Comparative Performances of Crossover Functions in Genetic Algorithms

Ezeani M I\textsuperscript{1}; Okonkwo O R; Onyesolu M O; Osita E G
Department of Computer Science, Nnamdi Azikiwe University,
PMB 5025, Awka, Anambra State, Nigeria

Abstract
Genetic Algorithms have been widely applied to various kinds of optimisation problems. In this work, a Genetic Algorithm is designed to solve the three classic numerical optimisation problems – Rastrigin, Schwefel and Griewank. An experiment to observe the comparative performances of five different crossover functions was conducted. Also, the possible effect of aging out some of the old individuals from the population was hinted at. A parameter set expected to give the optimal performance and a discussion on the design considerations are presented below.

1.0 Background Introduction
The field of evolutionary computation is a rapidly growing one. Mitchell [Mit96], in discussing the use of evolution as an inspiration for solving computational problems, observed that,

\textit{..the mechanism of evolution seem well suited for some of the most pressing computational problems in many fields. Many computational problems require searching through a huge number of possibilities for solutions.}

Genetic Algorithm is one of the most popular techniques used in evolutionary computation [Gol89]. However, designing and tuning a Genetic Algorithm to solve such problems have always involved trade-offs. This is often because the nature of the problem and desired quality of the result are often weighed

\textsuperscript{1}Corresponding Author: igezeani@yahoo.com
alongside the available resources for solving it. An overview of Genetic Algorithms is presented in the next section as well as the general discussion on the problem and approaches taken to tackle it.

**Genetic Algorithms (GAs)**

GAs are population based search algorithms, originally developed by John Holland (1975), based on the principles of the *Darwinian Theory of Evolution and Natural Selection* [Whi05]. They are currently among the most widely used heuristic approaches to multiobjective optimisation. A GA is said to be population based because it works with a population of individual solutions. These individual solutions are data structures encoded in chromosome-like forms i.e. a concatenation of genes. The gene values could be bit-strings, real values or symbols.

The first step in the canonical algorithm is to randomly generate individuals that form the initial population at generation $g:=0$. It then loops through the processes of selection of parents for the intermediated population (or mating pool); crossover and mutation that produces new offspring and finally, replacement of the parent population with the offspring population. This is done over a specified number of generations and stops when a certain target solution quality is reached or when any other termination criterion is met. Issues concerning the choice of the main operations of selection, crossover, mutation and replacement will be discussed in section 3: Methodology. Below is a sample structure of a typical GA:

1. Generate ($P(0)$)
2. $t := 0$
3. while not Termination_Criterion($P(t)$)
4. do
5. Evaluate($P(t)$)
6. $P'(t) := Selection(P(t))$
7. \( P'(t) := \text{Recombination}(P'(t)) \)
8. \( P'(t) := \text{Mutation}(P'(t)) \)
9. \( P(t+1) := \text{Replace}(P(t), P'(t)) \)
10. \( t := t+1 \)
11. return Best_Solution_Found

2.0 Task Description

The summary of the task in this assignment is to design and tune a single robust algorithm that can solve the three classic numerical optimisation test problems – Rastrigin, Schwefel and Griewank – up to given targets. This algorithm should be able to solve all the three problems using the same set of initial configuration parameters. Java is expected to be used for the development. The number of evaluations of the objective functions required to solve the three numeric functions should not exceed 30,000 and the performance should be improved as much as possible. Any additional feature implemented or experimented on may as well count for the overall assessment.

Overview of the functions (Rastrigin, Schwefel and Griewank)

The above given functions are classics that are often used as global numerical optimisation test problems. A brief overview of each of the functions in the context of our problem is presented below.

Rastrigin

- **Function definition:** \( f(x) = an + \sum_{i=1}^{n} [x_i^2 - a \cos(bx_i)] \)
- **Number of variables, \( n \):** \( n = 20 \)
- **Search domain for \( x \):** \( -5.12 \leq x_i \leq 5.12, \; i = 1, 2, \ldots, n. \)
- **Value of \( a \):** \( 10 \)
- **Value of \( b \):** \( 2\pi \)
- **Number of local minima:** many
- **Actual global minima:** \( x^* = (0, \ldots, 0), \; f(x^*) = 0. \)
- **Set target:** \( < 0.9 \)
Schwefel

- **Function definition:** \( f(x) = \sum_{i=1}^{n} x_i \sin(\sqrt{|x_i|}) \)
- **Number of variables, \( n \):** \( n = 10 \)
- **Search domain for \( x_i \):** \(-500 \leq x_i \leq 500, i = 1, 2, \ldots, n.\)
- **Number of local minima:** many
- **Actual global minima:** \( x^* = (1, \ldots, 1), f(x^*) = 0. \)
- **Set target value:** \(< -4187.5\)

Griewank

- **Function definition:** \( f(x) = 1 + \frac{\sum_{i=1}^{n} x_i^2}{4000} - \prod_{i=1}^{n} \cos\left(\frac{x_i}{\sqrt{i}}\right) \)
- **Number of variables, \( n \):** \( n = 10 \)
- **Search domain for \( x_i \):** \(-600 \leq x_i \leq 600, i = 1, 2, \ldots, n.\)
- **Number of local minima:** many
- **Actual global minima:** \( x^* = (0, \ldots, 0), f(x^*) = 0. \)
- **Set target value:** \(< 0.1\)
3.0 Methodology

This experiment was conducted with an already developed algorithm for the test problems. However, it was modified to introduce additional crossover operators and the age operator. The major tasks carried out in this work were tuning the algorithm to obtain a set of configuration parameters that will achieve the desired aim i.e. solving each of the problems to target and within the total maximum number of evaluations and then with the solution parameter set, the tests on the crossover functions and the age function were conducted. The structure of the algorithm used including its operators will be briefly presented and then followed by the experiment procedure, results and discussions.

The Algorithm

The GA performance is often determined by the choice of operators and parameters made during the design. The following defines the main aspects of the design of the GA used in this work.

Representation

Although most canonical\(^2\) GAs use binary encoding of chromosomes, other types of coding are possible. Real valued coding was used in this work. A good justification for that may be drawn from the works of [Wri91] and [ScE93].

Selection

This algorithm used tournament selection to pick the parents that will be recombined from the population specifically for the BLX-0.5, Linear and Uniform crossovers. Research has confirmed that this selection scheme can effectively improve performance [GoD91]. However, for the two other crossovers operators – Multi-Parent (1) and Multi-Parent (2) – normal random

\(^2\) The original structure of GA as proposed by Holland is often referred to as the canonical GA
selection was adopted because this approach seemed more explorative and yet produced good results.

**Crossover operators**

There were five different crossover operators featured in this work. The first (BLX-0.5 and Linear) two are typically used with real values because they also can introduce new information into the search space by producing new allele values. The others (multi-parents and uniform) can recombine based on the already existing information. They do not introduce any new traits into the population and so a more prominent with binary coding. All the five operators are discussed under *Testing the crossover operators* in section 3.

**Mutation**

Probabilistic creep mutation described by [Wri91] which is less disruptive than replacing a gene completely with a new random real value, was used. A perturbation is likely to increase or decrease the gene value but the maximum creep size was varied to obtain results during the tuning.

**Replacement**

The steady state replacement scheme of the *extinction of the worst* was used. Only one offspring is produced by the crossover and mutation and, if better, the new offspring replaces the worst individual in the population.

*The Experiment: Tuning the Algorithm*

The problem of selecting an optimum set of parameters for the GA is complex itself [Vin03]. The two main approaches to this are often building an adaptive GA or some kind of heuristic trial and error method. The latter was largely applied in this work.
Testing for Creep Size and Distribution (theta)
A quick evaluation of the performance of the mutation rate on the three functions was made (fig 4.) and the result indicated that Schwefel did not work at all but for some occasional noise at some points. Rastrigin was also shown to be sensitive to the mutation rate. However, mRate of 0.1 was used to examine the effect of maxCreep and mDist (Theta).

So for the three functions, the maxCreep and the theta were varied and the graph of the average number of evaluations that solved the problem with the maximum number of evaluation being 30,000 was plotted. (ref. to Appendix I for the data and figs 5,6,7)

Rastrigin gave good results for higher values of theta (8 – 15) and lower values of maxCreep (0.10 – 0.7). However, Schwefel seemed to be affected more by maxCreep than theta. Lower values of maxCreep generally produced bad results with occasional cases of exceptionally good performances (noise). Also, higher
values of $maxCreep$ (0.7 and above) almost guaranteed above average performance and helps the algorithm to better explore the search space and not get trapped in a local optima.

Griewank is somewhat different from the other two. Although, it gave comparatively good performance, the best results clustered around lower values of $maxCreep$ and the mid values of $\theta$. (see Appendix I)

The experiment, as well as the graphs, shows that there are quite a lot more issues with the nature of the individual functions. The target for Griewank can easily be met but with a combination of the other two functions makes the problem a little more demanding.

Fig 5: **Griewank: Max-Creep + Theta**: mRate: 0.1; popSize: 15; tSize: 2
However, tracing the data table Appendix I, the values identified were:

- \( \theta = 9 \) or \( 10 \)
- \( \text{maxCreep} = \text{between 0.8 and 0.9} \)

Fig 6: **Rastrigin: Max-Creep + Theta**

\[ \text{mRate: 0.1; popSize: 15; tSize: 2} \]

Fig 7: **Schwefel: Max-Creep + Theta**

\[ \text{mRate: 0.1; popSize: 15; tSize: 2} \]
Fine-tuning the initial values (*popSize*, *tSize* and *mRate*)

**popSize**
With the parameters for *theta* and *maxCreep* as shown above, values for *popSize* were tested. For the results, Schewefel and Griewank showed relative stability across varying *popSize* values but Rastrigin was obviously more sensitive to varying population sizes (*Fig 9*). However, for their combined number of evaluations, relatively good performance was observed for *popSize* values between 5 and 25 (*Fig 9*).

It is noteworthy though that in all these tests, the *overInit* remained constant i.e. 100, but it is believed to have some effects on the starting average quality of the population. Also, considering that the experiment is being run to target, accurate measurement of the quality of the solution was not part of the work.

**tSize**
With a population size of 10, the overall performance of the three functions in terms the number of evaluations was best at *tSize* = 7 (*see fig 10*). *Fig 11* shows their individual performances in terms of the number of evaluations. Considering the configuration set being used, the likely value for *tSize* could be anything between 6 and 8.

![Total Evaluation vs Population size](image-url)

*Fig 8: Total Evaluation vs Population size: mRate: 0.1; tSize: 2*
The actual effect of the mutation rate was the last to be tested with the configuration parameter values derived from the above experiments. popSize, tSize, theta and maxCreep were kept at 10, 7, 10, 0.9 respectively. The combined performance of the three functions in terms of evaluations seemed best at points between 0.8 and 0.16. However, the least number of evaluations was best at 0.12 and 0.16.

Also Schwefel and Griewank were tending toward reduced number of evaluations as the mutation rate increased but the performance of Rastrigin was not guaranteed (a shot at it actually showed worse performance). Generally, a mutation rate of 0.12 will be ideal considering the data collected. See Table 1 and Figs 12, 13.

mRate

Fig 9: Evaluation vs Population size: mRate= 0.1; tSize= 2
Fig 10: Total Evaluation vs Tournament size: mRate: 0.1; tSize: 2, popSize = 10

Fig 11: Evaluations vs tournament size: mRate: 0.1; popSize = 10

Fig 12: Total Evaluations vs Mutation Rate: tSize: 7, popSize = 10
Table 1: mRate vs Evaluation

<table>
<thead>
<tr>
<th>mRate</th>
<th>Rastrigin</th>
<th>Schwefel</th>
<th>Griewank</th>
<th>Total Evals</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.02</td>
<td>22387</td>
<td>20314</td>
<td>9024</td>
<td>51725</td>
</tr>
<tr>
<td>0.04</td>
<td>17291</td>
<td>15112</td>
<td>5011</td>
<td>37414</td>
</tr>
<tr>
<td>0.06</td>
<td>14534</td>
<td>10413</td>
<td>3467</td>
<td>28414</td>
</tr>
<tr>
<td>0.08</td>
<td>12943</td>
<td>7648</td>
<td>2764</td>
<td>23355</td>
</tr>
<tr>
<td>0.10</td>
<td>13623</td>
<td>8003</td>
<td>2125</td>
<td>23751</td>
</tr>
<tr>
<td>0.12</td>
<td>12930</td>
<td>7443</td>
<td>1602</td>
<td>21975</td>
</tr>
<tr>
<td>0.14</td>
<td>15142</td>
<td>6125</td>
<td>1468</td>
<td>22735</td>
</tr>
<tr>
<td>0.16</td>
<td>13824</td>
<td>6186</td>
<td>1418</td>
<td>21428</td>
</tr>
<tr>
<td>0.18</td>
<td>17388</td>
<td>6721</td>
<td>1611</td>
<td>25720</td>
</tr>
<tr>
<td>0.20</td>
<td>17329</td>
<td>5399</td>
<td>1413</td>
<td>24141</td>
</tr>
</tbody>
</table>

Fig 13: Evaluations vs Mutation Rate: tSize: 7, popSize = 10

Testing the crossover operators

The extended blend crossover – BLX-0.5

This is one of the commonly used crossover techniques for real number coding and it was used in this algorithm for the initial tuning phase. Point and uniform crossovers often used with bit-string representations do not introduce new information into the search [Vin03]. For this operator, the new offspring, $z$, is defined from the parents $x$ and $y$ as: -
\[ z = x + (2R - 0.5) \times (y - x) \text{ where } R = \text{random real number} \]

A sample code is shown below

```java
... if (xoverType == 1){ /* blx-0.5 crossover*/
    for (int i = 0; i < nPars; i++)
        offspr.gene[i]=parent1.gene[i]+(2*rand.nextDouble()-0.5)*(parent2.gene[i]- parent1.gene[i]);
}
... 

Linear crossover
Another operator used with real number coding – Linear crossover – was also introduced. Although this promised good results, it was not efficient because it requires additional evaluations. It works by producing three offspring from two parents and selection the best among them and this causes extra two evaluations for any one replacement. For instance, the three offspring \( z_1, z_2, z_3 \) produced by the parents \( x \) and \( y \) are defined by

\[ z_1 = \left( \frac{x}{2} \right) + \left( \frac{y}{2} \right); 
z_2 = 3 \left( \frac{x}{2} \right) - \left( \frac{y}{2} \right); 
z_3 = 3 \left( \frac{y}{2} \right) - \left( \frac{x}{2} \right) \]

A sample code is as shown below

```java
... else if(xoverType == 2){ /* linear crossover*/
    for(int i = 0; i < nPars; i++)
        offspr. gene [i] = parent1. gene [i]/2 + parent2. gene [i]/2;
        off2. gene [i] = 3*parent1. gene [i]/2 - parent2. gene [i]/2;
        off3. gene [i] = 3*parent2. gene [i]/2 - parent1. gene [i]/2;
    // select one of the three: 2 extra evaluations
    if (offspring.evaluate() > off2.evaluate()) offspring = off2;
    if (offspring.eval>off3.evaluate()) offspring = off3;
} // end linear crossover
... 

Multi parent crossover (1)
In the first version of the multi – parents crossover used, the number of parents used varies and so does the length of the genes each contributes. Parents are selected at random for as long as there is an ‘unfilled’ gene position
still remaining on the new offspring. Crossover points are determined by adding up the lengths of genes contributed by different parents. The sample code below illustrates the concept.

```java
if(xoverType == 3){
    // multi parent crossover(1): Some parents are used.
    int selLength;
    for(int x_point=0; x_point < nPars;){ /* start from the 1st gene*/
        // make new offsprings with some parents <= gene length
        parent1 = population[rand.nextInt(popSize)]; //get a parent
        selLength = rand.nextInt(nPars-1); // get a point on the parent
        if (x_point + selLength > nPars-1)
            selLength = (nPars) - x_point ;
        System.arraycopy(parent1.gene, x_point, offspring.gene, x_point, selLength);
        x_point += selLength; //get the new crossover point
    } //for
}

Multi parent crossover (2)
This is multi-parent complete. In this version of the multi–parent crossover, the number of parents used is more determined. In fact, as many parents as the gene length are used such that each gene value is contributed by a newly selected parent. Selection is also basically random not biased to better parents. See the code segment.

```java
else if(xoverType == 4){
    // multi parent crossover(2): No of parents = gene length.
    int selGene;
    for(int x_point=0; x_point < nPars;){ /* start from the 1st gene*/
        // make new offsprings with some parents = gene length
        parent = population[rand.nextInt(popSize)]; //get a parent
        selGene = x_point; // get the gene this parent donates
        System.arraycopy(parent.gene, x_point, offspring.gene, x_point, selLength);
        x_point += selGene; //get the new crossover point
    } //for
}
```

Uniform crossover
In uniform crossover, two parents were used and they were picked with tournament selection. A mask Boolean pattern is then randomly generated to
aid in deciding which parent donates which gene in producing a single offspring. An illustration and a sample code segment of two parents, x and y, producing an offspring, z, are given below.

```java
... else if (xoverType == 5) {
    //for uniform cross over
    boolean [] mask = new boolean [nPars];
    for (int i=0; i<nPars; i++) mask[i] = rand.nextBoolean(); //generate the Boolean values
    for(int x_point=0;x_point<nPars;x_point++){
        //decide which parent contributes the gene for the new offspring
        if (mask[x_point])
            System.arraycopy(parent1.gene,x_point, offspring.gene,x_point,1);
        else
            System.arraycopy(parent2.gene,x_point, offspring.gene,x_point,1);
    } //end of for
    } //uniform crossover
...
```

The analysis of the data from the comparison is as shown below.

<table>
<thead>
<tr>
<th>Crossover</th>
<th>Rastrigin</th>
<th>Schwefel</th>
<th>Griewank</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLX-0.5</td>
<td>13215</td>
<td>6927</td>
<td>1621</td>
<td>21763</td>
</tr>
<tr>
<td>Linear</td>
<td>21543</td>
<td>14610</td>
<td>3991</td>
<td>40144</td>
</tr>
<tr>
<td>M-Parents(1)</td>
<td>14567</td>
<td>7069</td>
<td>2003</td>
<td>23639</td>
</tr>
<tr>
<td>M-Parents(2)</td>
<td>12917</td>
<td>5707</td>
<td>1652</td>
<td>20276</td>
</tr>
<tr>
<td>Uniform</td>
<td>14052</td>
<td>5874</td>
<td>1746</td>
<td>21672</td>
</tr>
</tbody>
</table>

![Fig 14: Comparison of Different Crossover Functions](image)

---

3 Actually, two offspring can be generated but this work considered only one.
Expectedly, linear crossover, which has an additional two evaluations, was the worst in performance. However, the others were within the same range of performance with the BLX-0.5 but it is interesting to note that without any bias in the selection of parents (e.g. tournament), the multi-parent functions comparatively well.

The age function

One other function that seemed interesting to look at was the age function. The whole concept is about elimination individuals that have lasted in the population beyond a certain age limit (number of evaluations). The expectation is that replacing them with new offspring might improve the explorative aspect of the search. However, this experiment did not cover this in detail and therefore cannot give very sound report on its effect. The data (below) generated did not reveal much.

```java
public void age(Individual [] individuals, int maxAge){
    for (int i = 0; i < popSize; i++) {// go through the entire population
        individuals[i].age++; // increment age by 1
        if (individuals[i].age > maxAge){ // check if the age limit is exceeded
            Individual parent;
            for(int x_point=0;x_point<nPars;x_point++) { // create a new individual with
                parent = population[rand.nextInt(popSize)]; // multiple parents and make its
                //age = 0
                System.arraycopy(parent.gene, x_point, individuals[i].gene, x_point, 1);
            }
            individuals[i].age = 0;
        }
    }
}
```

<table>
<thead>
<tr>
<th>Age Limit</th>
<th>Rastrigin</th>
<th>Schwefel</th>
<th>Griewank</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>5000</td>
<td>13501</td>
<td>7266</td>
<td>2119</td>
<td>22886</td>
</tr>
<tr>
<td>10000</td>
<td>11803</td>
<td>7597</td>
<td>2102</td>
<td>21502</td>
</tr>
<tr>
<td>15000</td>
<td>13415</td>
<td>7419</td>
<td>1939</td>
<td>22773</td>
</tr>
<tr>
<td>20000</td>
<td>13152</td>
<td>6437</td>
<td>2208</td>
<td>21797</td>
</tr>
</tbody>
</table>
4.0 Conclusion

Generally, Rastrigin required far more time than the other two to reach the target. In summary, the optimum set of configuration parameters that was derived from this experiment are:

Population size: 10
Tournament size: 7
Mutation rate: 0.12
Max Creep: 0.9
Max Distribution (Theta) 10
Crossover: BLX-0.5, the “Multi-Parent”s, Uniform

The above set of configuration parameter values consistently guarantees a total number of evaluations less than 23,000 in meeting the required targets on all the three functions. The linear crossover naturally would not do well compared to the others if the key assessment factor is number of evaluations. But others performed well enough.

However, one of the weaknesses of this experiment is not considering the effect of other parameters on the performance of the crossover operator. Also BLX-0.5 remains the best choice for its ability to explore the search space better by producing new allele values. This goes also in considering the effect of the age function. Although, the age function did not show any interesting
performance pattern, it may be worth probing further into it. Future works on this need to explore those areas.
5.0 References


### 6.0 Appendix I: showing data collected on Theta, mCreep and No of Evaluations for three functions

<table>
<thead>
<tr>
<th>Theta</th>
<th>mCreep</th>
<th>No of Trials</th>
<th>Mutation Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAISTRIGIN</td>
<td>PSize = 15</td>
<td>TournamSize = 2</td>
<td>= 50</td>
</tr>
<tr>
<td>Theta</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>0.1</td>
<td>15207</td>
<td>15884</td>
<td>17102</td>
</tr>
<tr>
<td>0.2</td>
<td>14876</td>
<td>14429</td>
<td>15396</td>
</tr>
<tr>
<td>0.3</td>
<td>17904</td>
<td>16037</td>
<td>15619</td>
</tr>
<tr>
<td>0.4</td>
<td>19260</td>
<td>17533</td>
<td>16616</td>
</tr>
<tr>
<td>0.5</td>
<td>20736</td>
<td>17671</td>
<td>17566</td>
</tr>
<tr>
<td>0.6</td>
<td>23002</td>
<td>20700</td>
<td>19311</td>
</tr>
<tr>
<td>0.7</td>
<td>24892</td>
<td>20497</td>
<td>18809</td>
</tr>
<tr>
<td>0.8</td>
<td>24517</td>
<td>21563</td>
<td>21043</td>
</tr>
<tr>
<td>0.9</td>
<td>24519</td>
<td>22025</td>
<td>19794</td>
</tr>
<tr>
<td>1</td>
<td>25563</td>
<td>24177</td>
<td>22378</td>
</tr>
<tr>
<td>1.1</td>
<td>26448</td>
<td>23787</td>
<td>21259</td>
</tr>
<tr>
<td>Average</td>
<td>21539</td>
<td>19482</td>
<td>18627</td>
</tr>
<tr>
<td>Minimum</td>
<td>12190</td>
<td>Maximum</td>
<td>26448</td>
</tr>
<tr>
<td>Theta</td>
<td>mCreep</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-------</td>
<td>--------</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td></td>
<td>0.1</td>
<td>1782</td>
<td>1643</td>
</tr>
<tr>
<td></td>
<td>0.2</td>
<td>2669</td>
<td>1987</td>
</tr>
<tr>
<td></td>
<td>0.3</td>
<td>3794</td>
<td>2723</td>
</tr>
<tr>
<td></td>
<td>0.4</td>
<td>5421</td>
<td>3258</td>
</tr>
<tr>
<td></td>
<td>0.5</td>
<td>5334</td>
<td>3811</td>
</tr>
<tr>
<td></td>
<td>0.6</td>
<td>7596</td>
<td>4493</td>
</tr>
<tr>
<td></td>
<td>0.7</td>
<td>8360</td>
<td>4508</td>
</tr>
<tr>
<td></td>
<td>0.8</td>
<td>10503</td>
<td>4960</td>
</tr>
<tr>
<td></td>
<td>0.9</td>
<td>9769</td>
<td>5842</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>11607</td>
<td>7057</td>
</tr>
<tr>
<td></td>
<td>1.1</td>
<td>11037</td>
<td>6811</td>
</tr>
<tr>
<td>Average</td>
<td>7079</td>
<td>4281</td>
<td>3027</td>
</tr>
</tbody>
</table>
Schwefel

$pSize = 15 \quad tSize = 2 \quad No\ of\ Trials = 50 \quad Mutation\ Rate = 0.1000$

<table>
<thead>
<tr>
<th>Theta</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
</tr>
</thead>
<tbody>
<tr>
<td>mCreep</td>
<td>0.1</td>
<td>30000</td>
<td>30000</td>
<td>30000</td>
<td>30000</td>
<td>30000</td>
<td>30000</td>
<td>30000</td>
<td>30000</td>
<td>30000</td>
<td>30000</td>
<td>30000</td>
<td>30000</td>
</tr>
<tr>
<td></td>
<td>0.2</td>
<td>30000</td>
<td>30000</td>
<td>30000</td>
<td>30000</td>
<td>30000</td>
<td>30000</td>
<td>30000</td>
<td>30000</td>
<td>30000</td>
<td>30000</td>
<td>30000</td>
<td>3056</td>
</tr>
<tr>
<td></td>
<td>0.3</td>
<td>30000</td>
<td>30000</td>
<td>2057</td>
<td>30000</td>
<td>30000</td>
<td>30000</td>
<td>30000</td>
<td>30000</td>
<td>30000</td>
<td>30000</td>
<td>30000</td>
<td>2247</td>
</tr>
<tr>
<td></td>
<td>0.4</td>
<td>30000</td>
<td>30000</td>
<td>2661</td>
<td>1719</td>
<td>30000</td>
<td>30000</td>
<td>30000</td>
<td>30000</td>
<td>30000</td>
<td>30000</td>
<td>30000</td>
<td>22542</td>
</tr>
<tr>
<td></td>
<td>0.5</td>
<td>30000</td>
<td>30000</td>
<td>30000</td>
<td>30000</td>
<td>30000</td>
<td>903</td>
<td>30000</td>
<td>30000</td>
<td>30000</td>
<td>30000</td>
<td>30000</td>
<td>1199</td>
</tr>
<tr>
<td></td>
<td>0.6</td>
<td>30000</td>
<td>3874</td>
<td>30000</td>
<td>30000</td>
<td>30000</td>
<td>30000</td>
<td>30000</td>
<td>30000</td>
<td>30000</td>
<td>30000</td>
<td>30000</td>
<td>1492</td>
</tr>
<tr>
<td></td>
<td>0.7</td>
<td>14519</td>
<td>19388</td>
<td>18261</td>
<td>17229</td>
<td>18177</td>
<td>18158</td>
<td>21487</td>
<td>19883</td>
<td>13796</td>
<td>15659</td>
<td>22363</td>
<td>18775</td>
</tr>
<tr>
<td></td>
<td>0.8</td>
<td>8179</td>
<td>6029</td>
<td>6461</td>
<td>5692</td>
<td>6523</td>
<td>7136</td>
<td>6765</td>
<td>7578</td>
<td>7508</td>
<td>7647</td>
<td>8829</td>
<td>9423</td>
</tr>
<tr>
<td></td>
<td>0.9</td>
<td>8234</td>
<td>6161</td>
<td>5666</td>
<td>5289</td>
<td>5759</td>
<td>6659</td>
<td>7395</td>
<td>6625</td>
<td>7647</td>
<td>9151</td>
<td>8093</td>
<td>8357</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>9436</td>
<td>6171</td>
<td>5345</td>
<td>5855</td>
<td>5860</td>
<td>6450</td>
<td>8251</td>
<td>7578</td>
<td>7239</td>
<td>8056</td>
<td>8754</td>
<td>8086</td>
</tr>
<tr>
<td></td>
<td>1.1</td>
<td>9560</td>
<td>6754</td>
<td>5666</td>
<td>5326</td>
<td>5788</td>
<td>6008</td>
<td>6235</td>
<td>7066</td>
<td>7405</td>
<td>8965</td>
<td>7969</td>
<td>8532</td>
</tr>
<tr>
<td>Average</td>
<td>20903</td>
<td>18034</td>
<td>15102</td>
<td>17374</td>
<td>20192</td>
<td>20401</td>
<td>15686</td>
<td>18223</td>
<td>17710</td>
<td>20862</td>
<td>18864</td>
<td>13607</td>
<td>21424</td>
</tr>
</tbody>
</table>
| Minimum | 903  | Maximum | 30000 | Median | 25200.5

Note: In red italics are the values above the median of all the values while bolded blue are values below the median.
7.0 Appendix II: A version of the GA code used for the experiment

Evolution.Java runs the different crossover function for any of the functions selected with the pre-defined parameters. Evolution_Age.Java is the same only that it runs the with different age limits. The MersenneTwisterFast.Java is the random number generator class and Individual.Java defines the structure of each individual in the population.

There are also two output file that are generated after each run: result.txt which has the details of the run and data.txt which keeps the summary.