An Investigation on Genetic Algorithm Parameters

Siamak Sarmady
School of Computer Sciences,
Universiti Sains Malaysia,
11800 Penang, Malaysia
[P-COM0005/07(R) , P-COM0088/07]
{sarmady@cs.usm.my, shaher5481561@yahoo.com}

Abstract – *Genetic algorithms provide a simple and almost generic method to solve complex optimization problems. Despite simplicity of it, genetic algorithm needs careful selection of settings like parent selection methods, mutation methods, population size ... to be able to find good solutions. Choosing unsuitable parameters and methods might result into longer program runs or even bad optimization results. In this report we use genetic algorithm in a sample "Bin Packing" problem. We implement and run the algorithm using different configurations and compare results. We then identify the best configuration among the tested parameters.*

Keywords: Genetic algorithm, optimization, bin packing, parameter selection, mutation, parent selection.

1 Introduction

Choosing parameters and methods in genetic algorithm might result into very different results. A good configuration might cause the algorithm to converge to best results in a short time while a worse setting might cause the algorithm to run for a long time before finding a good solution or even it might never be able to find a good solution. In this report we implement GA with different parent selection, mutation, recombination methods and also different population sizes. We will then try to identify which settings will work better in this problem's case.

Bin packing problem is about separating bottles of different colors into separate boxes. We have chosen 10 colors and 10 boxes for this purpose. Initially bottles with different colors are inside each box. We will separate bottles into boxes in a way that each box contains only one color. Box have unlimited capacity and our purpose is to minimize the moves between boxes.

To be able to test the software we choose an initial data set (random number of bottles from different colors in each box). We use a fixed set of input data to be able to compare performance of different methods in finding the best solution. We will investigate different settings to see how fast they can reach or find a best solution.

1.1 Representation

As we described earlier, movement of bottles between boxes will cause boxes to contain only a single color at the end of the movements. If we have 10 colors and 10 boxes then we might have "10!" different possible variations of solutions (in which every box contains a single color).

10 options	9 options	8 options	7 options	6 options	5 options	4 options	3 options	2 options	1 options

We can represent solutions with a character string of the length 10. We can then represent each color with one of the alphabets "a to j". Because box colors are not repetitive, this is called a permutation representation.

1.2 Fitness function

Our target in this problem is to minimize number of movements between boxes. Perhaps the best fitness function can be based on the number of necessary movements to achieve each solution. Calculation of the number of moves (in any selected solution) could be easily achieved. We present a sample calculation below.

Initial Colors in boxes: (number of bottles with colors a,b,c,d,e,f,g,h,i,j respectively)

2	13	12	5	21	1	16	6	14	4
4	21	14	7	4	2	1	3	8	5
5	15	7	12	15	13	8	8	7	12
6	12	9	21	8	23	19	12	3	14
7	17	17	4	9	11	3	3	2	4
8	18	12	9	6	16	6	11	9	16
9	19	34	11	13	4	15	13	12	6
10	20	23	4	12	8	9	3	11	4
16	11	19	16	12	12	2	2	16	8
3	8	13	7	5	9	4	7	2	2

Solution to be evaluated: `ibgcaedjhf`

i	b	g	c	a	e	d	j	h	f

We just sum up the number of colors which do not match each box's color. This gives us the number of bottles which should move out from a specific box. Now if we calculate the sum of move outs from all the boxes we will have the total necessary movements. For example for above solution the Fitness (or actually unfitness) function can be calculated as below.

2	13	12	5	0	1	16	6	14	4
4	0	14	7	4	2	1	3	8	5
5	15	7	0	15	13	8	8	7	12
6	12	9	21	8	23	0	12	3	14
7	17	17	4	9	0	3	3	2	4
8	18	12	9	6	16	6	11	9	0
9	19	0	11	13	4	15	13	12	6
10	20	23	4	12	8	9	3	0	4
0	11	19	16	12	12	2	2	16	8
3	8	13	7	5	9	4	0	2	2

54	133	126	84	84	88	64	61	73	59

Sum = 826 (Unfitness).

In our genetic algorithm we will try to minimize this unfitness function. (Above sample is one of the best results our software has been able to find with that specific input set)

2 Experimenting with simulation parameters and methods

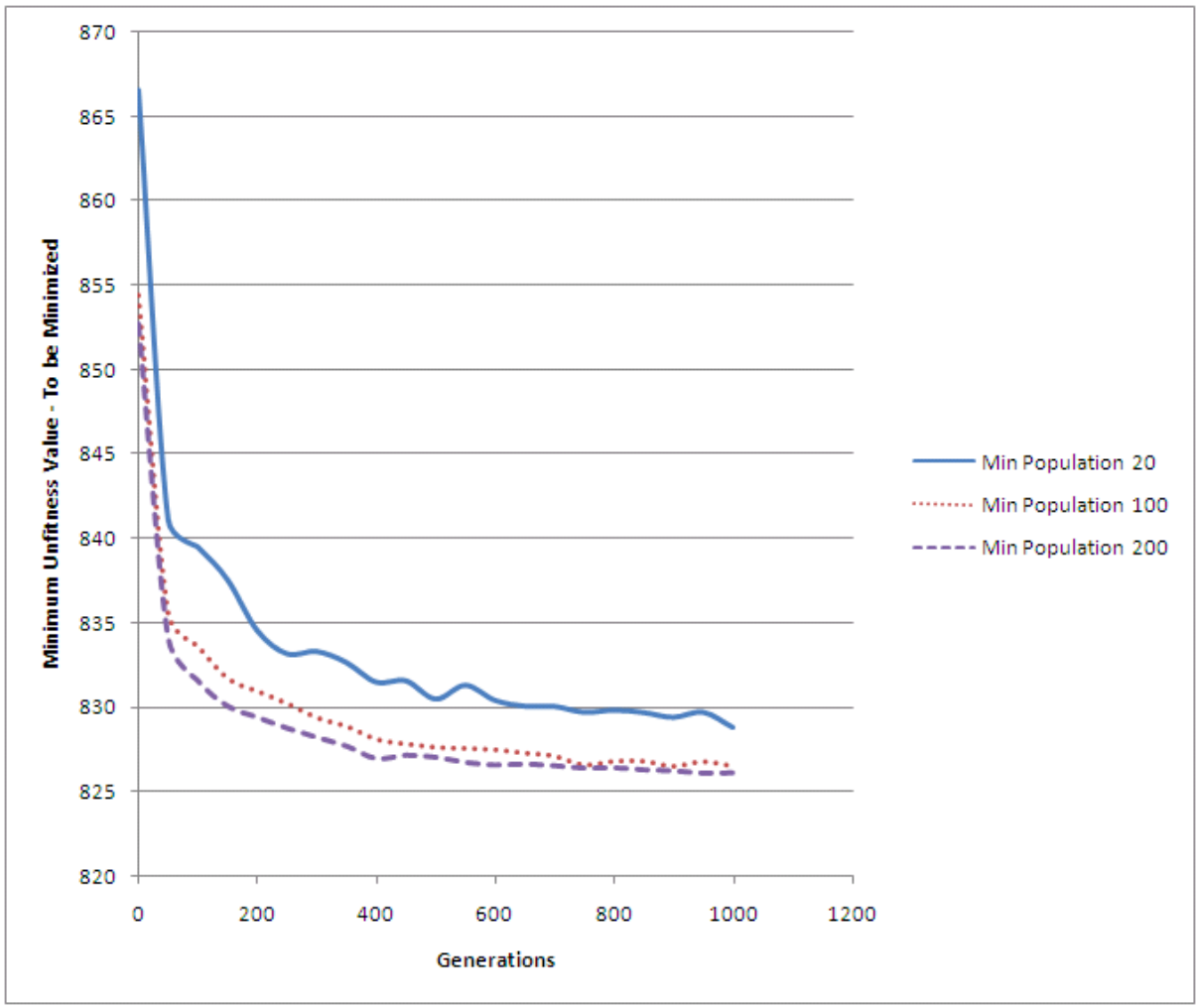
Because of the stochastic nature of the results we have developed the software in a way that each set of parameters has been run for 50 times to reach a reliable conclusion. For example to conclude about the "average fitness, maximum fitness, minimum fitness and diversity" of the population in Swap mutation method in 10 generations, we run the algorithm 50 times (every time with the same 10 generations) and calculate parameters by taking averages of them in those 50 runs. Next time we increase the generation number and repeat the 50 times to calculate parameters for that new generation level. This method has enabled us to run the algorithm more than 10,000 times with different settings to extract results for this report.

2.1 Population size

To be able to compare effect of changing the initial size of the population on genetic algorithm efficiency and results we needed to fix all the parameters except the population. The fixed configuration used for this

section is being described here. In this section, all of the individuals become parents and the product of recombination will double the size of the population. We then preserve the best half of the resulting population and put away the reaming. In this way size of the population will remain unchanged. Mutation is done by swapping two gene values. Recombination happen according "order 1" method which is one of the methods being used for permutation type of representations. Generation numbers are chosen from 0 (initial generated data without running the algorithm) to 1000 generations with the step size of 50 generations.

We have tested populations of 20,100 and 200. The algorithm has been run 50 times for each population size and each generation value. We have summarized the results in Graph 1. The graph only compares average of best found solutions (in population). Detailed Results are coming in Appendix A1 to A4.



Graph 1: Comparison of the effect of population size

Results show that higher population size provides a high diversity and therefore contains more sample solutions. As a result converging to better solutions happens sooner than smaller population sizes. Bigger population needs more time for the algorithm to run specially the time taken for sorting and evaluating the fitness of individuals is very CPU intensive. Smaller size of population looses the diversity very soon, before even finding a good solution. In our test the population size of 20 lost diversity (reached diversity of 1 individual type) before reaching an acceptable solution.

A population size of 200 does not provide much benefit over the population size of 100 with the consideration that the population size of 200 needs at least 2-4 times more CPU time. We therefore will use a population size of 100 in most of our experiments (when we need to fix population size and change and compare other parameters)

2.2 Mutation Method

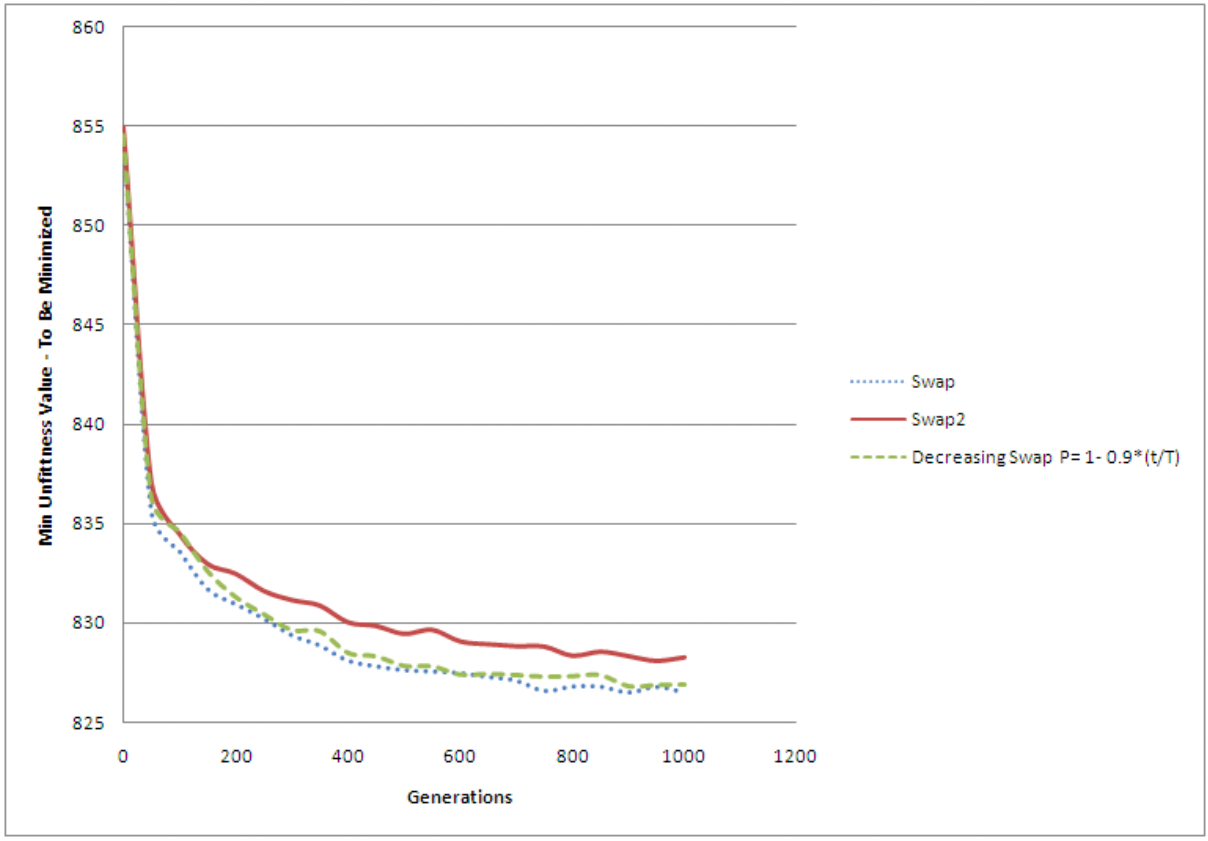
We compare mutation effect on converging to best answer again by fixing all other parameters except the mutation type. For this part again, all the individuals become parents and the product of recombination will double the size of the population. We then preserve the best half of the resulting population and put away the reaming. In this way size of the population will remain unchanged. "Order 1" method of recombination has been used in all of the tests in this section. Generation numbers are chosen from 0 (initial generated data) to 1000 generations with step size of 50. We have implemented and compare 3 types of mutations:

- Swap: In this method only 2 genes are being swapped in individuals and mutation is applied to 100% of the Children after recombination.
- Swap2: In this method 2 random genes are swapped with two other genes. Again 100% of the children are being mutated after recombination.
- Decreasing Swap: This is similar to Swap method with the difference that the mutation rate decreases as we go through the generations. The probability of applying mutation to children is determined by the formula: P= 1- 0.9 * (t/T)

  In above formula "T" is the total generations and "t" is the current generation number. As a result we will have less mutation in higher generations.

We run the algorithm 50 times for each setting the same as before. We have summarized the results in Graph 2. Detailed Results are coming in Appendix B1 to B4.

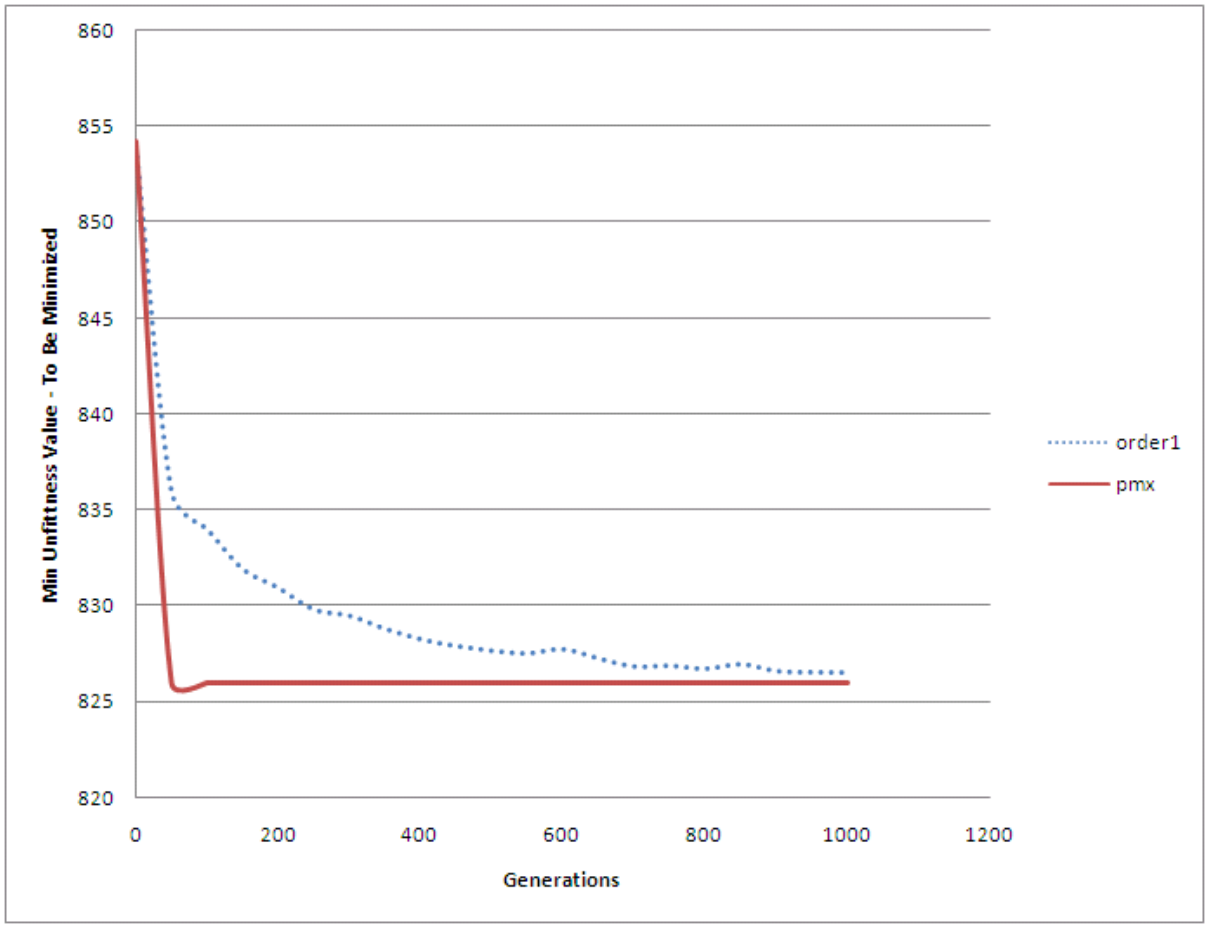
Graph 2 shows that Swap2 method has the least efficiency. This is perhaps because this method distorts the chromosomes more than needed. Normal swap and "decreasing swap" show very near results. We therefore use normal swap method in other parts of the report and change other parameters to investigate their effects.



Graph 2: Comparison of Mutation Methods on Converging to Best Answers (Min. Unfitness)

2.3 Recombination Method

We compare recombination effect on converging to best answer by fixing all other parameters except the crossover method. For this part again, all the individuals become parents and the product of recombination will double the size of the population. We then preserve the best half of the resulting population and put away the reaming. In this way size of the population will remain unchanged. Generation numbers are chosen from 0 (initial generated data) to 1000 generations with the step size of 50 generations. We have implemented and compared 2 types of crossover methods namely Oredr1 and PMX. We run the algorithm 50 times for each setting the same as before. The results are summarized in Graph 2. Detailed Results are coming in Appendix C1 to C3.



Graph 3: Comparison of Recombination Methods on Converging to Best Answers

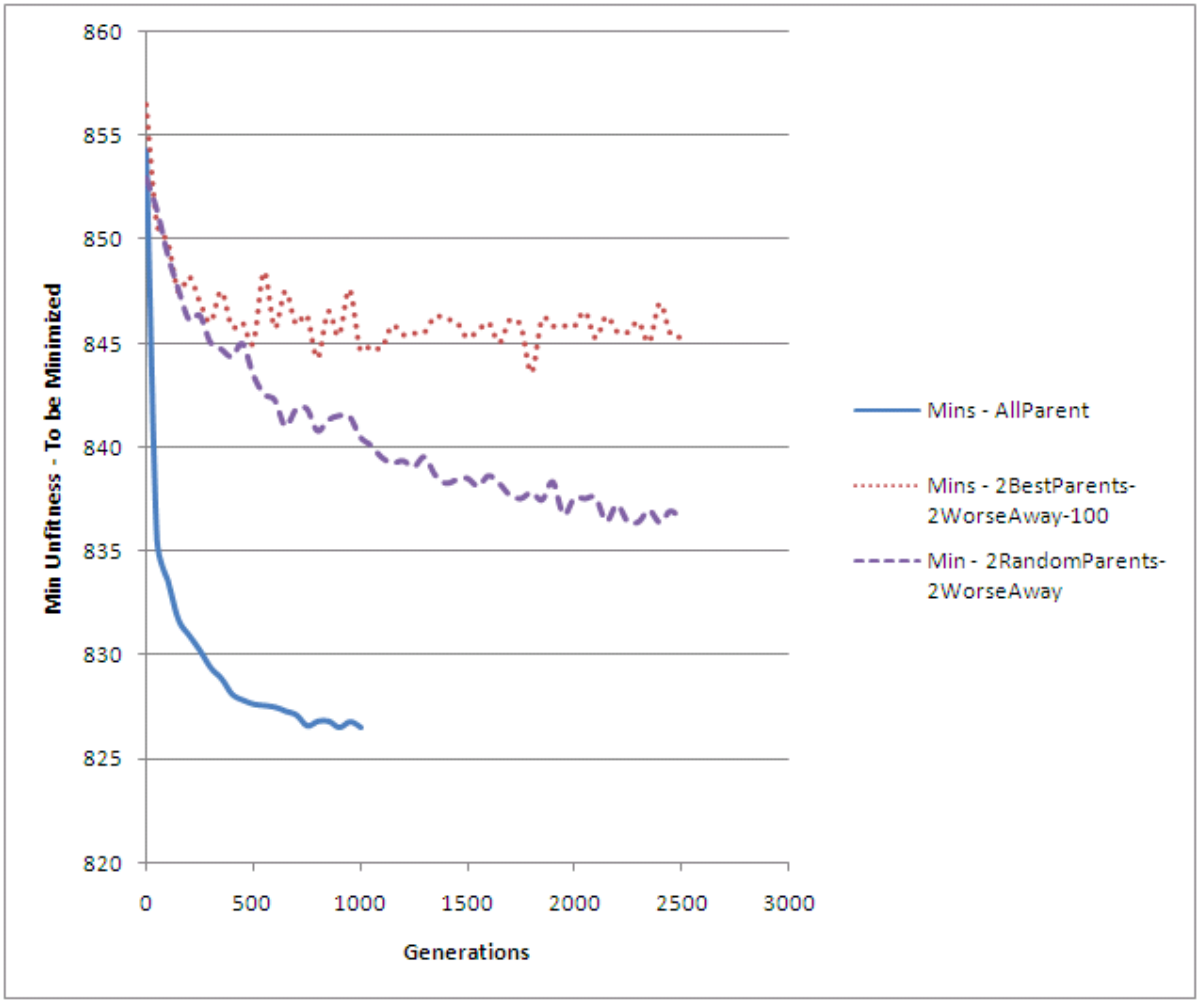
Graph 3 shows that PMX converges to good solutions much faster than “order 1” method. Actually using PMX method yield to best possible result (i.e. the one with the fitness of 826) in just 50 generations while the “order 1” method needs around 1000 generations to reach comparable results. This shows that the mutation and crossover methods can have very important effects on converging to good solutions and it worth to try other methods to see the effect.

2.4 Parent Selection Method

We compare parent selection method's effect on converging to best answer again by fixing all other parameters and methods except the parent selection. Population size is 100 individuals in this section, mutation type is normal swap and recombination is of the type "order 1".

- "AllParent-BestHalf": In this method which has been used in all other parts of this report, all the individuals become parents and produce the same amount of children as themselves. We then sort the new population (which its size has been doubled now) and put away the worst half and preserve the best half.
- "2BestParents-2WorseAway": In this method only 2 best individuals are chosen as the parents and produce 2 other individuals. We then sort the population (which now has 2 more individuals) and put away 2 worst individuals.
- "2RandomParents-2WorseAway": In this method only 2 random individuals are chosen as the parents and produce 2 other individuals. We then sort the population (which now has 2 more individuals) and put away 2 worst individuals.

We run the algorithm 50 times for each setting the same as before. We have summarized the results in Graph 4. Detailed Results are coming in Appendix D1 to D4.



Graph 4: Comparison of Parent Selection Methods on Converging to Best Answers

Graph 4 shows that "AllParent-bestHalf" method has been able to reach better solutions in considerably lower generations. This was predictable because larger number of individuals are being crossed over and mutated in this method and we are searching more areas of the solution space in each generation. Two other methods in comparison act only on 2 individuals each time and therefore chances of change and finding better results are very lower and in each generation because we only search 2 more spots in the solution space in every generation.

Between "2RandomParents-2WorseAway" and "2BestParents-2WorseAway" methods the first one is more convenient. This is because the method which acts on best 2 parents corrupts our best parents most of the time and therefore the evolution cannot take place very good. The other "2RandomParents-2WorseAway" method reaches better performance because it preserves best results and acts on 2 randomly chosen individuals and sometimes is able to add better parents to the population without distorting our best parents.

3 Software Implementation

We are using Sun JavaSE version 1.5 or higher to implement the software. Testing the software will need an installation of Java run time engine (JRE) and the "bin" directory of the JRE should be on PATH. Running the software is easily done by running the "run.bat" batch file. The entire software has been developed by us without using codes from internet.

3.1 Configuration file

The settings file of the software has a very flexible and easy format. Acceptable options and descriptions are provided in comments inside the configuration file. The configuration file should be inside the software directory under the name "settings.txt".

```
################################################################################
# Run Modes:
#       SingleRun, Runs only a single time and gives detailed results
#       StatisticalRun, runs several times and gives statisrical results
################################################################################
RunMode=StatisticalRun

################################################################################
# Global Settings, Applies to all run modes and settings:
#       Population, number of individuals in population
################################################################################
Population=100

################################################################################
# SingleRun Mode Settings:
#       Generations, number of generations
################################################################################
Generations=1000

################################################################################
# StatisticalRun Mode Settings:
#       NumRuns, run each exact setting how many times to extract avg of results
#       GenerationsStart, Start statistical run with how many generations
#       GenerationsIncrease, Increase number of generations with which step size
#       GenerationsEnd, End the statistical run in how many generations
################################################################################
NumRuns=50
GenerationsStart=0
GenerationsIncrease=50
GenerationsEnd=1000

################################################################################
# ParentSelection and Survival Method:
# AllParent-BestHalf ,All Individuals Parent - Best Half of the Population
# 2BestParents-2WorseAway , 2 best indivs become parent, two worst go away
# 2RandomParents-2WorseAway , 2 Random become parent, two worst go away
################################################################################
```

```
ParentSelectionMethod=AllParent-BestHalf

################################################################################
# MutuationMethod:
# Swap , Swap 2 genes
# DecreasingSwap , Swap genes with decreasing rate as generations increase
# Swap2 , swap 2 times meaning 4 genes will be swaped
################################################################################
MutuationMethod=Swap

################################################################################
# RecombineMethod:
# order1, order1 crossover method
# pmx, pmx crossover method
################################################################################
RecombineMethod=pmx
```

Code 1: Software Configuration File

3.2 Input and Output files format

To be able to change input data and also to make import of the output data into other software easy, we are using the "csv" delimited text format for both input and output files. Input file is always needed for the operation of the software while the output file is only created in "StatisticalRun" mode.

Each line of input file resembles initial bottles (of different colors) in each of 10 boxes. We have currently used a fixed data set for our entire tests but it is very easy to change the data.

As mentioned earlier, output file format is also in "csv" delimited text format. Columns of data are "Average of fitness of individuals in the population during test runs", "Average of minimum fitness during the test runs", "Average of maximum fitness during test runs" and "Diversity of population during test runs".

3.3 Sample results of "SingleRun" mode

A sample of the detailed results provided by the software is presented here. Understanding these results should be easy for anyone who knows about genetic algorithm and the problem we were trying to solve.

```
Parent Selection Method : AllParent-BestHalf
Crossover Method : pmx
Mutation Method : Swap
Population : 100
Generations : 1000

Listing All 100 members

1)Ind(92605): ibgdceajhf   UnFitness: 826
2)Ind(93722): ibgdceajhf   UnFitness: 826
3)Ind(95527): ibgdceajhf   UnFitness: 826
4)Ind(92348): ibgdceajhf   UnFitness: 826
... (removed from report to save space)...
64)Ind(94839): ibgdecajhf   UnFitness: 830
65)Ind(94093): ibgdcehjaf   UnFitness: 830
... (removed from report to save space)...
99)Ind(95362): ibgdcjaehf   UnFitness: 832
100)Ind(95362): ibgdcjaehf   UnFitness: 832

Average UnFitness = 828  Min UnFitness = 826  Max UnFitness = 832 Diverse types
are 23

Listings Diverse Types:
ibgdceajhf(826)
ibgcaedjhf(826)
hbgdceajif(827)
```

```
hbgcaedjif(827)
ebgicdajhf(828)
ibgdaecjhf(828)
ebgcadhjif(828)
ibgcedajhf(829)
hbgdaecjif(829)
ebgdachjif(829)
ibgcadejhf(830)
ibgjaedchf(830)
ihgcaedjbf(830)
ibgdecajhf(830)
ibgdcehjaf(830)
ebgdafhjic(830)
ihgdceajbf(830)
ebgiadcjhf(830)
ebgiafdjhc(830)
hbgcedajif(830)
ebgchdajif(830)
ibgdaejchf(831)
ibgdcjaehf(832)
```

Code 2: Output Sample in "SingleRun" mode

3.4 Sample results of "StatisticalRun" mode

In this section we bring a sample output of the program in "StatisticalRun" mode on console. Summerization of these details is saved in an "output.txt" file inside the software directory after each "StatisticalRun".

```
Parent Selection Method : AllParent-BestHalf
Crossover Method : pmx
Mutation Method : Swap
Population : 100
Generations : 0

Run# 1 : Average UnFitness = 894  Min UnFitness = 863  Max UnFitness = 930 Diverse types are 100
... (removed from report to save space)...
Run# 50 : Average UnFitness = 895  Min UnFitness = 854  Max UnFitness = 932 Diverse types are 100

Parent Selection Method : AllParent-BestHalf
Crossover Method : pmx
Mutation Method : Swap
Population : 100
Generations : 50

Run# 1 : Average UnFitness = 830  Min UnFitness = 826  Max UnFitness = 833 Diverse types are 39
Run# 2 : Average UnFitness = 829  Min UnFitness = 826  Max UnFitness = 832 Diverse types are 36
... (removed from report to save space)...
Run# 50 : Average UnFitness = 830  Min UnFitness = 826  Max UnFitness = 833 Diverse types are 44



... (removed from report to save space)...



Parent Selection Method : AllParent-BestHalf
Crossover Method : pmx
Mutation Method : Swap
Population : 100
Generations : 1000

Run# 1 : Average UnFitness = 829  Min UnFitness = 826  Max UnFitness = 833 Diverse types are 34
Run# 2 : Average UnFitness = 829  Min UnFitness = 826  Max UnFitness = 832 Diverse types are 30
... (removed from report to save space)...
Run# 50 : Average UnFitness = 828  Min UnFitness = 826  Max UnFitness = 830 Diverse types are 22


Generations = 0 , Average Results (50 runs): Avg UnFitness = 894.16  Min UnFitness = 854.86  Max
UnFitness = 930.34  Avg Diversity = 100.0
```

```
Generations = 50 , Average Results (50 runs): Avg UnFitness = 829.7  Min UnFitness = 826.0  Max
UnFitness = 832.68  Avg Diversity = 41.8
Generations = 100 , Average Results (50 runs): Avg UnFitness = 828.34  Min UnFitness = 826.0  Max
UnFitness = 831.5  Avg Diversity = 26.78
Generations = 150 , Average Results (50 runs): Avg UnFitness = 828.2  Min UnFitness = 826.0  Max
UnFitness = 831.42  Avg Diversity = 25.74
Generations = 200 , Average Results (50 runs): Avg UnFitness = 828.36  Min UnFitness = 826.0  Max
UnFitness = 831.6  Avg Diversity = 26.58
Generations = 250 , Average Results (50 runs): Avg UnFitness = 828.32  Min UnFitness = 826.0  Max
UnFitness = 831.38  Avg Diversity = 25.88
Generations = 300 , Average Results (50 runs): Avg UnFitness = 828.28  Min UnFitness = 826.0  Max
UnFitness = 831.54  Avg Diversity = 26.56
Generations = 350 , Average Results (50 runs): Avg UnFitness = 828.2  Min UnFitness = 826.0  Max
UnFitness = 831.2  Avg Diversity = 25.34
Generations = 400 , Average Results (50 runs): Avg UnFitness = 828.18  Min UnFitness = 826.0  Max
UnFitness = 831.48  Avg Diversity = 25.56
Generations = 450 , Average Results (50 runs): Avg UnFitness = 828.3  Min UnFitness = 826.0  Max
UnFitness = 831.84  Avg Diversity = 27.22

... (removed from report to save space)...

Generations = 900 , Average Results (50 runs): Avg UnFitness = 828.32  Min UnFitness = 826.0  Max
UnFitness = 831.54  Avg Diversity = 26.26
Generations = 950 , Average Results (50 runs): Avg UnFitness = 828.18  Min UnFitness = 826.0  Max
UnFitness = 831.58  Avg Diversity = 26.68
Generations = 1000 , Average Results (50 runs): Avg UnFitness = 828.3  Min UnFitness = 826.0  Max
UnFitness = 831.32  Avg Diversity = 25.92
```

Code 3: Output Sample in "StatisticalRun" mode

4 Conclusion and Future Work

In this report we experienced different parameter types and methods on a single problem and we were able to see the effects of changes. We want to mention that though we have tried to extract more reliable results by running each configuration for 50 times and then calculating the averages, these results are only valid for this specific problem and the specific chromosome representation we have chosen.

Even with this problem, effects of the changes are only valid if done with exact order we did. For example we have only tested different mutation and crossover methods on a population with a size of 100 individuals. Results might be different with other population sizes. Also effects of changing multiple parameters and methods are not predictable because parameters are not completely independent from each other and might have effects on others.

5 Acknowledgment

I want to thank "Associate Professor Dr. Tajudin Khader" for the evolutionary computing course we had with him and for the very enjoyable experience and valuable background it provided to us. We hope the knowledge will help us in our future work and research in computer science field.