Performance of Genetic Algorithm on Benchmark Functions

Sanjeev Shrestha
University of Idaho
Received 12 February 2014

ABSTRACT
In the field of artificial intelligence, evolutionary computing has become a promising research area for solving various kinds of problems. Genetic Algorithm is the branch of evolutionary computing that uses the phenomenon of natural selection for searching solutions to a given problem. In this report, I have covered the basic implementation features for representing a genetic algorithm .The methodologies such as steady state selection, uniform crossover, creep mutation and tournament selection have been used to get the desired results. We will find out how the corresponding best individuals go through the process of mutation and crossover so as to become better individuals who can better survive the environment and thus have better fitness. These individuals form a basis for converging towards a solution for our optimization problems. This report presents a brief overview for the performance of Genetic Algorithm on different optimization functions namely Spherical, Schwefel, Rosenbrock, Rastrigin, Ackley and Griewangk. These benchmark functions may have a number of local optimum but only one global optimum which thus helps us find out the search algorithm's ability to find the best result. This paper presents the results from various experiments carried out on these functions to converge to a given solution.

Keywords: Benchmark function, Creep Mutation, Tournament selection, Steady state selection, Uniform crossover, Fitness function.

Introduction

Genetic Algorithms (GAs) are adaptive heuristic search algorithm based on the evolutionary ideas of natural selection and genetics. GA is to use the power of evolution to solve optimization problems [11]. GA takes up Darwin's notion of biological evolution in beings and transforms that idea into a similar conjecture for solving real world mathematical problems. GA have the ability to dynamically change its behavior so as to get adapted to the environment and come up with its version of better individuals who can achieve their higher goals.

A genetic algorithm is a probabilistic algorithm which maintains a population of individuals, $P(t) = \{ x_1^t, \ldots, x_n^t \}$ For iterations t. Every individual represents a potential solution to a given problem and is implemented by some data structure S [5]. Every individual is subjected with a fitness value which is a measure of how good the individual truly is against a given problem. Evolution may take place for these problems in generally two ways. One method is to discard (selecting elites i.e., elitism) the old population and create a new population. Another way, which I have implemented, is to select a subset of the population using a selection methodology such as tournament selection or roulette wheel selection, perform crossover on the pair of individuals and also have small mutations done on the individual. After that, recombine the subset population into the original population, search for any possible solutions and remove extraneous individuals having "low fitness" value.

Genetic algorithm works on the basis that when two good parents in a certain population, during a generation, will be able to produce individuals which have better fitness than the parents and help to get to a solution later.

Firstly, I would like to discuss on the various benchmark functions used for testing the performance of GA. Then, I would like to outlines the advantage of using a GA, the implementation details for the genetic algorithm dealing with genetics operations for recombination, mutation, selection and replacement/insertion. Finally, I present the test results against various test conditions and the conclusion we may derive from these experiments.

Benchmark Functions Used

The optimization functions [4] (also known as artificial landscapes) are utilized to evaluate various characteristics of optimization algorithms, such as:

- Velocity of convergence
- Precision
- Robustness
- General Performance[13], [2], [1]

We will discuss six of the optimization functions against which we test our Genetic Algorithm.

Sphere function, $f_{Sph}(x) = \sum_{i=1}^{p} x_i^2$ where $x_i \in [-5.12, 5.12]$ and has global minimum at $x^* = (0, 0, \dots, 0)$ and $f_{Sph}(x^*) = 0$.

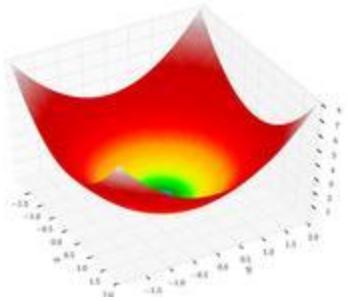


Fig: Sphere function for p = 2

Schwefel function,

$$f_{Sch}(x) = 419.9829 * p + \sum_{i=1}^{p} x_i * \sin\left(\sqrt{|x_i|}\right)$$

where $x_i \in [-512.03, 511.97]$ and has global minimum at $x^* = (-420.9867, \dots, -420.9867)$ and $f_{Sch}(x^*) = 0$.

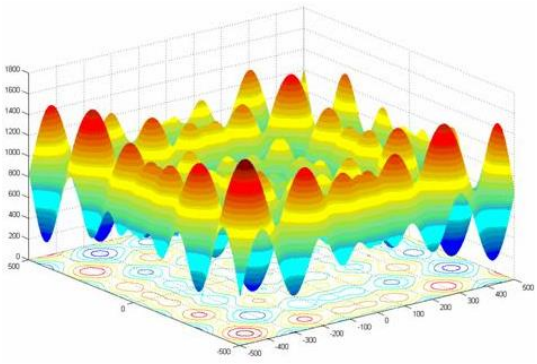


Fig: Schwefel function for p = 2

Ackley function, $f_{Ack}(x) = 20 + e - 20 * \exp\left(-0.2 * \sqrt{\frac{1}{p}\sum_{i=1}^{p} x_i^2}\right) - \exp\left(\frac{1}{p}\sum_{i=1}^{p} \cos(2\pi x_i)\right)$ where $x_i \in [-30, 30]$ and has global minimum at $x^* = (0, 0, \dots, 0)$ and $f_{Ack}(x^*) = 0$.

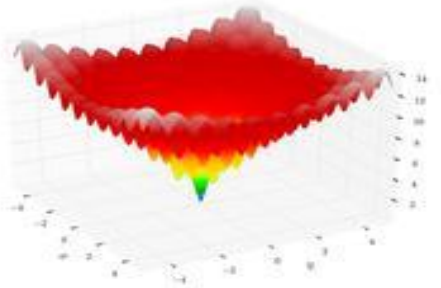


Fig: Ackley function for p=2

Rastrigin function, $f_{Ras}(x) = 10 * p + \sum_{i=1}^{p}(x_i^2 - 10 * \cos(2\pi x_i))$ where $x_i \in [-5.12, 5.12]$ and has global minimum at $x^* = (0, 0, \dots, 0)$ and $f_{Ras}(x^*) = 0$.

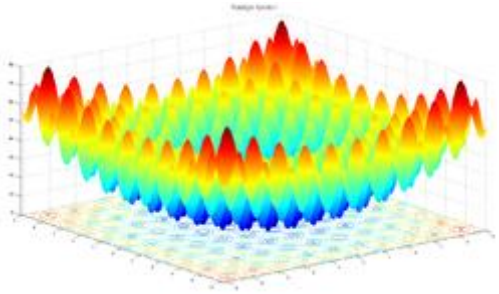


Fig: Rastrigin function for p=2

Rosenbrock function,

$$f_{Ros}(x) = \sum_{i=1}^{p-1}[100(x_{i+1} - x_i^2)^2 + (x_i - 1)^2]$$

where $x_i \in [-2.048, 2.048]$ and has global minimum at $x^* = (1, 1, \dots, 1)$ and $f_{Ros}(x^*) = 0$.

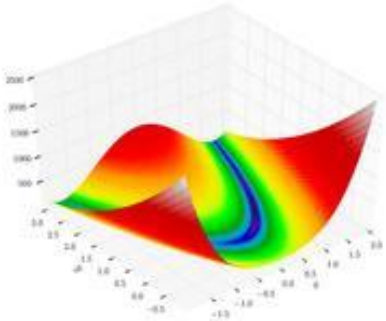


Fig: Rosenbrock function for p=2

Griewangk function,

$f_{Gri}(x) = 1 + \sum_{i=1}^{p} \frac{x_i^2}{4000} - \prod_{i=1}^{p} \cos\left(\frac{x_i}{\sqrt{i}}\right)$ where $x_i \in [-600, 600]$ and has global minimum at $x^* = (0, 0, \ldots, 0)$ and $f_{Gri}(x^*) = 0$.

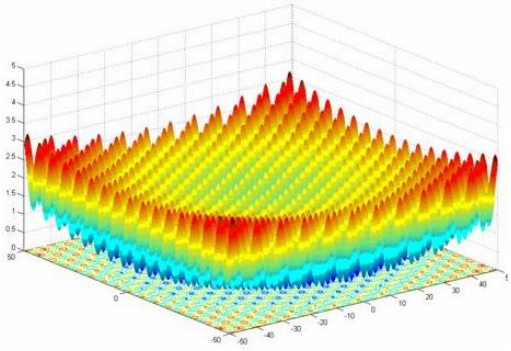


Fig: Griewangk function for p=2

In the implementation for these functions, each function has been differentiated into a separate class having its own range values, a function name identifier and methods to calculate the fitness value for all dimensions, fitness for single dimension, limit checking condition as well as mutation to be performed on the individuals. Following is a basic template for any function class:

```
public class FunctionName {

// Class Name for Identification
public static final String FUNCTION_NAME =
"FunctionName";

// Value Range Variable for function
public static final Range<Double> range =
Range.between(Function.MIN,Function.Max);

// returns fitness value of Individual
public static double
getfitnessValue(double[] varArray) {}

// return fitness value of Gene
public static Double
getSingleFitnessValue(Double var) {}

// mutate the given Individual
public static Individual
mutateIndividual(Individual
individualtoMutate) {}

// check is generated values are between
range
public static boolean
checkLimitCondition(double val) {}

}
```

Advantages of Genetic Algorithm

In context of artificial intelligence, there are a lot of searching algorithms. Some worth mentioning are the Hill Climbing, Simulated Annealing, etc. Even though many of these algorithms have a tendency to gradually get to a solution, genetic algorithms are most flexible for providing features for exploration and exploitation. This feature of genetic algorithm is a very powerful in terms that even though there can be a wide variety of fitness landscape and multiple solutions it will most successfully provide the optimum solution to the problem in hand. Genetic algorithms are more flexible in terms that they do not tend to break even in term of noise of input changes. Also, while searching in a large fitness landscape, multi-modal state-space, or n-dimensional surface, a GA may offer significant benefits over more typical search of optimization techniques. What makes GA so powerful is that it always works with a large number of individuals concurrently rather than taking into consideration only one data point [11].

Genetic Algorithm Implementation

The genetic algorithm is differentiated into a number of classes. At first we start out with the implementation details for some of these classes.

Individual Class

The Individual class consists of an array of double values for the floating point representation, a double value for the calculated fitness Value and an instance of the RandomArrayGenerator Class that is used to create random arrays in between the range of the Optimization function. The methods associated with this class are typically to generate the Individual, calculate the fitness value of the individual and perform mutation on the Individual.

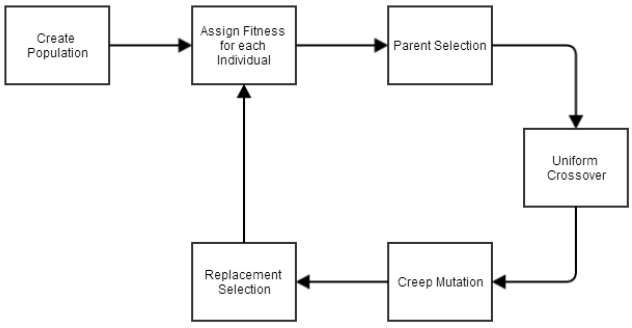


Fig: Evolution Strategy for the Genetic Algorithm

Population Class

The population class consists of a Set of Individuals, a subset population selected by selection mechanism to apply mutation and crossover, a value to store average fitness of the population, a string to get the function name as to which various calculations are branched out, the population size and offspring size and the Boolean flag to find if solution has been found and an Individual solution to the given problem. The methods in this class are:-

• generatePopulation(functionName): to generate the Population as per the functionName and range.

• selectedIndividuals(): returns the individuals who are selected for mutation and crossover.

• checkForSolution(): check if possible solution has been found as per the threshold.

• getTournamentWinner(): to get the Individual who has won the tournament.

• getRandomIndividual(population): get a random individual from the population.

•mutateAllOffSprings(individualsToMutate, functionName): mutate the Individuals as per the function given.

• appendToPopulation(populationSubSet): add the offsprings to the original population

• removeExtraPopulation(): remove the individuals with low fitness as to maintain the population size.

• populationReset(): remove all individuals from the population.

• evolve(functionName): evolve the population as per the given function name.

The evolve method evolves the population as to one generation. The evolve method is invoked from the main class up until a good solution has been found. The following code shows the implementation for evolution of the population.

```
public void evolve(String functionName){
populationSubSet=selectedIndividuals();

populationSubSet=
crossOver(populationSubSet, functionName);

populationSubSet =
mutateAllOffSprings(populationSubSet,
functionName);

appendToPopulation(populationSubSet);

removeExtraPopulation();

checkForSolution();}
```

FunctionTest Class

The FunctionTest class serves as the base class for testing our population against the benchmark functions. The main method invokes the test methods to run the tests against optimization functions. The skeleton for the class is given below:-

```
public class FunctionTest {

    /**
     * @param args
     */
    public static void main(String[] args) {
        FunctionTest ft = new FunctionTest();
        ft.testForSphere();
        ft.testForSchwefel();
        ft.testForAckley();
        ft.testForGriewangk();
        ft.testForRastrigin();
        ft.testForRosenbrock();
    }

    private void testForSphere() {}

    private void testForSchwefel() {}

    private void testForRosenbrock() {}

    private void testForRastrigin() {}

    private void testForAckley() {}

    private void testForGriewangk() {}

    private void functionTester(String
    functionName, int offSpringSize,int
    populationSize, int maxIterations) {}
}
```

Genetic Operators

Parent Selection: Tournament Selection Algorithm

Tournament selection algorithm basically selects some random individuals, then as if in a tournament selects the one with the most optimal fitness and returns that Individual as the parent fit selection.

To select each candidate solution, select a random subset of k solutions from the original population and then select the best solution out of this subset. After tournament competition, the winner is then inserted into the mating pool. Basic Pseudo code is given below:

```
private Individual getTournamentWinner() {
Individual winner = new Individual();
Individual temp = new Individual();
winner = getRandomIndividual(
getPopulation());
for (int i = 0; i < offSpringSize; i++) {
temp = getRandomIndividual
(getPopulation());
        if (temp.getFitnessValue() < winner.
        getFitnessValue()) {winner = temp;}
        }
return winner;}
```

Crossover: Uniform Crossover

Unlike One point and Two point crossover where you select a point to divide the genes of individuals at point and perform information substitution, uniform crossover takes each gene into consideration and perform swapping on individual genes. The genes are swapped on the basis of some probability. The basic idea is that the children generated from crossover has best features from both parents and is therefore closer to the actual solution. This is a key step in genetic algorithm as this notion involves selecting two best individuals, performing crossover and getting an even better individual. The convergence rate for the actual function was nearly twice as fast as compared to when only mutation was done.

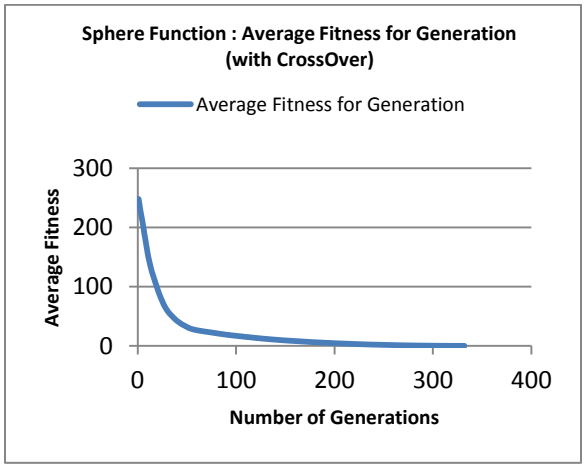


Fig: Average Fitness of Individuals with Crossover

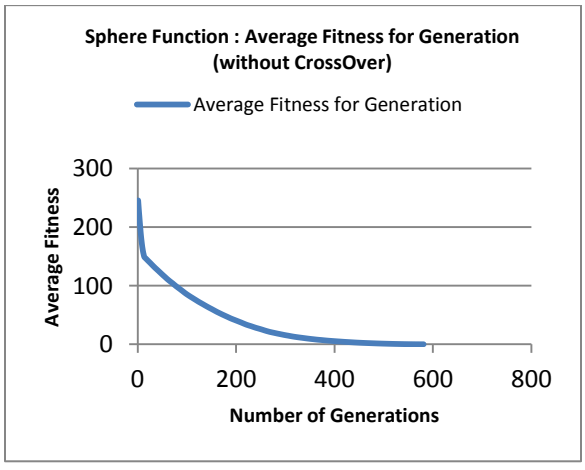


Fig: Average Fitness of Individuals without Crossover

Notice, how the graph is smooth in the one with crossover and how it takes a deep dive in case of one with no crossover. The number of iterations was found to be more less when crossover was done on the individuals. The pseudo code for uniform crossover is provided below:

```
private Set<Individual> crossover
(Set<Individual> popSubSet,String
functionName) {

Set<Individual> resultSet = new
HashSet<Individual>();

Individual child[] = new Individual[2];

Individual[] indvArray = new
Individual[popSubSet.size()];

popSubSet.toArray(indvArray);

for (int i = 0; i < indvArray.length; i = i
+ 2) {
  child = exchangeInfo(indvArray[i],
indvArray[i + 1], functionName);
  resultSet.add(child[0]);
  resultSet.add(child[1]);
}

return resultSet;
}
```

Note that in my implementation the parent selection size must be even so as to create an even number of off springs.

Mutation: Creep Mutation

Mutation is one of the most important genetic operators. Mutation is the factor for actual change in the population. The idea that even though we have characteristics from our parents but we may differ from them somehow brings about the very essence of change. This change is the reason that leads the individuals to become better beings or solutions in this case.

Creep mutation is the form of mutation in which some small value is added or subtracted from the genes of the chromosome or individual. The genes which undergo this change may be selected on the basis of some random probability. However in my implementation I have randomly made the changes to each and every individual so as to make the convergence rate faster. Also, mutation is done on the basis that the calculated fitness after addition of the small change yields better results than the current fitness value of the individual. Below is a case without mutation:

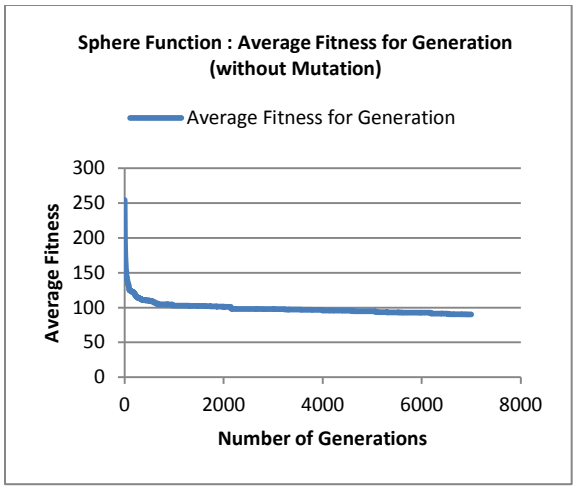


Fig: Average Fitness for Individuals without Mutation

We can observe that the algorithm will not be able to converge after a certain point. Why? This is because the population is used all over again so even though we do crossover there may come a time when even though we swap the genes no new combination will come up. This behavior will be seen in both the Generational and Steady State algorithm if no new individuals are created randomly. Pseudo code for the creep mutation used for GA is given below:

```
Public Individual mutateIndividual
(Individual indvtoMutate) {
// small change delta
double delta = small_change;
Operator op = ±;
for(int i=0;i<DIMENSIONS;i++){
if(getSingleFitness(indvtoMutate[i]        op
delta)   <   indvtoMutate[i].getSingleFitness
&& checkBounds(indvtoMutate op delta) ){
        indvtoMutate[i] op= delta;
}
}
Return indvtoMutate;
}
```

Replacement Selection: Remove Worst Fit Individuals

Whenever we generate some offspring we add it to the population. However, we will probably want to create our population size constant. The individuals who have low fitness don't have much value in our population set. So we basically remove these individuals from our population so as to limit our population. The basic pseudo code for this operation is given below:

```
public void removeExtraPopulation() {
while (population.size() != populationSize)
{

Individual toRemove = new Individual();
toRemove.setFitnessValue(-50000.0);
for (Individual indv : population)
{
if (indv.getFitnessValue() >
toRemove.getFitnessValue())
        {   toRemove = indv;   }
}
        population.remove(toRemove);
        }
}
```

Test Results for GA and Benchmark Problems

Following are the test execution results so as to find the solution of the corresponding benchmark function and also are the execution results for 5000 evaluations. The graphs taken into consideration are the (Average Fitness & Best Individual Fitness) vs. Number of Generations graph.

Three tests are executed:

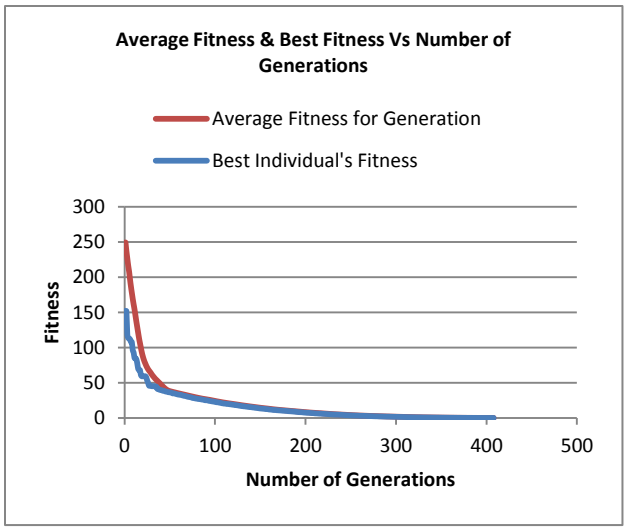
1. **Case I**: Firstly, the population size (p) is 100, the tournament size (t) is 10 and the number of generations (g) is 5000 (for getting solution).
2. **Case II**: Secondly, the population size (p) is 50, the tournament size (t) is 10 and the number of generations (g) is 100.
3. **Case III**: Finally, the population size (p) is 100, the tournament size (t) is 10 and the number of generations (g) is 50.

COMP572 Extra Analysis

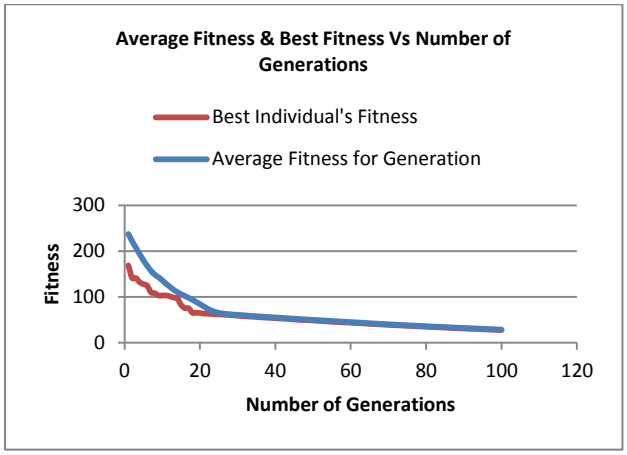
For all the graphs below, we see that the Average fitness is seen to smoother than the Best Individual's Fitness. This may be expected as Population has a lot of Individuals and their fitness value may highly vary. For Case II (p=50 & g=100), the transitions were more abrupt. Smoother transitions were seen for Case III (p=100 & g=50). This was observed for both the Average Individual's Fitness and Best Individual's Fitness. I also came to find that in Case II, the graph between Average Individual's Fitness and Best Individual's Fitness overlapped faster rather than in Case III.

Test for Spherical Function

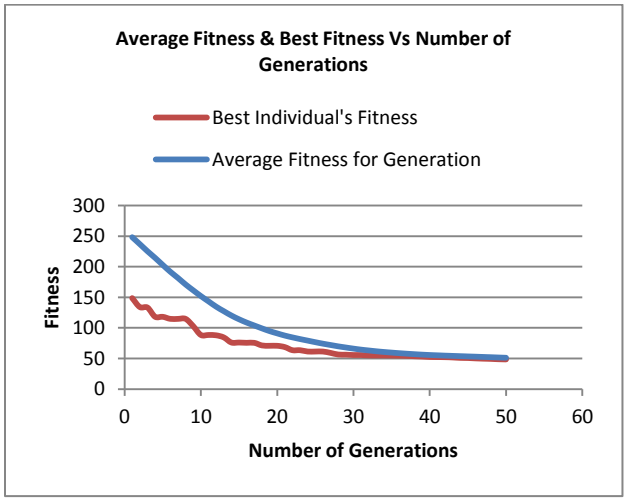
- Test for Case I



- Test for Case II

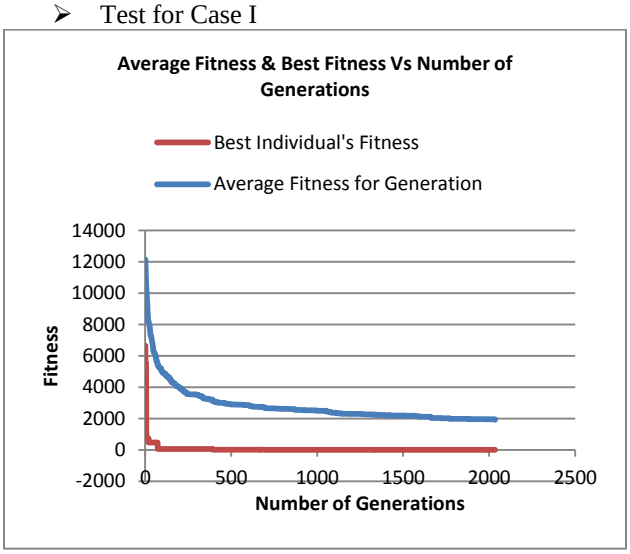


- Test for Case III

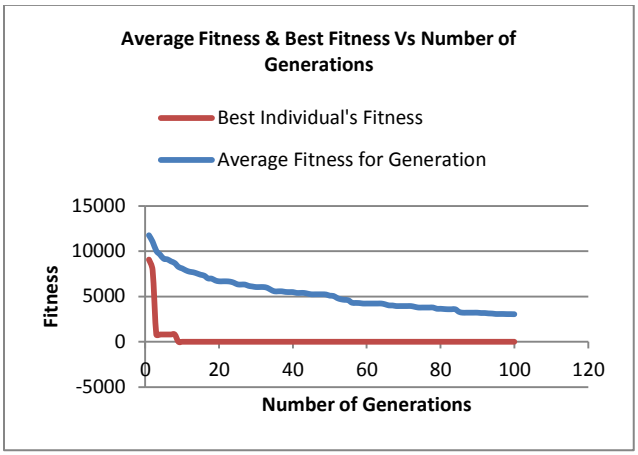


Test for Schwefel Function

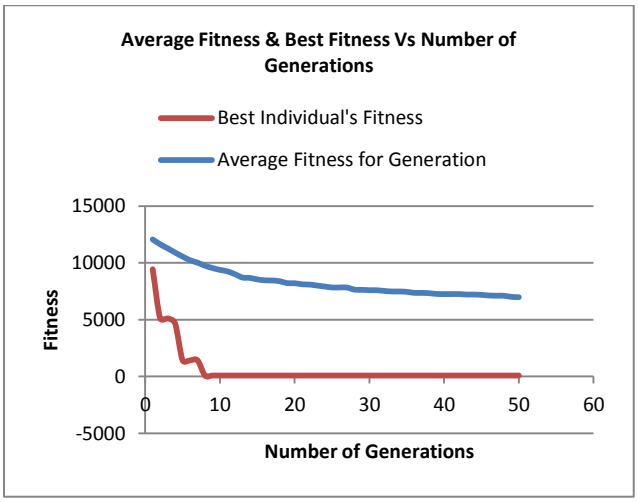
- Test for Case I



- Test for Case II

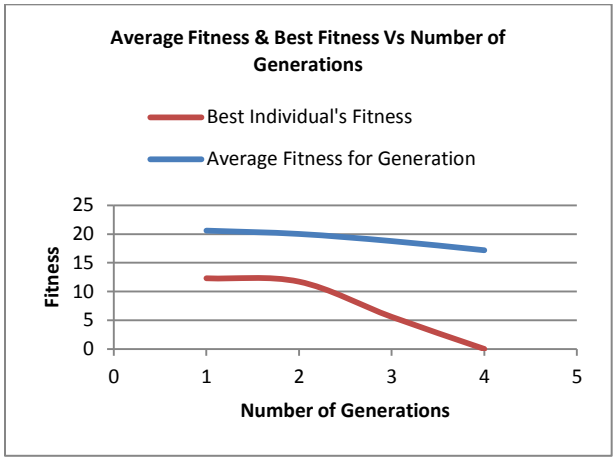


- Test for Case III

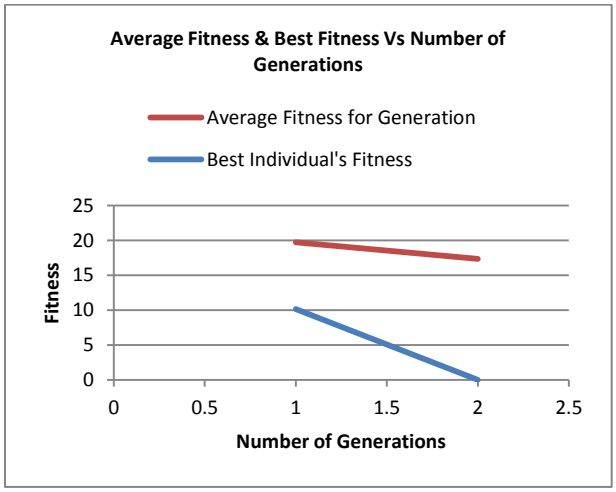


Test for Ackley Function

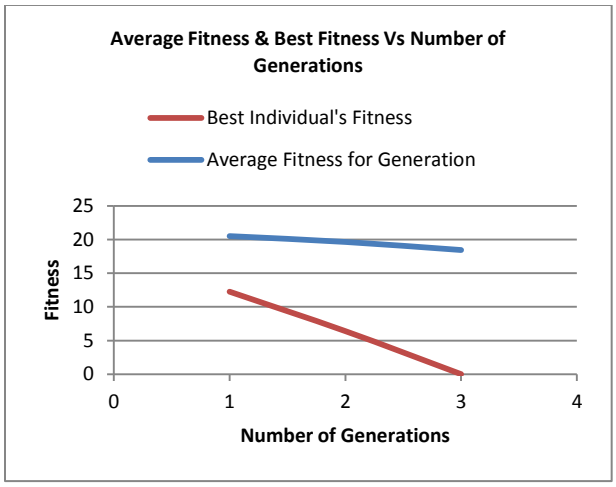
- Test for Case I



- Test for Case II

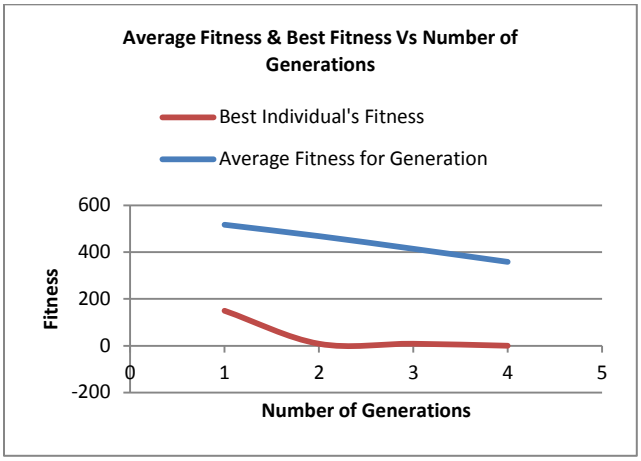


- Test for Case III

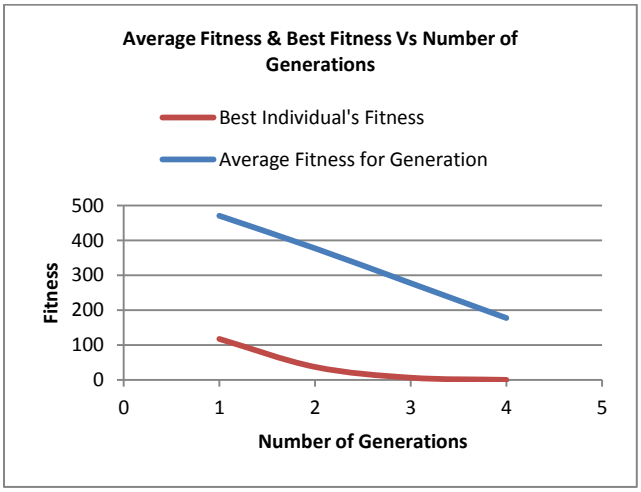


Test for Rastrigin Function

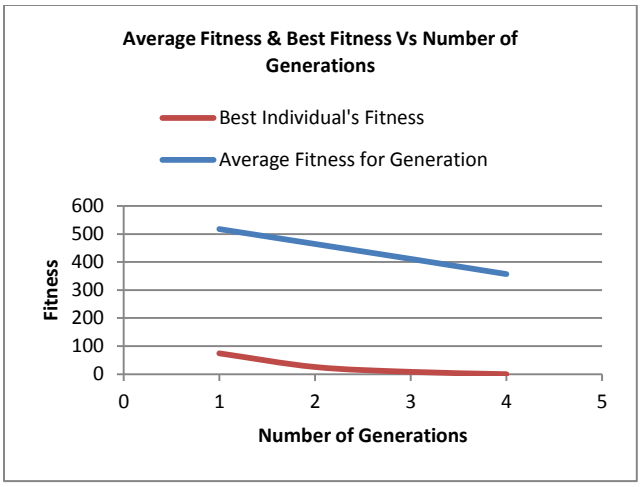
- Test for Case I



- Test for Case II

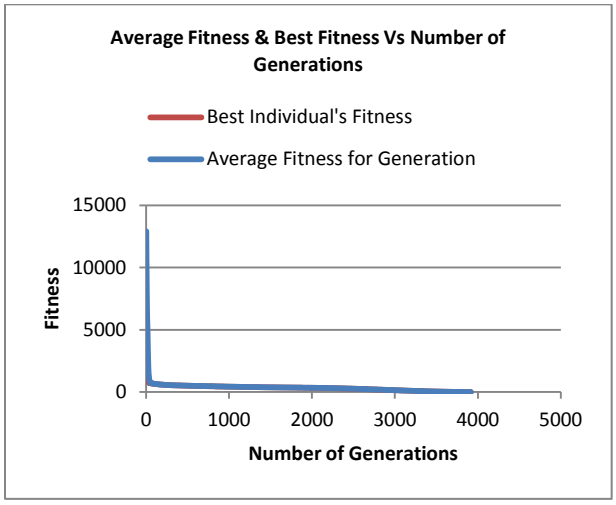


- Test for Case III

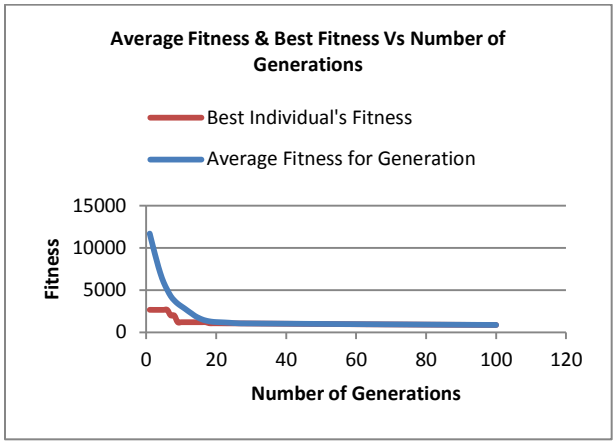


Test for Rosenbrock Function

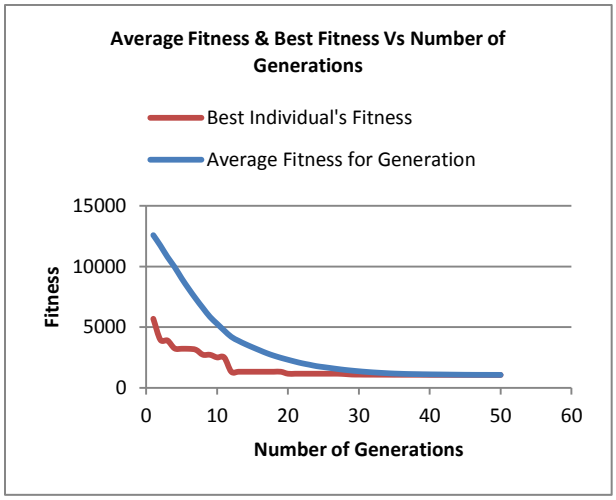
- Test for Case I



- Test for Case II

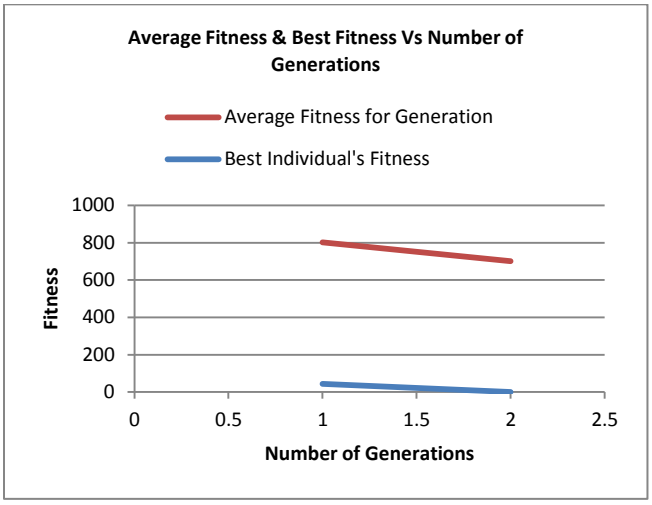


- Test for Case III

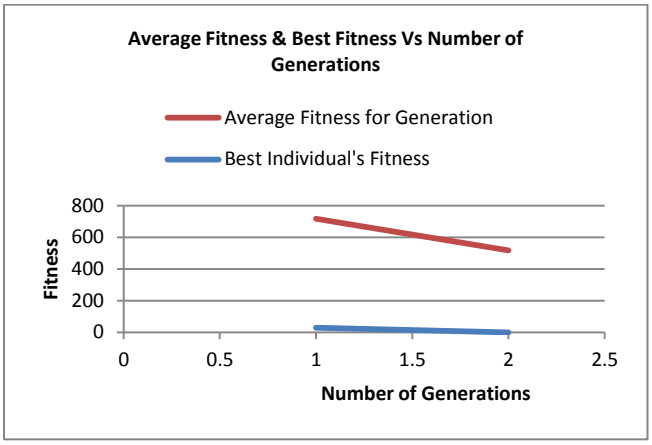


Test for Griewangk Function

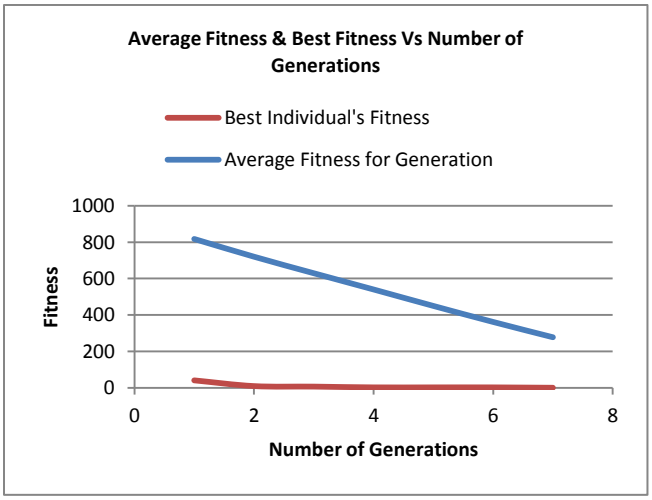
- Test for Case I



- Test for Case II



- Test for Case III



Conclusion

In conclusion, Genetic algorithms prove a much better alternative for search algorithms. They worked very well on the Benchmark problems while the other search algorithms struggled to converge to a solution. The analysis and implementation of genetic operators was a very important part of the project as well. These operators were actually through which we could control the artificial evolution process and eventually come up to a solution. The variables such as population size, offspring size, mutation rate really controlled how the population behaved at a certain generation. Mutation was in my opinion one of the most important genetic operators for genetic algorithm. Solutions for Spherical, Rastrigin, Ackley and Griewangk were found very quickly. However for the functions Schwefel and Rosenbrock finding the solution actually took a considerable amount of iterations. When the evaluations were limited to 5000 evaluation, the fluctuations seen in fitness values were comparatively less in Case II (p=50, g=100) than in Case II (p=100, g=50). Hence, we can conclude that Genetic algorithms are truly able to explore the fitness landscape and find the optimum solution quickly.

References

1. "Benchmark Problems." Benchmark Problems. N.p., n.d. Web. 11 Feb. 2014. <http://www.cs.cmu.edu/afs/cs/project/jair/pub/volume24/ortizboyer05a-html/node6.html#tabla:DefFunc>.
2. "Benchmarking." Wikipedia. Wikimedia Foundation, 2 Oct. 2014. Web. 11 Feb. 2014. <http://en.wikipedia.org/wiki/Benchmarking>.
3. Eiben, Agoston E., and J. E. Smith. "1-3." Introduction to evolutionary computing. New York: Springer, 2003. 1-69. Print.
4. "GEATbx: Examples of Objective Functions." GEATbx: Examples of Objective Functions. N.p., n.d. Web. 12 Feb. 2014. <http://www.pg.gda.pl/~mkwies/dyd/geadocu/fcnfun7.html>.
5. Jong, Kenneth A.. "1-2." Evolutionary computation: a unified approach. Cambridge, Mass.: MIT Press, 2006. 1-26. Print.
6. Kuniyil, Mithun. "Tournament Selection(example) for Introduction to Genetic Algorithms." Scribd. N.p., n.d. Web. 13 Feb. 2014. <http://www.scribd.com/doc/54585910/18/Tournament-Selection-example>.
7. Michalewicz, Zbigniew. "1." Genetic algorithms + data structures = evolution programs. 3rd rev. and extended ed. Berlin: Springer-Verlag, 1996. 1-2. Print.
8. "Rastrigin function." Wikipedia. Wikimedia Foundation, 23 Jan. 2014. Web. 14 Feb. 2014. <http://en.wikipedia.org/wiki/Rastrigin_function>.
9. "Rosenbrock function." Wikipedia. Wikimedia Foundation, 2 Oct. 2014. Web. 14 Feb. 2014. <http://en.wikipedia.org/wiki/Rosenbrock_function>.
10. Soule, Terence. "Project 1." Porject 1. N.p., n.d. Web. 10 Feb. 2014. <http://www2.cs.uidaho.edu/~cs472_572/s14/GAProject.html>.
11. "Test functions for optimization." Wikipedia. Wikimedia Foundation, 2 Jan. 2014. Web. 14 Feb. 2014. <http://en.wikipedia.org/wiki/Test_functions_for_optimization>.
12. "Tournament selection." Wikipedia. Wikimedia Foundation, 2 May 2014. Web. 14 Feb. 2014. <http://en.wikipedia.org/wiki/Tournament_selection>.
13. Yadav, Ruby , and Waseem Ahmad. "Benchmark Function Optimization using Genetic Algorithm." Benchmark Function Optimization using Genetic Algorithm 2.2319â€ • 5606 (2013): 1-4. http://borjournals.com/Research_papers/Jun_2013/1323IT.pdf. Web. 12 Feb. 2014.