true then the element of the noise vector is selected and put into a trial vector, else the parameter from the initial population is selected and put into the trial vector. Once all of the elements have been checked and selected according and the trial vector is filled.

Step 7: Apply Selection between the trial vector and the initial population individual. The selection is done by calculating the cost value of the trial vector and comparing the cost value to the cost value of the initial population. Since the algorithm is set up to minimize the problem we check to see if the cost of the trial vector is smaller than or equal to that of the initial population, if it is then the trial vector selected and added to the new population, else the initial population individual is left unchanged.

We have implemented a similar way in out differential evolution algorithm; the only difference is the cost calculation of the initial population. In the original differential evolution the cost is calculated right after the initial population has been generated, we decided to calculate the cost of the initial population within the selection operation which is step 7, so that the comparison between the trial vector cost and the initial cost can be handled properly.

Implementation



Results

a) Solving Five Benchmark Functions

Benchmark Function	Average Best Fitness Value (over 50 runs)	Standard Deviation (over 50 runs)
f1 – 1st De Jong	0.00000	0.00000
f2 – Axis Parallel Hyper-Ellipsoid	0.00000	0.00000
f3 – Schwefel's Problem 1.2	0.00000	0.00000
f4 – Rosenbrock's Valley	0.00002	0.00011
f5 – Rastrigin's Function	22.65504	0.00000

Analysis

Function 1, Function 2 and Function 3

The 1st De Jong , Axis parallel Hyper-Ellipsoid, Schwefel's Problem 1.2 functions had an average best fitness value of 0 after 50 runs this shows that the functions are reaching the global minimum using differential evolution. The standard deviation shows that the average best fitness value is reached in every run of the algorithm. This shows that the differential evolution algorithm is functioning as expected, to retrieve the optimal solution of a problem.

Function 4

The Rosenbrock's Valley had an average best fitness value of 0.00002 after 50 runs, this shows that the function is reaching the global minimum but needs more generations to achieve. Currently the implemented differential evolution algorithm is using a problem dimensionality of 30, resulting in 30000 generations, which is not enough for this function to reach the global minimum. The standard deviation shows that in some of the runs the global optimal could have been reached, but just not enough times to get a result of 0 as the average best fitness value.

Function 5

The Rastrigin's function had an average best fitness value of 22.65504 after 50 runs, this shows that the function is approaching a global minimum but require more generations to achieve. The current implementation of the differential evolution algorithm does not allow more than 30000 generations per run which results in the function not reaching the global minimum. Since the standard deviation is 0 is this cases that means that never in the 50 runs does the function shift from the average best fitness value.

b) Performance Graph

Function 1 – 1st De Jong

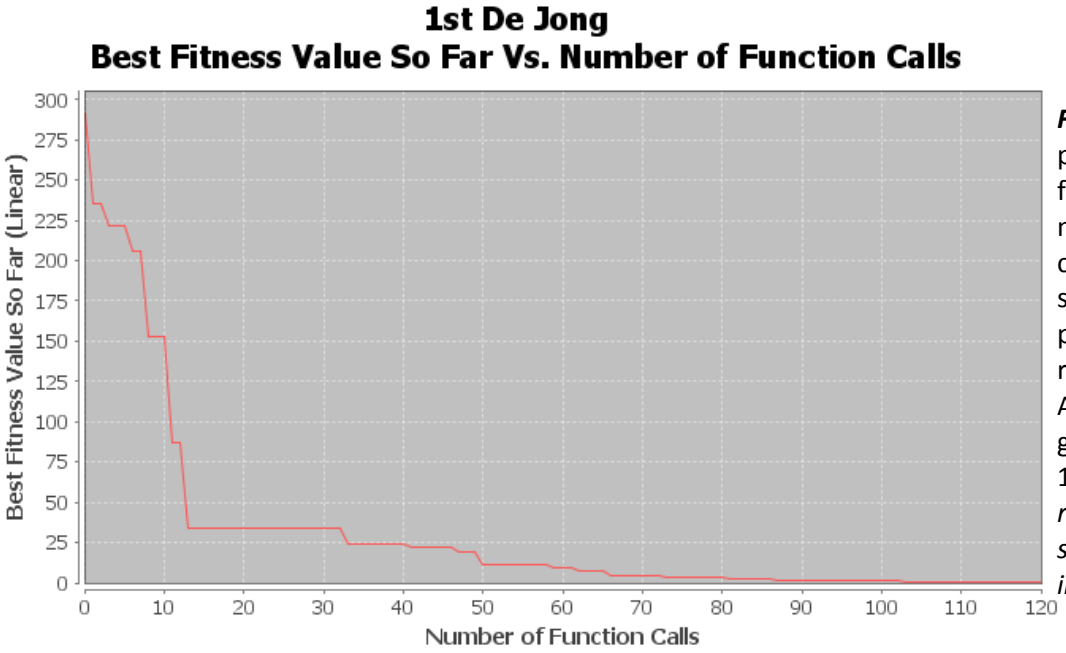
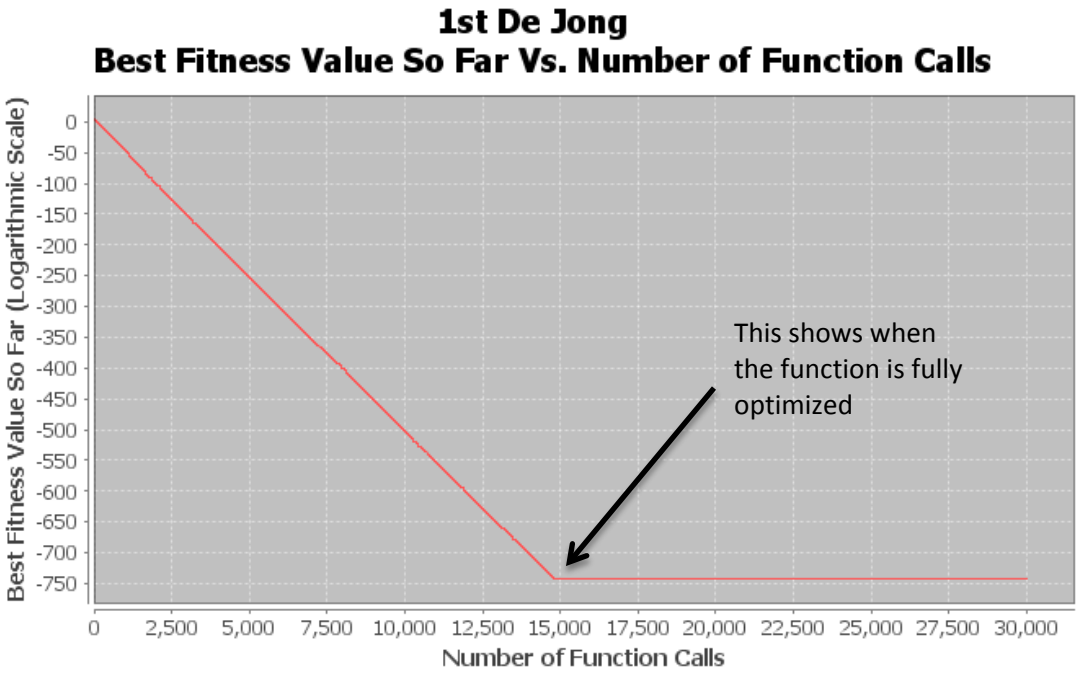


Figure 1: This represents the linear performance graph of the 1st De Jong function. The x-axis consists of the number of function calls and y-axis consists of the best fitness value of a single function call. For this specific performance graph we adjusted the range so that the e^{-x} can be seen clearly. Also from this graphs we can see that the graphs is approaching 0 starting at around 103 function calls. *To see the full graph refer to the attached jar file to run the simulation, with the instructions provided in the readme file.*

Figure 2: This represents the logarithmic performance graph of the 1st De Jong function. The x-axis consists of the number of function calls and y-axis consists of the log of the best fitness value from ***figure 1***. For this specific performance graph it is clearly seen that the graph is in fact e^{-x}, as seen in the graph on the right the log of the above graph gives a straight line showing that it is e^{-x}. The log graph also shows that the 1st De Jong function reaches a minimum value of 0.0.



Analysis

The differential evolution algorithm that was applied on this function was meant to optimize the function to its global minimum of 0. The time taken for the 1st De Jong function to optimize was approximately 18s, with a complexity of O (n). From ***figure 1,*** the 1st De Jong function is gradually approaching the global minimum of 0. Therefor we concluded that the 1st De Jong function reached the global minimum after 14500 function calls seen in ***figure 2***, the log graph starts to become a straight line meaning that at those points the y value is 0 and a log of 0 is –infinity so the graph just plots the previous value until all the function calls are completed.

Function 2 – Axis Parallel Hyper-Ellipsoid

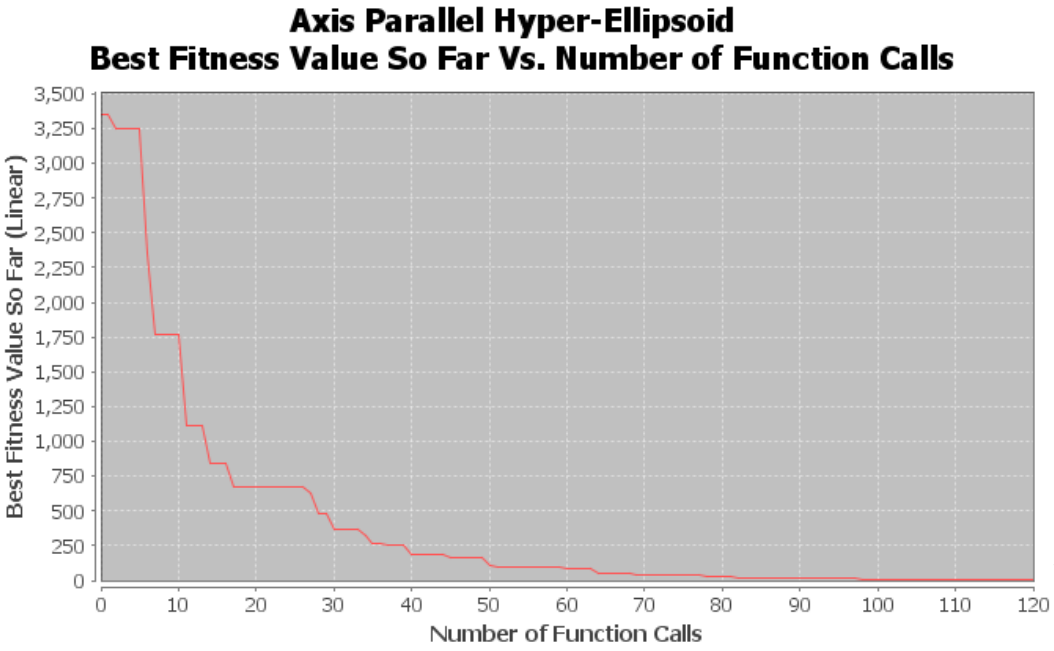
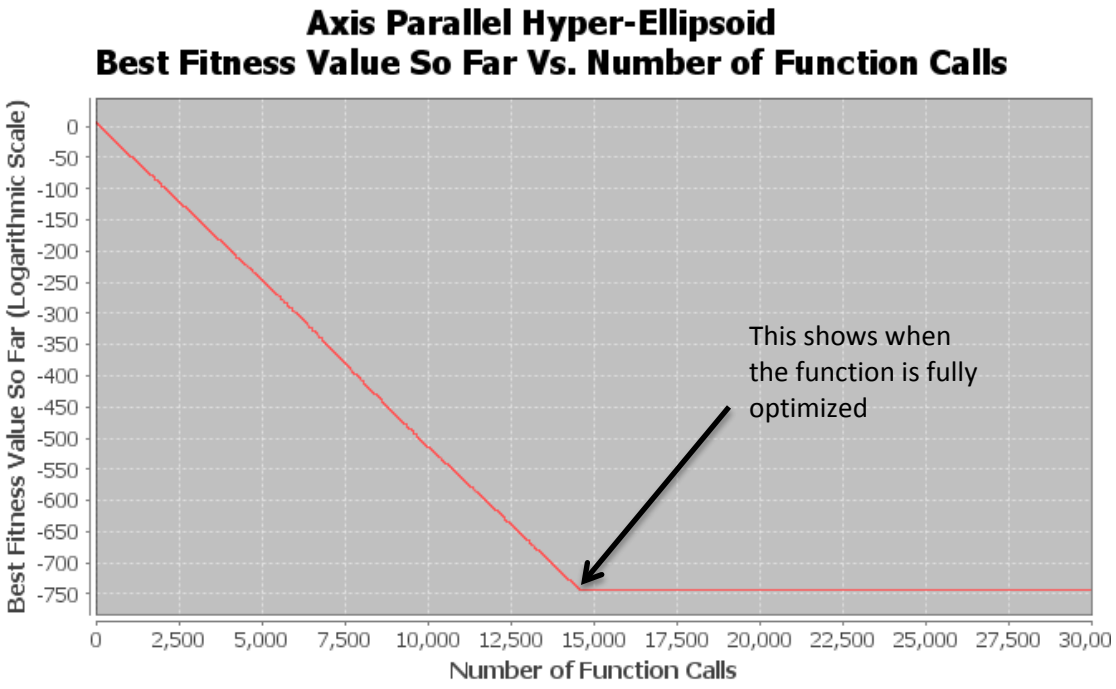


Figure 3: This represents the linear performance graph of the Axis Parallel Hyper-Ellipsoid function. The x-axis consists of the number of function calls and y-axis consists of the best fitness value of a single function call. For this specific performance graph we adjusted the range so that the e^{-x} can be seen clearly. Also from this graphs we can see that the graphs is approaching 0 at around starting at 97 function calls. *To see the full graph refer to the attached jar file to run the simulation, with the instructions provided in the readme file.*

Figure 4: This represents the logarithmic performance graph of the Axis Parallel Hyper-Ellipsoid function. The x-axis consists of the number of function calls and y-axis consists of the log of the best fitness value from ***figure 3***. For this specific performance graph it is clearly seen that the graph is in fact e^{-x}, as seen in the graph on the right the log of the above graph gives a straight line showing that it is e^{-x}. The log graph also shows that the function reaches a global minimum at 14280 function calls where the graph starts to straighten out.



Analysis

The differential evolution algorithm that was applied on this function was meant to optimize the function to its global minimum of 0. It took approximately 18 seconds to optimize the Axis parallel Hyper-Ellipsoid function using differential evolution, also has a complexity of O (n). From ***figure 3,*** it is clearly seen that the Axis Parallel Hyper-Ellipsoid function is gradually approaching the global minimum of 0. Therefor we concluded that the Axis Parallel Hyper-Ellipsoid function reached the global minimum after 14280 function calls seen in ***figure 4***, the log graph starts to become a straight line meaning that at those points the y value is 0 and a log of 0 is –infinity so the graph just plots the previous value until all the function calls are completed.

Function 3 – Schwefel's Problem 1.2

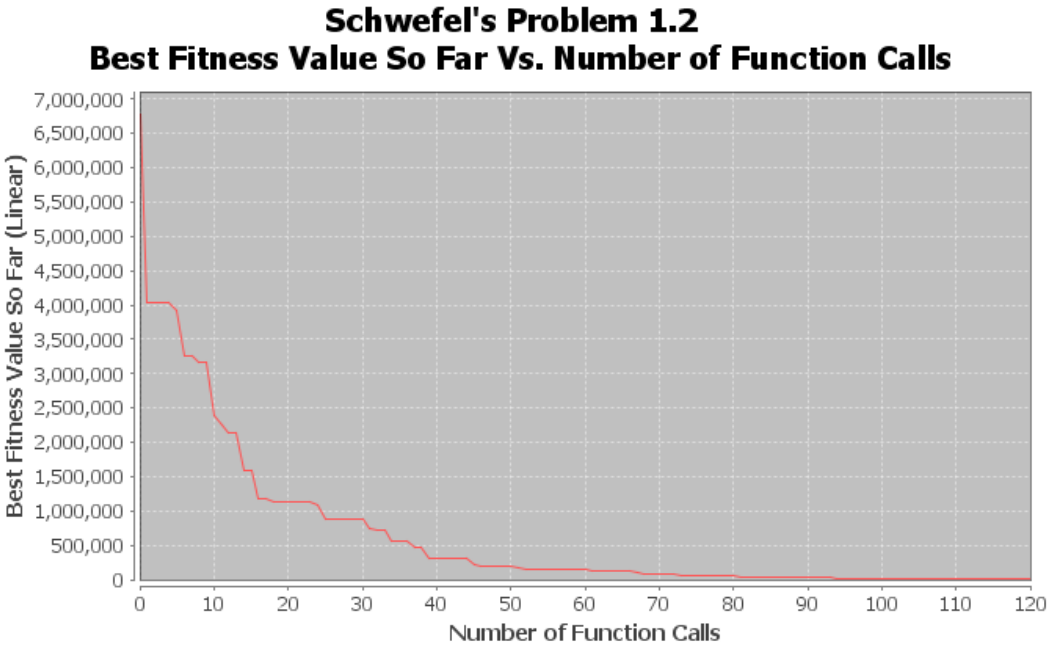
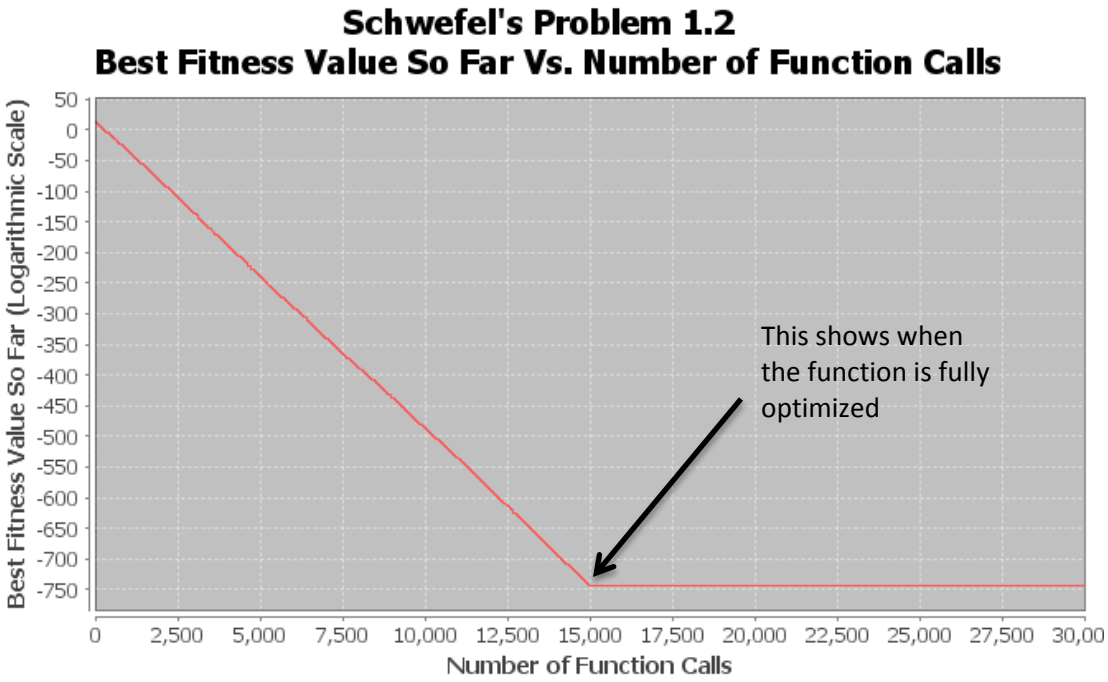


Figure 5: This represents the linear performance graph of the Schwefel's Problem 1.2 function. The x-axis consists of the number of function calls and y-axis consists of the best fitness value of a single function call. For this specific performance graph we adjusted the range so that the e^{-x} can be seen clearly. Also from this graphs we can see that the graphs is approaching 0 starting at around 94 function calls. *To see the full graph refer to the attached jar file to run the simulation, with the instructions provided in the readme file.*

Figure 6: This represents the logarithmic performance graph of the Schwefel's Problem 1.2 function. The x-axis consists of the number of function calls and y-axis consists of the log of the best fitness value from ***figure 5***. For this specific performance graph it is clearly seen that the graph is in fact e^{-x}, as seen in the graph on the right the log of the above graph gives a straight line showing that it is e^{-x}. The log graph also shows that the function reaches a global minimum at 15000 function calls where the graph starts to straighten out.



Analysis

The differential evolution algorithm that was applied on this function was meant to optimize the function to its global minimum of 0. It took approximately 25 seconds to optimize the Schwefel's problem 1.2 function using differential evolution, also has a complexity of O (n^2). From ***figure 5,*** it is clearly seen that the Schwefel's problem 1.2 is gradually approaching the global minimum of 0. Therefor we concluded that the Schwefel's problem 1.2 has reached the global minimum after 14280 function calls seen in ***figure 6***, the log graph starts to become a straight line meaning that at those points the y value is 0 and a log of 0 is –infinity so the graph just plots the previous value until all the function calls are completed. This particular function has a complexity of O(n^2), to compute the fitness value of a set of points.

Function 4 – Rosenbrock's Valley

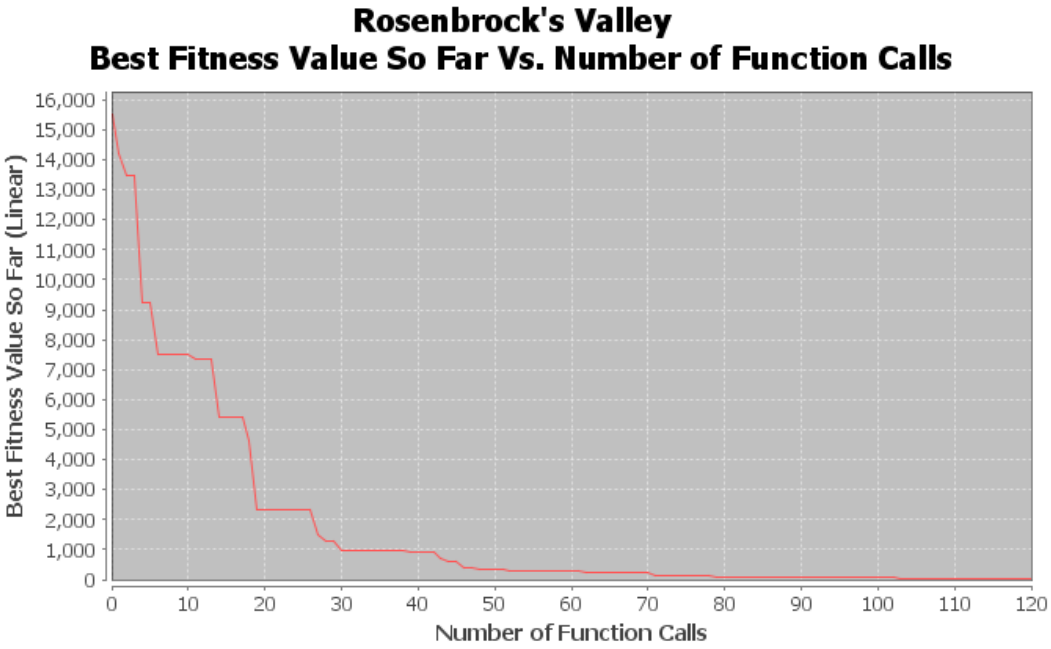
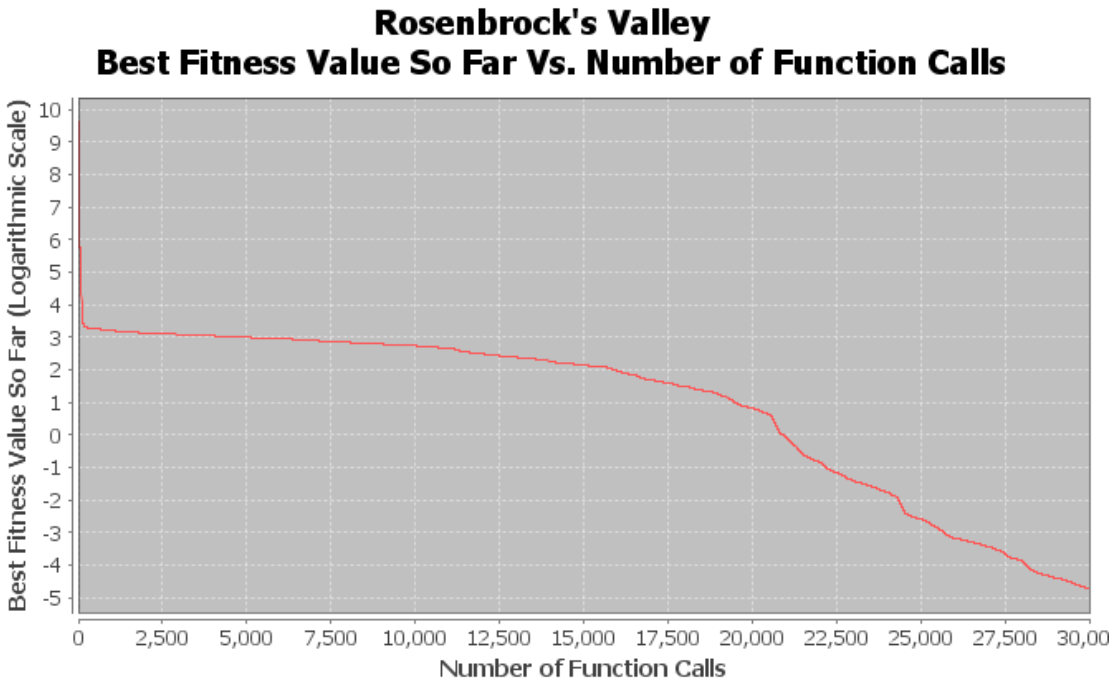


Figure 7: This represents the linear performance graph of the Rosenbrock's Valley function. The x-axis consists of the number of function calls and y-axis consists of the best fitness value of a single function call. For this specific performance graph we adjusted the range so that the e^{-x} can be seen clearly. The Rosenbrock's Valley graph does not start to minimize to 0 after 30000 function calls. *To see the full graph refer to the attached jar file to run the simulation, with the instructions provided in the readme file.*

Figure 8: This represents the logarithmic performance graph of the Rosenbrock's Valley function. The x-axis consists of the number of function calls and y-axis consists of the log of the best fitness value from ***figure 7***. For this specific performance graph the optimal minimum is being reached as seen to the right the graph is slowly dipping towards the negative showing that the optimal is in fact reached.



Analysis

The differential evolution algorithm that was applied on this function was meant to optimize the function to its global minimum of 0. It took approximately 22 seconds to optimize the Rosenbrock's Valley function using differential evolution, also has a complexity of O (n). From ***figure 7,*** the Rosenbrock's Valley function is gradually approaching the global minimum of 0. In ***figure 8,*** the log graph for this functions is pretty hard to distinguish, but by looking at the tremendous falling of the graph shows that the functions is reaching its global minimum. The optimal is reached when the fitness values reaches o for this function that occurs when the value inputted in to the function is 1. Looking at the ***figure 7,*** the initial fitness value it tremendous high and as the function calls go buy the fitness value slowly reached 0.

Function 5 – Rastrigin's Function

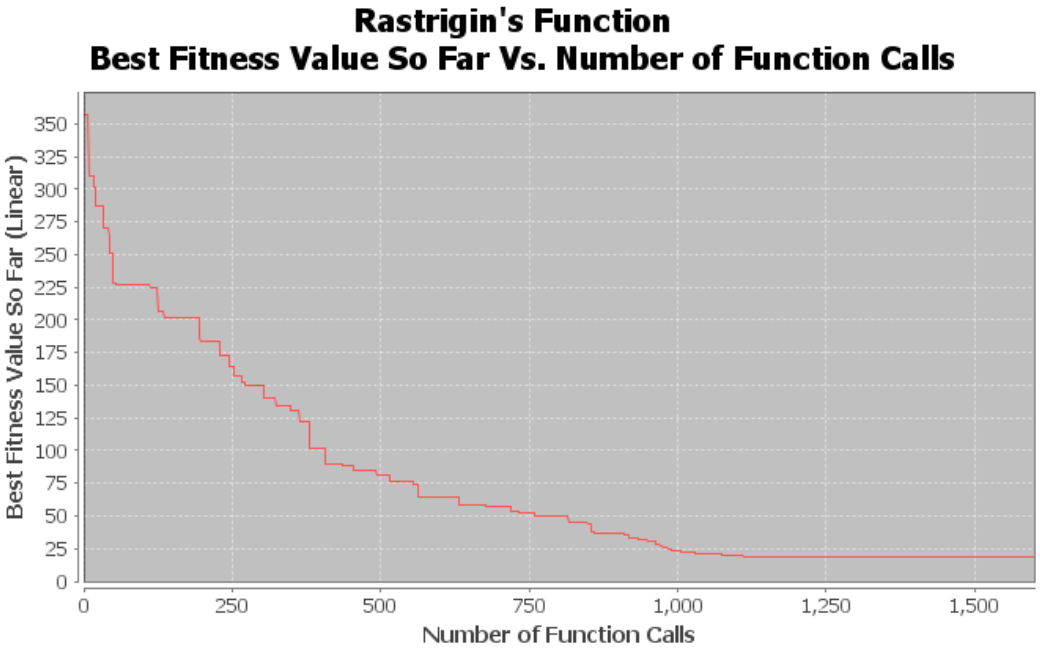


Figure 9: This represents the linear performance graph of the Rastrigin's function. The x-axis consists of the number of function calls and y-axis consists of the best fitness value of a single function call. For this specific performance graph we adjusted the range so that the e^{-x} can be seen clearly. The Rastrigin's function graph does not minimize in this image. *To see the full graph refer to the attached jar file to run the simulation, with the instructions provided in the readme file.*

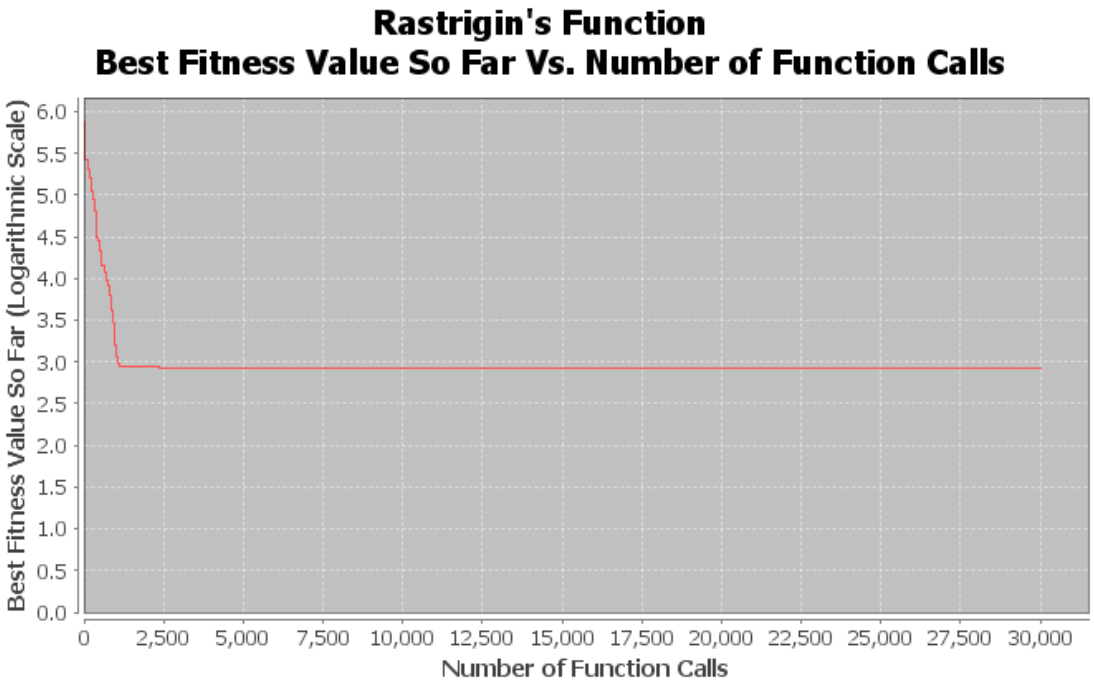


Figure 10: This represents the logarithmic performance graph of the Rastrigin's function. The x-axis consists of the number of function calls and y-axis consists of the log of the best fitness value from ***figure 9***. For this specific performance graph the optimal minimum is in fact never reached it continues to stay at a fixed fitness values throughout all of the runs.

Analysis

The differential evolution algorithm that was applied on this function was meant to optimize the function to its global minimum of 0. It took approximately 35 seconds to optimize the Rastrigin's function using differential evolution, also has a complexity of O (n). Using differential evolution, with only a problem dimensionality of 30 was not enough to get the Rastrigin's function to the global minimum. This can be due to the fact that the function is so complex and the minimum peaks are highly unlikely to reach using only a problem dimensionality of 30.

Conclusion

In conclusion, using the differential evolution algorithm to solve benchmark functions and try to retrieve the global minimum value for each function was a good test to see if the algorithm was functioning correctly. When the differential evolution algorithm was applied to the functions it was seen that some of the functions did reach the global minimum value as expected. Some function did not reach the global minimum because the differential evolution algorithm that was applied was only of problem dimensionality of 30. As seen in all of the graphs that were mentioned above that the initial population contained a high fitness value and as the differential evolution was applied all the fitness values started to reach a global minimum. This was accomplished with the use of the mutation, crossover and selection operations.