NEAR EAST UNIVERSITY

GRADUATE SCHOOL OF APPLIED AND SOCIAL SCIENCES

COMPARISON OF SELECTION SCHEMES OF GENETIC ALGORITHM IN NUMERICAL OPTIMIZATION

BSc. Ozan Akkoca

MASTER THESIS

Department of Computer Engineering

Nicosia-2007

ACKNOWLEDGEMENTS

I would like to express my gratitude to my supervisor Assoc. Professor Dr. Adil Amircanov for his constant guidance and continuous support. He followed the whole process of this thesis very closely and suggested useful ideas and insights all the way long. The frequent and stimulating discussions we had at our regular meetings made it all happen. Many thank to him.

I am very grateful to Assoc. Professor Dr. Rahib Abiyev for his help and answering any question I asked him.

I would like to thank my best friends Yusuf Şen, Engin Çiloğlu, Özgür Pehlivan, Hüseyin Mısır and Hayrettin Tektunalı for their support, also expressing my apology to all my friends that I could not mention here personally one by one.

Especially, I would like to express my apologies to my friend Begüm Gül. I did not want to hurt and disappoint her while preparing this thesis. Really, I am so sorry.

Finally, I could never have prepared this thesis without encouragement, unconditional support and love of my family. Many thank to them.

ABSTRACT

Genetic Algorithms are a common probabilistic optimization methods based on the model of natural evolution. A genetic algorithm mainly composed of three genetic operations, which are selection, crossover and mutation operation. Selection operation uses some schemes such as proportionate-based selection scheme and ordinal-based selection scheme to select "good" individuals from the population to insert into mating pool. Then selected individuals from the mating pool are used by a recombination operator "crossover and mutation" to generate new fitness offspring for next generation.

This thesis presents the comparison of selection schemes of genetic algorithm in numerical optimization with various selection schemes. The simulation results are studied by using various test cases. The quantitative analysis of the selection strategies is depicted and the numerical experiments show that genetic algorithm with ordinal-based selection strategy converges faster than with proportionate-based selection, schemes. But for the some multimodal test functions the quality of the solution obtained is better for proportionate-based selection schemes than for ordinal-based one.

CONTENTS

ACKNOWLEDGEMENTS .... **i**

ABSTRACT .... **ii**

CONTENTS .... **iii**

LIST of TABLES .... **v**

LIST of FIGURES .... **vi**

INTRODUCTION **1**

CHAPTER ONE GENETIC ALGORITHM **4**

1.1- Genetic Algorithm 4

1.1.1- Termination Conditions of Genetic Algorithm 6

1.1.2- Pseudo Code of Genetic Algorithm 7

1.2- Working Principle of Genetic Algorithm 8

1.3- Steps of Genetic Algorithm 9

1.4- Parameters of Genetic Algorithm 12

1.5- Representation of a Chromosome in Search Space 14

1.5.1- Encoding and Decoding of Parameters in Genetic Algorithm 15

1.5.2- Parameter Mapping 15

1.5.3- Encoding Multi Parameters 17

1.5.4- Decoding Multi Parameters 17

CHAPTER 2 SELECTION OPERATORS **18**

2.1- Selection Schemes 18

2.1.1- Basic Selection Schemes 19

2.2- Proportionate-Base Selection 19

2.2.1- Proportionate (Roulette-Wheel) Selection 19

2.2.2- Stochastic Universal Sampling Selection 23

2.3- Ordinal-Base Selection 25

2.3.1- Tournament Selection 25

2.3.2- Truncation Selection 28

2.3.3- Linear Ranking Selection 30
2.3- Comparison of Selection Schemes 30
2.3.1- Selection Probability 30
2.3.1.1- Selection Probability for Tournament Selection 30
2.3.1.2- Selection Probability for Linear Ranking Selection 30
2.3.1.3- Selection Probability for Roulette-Wheel Selection 31
2.3.1.4- Selection Probability for S.U.S Selection 31
2.3.1.5- Comparison of Selections Probability 32
2.3.2- Selection Intensity 32
2.3.2.1- Selection Intensity for Tournament Selection 33
2.3.2.2- Selection Intensity for Linear Ranking Selection 33
2.3.2.3- Selection Probability for Proportionate-Base Selections 33
2.3.2.4- Comparison of Selections Intensity 34

CHAPTER 3 RECOMBINATION OPERATORS 35
3.1- Crossover Operators 35
3.1.1- K-Point Crossover 36
3.1.2- Uniform Crossover 38
3.1.3- Uniform Order Based Crossover 40
3.1.4- Order Based Crossover 41
3.1.5- Partially Matched Crossoer (PMX) 42
3.1.6- Cycle Crossover (CX) 43
3.2- Mutation Operators 44
3.2.1- Flip Bit Mutation 45
3.2.2- Boundary Mutation 45
3.2.3- Uniform Mutation 45
3.2.4- Non-Uniform Mutation 46
3.2.5- Gaussian Mutation 46

CHAPTER 4 BENCHMARK TEST FUNCTIONS 47
4.1- F1- Rosenbrock's Function 47

4.2- F2- De Jong's Test Function 1 49
4.3- F3- Griewangk's Function 50
4.4- F4- Michalewicz's Function 53
4.5- F5- Shekel's Function 56

CHAPTER 5 IMPLEMENTATION of GENETIC ALGORITHM 58
5.1- Implementation of Genetic Algorithm 58
5.1.1- Parameter Selection for a Genetic Algorithm Program 59
5.2- C Codes of Genetic Algorithm's for Program Design 60
5.2.1- Genetic Algorithm Program Files 60
5.2.2- Genetic Algorithm Program Functions 61

CHAPTER 6 ANALYSIS of COMPUTATIONAL RESULTS 69
6.1- Computational Test Results 69
6.1.1- Roulette-Wheel Selection Test Results 70
6.1.2- Tournament Selection Test Results 80
6.1.3- Comparison of Computational Test Results 90
6.2- Revision 92

CONCLUSION 94
REFERENCES 95
APPENDIX A A-1
APPENDIX B B-1

List of Tables

Table 2.1- Individuals fitness values for roulette-wheel selection. 22
Table 2.2- Individuals fitness values of tournament selection. 27
Table 2.3- Comparison of selection schemes according to selection intensity. 34

Table 5.1- Comparison of empirically determined GA parameter settings [25]. 59

Table 6.1- List of five test functions. 69
Table 6.2- Test results of roulette-wheel selection for F1. 70
Table 6.3- Test results of roulette-wheel selection for F2. 72
Table 6.4- Test results of roulette-wheel selection for F3. 74
Table 6.5- Test results of roulette-wheel selection for F4. 76
Table 6.6- Test results of roulette-wheel selection for F5. 78
Table 6.7- Test results of tournament selection for F1. 80
Table 6.8- Test results of tournament selection for F2. 82
Table 6.9- Test results of tournament selection for F3. 84
Table 6.10- Test results of tournament selection for F4. 86
Table 6.11- Test results of tournament selection for F5. 88
Table 6.12- Convergence speed statistics for roulette-wheel selection. 90
Table 6.13- Convergence speed statistics for tournament selection. 90
Table 6.14- Comparison of convergence speed. 90
Table 6.15- Roulette-wheel selection fitness statistics. 91
Table 6.16- Tournament selection fitness statistics. 91
Table 6.17- Comparison of comparison of quality of global optimum. 92

Table A.1- Test results of stochastic universal sampling selection for F1. .... A-1
Table A.2- Test results of stochastic universal sampling selection for F2. .... A-3
Table A.3- Test results of stochastic universal sampling selection for F3. .... A-5
Table A.4- Test results of stochastic universal sampling selection for F4. .... A-7
Table A.5- Test results of stochastic universal sampling selection for F5. .... A-9
Table A.6- Convergence speed statistics for stochastic universal sampling selection. A-11
Table A.7- Stochastic universal sampling selection fitness statistics. A-11

List of Figures

Figure 1.1- Flowchart of genetic algorithm. 5

Figure 1.2- This flowchart illustrates the basic steps in a genetic algorithm....................12
Figure 1.3- Chromosome and gene representation in a population. 23
Figure 1.4- Decoding and encoding process. 15

Figure 2.1- Pseudo code of proportionate selection. 20
Figure 2.2- Flowchart of roulette-wheel selection. 21
Figure 2.3- Proportions of individuals in a population..23
Figure 2.4- Pseudo code of stochastic universal sampling selection. 24
Figure 2.5- Stochastic universal sampling selection example. 24
Figure 2.6- Pseudo code of tournament selection. 25
Figure 2.7- Flowchart of tournament selection. 26
Figure 2.8- Pseudo code of truncation selection. 28
Figure 2.9- Pseudo code of linear ranking selection. 29

Figure 3.1- One point crossover. 37
Figure 3.2- Two point crossover. 38
Figure 3.3- Uniform crossover. 39
Figure 3.4- Uniform order based crossover. 41
Figure 3.5- Order based crossover. 42
Figure 3.6- Partially matched crossover. 43
Figure 3.7- Cycle crossover. 44
Figure 3.8- Flip bit mutation. 45

Figure 4.1- Full definition range of the function. 48
Figure 4.2- Focus around the area of the global optimum at [1, 1]. 48
Figure 4.3- Surf plot of the function in a very large area from -500 to 500 for each of both variables. 49
Figure 4.4- The function at a smaller area from -10 to 10. 50
Figure 4.5- Full definition area from -500 to 500. 51
Figure 4.6- Inner area of the function from -50 to 50. 52
Figure 4.7- Area from -8 to 8 around the optimum at [0, 0]. 53

Figure 4.8- Surf plot in an area from 0 to 3 for the first and second variable. 54
Figure 4.9- Area around the optimum. 54
Figure 4.10- Same as Figure 4.8 for the third and forth var., var. 1 and 2 are set 0. 55
Figure 4.11- Graphics of Shekel's function. 56

Figure 5.1- C code of roulette-wheel selection. 61
Figure 5.2- C code of tournament selection. 62
Figure 5.3- C code of flip bit mutation. 63
Figure 5.4- C code of one point crossover with flip bit mutation. 64
Figure 5.5- C code of decoding a parameter. 64
Figure 5.6- C code of mapping a parameter. 65
Figure 5.7- C code of extracting a parameter from a full string. 65
Figure 5.8- C code of parameter decoding. 66
Figure 5.9- C code of flipping a biased coin. 67
Figure 5.10- C code of main function. 68

Figure 6.1- Roulette-wheel selection graphic for F1. 71
Figure 6.2- Roulette-wheel selection graphic for F2. 73
Figure 6.3- Roulette-wheel selection graphic for F3. 75
Figure 6.4- Roulette-wheel selection graphic for F4. 77
Figure 6.5- Roulette-wheel selection graphic for F5. 79
Figure 6.6- Tournament selection graphic for F1. 81
Figure 6.7- Tournament selection graphic for F2. 83
Figure 6.8- Tournament selection graphic for F3. 85
Figure 6.9- Tournament selection graphic for F4. 87
Figure 6.10- Tournament selection graphic for F5. 89

Figure A.1- Stochastic universal sampling selection graphic for F1. .... A-2
Figure A.2- Stochastic universal sampling selection graphic for F2. .... A-4
Figure A.3- Stochastic universal sampling selection graphic for F3. .... A-6
Figure A.4- Stochastic universal sampling selection graphic for F4. .... A-8

Figure A.5- Stochastic universal sampling selection graphic for F5. .... A-10

INTRODUCTION

Genetic algorithms are robust and adaptive methods for solving a wide range of the global numerical optimization problems [14].

By using the genetic algorithm to solve a problem, first present the candidate solutions as a sequence of values, and define an evaluation function to evaluate the candidate solutions. One population consists of a certain number of individuals which serve as candidate solutions. New generation of population is created by genetic operations such as selection, crossover and mutation in iteration.

According to the Darwinian principles of survival, which is called "the survival of the fittest", the excellent individuals have far more chances to adapt themselves to the environment and survive, while the inferior ones die out [8]. The survivals reproduce new individuals with better genes which make the new generation more endurable to the nature. The genetic algorithm uses this process of reproduction as a basic genetic operation for the algorithms.

The genetic algorithm uses a selection scheme to select individuals from the population to insert into a mating pool. Individuals from the mating pool are used by a recombination operator to generate new offspring, with the resulting offspring forming the basis of the next generation. Since the individuals in the mating pool pass their genes on to the next generation, it is desirable that the mating pool comprise "good" individuals. A selection scheme in the genetic algorithm is simply a process that favors the selection of better individuals in the population for the mating pool. The selection operator is intended to improve the average quality of the population by giving individuals of higher quality a higher probability to be copied in to the next generation. So selection schemes are very important for the genetic algorithm to reach the better solutions.

There are two basic types of selection schemes commonly used today: proportionate-base selection and ordinal-based selection [2].

Proportionate-based selection selects individuals on the basis of their fitness values relative to the fitness of the other individuals in the population. Some common proportionate-based schemes are proportionate selection (roulette-wheel) [4] and stochastic universal selection [12, 6].

Ordinal-based selection schemes select individuals not according to their fitness, but on the basis of their rank within the population. The individuals are ranked according to their fitness. Some common ordinal-based selection schemes are tournament selection [9, 10], linear ranking selection and truncation selection [11].

The aim of this thesis is comparison of selection schemes of genetic algorithm in numerical optimization with various selection schemes such as proportionate-based scheme and ordinal-based scheme. For proportionate-based scheme roulette-wheel selection, and for ordinal-based scheme binary tournament selection is taken as representative because for each group, chosen selection methods has less selection intensity for its group. Also for proportionate selection scheme, stochastic universal sampling selection method simulated but not considered for the thesis purpose but also its simulated test result are given in Appendix A. The simulation results are studied by using various test cases such as speed of convergence to the global optimum, and the quality of optimal solution.

The first chapter gives brief information about idea of a genetic algorithm and working principle step by step. In addition to this, discusses of chromosome representation in search space and encoding, decoding, mapping techniques of the parameters in a search space.

The second chapter discusses briefly the proportionate-base selection and ordinal-based selection schemes. The subsequent titles deal with the selection methods such as,

proportionate selection, stochastic universal sampling selection, linear ranking selection, tournament selection and truncation selection algorithms and gives selection algorithms pseudo codes. Also mentions the comparisons of selection schemes according to selection intensity and selection probability.

The third chapter shows the recombination operators “crossover and mutation”. Crossover techniques such as k point crossover, uniform crossover and order based crossover and mutation techniques as flip bit mutation, uniform mutation are given.

The forth chapter describes the test functions “Rosenbrock Function, De Jong’s Function 1 (Sphere Model), Griewank’s Function, Michalewicz’s Function, Shekel’s Function”. The test functions properties and their design are mentions.

The fifth chapter gives information about the designing a computer program for genetic algorithm, the genetic algorithm program is written with C programming language and discusses program files and program functions. Subtitles give information about implementation of genetic algorithm operators and parameter selection for recombination operators.

The last chapter gives the computational test results of tournament selection and roulette-wheel selection. Also comparison of the selection schemes given according to test results and analyze.

CHAPTER ONE
GENETIC ALGORITHM

1.1- Genetic Algorithm

The genetic algorithm is a model of machine learning which derives its behavior from a metaphor of the processes of evolution in nature. This is done by the creation within a machine of a population of individuals represented by chromosomes, in essence a set of character strings that are analogous to the base-4 chromosomes that seen in humans DNA. The individuals in the population then go through a process of evolution.

In nature, the encoding of genetic information (genome) is done in a way that admits asexual reproduction (such as by budding) typically results in offspring that are genetically identical to the parent. Sexual reproduction allows the creation of genetically radically different offspring that are still of the same general flavor (species).

At the molecular level what occurs is that a pair of chromosomes bump into one another, exchange chunks of genetic information and drift apart. This is the recombination operation, which genetic algorithm generally refers to as crossover because of the way that genetic material crosses over from one chromosome to another.

The crossover operation happens in an environment where the selection of who gets to mate is a function of the fitness of the individual, i.e. How good the individual is at competing in its environment. Some genetic algorithms use a simple function of the fitness measure to select individuals (probabilistically) to undergo genetic operations such as crossover or asexual reproduction [4]. This is fitness-proportionate selection. Other implementations use a model in which certain randomly selected individuals in a subgroup competes and the fittest is selected [9]. This is called tournament selection. The two processes that most contribute to evolution are crossover and fitness based selection/reproduction.

Mutation also plays a role in this process, although how important its role is continues to be a matter of debate (some refer to it as a background operator, while others view it as playing the dominant role in the evolutionary process) [13]. It cannot be stressed too strongly that the genetic algorithm (as a simulation of a genetic process) is not a random search for a solution to a problem (highly fit individual). The genetic algorithm uses stochastic processes, but the result is distinctly non-random (better than random).

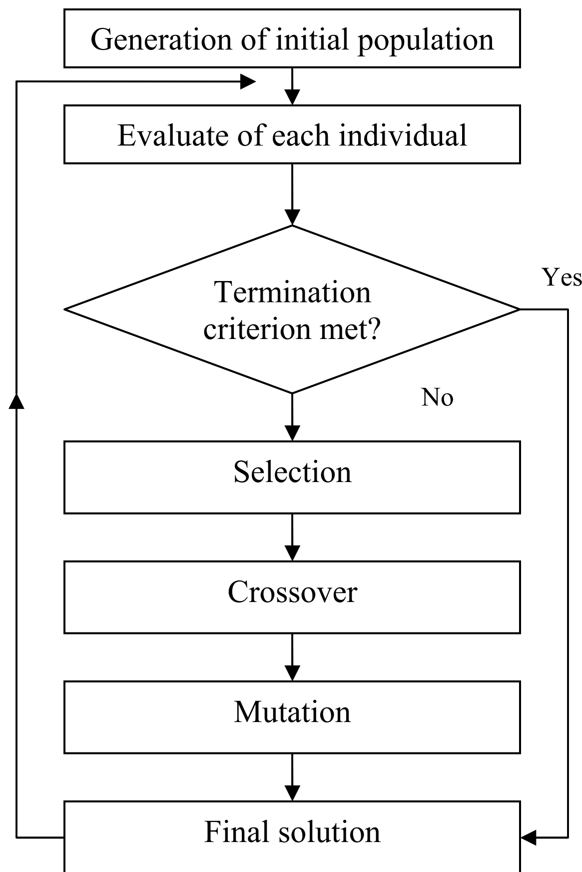

Figure 1.1- Flowchart of genetic algorithm.

Genetic algorithms are used for a number of different application areas [1]. An example of this would be multidimensional optimization problems in which the character string of the chromosome can be used to encode the values for the different parameters being optimized.

In practice, therefore, the implementation of this genetic model of computation by having arrays of bits or characters to represent the chromosomes. Simple bit manipulation operations allow the implementation of crossover, mutation and other operations. Although a substantial amount of research has been performed on variable- length strings and other structures, the majority of work with genetic algorithm is focused on fixed-length character strings.

When the genetic algorithm is implemented it is usually done in a manner that involves the following cycle: Evaluate the fitness of all of the individuals in the population. Create a new population by performing operations such as crossover, fitness-proportionate reproduction and mutation on the individuals whose fitness has just been measured. Discard the old population and iterate using the new population.

An iteration of this loop is referred to as a generation. There is no theoretical reason for this as an implementation model. Indeed, do not see this punctuated behavior in populations in nature as a whole, but it is a convenient implementation model.

The first generation (generation 0) of this process operates on a population of randomly generated individuals. From there on, the genetic operations, in concert with the fitness measure, operate to improve the population.

1.1.1- Termination Conditions of Genetic Algorithm

The above generational process is repeated until a termination condition has been reached.
Common terminating conditions are;
A solution is found that satisfies minimum or maximum criteria.
Fixed number of generations reached.
Allocated budget (computation time/money) reached.
The highest ranking solution's fitness is reaching or has reached a plateau such that successive iterations no longer produce better results.
Combinations of the above conditions.

1.1.2- Pseudo Code of Genetic Algorithm

```
BEGIN GA

        // start with an initial time
        t := 0;
        // initialize a usually random population of individuals
        initpopulation P (t);
        // evaluate fitness of all initial individuals of population
        evaluate P (t);
        // test for termination criterion (time, fitness, etc.)
        while not done do

              // increase the time counter
              t := t + 1;
              // select a sub-population for offspring production
              P' := selectparents P (t);
              // recombine the "genes" of selected parents
              recombine P' (t);
              // perturb the mated population stochastically
              mutate P' (t);
              // evaluate it's new fitness
              evaluate P' (t);
              // select the survivors from actual fitness
              P := survive P, P' (t);

        Od

   END GA.
```

1.2- Working Principle of Genetic Algorithm

Genetic algorithm encodes the decision variables of a search problem into finite-length strings of alphabets of certain cardinality. The strings which are candidate solutions to the search problem are referred to as chromosomes, the alphabets are referred to as genes and the values of genes are called alleles. For example, in contrast to traditional optimization techniques, Genetic algorithm works with coding of parameters, rather than the parameters themselves [13].

To evolve good solutions and to implement natural selection needed a measure for distinguishing good solutions from bad solutions. The measure could be an objective function that is a mathematical model or a computer simulation, or it can be a subjective function where humans choose better solutions over worse ones. In essence, the fitness measure must determine a candidate solution's relative fitness, which will subsequently be used by the genetic algorithm to guide the evolution of good solutions.

Another important concept of genetic algorithm is the notion of population. Unlike traditional search methods, genetic algorithm relies on a population of candidate solutions. The population size, which is usually a user-specified parameter, is one of the important factors affecting the scalability and performance of genetic algorithms. For example, small population sizes might lead to premature convergence and yield substandard solutions [14]. On the other hand, large population sizes lead to unnecessary expenditure of valuable computational time.

Once the problem is encoded in a chromosomal manner and starts with a set of solutions (represented by chromosomes) called population. Solutions from one population are taken and used to form a new population. This is motivated by a hope, that the new population will be better than the old one. Solutions which are selected to form new solutions (offspring) are selected according to their fitness - the more suitable they are the more chances they have to reproduce.

The genetic algorithm is composed of three genetic operations: selection, crossover and mutation .The genetic algorithm uses the steps as below:

Step1. Generate initial random population with n chromosomes.
Step2. Evaluate the fitness value of each individual in the population.
Step3. Perform sub-steps as follows to create new population. Repeat until the new population is completed.

3.1. According to the fitness value, individuals are chosen with a probability. Replicate the selected ones to form a new population (Selection).

3.2. Create two new individuals by two parents who are selected probabilistically from the population and recombine them at the crossover point (Crossover).

3.3. Create a new individual by mutating an existing individual with the probabilistically selected (Mutation).

3.4. Place new offspring in a new population (Replacement).

Step4. Use new generated population for next generation.
Step5. If the end condition is satisfied, stop, and return the best solution in current population. Else go to step2.

1.3- Steps of Genetic Algorithm

Step1. Initialization

The initial population of candidate solutions is usually generated randomly across the search space. However, domain specific knowledge or other information can be easily incorporated.

Step2. Evaluation

Once the population is initialized or an offspring population is created, the fitness values of the candidate solutions are evaluated.

Step3. Perform sub-steps as follows to create new population. Repeat until the new population is completed.

Step3.1. Selection

Selection allocates more copies of those solutions with higher fitness values and thus imposes the "survival of the fittest" mechanism on the candidate solutions.

The main idea of selection is to prefer better solutions to worse ones, and many selection procedures have been proposed to accomplish this idea, including roulette-wheel selection, stochastic universal selection, linear ranking selection and tournament selection (some of the selection methods discussed in chapter 2).

Step3.2. Crossover

Crossover combines parts of two or more parental solutions to create new, possibly better offspring. There are many ways of accomplishing this (some of which are discussed in chapter 3), and competent performance depends on a properly designed recombination mechanism. The offspring under recombination will not be identical to any particular parent and will instead combine parental traits in a novel manner.

Step3.3. Mutation

While recombination operates on two or more parental chromosomes, mutation locally but randomly modifies a solution. Again, there are many variations of mutation, but it usually involves one or more changes being made to an individual's trait or traits. In other words, mutation performs a random walk in the vicinity of a candidate solution (some of the mutation techniques are discussed in chapter 3).

Step3.4. Replacement Techniques

Once the new offspring solutions are created using crossover and mutation then need to introduce them into the parental population. There are many ways to approach this. Bear in mind that the parent chromosomes have already been selected according to their fitness, so hoping that the children (which include parents which did not undergo crossover) are among the fittest in the population and so hope that the population will gradually, on average, increase its fitness. Some of the most common replacement techniques are outlined below.

3.4.1- Delete All

This technique deletes all the members of the current population and replaces them with the same number of chromosomes that have just been created. This is probably the most common technique and will be the technique of choice for most people due to its relative ease of implementation. It is also parameter-free, which is not the case for some other methods.

3.4.2- Steady State

This technique deletes n old members and replaces them with n new members. The number to delete and replace, n, at any one time is a parameter to this deletion technique. Another consideration for this technique is deciding which members to delete from the current population. Delete the worst individuals, pick them at random or delete the chromosomes that used as parents. Again, this is a parameter to this technique.

3.4.3- Steady State No Duplicates

This is the same as the steady state technique but the algorithm checks that no duplicate chromosomes are added to the population. This adds to the computational overhead but can mean that more of the search space is explored.

Step4. Use new generated population for next generation.

Step5. Repeat steps 2–6 until a terminating condition is met.

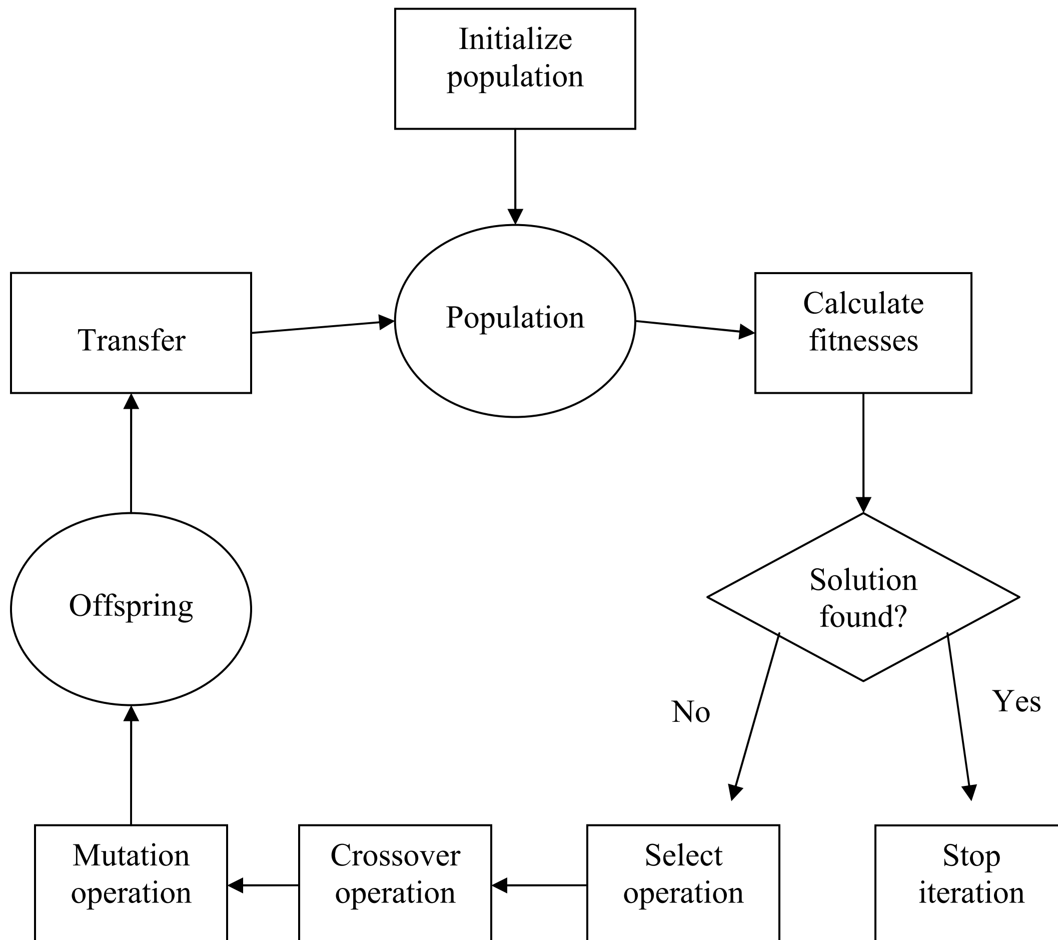

Figure 1.2- This flowchart illustrates the basic steps in a genetic algorithm .

1.4- Parameters of Genetic Algorithm

Number of Generation: indicates, how many times the genetic algorithm will be repeated.

Population Size: indicates, number of chromosomes are in a population. If there are too few chromosomes, Genetic Algorithm has a few possibilities to perform crossover and only a small part of search space is explored. On the other hand, if there are too many chromosomes, Genetic Algorithm slows down. Also if the population size is too small, the genetic algorithm may not explore enough of the solution space to consistently find good solutions [17]. Figure 1.3 indicates a population which has 3 chromosomes and each chromosome has 5 genes.

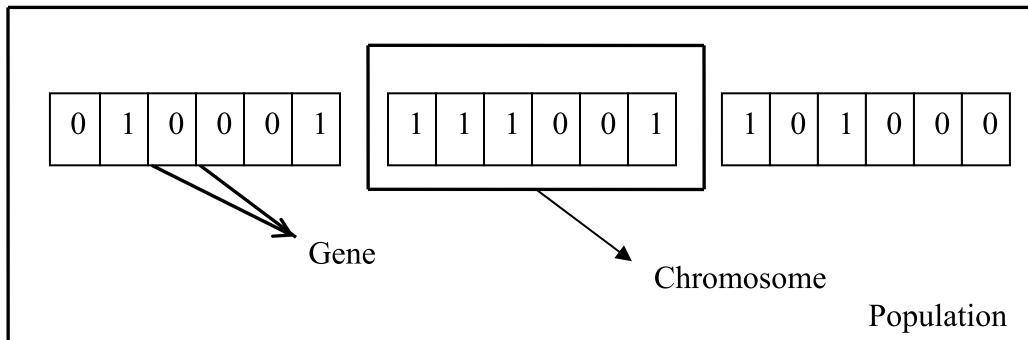

Figure 1.3- Chromosome and gene representation in a population.

Chromosome Length: Each chromosome represents a point in the search space and is composed of a string of genes. The binary alphabet {0, 1} is often used to represent these genes but sometimes, depending on the application, integers or real numbers are used. In fact, almost any representation can be used that enables a point to be encoded as finite length string. Figure 1.3 indicates chromosomes length as 5 (genes).

* m bit string parameters each of length n bit

Example: 5*10 bits

[[0001010101]p1[0101010101]p2[1110010101]p3[0101100001]p4[1111110000]]p5

* Single parameter bits of length n bit

Example: 50 bits

[000101010101010101011110010101010110000111111110000]p1

Crossover probability: indicates, how often the crossover will be performed. If there is no crossover, offspring is exact copy of parents. If there is a crossover, offspring is made from parts of parents “chromosome”. If crossover probability is equal to 1, then all offspring is made by crossover. If it is equal to 0, whole new generation is made from exact copies of chromosomes from old population (but this does not mean that the new generation is the same).

Crossover is made in hope that new chromosomes will have good parts of old chromosomes and maybe the new chromosomes will be better. However it is good to leave some parts of population survive to next generation.

Mutation probability: indicates, how often the parts of the chromosome will be mutated. If there is no mutation, offspring is taken after crossover (or copy) without any change. If mutation is performed, part of chromosome is changed. If mutation probability is equal to 1, whole chromosome is changed, if it is equal to 0, nothing is changed. Mutation is made to prevent falling genetic algorithm into local extreme, but it should not occur very often, because then genetic algorithm will in fact change to random search.

1.5- Representation of a Chromosome in Search Space

While solving some problem, usually looking for some solution, which will be the best among others. The space of all feasible solutions is called search space. Each point in the search space represents one feasible solution. Each feasible solution can be "marked" by its value or fitness for the problem. Looking for solution which is the one point (or more)

among feasible solutions - that is one point in the search space. And the looking for a solution is then equal to a looking for some extreme (minimum or maximum) in the search space.

Each individual in a genetic algorithm population is represented by a chromosome. In nature this chromosome contains genetic information relating to each individual characteristic. For simple genetic algorithms the chromosome is often represented as a binary string. But representation is depends on the problem.

1.5.1- Encoding and Decoding of Parameters in Genetic Algorithm

The mapping from phenotype to the genotype space is encoding process. The inverse mapping from genotypes to phenotypes is usually called decoding.

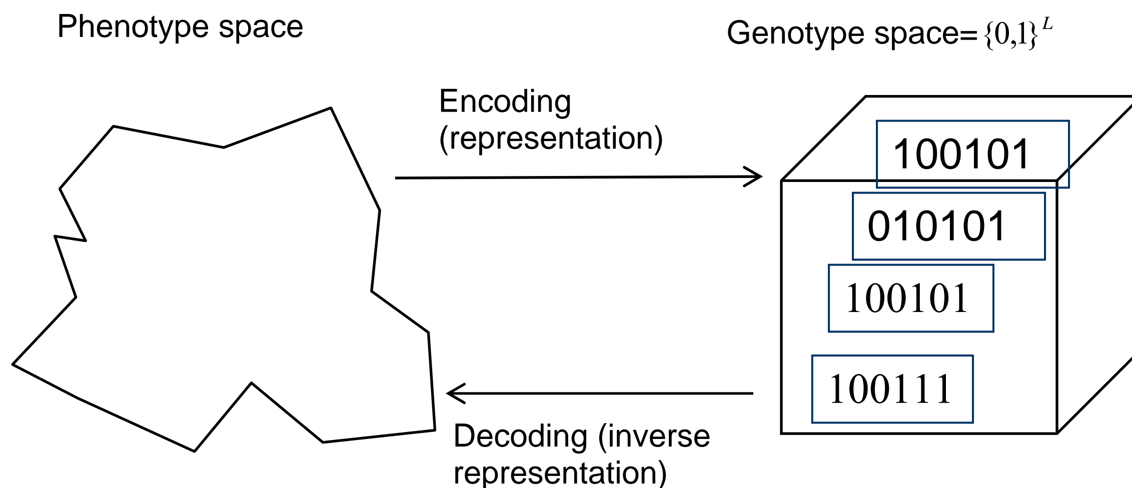

Figure 1.4- Decoding and encoding process.

1.5.2- Parameter Mapping

$z \in [x,y] \subseteq \Re$ represented by $\{a_1,...,a_L\} \in \{0,1\}^L$

L = number of binary bits in each individual.

$[x,y] \rightarrow \{0,1\}^L$ must be invertible (one phenotype per genotype).

Γ: $\{0,1\}^L \rightarrow [x,y]$ defines the representation.

$$\Gamma(a_1,...,a_L) = x + \frac{y-x}{2^L - 1}.\left(\sum_{j=0}^{L-1} a_{L-j}.2^j\right)\varepsilon[x,y]$$

Only 2^L values out of infinite are represented. L determines possible maximum precision of solution. High precision requires long chromosomes (but makes slow evolution).

Let's look at a specific example, and see how the analogy would apply to finding the minimum of the function. The polynomial to be minimized is:

$$y = x^4 + 9x^3 - 20x^2 + 4x - 12$$

This is the objective function, and has a single parameter x. When using a genetic algorithm, the parameters are represented as bit strings, and so the length of the string as well as the possible bounds on the parameter must be specified. A length of 10 bits will be assumed, and x will be allowed to range between -10 and +10. This means, for example, that the bit string "0000000000" will correspond to the value -10. Some function to perform the mapping between integer and bit string must exist: it will be called encode.

The encode function assigns parameter values to bit strings in an obvious way. If the parameter is an integer, its value is simple the bit string interpreted as such. If the parameter is real, a fixed point representation is used, wherein each increment to the bit string corresponds to a real increment in the parameter. In this case, as an example, the range is 20.0, which is to be stored in 10 bits. The number of different bit strings of 10 bits is 210, or 1024; thus, the increment is 20.0/1024, or 0.01953125. The space of real values allowed by this representation is:

0000000000 - 10.0

0000000001 - 9.980469

0000000010 - 9.960938

. . .

1000000000 - 0.00

1000000001 - 0.019531

. . .

1111111111 - 9.980469

Note that 10.0 is not a represent able value. Parameters cannot have a value between two consecutive increments, because the representation does not permit it. Therefore the choice of chromosome size limits the accuracy of the final result.

The next step is the creation of a population. Initially, a population consisting of randomly generated bit strings is used. The size of the population, which is the number of randomly generated bit strings, is also a parameter to the algorithm. Now each bit string parameter (gene) is passed as a parameter to the objective function, and the result is a measure of the gene's fitness; in this case, large values correspond to poor fitness, and vice versa. The best P percent of the parameters can reproduce, where a value for P must also be specified to the genetic algorithm, and the remainder of the parameters will not be propagated into the next generation (iteration).

1.5.3- Encoding Multi Parameters

Suppose n parameters: $f(x_1, x_2, ..., x_n)$;

Encode each parameter from full binary string then concatenate encoded parameters to form a chromosome.

1.5.4- Decoding Multi Parameters

Decode each parameter and then calculate function value

CHAPTER 2
SELECTION OPERATORS

2.1- Selection Schemes

Genetic algorithms use a selection scheme to select individuals from the population to insert into a mating pool. Individuals from the mating pool are used by a recombination operator to generate new offspring, with the resulting offspring forming the basis of the next generation.

Since the individuals in the mating pool pass their genes on to the next generation, it is desirable that the mating pool comprise “good” individuals. A selection scheme in genetic algorithms a simply a process that favors the selection of better individuals in the population for the mating pool. The selection pressure is the degree to which the better individuals are favored: the higher the selection pressure, the more the better individuals are favored. This selection pressure drives the genetic algorithm to improve the population fitness over succeeding generations [30].

The convergence rate of a genetic algorithm is largely determined by the magnitude of the selection pressure, with higher selection pressures resulting in higher convergence rates. Genetic algorithms are able to identify optimal or near-optimal solutions under a wide range of selection pressure values [30]. However, if the selection pressure is too low, the convergence rate will be slow, and the genetic algorithm will unnecessarily take longer to find the optimal solution. If the selection pressure is too high, there is an increased chance of the genetic algorithm prematurely converting to a suboptimal solution. In addition to providing selection pressure, selection schemes should also preserve population diversity because this held to avoid premature convergence.

2.1.1- Basic Selection Schemes

There are two basic types of selection schemes commonly used today: proportionate-base selection and ordinal-base selection [18].

Proportionate-based selection scheme selects individuals on the basis of their fitness values relative to the fitness of the other individuals in the population. Some common proportionate-based selection schemes are proportionate selection [4, 7], stochastic universal sampling selection [5, 6].

Ordinal-based selection scheme selects individuals not according to their fitness, but on the basis of theirs rank within the population. The individuals are ranked according to their fitness. This entails that the selection pressure is independent of the fitness distribution of the population and is solely based on the relative ordering (ranking) of the population. Some common ordinal-based selection schemes are tournament selection [9, 10, and 27], truncation selection [11] and linear ranking selection [18].

2.2- Proportionate-Base Selection

Proportionate-based selection selects individuals according to their fitness values relative to the fitness of the other individuals in the population.

2.2.1- Proportionate (Roulette-Wheel) Selection

Proportionate selection is also known as roulette-wheel selection which is the original selection method proposed for genetic algorithms by Holland [4]. This method can be represented as a game of roulette-wheel. According to game of roulette-wheel, wheel is partitioned into several sectors in different area corresponding to different amount of money. When spun of the roulette-wheel stops, the sector which the pointer points at is chosen and the player gets the amount of money corresponding to the sector. Although

can't estimated which sector the pointer will point to but known sectors are proportional to the magnitude of the central angle of the sectors. The bigger central angle of the sector is the higher probability the pointer will point at the sector.

Similarly, in the roulette-wheel selection method, the whole population is partitioned by the individuals, each sector representing an individual. The proportion of the individual's fitness value to the total fitness values of the whole population decides the area of the sector corresponding to the individual and decides the probability of the individual to be selected for the next generation.

Input: The population $P(\tau)$

Output: The population after selection $P(\tau)'$

Proportional$\left(J_1,...,J_N\right)$:

$s_0 \leftarrow 0$

for $i \leftarrow 1$ **to** N **do**

$s_i \leftarrow s_{i-1} + \frac{f_i}{M}$

od

for $i \leftarrow 1$ **to** N **do**

$r \leftarrow random[0, s_N]$

$J_i^{'} \leftarrow J_l$ such that $s_{l-1} \le r \le s_l$

return $\{J_1^{'},...,J_N^{'}\}$

Figure 2.1- Pseudo code of proportionate selection.

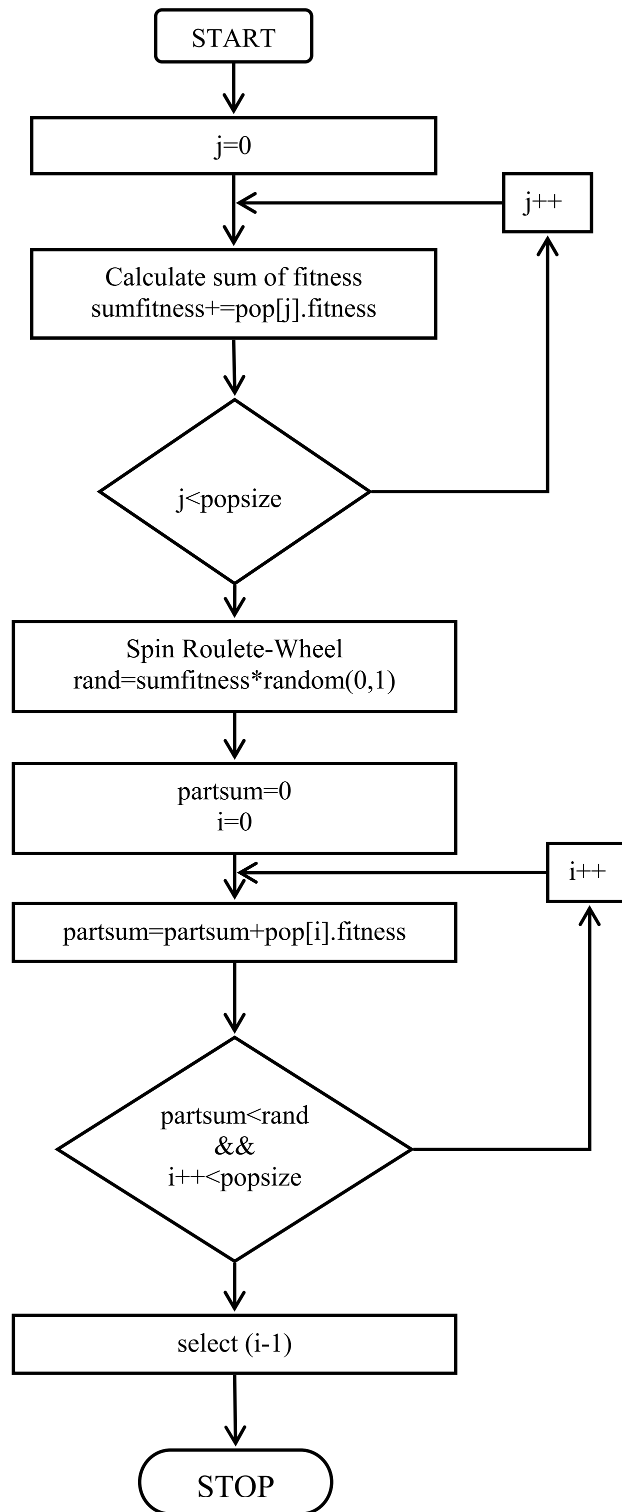

Figure 2.2- Flowchart of roulette-wheel Selection

The selection performs following the steps as below:

Step 1 : Calculate sum of the fitness value of every individual in the population.
Step 2 : Calculate the proportion of each individual's fitness value to the sum of fitness value of all individuals in the population. The proportions of individuals represent the probability of the individuals to be selected.

Step 3 : Spin the roulette-wheel n times where n is the number of individuals of the population. When the spun roulette stops, the sector where pointer pointing at represents the corresponding individual being selected.

As an example;

Step1: Calculate sum of the fitness value of every individual in the population.

Table 2.1- Individuals fitness values for roulette-wheel selection.

Individuals	**Chromosome**	**Value**	**X1**	**Fitness**	**% of Total**
1	1111100001	543	0.307918	0.478978	0.426898
2	0001010100	168	-3.357771	18.990170	16.92534
3	1100011100	227	-2.781036	14.296234	12.74178
4	0101101011	858	3.387097	5.698232	5.078654
5	0001001110	456	-0.542522	2.379374	2.120661
6	1011110101	701	1.852395	0.726577	0.647575
7	1100100000	19	-4.814272	33.805759	30.13
8	0111000000	14	-4.863148	34.376503	30.63869
9	1110101001	599	0.855327	0.020930	0.018654
10	0000011101	736	2.194526	1.426892	1.271744
				SUM of Fitness	**SUM of individuals percentage**
				112.1996	100

Step 2: Calculate the proportion of each individual's fitness value to the sum of fitness value of all individuals in the population. The proportions of individuals represent the probability of the individuals to be selected.

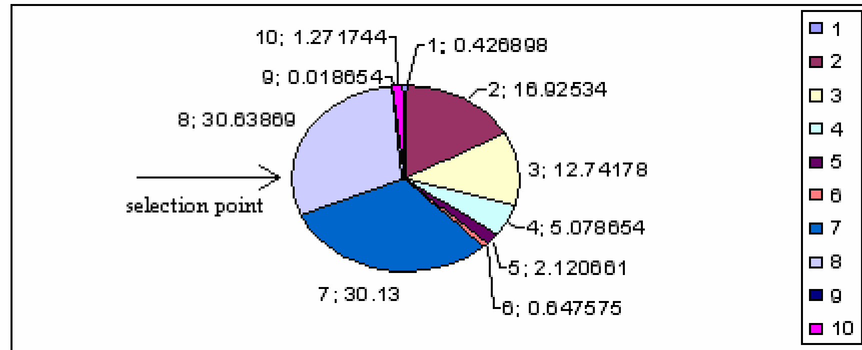

Figure 2.3- Proportions of individuals in a population.

Weakest individual "9" has smallest area on the wheel. Fittest individual "8" has largest area on the wheel.

Step 3: Spin the roulette-wheel n times where n is the number of individuals of the population. When the spun roulette stops, the sector where pointer pointing at represents the corresponding individual being selected.

2.2.2- Stochastic Universal Sampling Selection

Stochastic universal sampling selection [12] provides zero bias and minimum spread. The individuals are mapped to contiguous segments of a line, such that each individual's segment is equal in size to its fitness exactly as in roulette-wheel selection.

Here equally spaced pointers are placed over the line as many as there are individuals to be selected. Consider NPointer the number of individuals to be selected, then the distance between the pointers are 1/NPointer and the position of the first pointer is given by a randomly generated number in the range [0, 1/NPointer].

Input: The population $P(\tau)$ and the reproduction rate for each fitness value $R_i \varepsilon [0, N]$.

Output: The population after selection $P(\tau)'$

SUS $(R_1,...,R_N, J_1,...,J_N)$:

$sum \leftarrow 0$

$j \leftarrow 1$

$ptr \leftarrow random[0,1)$

for $i \leftarrow 1$ **to** N **do**

$sum \leftarrow sum + R_i$ where R_i is the reproduction rate of individual J_i

while $(sum > ptr)$ **do**

$J'_j \leftarrow J_i$

$j \leftarrow j+1$

$ptr \leftarrow ptr + 1$

od

od

return $\{J'_1,...,J'_N\}$

Figure 2.4- Pseudo code of stochastic universal sampling selection.

For 6 individuals to be selected, the distance between the pointers is 1/6=0.167. Figure 2.5 shows the selection for the above example in section 2.2.1. Sample of 1 random number in the range [0, 0.167]: 0.1

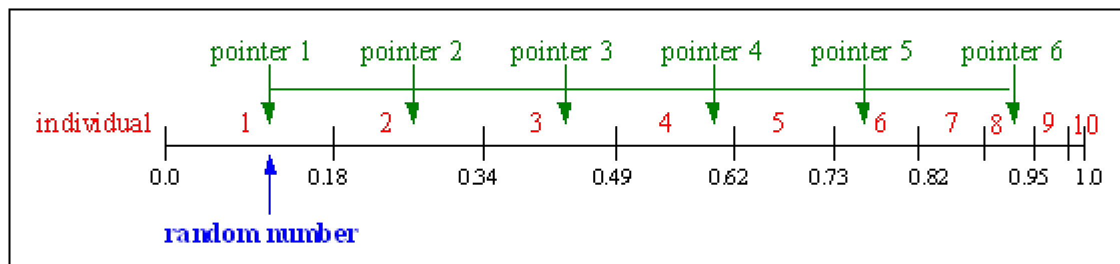

Figure 2.5- Stochastic universal sampling selection example.

After selection the mating population consists of the individuals: 1, 2, 3, 4, 6, and 8. Stochastic universal sampling selection ensures a selection of offspring which is closer to what is deserved then roulette-wheel selection.

2.3- Ordinal-Base Selection

Ordinal-based selection schemes select individuals not according to their fitness, but on the basis of their rank within the population. The individuals are ranked according to their fitness. This entails that the selection pressure is independent of the fitness distribution of the population and is solely based on the relative ordering (ranking) of the population.

2.3.1- Tournament Selection

The basic idea of this strategy is to select the individual with the highest fitness value from a certain number of individuals in the population. In the tournament selection, there is only comparison between individuals by fitness value [9]. The number of the individuals taking part in the tournament is called tournament size for binary tournament selection tournament size is 2.

Input: The population $P(\tau)$ the tournament size $T\ \varepsilon\{1,2,...,N\}$

Output: The population after selection $P(\tau)^{'}$

Tournament$\left(t, J_1,...,J_N\right)$:

for $i \leftarrow 1$ to N **do**

$J_i^{'} \leftarrow$ best fit individual out of t randomly picked individuals from $\{J_1,...,J_N\}$

od

return $\{J_1^{'},...,J_N^{'}\}$

Figure 2.6- Pseudo code of tournament selection.

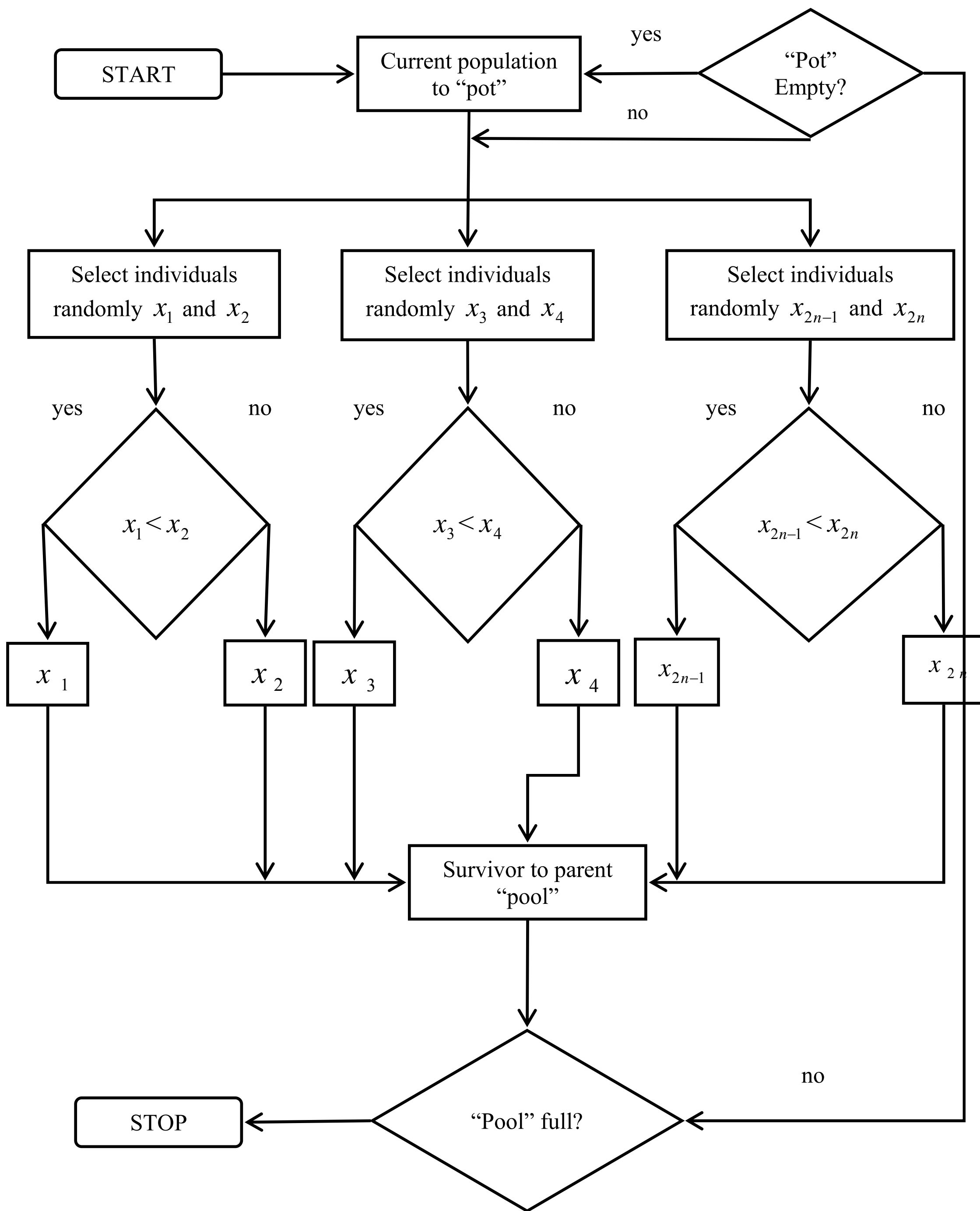

Figure 2.7- Flowchart of binary tournament selection.

The selection performs following the steps as below:

Step 1: Randomly select several individuals “tournament size=2” from the population to take part in the tournament. Choose the individual that has the highest fitness value from the individuals selected above by comparing the fitness value of each individual. Then the chosen one is copied for the next generation of the population.

Step 2: Repeat step1 n time where n is the number of individuals of the population.

As an example;

Step 1: Randomly select 2 (tournament size) individuals from the population.

Table 2.2- Individuals fitness values of tournament selection.

Individuals	Chromosome	Value	X1	Fitness
1	1111100001	543	0.307918	0.478978
2	0001010100	168	-3.357771	18.990170
3	1100011100	227	-2.781036	14.296234
4	0101101011	858	3.387097	5.698232
5	0001001110	456	-0.542522	2.379374
6	1011110101	701	1.852395	0.726577
7	1100100000	19	-4.814272	33.805759
8	0111000000	14	-4.863148	34.376503
9	1110101001	599	0.855327	0.020930
10	0000011101	736	2.194526	1.426892

1st individual = 1.426892 and 2nd individual = 5.698232
Compare first and second and choose highest fitness.

Step 2: Repeat step 1 up to population size.

2.3.2- Truncation Selection

In truncation selection [11] individuals are sorted according to their fitness. Only the best individuals are selected for parents. These selected parents produce uniform at random offspring. The parameter for truncation selection is the truncation threshold Trunc.

Trunc indicates the proportion of the population to be selected as parents and takes values ranging from 50%-10%. Individuals below the truncation threshold do not produce offspring [11].

Input: The population $P(\tau)$ the truncation threshold $T \ \varepsilon \{1,2,...,N\}$

Output: The population after selection $P(\tau)'$

Truncation$(T, J_1,...,J_N)$:

$\overline{J} \leftarrow$ Sorted population J according fitness with worst individual at the first position.

for $i \leftarrow 1$ to N **do**

$r \leftarrow random\{[(1-T),...,N]\}$

$J_i^{'} \leftarrow \overline{J}_r$

od

return $\{J_1^{'},...,J_N^{'}\}$

Figure 2.8- Pseudo code of truncation selection.

2.3.3- Linear Ranking Selection

For rank selection the individuals in the population are sorted to the objective values and the rank N is assigned to the best individual and the rank 1 to the worst individual.
Consider N is the number of individuals in the population and the selection probability is linearly assigned to the individuals according to their rank:

$$p(i) = \frac{1}{N} * \left(nu_max - (nu_max - nu_min) * \frac{i-1}{N-1} \right)$$

where $1 \le nu_max \le 2$;

$nu_min = 2 - nu_max$.

nu_max = rank factor (maximum 2).

Input: The population $P(\tau)$ and the reproduction rate of the worst individual $\eta^{-} \varepsilon [0,1]$

Output: The population after selection $P(\tau)'$

Linear Ranking $(\eta^{-}, J_1, \ldots, J_N)$

$\overline{J} \leftarrow$ Sorted population J according fitness with worst individual at the position

$s_0 \leftarrow 0$

for $i \leftarrow 1$ to N **do**

$s_i \leftarrow s_{i-1} + p_i$

od

for $i \leftarrow 1$ to N **do**

$r \leftarrow random[0, s_N[$

$J_i' \leftarrow \overline{J}_l$ such that $s_{l-1} \le r < s_l$

od

return $\{J_1', \ldots, J_N'\}$

Figure 2.9- Pseudo code of linear ranking selection.

2.3- Comparison of Selection Schemes

This section, analyses the comparative performance of the genetic algorithm on ordinal-based selection and proportionate-based selection, according to selection probability and selection intensity.

2.3.1- Selection Probability

2.3.1.1- Selection Probability for Tournament Selection

Tournament selection computes selection probabilities based only on a random subset of the whole population. The size of the subset is often referred to as the tournament size. The selection probability can be determined by one or more tournaments. That is, more than one tournament can be held in order to determine the selection probability of an individual. Different tournaments are independent of each other and can be held in parallel [28, 30, and 3].

2.3.1.2- Selection Probability for Linear Ranking Selection

In ranking selection at each generation, the individuals in the population are sorted according to their fitness and each individual is assigned a rank in the sorted population. The worst individual gets the rank 1 while the best receives the rank N (N = population size). The selection probabilities of the individuals x_k (k=1,…,N) are given by some function(most commonly, linear) of their rank.

Let $\{x_1^{(t)}, x_2^{(t)}, ..., x_N^{(t)}\}$ denote the population at generation t. Then in linear ranking selection the probabilities of selecting individual $x_k\,(k = 1,2,...,N)$ is given by

$$p\left(x_k^{(t)}\right) = \frac{1}{N}\left(\min + \frac{(\max - \min)(rank(x_k^{(t)}) - 1)}{}\right)$$

Where $\max + \min = 2$ and $1 \le \max \le 2$ .The $\{p(x_k^{(t)})\}$ is a proper probability distribution ($\sum_{k=1}^{N} p(x_k^{(t)}) = 1$ for each t), and sampling N individuals according to this probability distribution yields the next generation.

2.3.1.3- Selection Probability for Roulette-Wheel Selection

Roulette-wheel selection calculates the selection probability of an individual using the individual's fitness directly. Let f_i , $1 \le i \le n$, be the fitness of n individuals in a population. Then the probability of selecting individual i as a parent is :

$$p_i = \frac{f_i}{\sum_{j=1}^{n} f_j}$$

2.3.1.4- Selection Probability for Stochastic Universal Sampling Selection

Suppose the population size is N. The selection probability of an individual uses the individual's fitness. In stochastic universal sampling selection the wheel is first divided into N sections according to selection probability of individual i $p_i = \frac{f_i}{avg \sum_{j=1}^{N} f_j}$ so the central angle of that sector is $2\pi p_i$ .Then one random number *rand* within the interval (0, 1) is generated and the pointers begin to rotate a certain angle from the base 0° one by one. Pointer i should rotate the angle of $2\pi * rand + (i-1) * 2\pi / N, (i = 1,..., N)$. If the sector i contains one pointer after rotation, the individual i will be selected twice in the selection operation. Under this principle exactly N individuals will be selected but only one random number is needed.

2.3.1.5- Comparison of Selections Probability

Although roulette wheel selection and stochastic universal sampling selection is simple, it suffers from problems of super individuals and slow convergence. In a population where a few individuals (super individuals) have a substantially higher fitness than others, these super individuals will quickly dominate the whole population due to their extremely high selection probabilities. Such dominance prevents the population from exploring other regions of the search space. If a super individual is a global optimum, this is good news. Or else it is a disaster. On the other hand, if the fitter individuals in a population have very similar fitness, they will have very similar selection probabilities. Hence it will be very slow for the population to converge to the best one. In general, the selection pressure induced by roulette wheel selection fluctuates too much as the fitness distribution in a population changes. It is very difficult to make a suitable trade-off between achieving faster global convergence and avoiding premature convergence [28, 30].

2.3.2- Selection Intensity

The selection intensity I measures the magnitude of the selection pressure provided by a selection scheme [30]. The selection intensity of genetic algorithm, as defined by Mühlenbein and Schierkamp-Voosen [11], is expected average fitness of a population after selection is performed on a population whose fitness is distributed according to the unit normal distribution $N(0,1)$.

If the selection intensity I of a selection scheme is known, and the population fitness at generation t is distributed $N(\mu_t, \sigma_t^2)$, the expected mean fitness of a population after selection can be determined:

$$\mu_{t+1} = \mu_t + \mathrm{I}\sigma_t$$

an important assumption of this model is that population fitness is normally distributed before selection. In practice, this is true or approximately true for many domains because

recombination and mutation operators have a normalizing effect on the population fitness distribution.

2.3.2.1- Selection Intensity for Tournament Selection

Bäck (1995) [27] and Miller and Goldberg (1996) [18] independently applied order statistics to derive the selection intensity for tournament selection. The order statistics are for the unit normal distribution $N(0,1)$; thus, $\mu_{i,j}$ represents the expected value of the i th biggest sample from a sample of size j drawn from the unit normal distribution. The maximal order statistics $\mu_{s:s}$ determines the selection pressure of a tournament of size s.

2.3.2.2- Selection Intensity for Linear Ranking Selection

The study by Bäck (1995) also derives the selection intensity for (μ,λ) selection [18]. In (μ,λ) selection, the best μ individuals are selected from a random sample size λ. The selection pressure is simply the mean of the top μ th-order statistics of sample size λ. The selection intensity of linear ranking is given by Blickle and Thiele (1995), where n^+ denotes the number of desired copies of the best individual. Linear ranking selects each individual in the population with a probability linearly proportional to the rank of the individual. Implicit in the selection intensity value for linear ranking is that $1 \le n^+ \le 2$, and $n^+ + n^- = 2$, where n^- is the number of desired copies of the worst individual.

2.3.2.3- Selection Intensity for Proportionate-Based Selections

Mühlenbein and Schlierkamp-Voosen [11] derived the selection intensity for proportionate-based schemes, which directly depends on the current mean μ_t and standard deviation σ_t of the population in generation t. Proportionate selection selects individuals for the mating pool with a probability directly proportional to the individuals' fitness. The selection intensity equation for proportionate selection is used to predict the

performance of stochastic universal sampling selection, one of a handful of different proportionate selection schemes. The selection intensity of proportionate selection is unique in that it is the only one that is sensitive to the current population distribution.

2.3.2.4- Comparison of Selections Intensity

According to Table 2.3 the selection intensity in proportional selection decreases with the inverse of the average fitness and proportionately to the standard deviation. The closer the population comes to the optimum, the less severe is the selection. Proportionate selection is afraid of reaching the goal. The selection intensity in tournament selection is proportional with the average fitness of tournament size and in linear ranking selection each individual in the population with a probability linearly proportional to the rank of the individual. This result explains why proportionate-base selections are not a good strategy for optimization purposes.

Table 2.3 gives the selection intensity for tournament, linear ranking and proportionate selection schemes.

Table 2.3- Comparison of selection schemes according to selection intensity.

Selection Scheme	**Parameters**	**Selection Intensity** I
Tournament selection	s	$\mu_{s:s}$
Linear ranking selection	n^+	$(n^+ - 1)\frac{1}{\sqrt{\pi}}$
Proportionate-base selections	σ_t, μ_t	σ_t / μ_t

CHAPTER 3
RECOMBINATION OPERATORS

The next step is to generate a second generation population of solutions from those selected through genetic operators: crossover (also called recombination) and/or mutation.

For each new solution to be produced, a pair of "parent" solutions is selected for breeding from the pool selected previously. By producing a "child" solution using the below methods of crossover and mutation, a new solution is created which typically shares many of the characteristics of its "parents". New parents are selected for each child, and the process continues until a new population of solutions of appropriate size is generated. These processes ultimately result in the next generation population of chromosomes that is different from the initial generation. Generally the average fitness will have increased by this procedure for the population, since only the best organisms from the first generation are selected for breeding, along with a small proportion of less fit solutions, for reasons already mentioned above [17].

3.1- Crossover Operators

After selection, the crossover process is used to breed a pair of children “offspring” from a pair of parents. The idea behind crossover operation is that, the new offspring may be better than both of the parents if it takes the best characteristics from each of the parents.

Crossover operation is a probabilistic operation, flipping a coin according to user defined crossover probability is determined if the operation will be performed or not. If the result is true new chromosomes are generated otherwise copies of the parents are retuned as children and also crossover probability defines how often will be crossover performed.

3.1.1- K-Point Crossover

One Point and Two Point Crossover

One-point and two-point crossovers are the simplest and most widely applied crossover methods.

In one-point crossover, as illustrated in Figure 3.1, a crossover point is selected at random over the string length, and the alleles on one side of the site are exchanged between the individuals.

The one point crossover takes two vectors;

$$individual1 = (a_1, a_2, ..., a_n)$$
$$individual2 = (b_1, b_2, ..., b_n)$$

Then generate a random number k where $1 \le k \le n-1$, this means that both vectors are split at the same point and assembled with swapped second parts.

$$offspring1 = (a_1, ..., a_k, b_{k+1}, ..., b_n)$$
$$offspring2 = (b_1, ..., b_k, a_{k+1}, ..., a_n)$$

Consider the following two individual with 6 binary variables each:

$$individual1 = 000100$$
$$individual2 = 101111$$

The chosen crossover position is:
Cross position (m=1) = 2
After crossover the new individuals are created:

$offspring1 = 001111$
$offspring2 = 100100$

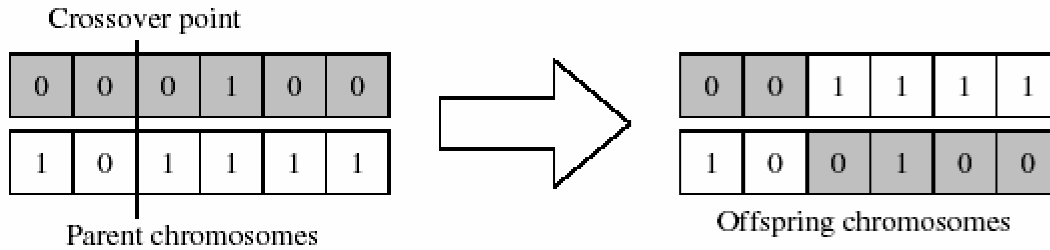

Figure 3.1- One point crossover.

In two-point crossover, two crossover points are randomly selected, as in one point crossover. The alleles between the two sites are exchanged between the two randomly paired individuals. Two-point crossover is as illustrated in Figure 3.2.

Two points crossover takes two vectors:

$individual1 = (a_1, ..., a_n)$
$individual2 = (b_1, ..., b_n)$

-Then generate two random number i and j where $1 \leq i < j \leq n-1$, this means that both vectors are split at the same two points and assembled with swapped middle parts.

$offspring1 = (a_1, ..., a_i, b_{i+1}, ..., b_j, a_{j+1}, ..., a_n)$
$offspring2 = (b_1, ..., b_i, a_{i+1}, ..., a_j, b_{j+1}, ..., b_n)$

Consider the following two individual with 6 binary variables each:

$individual1 = 000100$
$individual2 = 101111$

The chosen crossover position is:

Cross position (m=2) = 2 & 6

After crossover the new individuals are created:

$offspring1 = 001110$

$offspring2 = 100101$

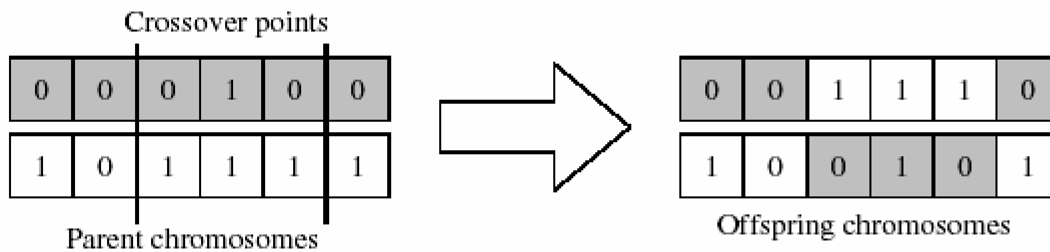

Figure 3.2- Two point crossover.

The concept of one-point crossover can be extended to k-point crossover, where k crossover points are used, rather than just one or two.

The idea behind multi-point, and indeed many of the variations on the crossover operator, is that parts of the chromosome representation that contribute most to the performance of a particular individual may not necessarily be contained in adjacent substrings.

Further, the disruptive nature of multi-point crossover appears to encourage the exploration of the search space, rather than favoring the convergence to highly fit individuals early in the search, thus making the search more robust.

3.1.2- Uniform Crossover

Single and multi-point crossover defines cross points as places between loci where an individual can be split. Uniform crossover generalizes this scheme to make every locus a

potential crossover point. A crossover mask, the same length as the individual structure is created at random and the parity of the bits in the mask indicate which parent will supply the offspring with which bits [19, 20].

Consider the following two individuals with 6 binary variables each:

$individual1 = 000100$

$individual2 = 101111$

For each variable the parent who contributes its variable to the offspring is chosen randomly with equal probability. Here, the offspring 1 is produced by taking the bit from parent 1 if the corresponding mask bit is 1 or the bit from parent 2 if the corresponding mask bit is 0. Offspring 2 is created using the inverse of the mask, usually.

$mask = 100001$

After crossover the new individuals are created:

$offspring1 = 100101$

$offspring2 = 001110$

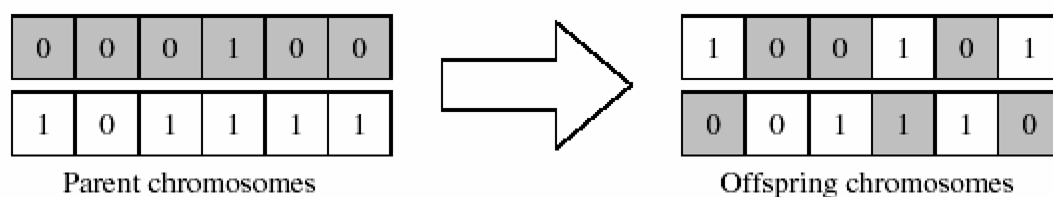

Figure 3.3- Uniform crossover.

Uniform crossover, like multi-point crossover, has been claimed to reduce the bias associated with the length of the binary representation used and the particular coding for a given parameter set. This helps to overcome the bias in single-point crossover towards

short substrings without requiring precise understanding of the significance of the individual bits in the individual representation.

Demonstrated how uniform crossover may be parameterized by applying a probability to the swapping of bits. This extra parameter can be used to control the amount of disruption during recombination without introducing a bias towards the length of the representation used.

3.1.3- Uniform Order Based Crossover

The k-point and uniform crossover methods described above are not well suited for search problems with permutation codes such as the ones used in the traveling salesman problem. They often create offspring that represent invalid solutions for the search problem. Therefore, when solving search problems with permutation codes, a problem-specific repair mechanism is often required (and used) in conjunction with the above recombination methods to always create valid candidate solutions.

Another alternative is to use recombination methods developed specifically for permutation codes, which always generate valid candidate solutions. Several such crossover techniques are described in the following paragraphs starting with the uniform order-based crossover. In uniform order-based crossover, two parents (say P1 and P2) are randomly selected and a random binary template is generated (see Figure 3.4).

Some of the genes for offspring C1 are filled by taking the genes from parent P1 where there is a one in the template. At this point C1 partially filled, but it has some “gaps”. The genes of parent P1 in the positions corresponding to zeros in the template are taken and sorted in the same order as they appear in parent P2. The sorted list is used to fill the gaps in C1. Offspring C2 is created by using a similar process (see Figure 3.4).

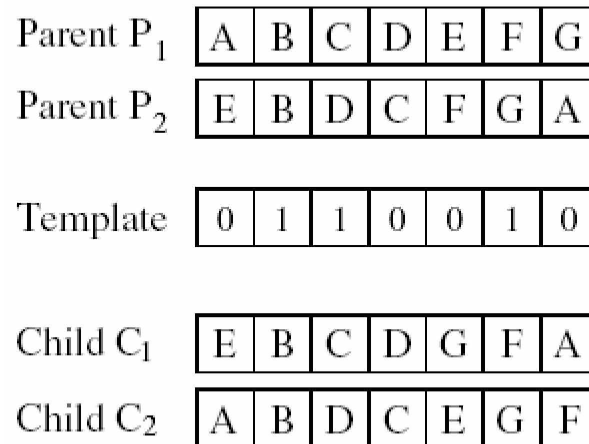

Figure 3.4- Uniform order based crossover.

3.1.4- Order Based Crossover

The order-based crossover operator [21] is a variation of the uniform order-based crossover in which two parents are randomly selected and two random crossover sites are generated (see Figure 3.5). The genes between the cut points are copied to the children. Starting from the second crossover site copy the genes that are not already present in the offspring from the alternative parent (the parent other than the one whose genes are copied by the offspring in the initial phase) in the order they appear.

For example, as shown in Figure 3.5, for offspring C1, since alleles C, D, and E are copied from the parent P1, get alleles B, G, F, and A from the parent P2. Starting from the second crossover site, which is the sixth gene, copy alleles B and G as the sixth and seventh genes respectively. Then wrap around and copy alleles F and A as the first and second genes.

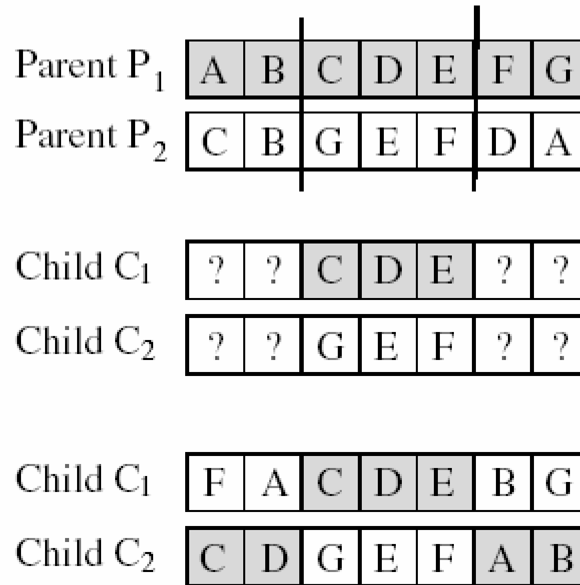

Figure 3.5- Order based crossover.

3.1.5- Partially Matched Crossover (PMX)

Apart from always generating valid offspring, the PMX operator [29] also preserves orderings within the chromosome. In PMX, two parents are randomly selected and two random crossover sites are generated. Alleles within the two crossover sites of a parent are exchanged with the alleles corresponding to those mapped by the other parent.

For example, as illustrated in Figure 3.6 (reproduced from Goldberg [14] with permission), looking at parent P1, the first gene within the two crossover sites, 5, maps to 2 in P2. Therefore, genes 5 and 2 are swapped in P1. Similarly swap 6 and 3, and 10 and 7 to create the offspring C1. After all exchanges it can be seen that have achieved a duplication of the ordering of one of the genes in between the crossover point within the opposite chromosome, and vice versa.

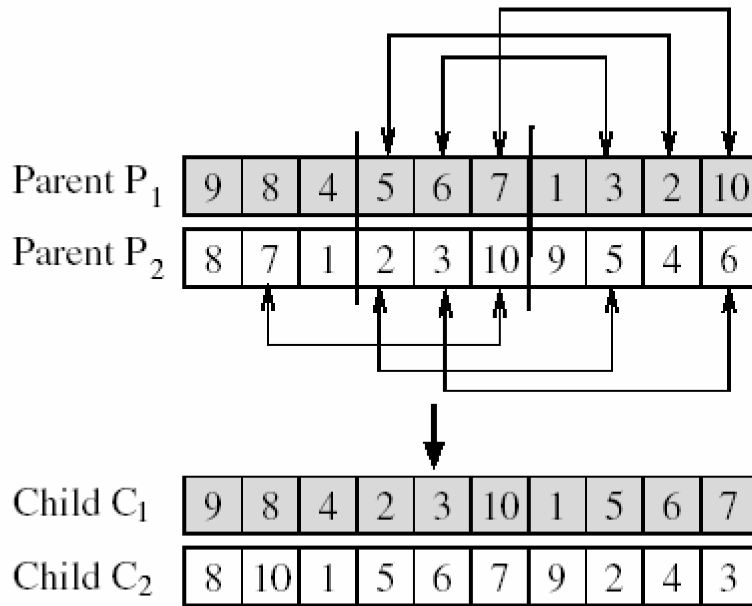

Figure 3.6- Partially matched crossover.

3.1.6- Cycle Crossover (CX)

To describing cycle crossover [15] with help of a simple illustration reproduced from Goldberg [14] with permission). Consider two randomly selected parents P1 and P2 as shown in Figure 3.7 that are solutions to a traveling salesman problem. The offspring C1 receives the first variable (representing city 9) from P1. Then choose the variable that map onto the same position in P2. Since city 9 is chosen from P1 which maps to city 1 in P2, choose city 1 and place it into C1 in the same position as it appears in P1 (fourth gene), as shown in Figure 3.7. City 1 in P1 now maps to city 4 in P2, so place city 4 in C1 in the same position it occupies in P1 (sixth gene). Continue this process once more and copy city 6 to the ninth gene of C1 from P1. At this point, since city 6 in P1 maps to city 9 in P2, take city 9 and place it in C1, but this has already been done, so completed a cycle; which is where this operator gets its name. The missing cities in offspring C1 is filled from P2. Offspring C2 is created in the same way by starting with the first city of parent P2 (see Figure 3.7).

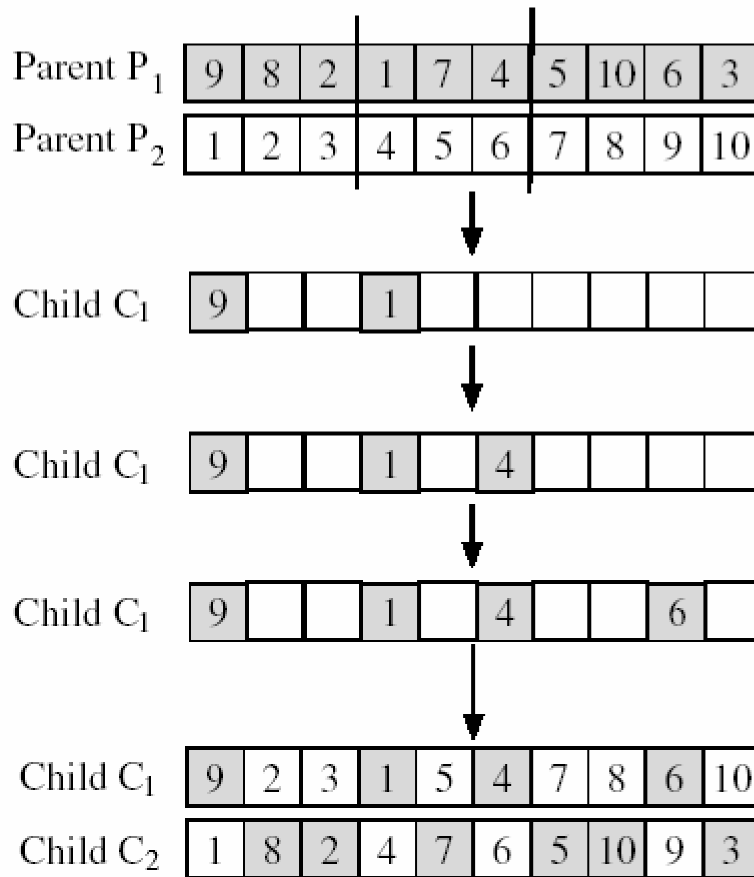

Figure 3.7- Cycle crossover.

3.2- Mutation Operators

Mutation is a genetic operator that alters one ore more gene values in a chromosome from its initial state. This can result in entirely new gene values being added to the gene pool. With these new gene values, the genetic algorithm may be able to arrive at better solution than was previously possible.

Mutation is an important part of the genetic search as help helps to prevent the population from stagnating at any local optima. Mutation occurs during evolution according to a user-definable mutation probability. This probability should usually be set fairly low. If it is set to high, the search will turn into a primitive random search.

3.2.1- Flip Bit Mutation

A mutation operator that simply inverts the value of the chosen gene (0 goes to 1 and 1 goes to 0). This mutation operator can only be used for binary genes.

For binary valued individuals mutation means the flipping of variable values, because every variable has only two states. Thus, the size of the mutation step is always 1. For every individual the variable value to change is chosen (mostly uniform at random). Figure 3.8 shows an example of a binary mutation for an individual with 11 variables, where variable 4 is mutated.

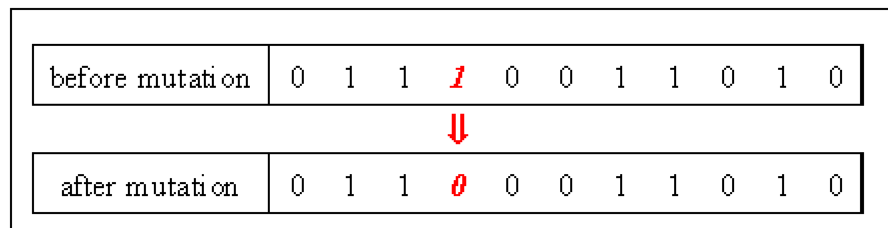

Figure 3.8- Flip bit mutation.

3.2.2- Boundary Mutation

A mutation operator that replaces the value of the chosen gene with either the upper or lower bound for that gene (chosen randomly).

At random choose $i \varepsilon N$. Set either $X_i = \underline{x}_i$ or $X_i = \overline{x}_i$, with probability ½ of using each value.

3.2.3- Uniform Mutation

Uniform mutation changes the value of the element to a value chosen from the uniform distribution on the interval between the lower and upper bounds specified for the element

At random choose $i \varepsilon N$. Select a value $x_i \sim U\left(\underline{x}_i, \overline{x}_i\right)$. Set $X_i = x_i$

3.2.4- Non-Uniform Mutation

A mutation operator that increases the probability that the amount of the mutation will be close to 0 as the generation number increases. This mutation operator keeps the population from stagnating in the early stages of the evolution then allows the genetic algorithm to fine tune the solution in the later stages of evolution. This mutation operator can only be used for integer and float genes.

At random choose i Є N. Compute p = $(1\text{-}t/T)^{B}$ u, where t is the current generation number, T is the maximum number of generations, B>0 is a tuning parameter and $u \sim U(0,1)$. Set either $X_i = (1-p)x_i + p\underline{x_i}$ or $X_i = (1-p)x_i + p\overline{x_i}$, with probability ½ of using each value.

3.2.5- Gaussian Mutation

A mutation operator that adds a unit Gaussian distributed random value to the chosen gene. The new gene value is clipped if it falls outside of the user-specified lower or upper bounds for that gene. This mutation operator can only be used for integer and float genes.

CHAPTER 4
BENCHMARK TEST FUNCTIONS

4.1- F1- Rosenbrock's Function

$$f_1(x) = 100 * (x_1^2 - x_2)^2 + (1 - x_1)^2$$

It is a standard test function in the optimization literature, first proposed by Rosenbrock. It is a continuous, non-convex, uni-modal, low-dimensional quartic function with a minimum of zero at $(1,1)$. [22].

It is a difficult minimization problem because it has a deep parabolic valley along the curve $x_2 = x_1^{\ 2}$. For testing purposed, it was restricted to the space as $-2.048 \le x_i \le 2.048$, x_1, x_2 with resolution factor of $\Delta x_i = 0.001$ along each axis.

Range of x_1 and $x_2 = -2.048 \le x_i \le 2.048$.

Global minimum of function = $\min f(x) = 0$ at $x_1 = 1$ $x_2 = 1$

A common multidimensional extension is:

$$f(x) = \sum_{i=1}^{N-1} \left[(1 - x_i)^2 + 100 \left(x_{i+1} - x_i^2 \right)^2 \right] \quad \forall x \varepsilon R^N$$

Visualization of Rosenbrock's function;

}

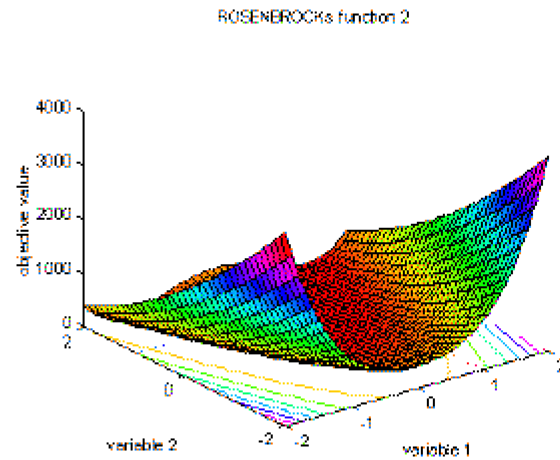

Figure 4.1- Full definition range of the function.

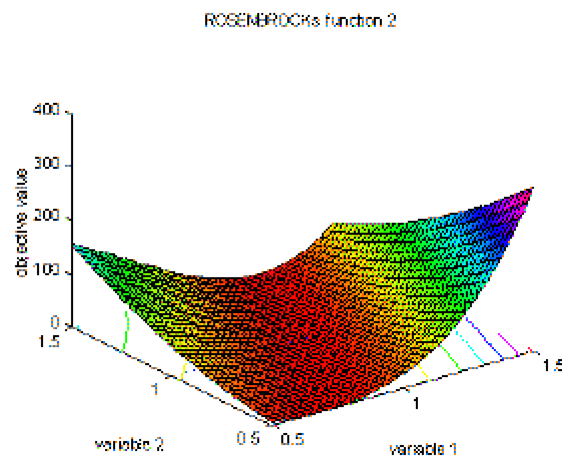

Figure 4.2- Focus around the area of the global optimum at [1,1].

C Codes of Rosenbrock Function

```
float objfunc(float x[], int n){
        double result;
        result=100*pow(x[0]*x[0]-x[1],2)+pow(1-x[0],2);
        return(result);
}
```

4.2- F2- De Jong's Test Function 1

$$f_2(x) = \sum_{i=1}^{N} (x_i)^2$$

It is a simple 5 dimensional parabola with. It is a continuous, convex, uni-modal, low-dimensional quadratic function with a minimum of zero at the origin [22]. Because of its simplicity and symmetry, provides an easily analyzable test for an adaptive plan. For testing purposes it was restricted to the space as $-5.12 \le x_i \le 5.12$. $i = 1:5$ with a resolution factor Δx_i =0.01 on each axis.

$N = 5$

Range of $x_i = -5.12 \le x_i \le 5.12 \quad x(i) = 0 \;\; i = 1:N$

Global minimum of function = $\min f(x) = 0$ at $x(i) = 0 \;\; i = 1:N$

Visualization of De Jong's function 1 using different domains of the variables:

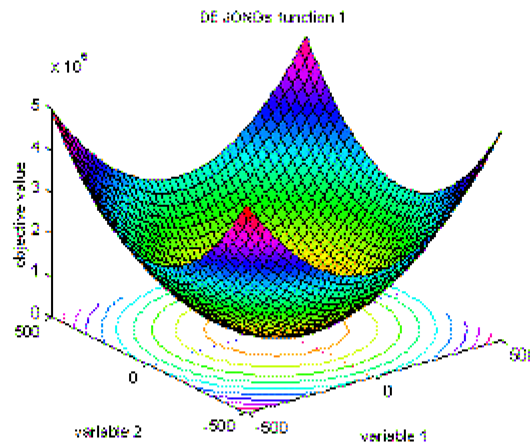

Figure 4.3- Surf plot of the function in a very large area from -500 to 500 for each of both variables.

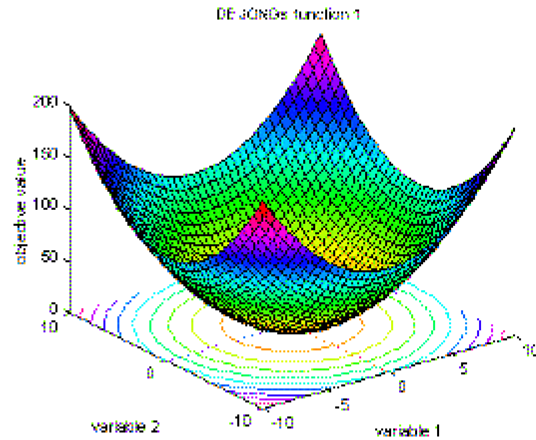

Figure 4.4- The function at a smaller area from -10 to 10.

C Codes of De Jong's Function 1

```
float objfunc(float x[], int n){
double result;
int i;
result=0.0;
for (i=0;i<n;i++){
        result+=x[i]*x[i];
        }
return(result);
}
```

4.3- F3- Griewangk's Function

$$f_3(x) = \frac{1}{4000}\sum_{i=1}^{N}(x_i - 100)^2 - \prod_{i=1}^{N}\cos\left(\frac{x_i}{\sqrt{i}}\right) + 1$$

It is a 5 dimensional function multi modal test function. It has many widespread local minima. However, the locations of the minima are regularly distributed. For testing purpose it was restricted to the space as $-600 \le x_i \le 600$, $i = 1:5$ with a resolution factor $\Delta x_i = 0.01$ on each axis.

$N = 5$

Range of x_i $= -600 \le x_i \le 600 \quad x(i) = 0 \;\; i = 1:N$

Global minimum of function = $\min f(x) = 0$ at $x(i) = 0 \;\; i = 1:N$

The graphics in depict Griewangk's function using three different resolutions. Each of the graphics represents different properties of the function. The Figure 4.5 shows the full definition range of the function. Here, the function looks very similar to De Jong's function 1. When approaching the inner area, the function looks different. Many small peaks and valleys are visible in the Figure 4.6. When zooming in on the area of the optimum, Figure 4.7, the peaks and valleys look smooth.

Visualization of Griwangk's function:

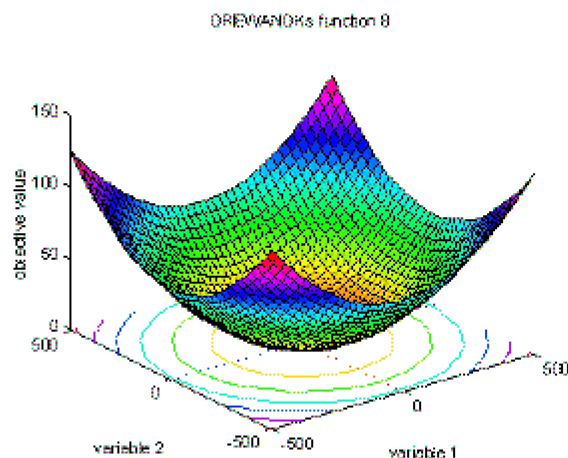

Figure 4.5- Full definition area from -500 to 500.

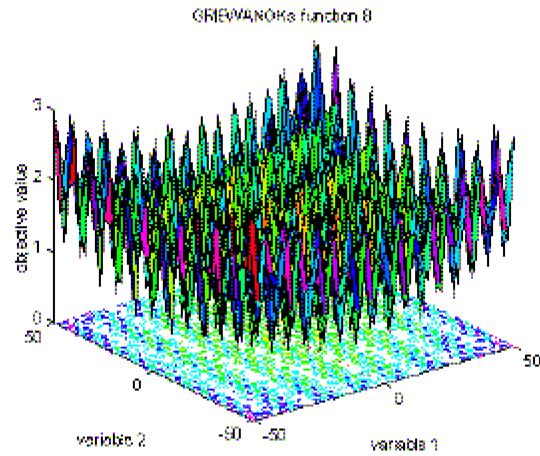

Figure 4.6- Inner area of the function from -50 to 50.

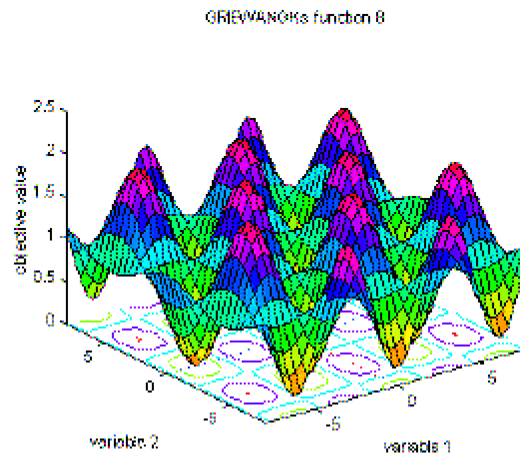

Figure 4.7- Area from -8 to 8 around the optimum at [0, 0].

C Codes of Griewangk's Function

```
float objfunc(float x[], int n){
const int D=4000.0;
int i;
```

```
double Val1,Val2,Sum;
for (Val1 = 0.0, Val2 = 1.0, i = 0; i < n; i++) {
        Val1 += (x[i]-100.0) * (x[i]-100.0);
        Val2 *= cos((x[i]-100.0) / sqrt((float) (i + 1)) );
        }
Sum = Val1 / D - Val2 + 1.0;
return (Sum);
}
```

4.4- F4- Michalewicz's Function

$$f_4(x) = -\sum_{i=1}^{N} \sin(x_i) * \left(\sin\left(\frac{i * x_i^2}{\pi} \right) \right)^{2*m}$$

It is a 5 dimensional function. It has many widespread local minima. However, the locations of the minima are regularly distributed. For testing purpose it was restricted to the space as $-600 \le x_i \le 600$, $i = 1:5$ with a resolution factor $\Delta x_i = 0.01$ on each axis.

The Michalewicz function [16] is a multimodal test function. The parameter m defines the "steepness" of the valleys or edges. Larger m leads to more difficult search. For very large m the function behaves like a needle in the haystack (the function values for points in the space outside the narrow peaks give very little information on the location of the global optimum). For testing purpose it was restricted to the space as $0 \le x_i \le \pi$, $i = 1:5$ with a resolution factor $\Delta x_i = 0.01$ on each axis.

$N = 5$

Range of $x_i = 0 \le x_i \le \pi$ $\quad x(i) = 0 \;\; i = 1:N$

Global minimum of function = $\min f(x) = -4.687 \;\; x(i) = ? \;\; i = 1:N$

The first two graphics below represent a global and a local view to Michalewicz's function, both for the first two variables. The third graphic displays the function using the third and fourth variable; the first two variables were set to 0. By comparing the figure 4.8 and the figure 4.9 the increasing difficulty of the function can be seen. As higher the dimension as more valleys are introduced into the function.

Visualizations of Michalewicz's function:

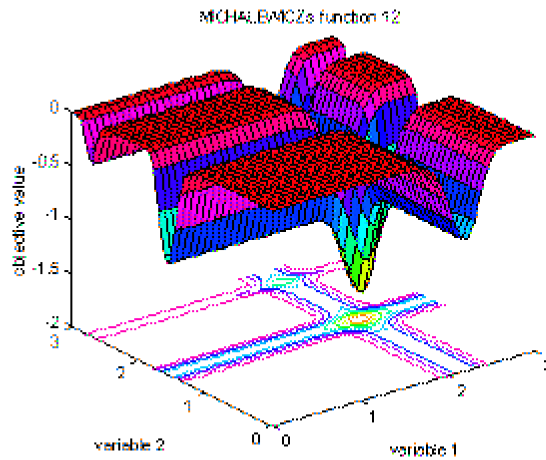

Figure 4.8- Surf plot in an area from 0 to 3 for the first and second variable.

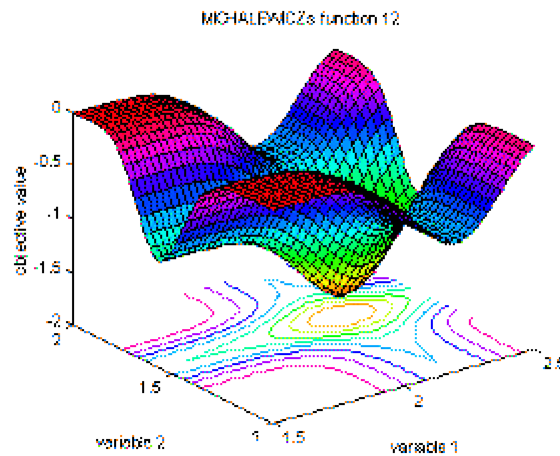

Figure 4.9- Area around the optimum.

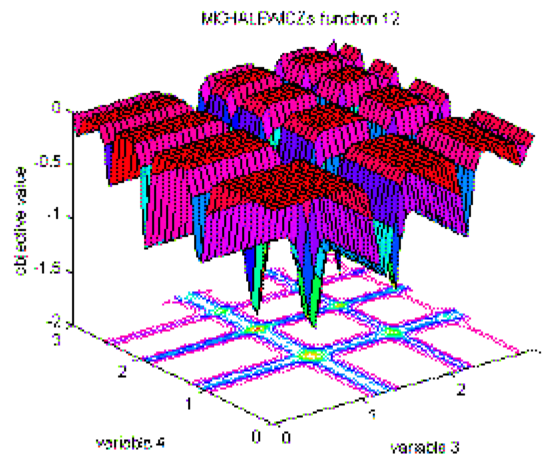

Figure 4.10- Same as Figure 4.8 for the third and forth variable, variable 1 and 2 are set 0.

C Codes of Michalewicz's Function

```
float objfunc(float x[], int n){
const int m=10.0;
double   u;
int     i;
u=0;
for (i=0;i<n;i++) {
          u = u + sin(x[i])* pow(sin((i+1)*x[i]*x[i]/PI),2*m);
          }
return(-u);
}
```

4.5- F5- Shekel's Function

$$f_5(x) = -\sum_{i=1}^{30} \frac{1}{\sum_{j=1}^{N} (x_i - a_{ij})^2 + c_i}$$

Shekel's function is an interesting multimodal function synthesized as suggested by Shekel (1971). It is a continuous, non convex, non quadratic, 5 dimensionally function with 30 local minima approximately at the points $\{(a_{1j}, a_{2j}, a_{3j}, a_{4j}, a_{5j})\}_{j=1}^{30}$. For testing purposes, (a_{1j}) were defined in shekel.h file. It was restricted to the space Δ defined by $0 \le x_i \le 10$, $i = 1:5$ with a resolution factor of $\Delta_i = 0.01$ on each axis.

$N = 5$

Range of x_i = $0 \le x_i \le 10$ $\quad x(i) = 0 \;\; i = 1:N$

Global minimum of function = $f(x) = -10.40561$

$x_1 = 8.0249 \quad x_2 = 9.1517 \quad x_3 = 5.1139 \quad x_4 = 7.6208 \quad x_5 = 4.5640$

Visualizations of Shekel's function:

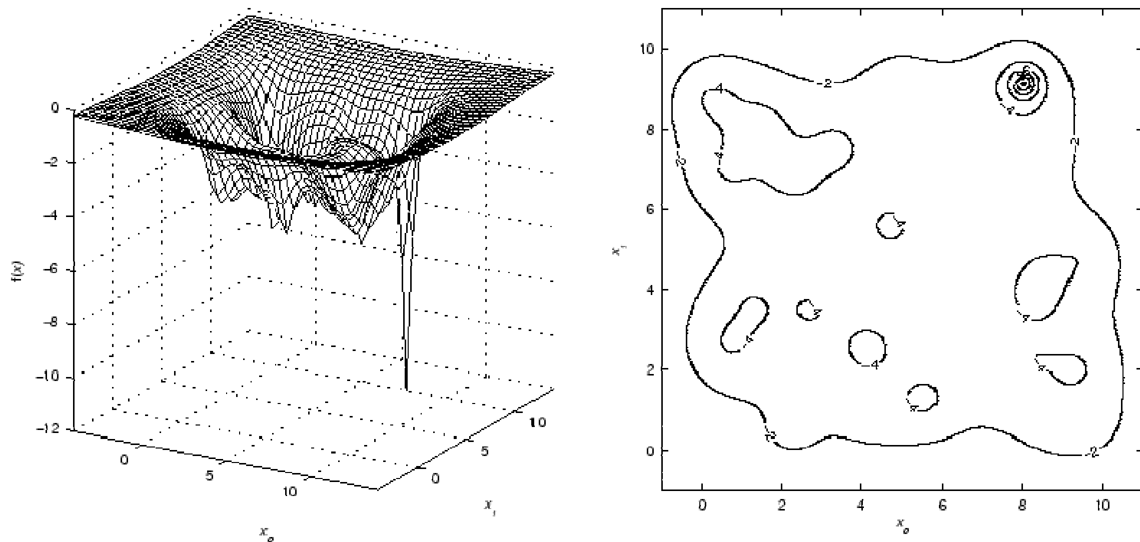

Figure 4.11- Graphics of Shekel's function.

C Codes of Shekel's Function

```
float objfunc(float x[],int n){
int i, j;
double sp, h, result = 0.0;
for (i = 0; i < 30; i++) {
        sp = 0.0;
for (j = 0; j < n; j++) {
        h   = x[j] - Shekel_a[i][j];
        sp += h * h;
                }
result += 1.0 / (sp + Shekel_c[i]);
        }
return(-result);
}
```

CHAPTER 5
IMPLEMENTATION of GENETIC ALGORITHM

5.1- Implementation of Genetic Algorithm

This section, discusses the design of genetic algorithm. When using a different selection strategy "binary tournament, linear ranking, stochastic universal and roulette-wheel selection" for performing the test functions, genetic algorithm use the same crossover "one point crossover" and mutation operation " flip bit mutation". The genetic algorithm implementation as below:

```
BEGIN
    WHILE (Times<30)  Do
    BEGIN GA
        Initialization;
        Evaluation;
        Keep the best fitness value;
        generate:=0;
        WHILE (generation<MAXGENS)   Do
            BEGIN
                generation++;
                if selection:= tour_sel tournament_sel()
                else     rouelette_wheel_sel()
                crossover;
                mutation;
                evaluation;
        END;
    END GA;
    Keep the best fitness value for each test (times).
END;
```

In the main algorithm above, Times<30 is the termination condition, which indicates the times for executing genetic algorithm. Because genetic algorithms are stochastic

computational techniques, performing of genetic algorithm many times for one problem so that gives a statistically good result. The genetic algorithm program popsize represents the number of individuals of the population. Maxgens is the maximum number of the generations in the evolutionary process. Pcross is the probability of the individuals selected to crossover and Pmutation is the probability of the individual selected to mutate.

5.1.1- Parameter Selection for a Genetic Algorithm Program

The first intensive study of genetic algorithm parameters was done by De Jong [22] and is nicely summarized in Goldberg [14]. De Jong found that a small population size improved initial performance while large population size improved long-term performance and a high mutation rate was good for off-line performance while low mutation rate was good for on-line performance. Grefenstette [24] used a genetic algorithm. He found the best genetic algorithm had a population size of 30 and mutation rate of 0.01. Schaffer found a population size = 20 to 30 and a mutation rate = 0.005 to 0.01 are best. In [23] both binary and continuous parameter of genetic algorithm, a small population size allowed to evolve for many generations produced the best results. Similar sensitivity studies with mutation rate suggested that mutation rates in the range of 0.05 to 0.35 found the best minima. Table 5.1 lists the parameters suggested by Schaffer and those proposed earlier by De Jong [22] and Greffenstette [24].

Table 5.1- Comparison of empirically determined genetic algorithm parameter settings [25].

Author	Population Size	Crossover Rate	Mutation Rate
Schaffer	20-30	0.75-0.95	0.005-0.01
De Jong	50-100	0.60	0.001
Grefenstette	30	0.95	0.01

According to [22, 24, 26] for each test functions with using genetic algorithm maxgens is taken 1000 times, set the value of population size as 50, the value of crossover probability as 0.8 and the value of mutation probability as 0.05. And the algorithm executed 30 times.

The algorithm uses roulette-wheel and binary tournament selection for the operation of selection. Specially, the number of the individuals taking part in the tournament is 2 for the binary tournament selection strategy and flip bit mutation is used.

5.2- C Codes of Genetic Algorithm's for Program Design

5.2.1- Genetic Algorithm Program Files

Definition.h : contains type definitions of data types.
Generate.h : contains new population generation routine.
Initialize.h : contains initdata(), initpop(), initreport() routines.
Kernel.h : contains three operators Reproduction (selection), Crossover (crossover), Mutation (mutation).
ObjectFunction.h : contains objectfunction() and decode() routines.
Parameters.h : cotains extract_parm(), map_parm(), decode_parms() routines.
Random.h : contains random number utility programs.
Reports.h : contains routines used to print a report from each cycle of genetic algorithm's operation.
Statistics.h : contains the routine statistics(), which calculates population statistics for each generation.
Main.cpp : contains the main GA program loop, main().

5.2.2- Genetic Algorithm Program Functions

Initialize.h

void initdata(): is a routine to prompt the user for genetic algorithm parameters.
void initreport(): is a routine that prints a report after initialization and before the first genetic algorithm cycle.
void initpop(): is routine that generates a random population.
void initialize(): is the central initialization routine called by main().

Kernel.h

int select_rws(): contains routines for roulette-wheel selection.

```
int select_rws(int popsize, float sumfitness, population pop){
        float rand, partsum; // Random point on wheel, partial sum
        int j; // population index

        partsum=0.0; j=0; // Zero out counter and accumulator
        rand=sumfitness*random(); // Wheel point calc. uses random number [0,1]
        do{     // Find wheel slot
                partsum=partsum+pop[j].fitness;
                j++;
                }while(partsum>rand&&j<popsize);
        return (j-1);  // Return individual number
}
```

Figure 5.1- C code of roulette-wheel selection.

void reset(): shuffles the tournament selection tourneylist at random.

int select_ts(): contains the routines for tournament selection. Tournaments of any size up to the population size can be held with this implementation. But for experiments tournament size is taken 2.

```
int select_ts(int popsize,population pop){
        int pick, winner, i;
        tourneysize=2;
// If remaining members not enough for a tournament, then reset list
   if((popsize – tourneypos) < tourneysize){
                reset();
                tourneypos = 0;
   }
// Select tourneysize structures at random and conduct a tournament
        winner=tourneylist[tourneypos];
        for(i=1; i<tourneysize; i++){
                pick=tourneylist[i+tourneypos];
                if(pop[pick].fitness < pop[winner].fitness)
                winner=pick;
                }
// Update tourneypos
   tourneypos += tourneysize;
   return(winner);
}
```

Figure 5.2- C code of tournament selection.

allele mutation(): performs a flip bit mutation.

```
allele mutation(allele alleleval, float pmutation, int &nmutation){
bool mutate;
mutate=flip(pmutation); // Flip the biased coin
          if(mutate){
                    nmutation++;
          return !alleleval; // Change bit value
   }
  else  return alleleval; // No change
}
```

Figure 5.3- C code of flip bit mutation.

void crossover(): performs single point crossover on two mates, producing two children.

```
void crossover(chromosome parent1, chromosome parent2, chromosome child1,
   chromosome child2,int &lchrom,int &ncross,int &nmutation,int &jcross,
   float &pcross,float &pmutation){
int j;
//Do crossover with probability crossover
if(flip(pcross)){
          jcross=rnd(1,lchrom-1);  //Cross between 1 and lchrom-1
                    ncross++;     // Increment crossover counter
   }
  else jcross=lchrom;
  // first exchange, 1 to 1 and 2 to 2
  for(j=0;j<jcross;j++){
     child1[j]=mutation(parent1[j],pmutation,nmutation);
     child2[j]=mutation(parent2[j],pmutation,nmutation);
   }
  // second exchange, 1 to 2 and 2 to 1
```

```
 if(jcross!=lchrom)
   for(j=jcross;j<lchrom;j++){
       child1[j]=mutation(parent2[j],pmutation,nmutation);
       child2[j]=mutation(parent1[j],pmutation,nmutation);
     }
}
```

Figure 5.4- C code of one point crossover with flip bit mutation.

ObjectFunction.h

float objfunc(): The objective function for the specific application. This routine is called by generation().

float decode(): decodes chromosomes strings as unsigned binary number. True=1, False=0.

```
float decode(chromosome chrom,int lbits){
int j;
float accum=0.0;
powerof2=1.0;
for(j=0;j<lbits;j++){
if(chrom[j])
        accum=accum+powerof2;
powerof2=powerof2*2;
   }
return accum;
}
```

Figure 5.5- C code of decoding a parameter.

Parameters.h

float map_parm(): an unsigned binary integer to range [minparm, maxparm]

```
float map_parm(float x,float maxparm,float minparm,float fullscale){
        return (minparm+(maxparm-minparm)/fullscale*x);
}
```

Figure 5.6- C code of mapping a parameter.

void extract_parm(): extracting a substring from a full string.

```
void extract_parm(chromosome chromfrom,chromosome chromto,int &jposition,int &lchrom,int &lparm){
int j,jtarget;
j=0;
jtarget=jposition+lparm;
if(jtarget>lchrom)
jtarget=lchrom;
// Clamp if excessive
        while(jposition<jtarget){
                chromto[j]=chromfrom[jposition];
                jposition++;
                j++;
  }
}
```

Figure 5.7- C code of extracting a parameter from a full string.

void decode_parms(): decode parameters.

```
void decode_parms(int &nparms,int &lchrom,chromosome chrom,parmspecs parms){
int j,jposition;
chromosome chromtemp;
// Temporary string buffer
j=0;
// Parameter counter
jposition=0;
// String position counter
do{
if(parms[j].lparm>0){
        extract_parm(chrom,chromtemp,jposition,lchrom,parms[j].lparm);
        parms[j].parameter=map_parm(decode(chromtemp,parms[j].lparm),
parms[j].maxparm,parms[j].minparm,pow(2.0,parms[j].lparm)-1);
        }
        else
parms[j].parameter=0.0;
j++;
}while(j<nparms);
}
```

Figure 5.8- C code of parameter decoding.

Random.h

void advance_random(): generates a new batch of 55 random numbers.
void warmup_random(): primes the random number generator.

float random(): returns a single, uniformly-distributed, real, pseudo-random number between 0 and 1.

bool flip(): flips a biased coin,returning 1 with probability p,and 0 with probability 1-p.

```
Bool flip(float probability){
        if(probability==1.0)
        return true;
        else return (random()<=probability);
}
```

Figure 5.9- C code of flipping a biased coin.

int rnd(low,high): returns an uniformly-distributed integer number between low and high.

void randomize(): srand() function changes the number seed.

Reports.h

void writechrom(): writes out the chromosomes as a string of ones and zeros.

void report(): controls overall reporting.

Statistics.h

void statistic(): which calculates population statistics for each generation.

Generate.h

void generation(): is a routine which generates and evaluates a new genetic algorithm population.

Main.cpp

void main(): GA program loop.

```
Void main(){
        gen=0;  // Set things up
        initialize();
        do{
        gen++;
        generation();
        statistics(popsize,max,avg,min,sumfitness,newpop);
        report(gen);
        for(int i=0;i<popsize;i++)
                oldpop[i]=newpop[i]; // advance the generation
        }while(gen<maxgen);
}
```

Figure 5.10- C code of main function.

Susandlrnk.h

int select_sus(): contains the routines for stochastic universal sampling selection.

CHAPTER 6
ANALYSIS of COMPUTATIONAL RESULTS

6.1- Computational Test Results

Five test functions are taken for experiments using genetic algorithm. Each function has a prescribed search domain given in range column of the Table 6.1. Problem column indicates the solution wanted to obtain, minimization or maximization. The optimal solution of each function is known beforehand. The resolution factor value is shown in the Precision column.

Table 6.1- List of five test functions.

Test Function	Range	Problem	Precision	Dim
$f_1(x) = 100*(x_1^2 - x_2)^2 + (1-x_1)^2$	[-2.048, 2.048]	Min	0.001	2
$f_2(x) = \sum_{i=1}^{N}(x_i)^2$	[-5.12,5.12]	Min	0.01	5
$f_3(x) = \frac{1}{4000}\sum_{i=1}^{N}(x_i - 100)^2 - \prod_{i=1}^{N}\cos\left(\frac{x_i}{\sqrt{i}}\right) + 1$	[-600,600]	Min	0.01	5
$f_4(x) = -\sum_{i=1}^{N}\sin(x_i)*\left(\sin\left(\frac{i*x_i^2}{\pi}\right)\right)^{2*m}$	[0,π]	Min	0.01	5
$f_5(x) = -\sum_{i=1}^{30}\frac{1}{\sum_{j=1}^{N}\left(x_i - a_{ij}\right)^2 + c_i}$	[0,10]	Min	0.01	5

For a given function, each time of the 1000 times for executing genetic algorithm, keep the value of generation when the process of genetic algorithm terminates. The value of generation is in integer ranged from 0 to 1000. Each value of generation has a corresponding counter preserving the times for genetic algorithm obtaining this generation.

6.1.1- Roulette-Wheel Selection Test Results

F1 = $f_1(x) = 100*(x_1^2 - x_2)^2 + (1 - x_1)^2$.

Range of x_1 and $x_2 = -2.048 \le x_i \le 2.048$.

x_1, x_2 with resolution factor of $\Delta x_i = 0.001$.

Global minimum of function = $\min f(x) = 0$ at $x_1 = 1$ $x_2 = 1$.

Table 6.2- Test results of roulette-wheel selection for F1.

#TEST	#Generation	Best_min Fitness
1	36	0.021625
2	689	0.012379
3	107	0.003115
4	380	0.010103
5	133	0.013182
6	725	0.006524
7	381	0.006387
8	779	0.003378
9	144	0.007762
10	337	0.003148
11	96	0.002099
12	40	0.002759
13	872	0.011802
14	834	0.007192
15	97	0.008316
16	963	0.000392
17	836	0.003000
18	49	0.025836
19	30	0.006666
20	63	0.000305

21	268	0.062003
22	65	0.000501
23	295	0.003207
24	412	0.000534
25	226	0.044440
26	980	0.015776
27	752	0.031173
28	899	0.018574
29	201	0.012611
30	658	0.009197

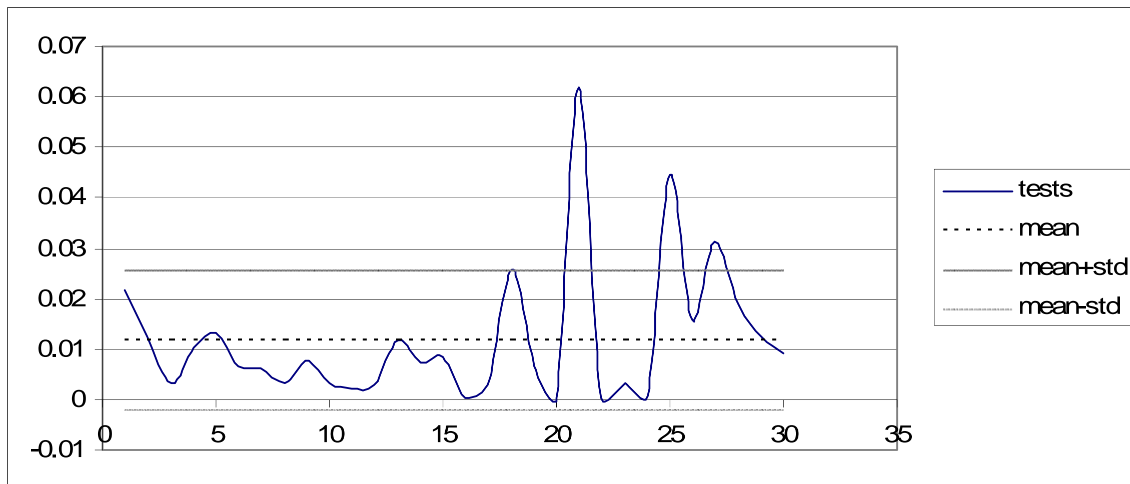

Figure 6.1- Roulette-wheel selection graphic for F1.

F2 = $f_2(x) = \sum_{i=1}^{N}(x_i)^2$.

$N = 5$.

Range of $x_i = -5.12 \le x_i \le 5.12 \quad x(i) = 0 \;\; i = 1:N$.

$i = 1:5$ with a resolution factor $\Delta x_i = 0.01$ on each axis.

Global minimum of function: $\min f(x) = 0$ at $x(i) = 0 \;\; i = 1:N$.

Table 6.3- Test results of roulette-wheel selection for F2.

#TEST	#Generation	Best_min Fitness
1	502	2.407425
2	313	0.622540
3	43	1.143557
4	538	0.689070
5	301	1.616681
6	403	1.991413
7	903	1.328118
8	123	1.801241
9	368	1.230728
10	440	0.539178
11	187	1.294252
12	75	1.592233
13	4	1.307477
14	464	0.572843
15	534	1.368196
16	259	0.522345
17	368	0.481465
18	638	3.219810
19	90	2.071168
20	798	0.528156

21	103	2.026481
22	870	0.810908
23	857	1.994218
24	519	0.831548
25	769	0.226367
26	763	0.988855
27	445	1.462179
28	118	1.331725
29	293	0.277867
30	860	4.204733

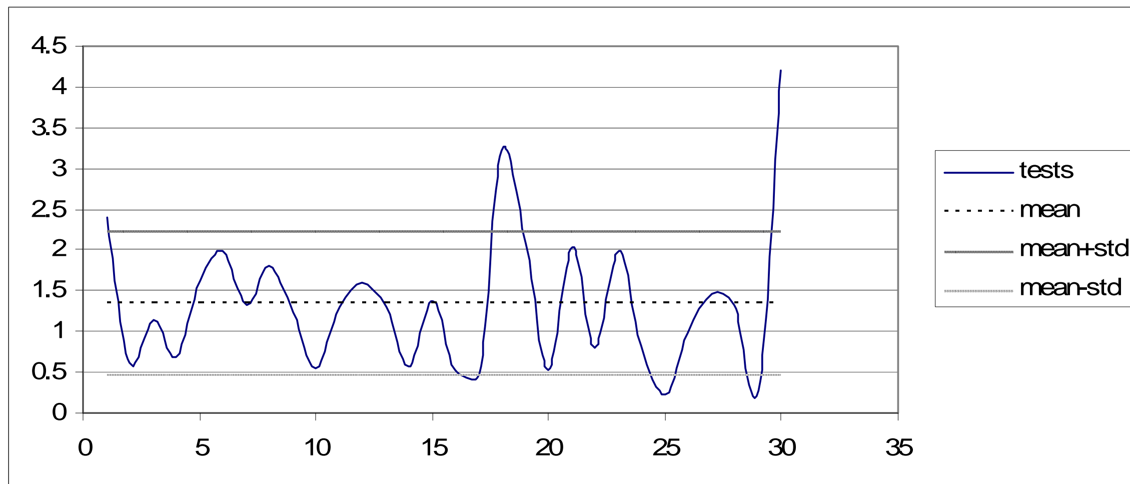

Figure 6.2- Roulette-wheel selection graphic for F2.

F3 $= f_3(x) = \frac{1}{4000}\sum_{i=1}^{N}(x_i - 100)^2 - \prod_{i=1}^{N}\cos\left(\frac{x_i}{\sqrt{i}}\right) + 1$.

$N = 5$.

Range of $x_i = -600 \le x_i \le 600 \;\; x(i) = 0 \;\; i = 1:N$.

$i = 1:5$ with a resolution factor $\Delta x_i = 0.01$ on each axis.

Global minimum of function: $\min f(x) = 0$ at $x(i) = 0 \;\; i = 1:N$.

Table 6.4- Test results of roulette-wheel selection for F3.

#TEST	**#Generation**	**Best_min Fitness**
1	15	4.221037
2	197	2.760204
3	226	5.355815
4	106	4.124454
5	437	6.183483
6	916	7.642037
7	342	11.04009
8	777	7.214392
9	234	6.127833
10	503	7.639765
11	11	6.283434
12	638	2.684016
13	711	7.221925
14	318	9.988311
15	721	6.668437
16	919	9.444723
17	219	10.404618
18	47	7.199335
19	755	2.507251
20	642	6.858248

21	190	6.036229
22	849	11.730000
23	774	5.069771
24	730	9.595348
25	863	8.863648
26	277	7.621606
27	683	6.180864
28	222	4.278033
29	672	8.173793
30	220	10.038148

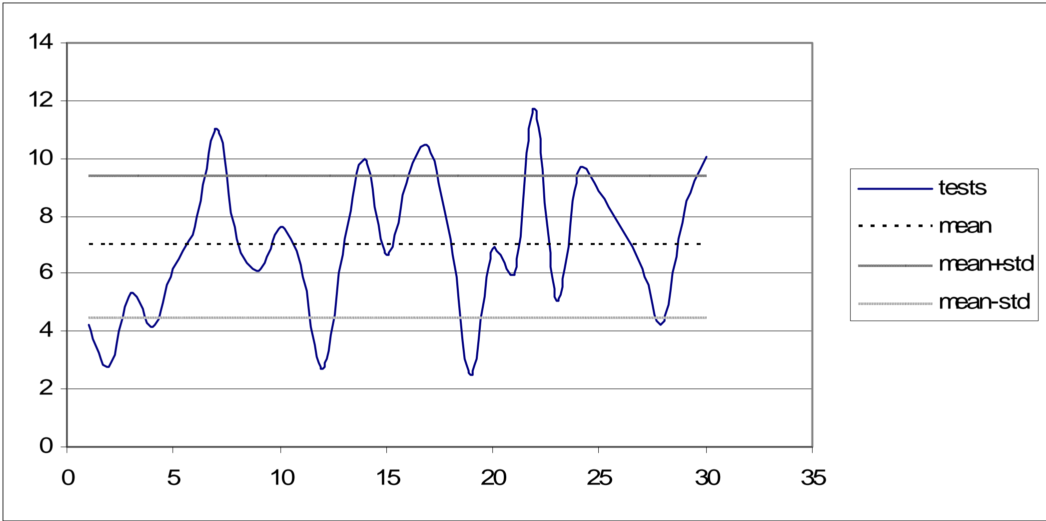

Figure 6.3- Roulette-wheel selection graphic for F3.

$$\mathbf{F4} = f_4(x) = -\sum_{i=1}^{N} \sin(x_i) * \left(\sin\left(\frac{i * x_i^2}{\pi} \right) \right)^{2*m} .$$

$N = 5$.

Range of $x_i = 0 \le x_i \le \pi$ $\quad x(i) = 0 \;\; i = 1 : N$.

$i = 1:5$ with a resolution factor $\Delta x_i = 0.01$ on each axis.

Global minimum of function = $\min f(x) = -4.687 \;\; x(i) = ? \;\; i = 1 : N$.

Table 6.5- Test results of roulette-wheel selection for F4.

#TEST	#Generation	Best_min Fitness
1	612	-4.586186
2	692	-4.570916
3	430	-4.625953
4	174	-4.626780
5	481	-4.626883
6	131	-4.630258
7	125	-4.619846
8	932	-4.606482
9	292	-4.612538
10	313	-4.613001
11	536	-4.573160
12	895	-4.620372
13	165	-4.669135
14	156	-4.651917
15	641	-4.594432
16	803	-4.612193
17	187	-4.627946
18	894	-4.617115
19	108	-4.667325

20	284	-4.603973
21	281	-4.634270
22	348	-4.608518
23	326	-4.654795
24	573	-4.628489
25	327	-4.631560
26	901	-4.618083
27	414	-4.622041
28	704	-4.665676
29	209	-4.670503
30	201	-4.621484

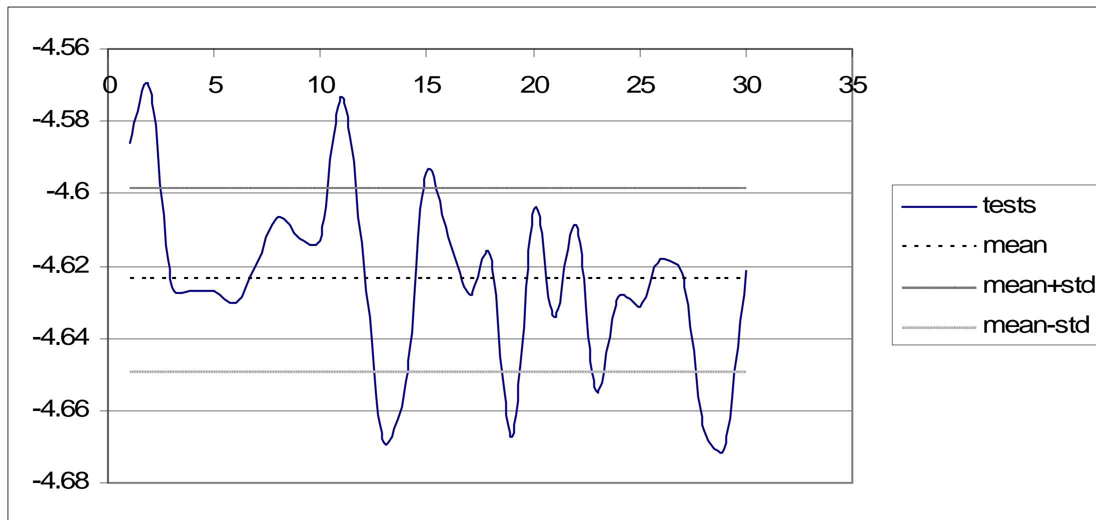

Figure 6.4- Roulette-wheel selection graphic for F4.

F5 $= f_5(x) = -\sum_{i=1}^{30} \frac{1}{\sum_{j=1}^{N} (x_i - a_{ij})^2 + c_i}$.

$N = 5$.

Range of $x_i = 0 \le x_i \le 10 \quad x(i) = 0 \;\; i = 1:N$.

$i = 1:5$ with a resolution factor of $\Delta_i = 0.01$ on each axis.

Global minimum of function = $f(x) = -10.40561$

$x_1 = 8.0249 \quad x_2 = 9.1517 \quad x_3 = 5.1139 \quad x_4 = 7.6208 \quad x_5 = 4.5640$

Table 6.6- Test results of roulette-wheel selection for F5.

#TEST	#Generation	Best_min Fitness
1	880	-3.403414
2	143	-2.504386
3	845	-2.528651
4	556	-2.194961
5	465	-2.532915
6	638	-2.544109
7	936	-2.197939
8	661	-3.106445
9	118	-2.855063
10	176	-2.656415
11	848	-9.141911
12	840	-2.652430
13	833	-2.543444
14	847	-2.655011
15	725	-3.170594
16	977	-2.642898
17	997	-2.537261
18	560	-2.529057

19	756	-2.661852
20	505	-2.186500
21	565	-3.235952
22	321	-2.195720
23	674	-9.165722
24	784	-2.538344
25	469	-9.059353
26	515	-2.657397
27	233	-2.533310
28	897	-2.639995
29	466	-1.827188
30	873	-2.536064

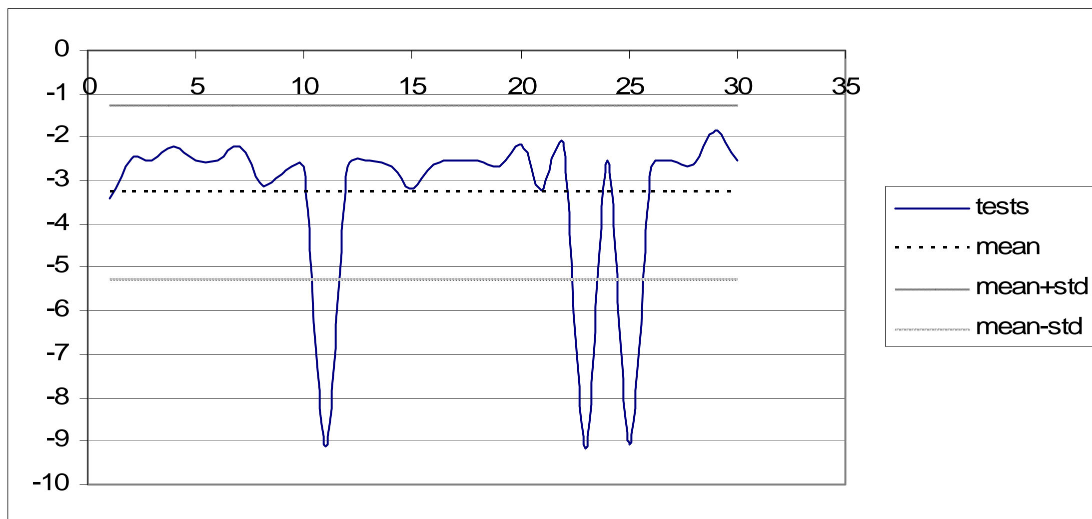

Figure 6.5- Roulette-wheel selection graphic for F5.

6.1.2- Tournament Selection Test Results

F1 = $f_1(x) = 100*(x_1^2 - x_2)^2 + (1 - x_1)^2$.

Range of x_1 and $x_2 = -2.048 \le x_i \le 2.048$.

x_1, x_2 with resolution factor of $\Delta x_i = 0.001$.

Global minimum of function = $\min f(x) = 0$ at $x_1 = 1$ $x_2 = 1$.

Table 6.7- Test results of tournament selection for F1.

#TEST	#Generation	Best_min Fitness
1	942	0.000020
2	9	0.136140
3	716	0.000072
4	7	0.008419
5	80	0.080672
6	0	0.045002
7	25	0.006413
8	12	0.002652
9	258	0.026434
10	69	0.061288
11	6	0.123862
12	13	0.238514
13	108	0.026434
14	834	0.061068
15	75	0.000007
16	637	0.055184
17	574	0.069519
18	878	0.002302
19	860	0.002133
20	8	0.009033

21	10	0.002764
22	13	0.067792
23	80	0.017201
24	14	0.006000
25	0	0.079174
26	601	0.037760
27	7	0.947822
28	209	0.000794
29	0	0.015277
30	15	0.012311

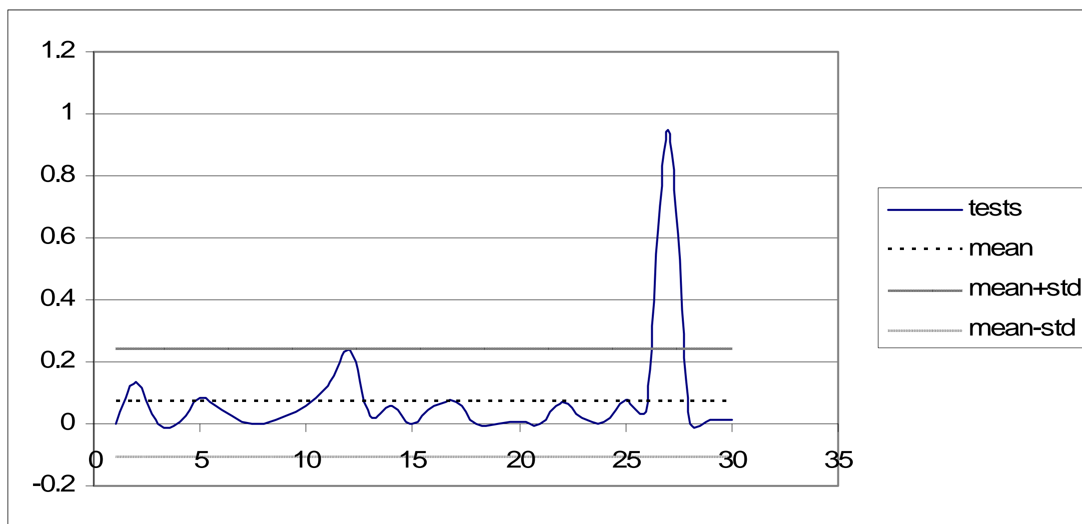

Figure 6.6- Tournament selection graphic for F1.

F2 = $f_2(x) = \sum_{i=1}^{N} (x_i)^2$.

$N = 5$.

Range of $x_i = -5.12 \le x_i \le 5.12 \quad x(i) = 0 \;\; i = 1:N$.

$i = 1:5$ with a resolution factor $\Delta x_i = 0.01$ on each axis.

Global minimum of function: $\min f(x) = 0$ at $x(i) = 0 \;\; i = 1:N$.

Table 6.8- Test results of tournament selection for F2.

#TEST	**#Generation**	**Best_min Fitness**
1	328	0.001929
2	997	0.003933
3	822	0.004534
4	792	0.005536
5	698	0.005335
6	416	0.006337
7	850	0.006137
8	637	0.003131
9	689	0.009143
10	753	0.005135
11	577	0.001127
12	821	0.000326
13	76	0.003131
14	584	0.004133
15	265	0.003532
16	789	0.000726
17	689	0.006337
18	302	0.005937
19	234	0.006337
20	104	0.004734

21	312	0.003332
22	606	0.005335
23	711	0.010145
24	206	0.003532
25	415	0.004935
26	499	0.007139
27	642	0.004734
28	379	0.005335
29	435	0.009343
30	921	0.004935

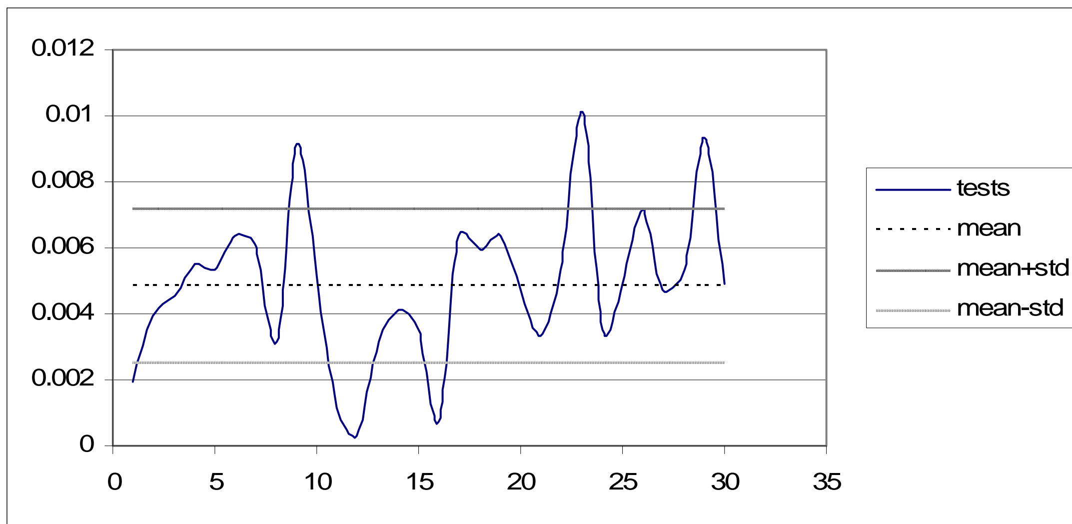

Figure 6.7- Tournament selection graphic for F2.

$$\textbf{F3} = f_3(x) = \frac{1}{4000}\sum_{i=1}^{N}(x_i - 100)^2 - \prod_{i=1}^{N}\cos\left(\frac{x_i}{\sqrt{i}}\right) + 1\,.$$

$N = 5$.

Range of x_i = $-600 \le x_i \le 600$ $x(i) = 0$ $i = 1:N$.

$i = 1:5$ with a resolution factor $\Delta x_i = 0.01$ on each axis.

Global minimum of function: $\min f(x) = 0$ at $x(i) = 0$ $i = 1:N$.

Table 6.9- Test results of tournament selection for F3.

#TEST	**#Generation**	**Best_min Fitness**
1	754	0.250106
2	210	0.229570
3	244	0.176016
4	599	0.306603
5	323	0.189994
6	276	0.294754
7	485	0.438590
8	559	0.276699
9	276	0.277848
10	786	0.362579
11	479	0.329012
12	670	0.334033
13	594	0.253912
14	281	0.225611
15	434	0.390381
16	364	0.255767
17	250	0.143045
18	725	0.260716
19	652	0.281300
20	429	0.356857

21	107	0.344653
22	478	0.351318
23	996	1.027333
24	482	0.190134
25	182	0.279729
26	670	0.362408
27	433	0.205664
28	464	0.310888
29	397	0.297143
30	165	0.195263

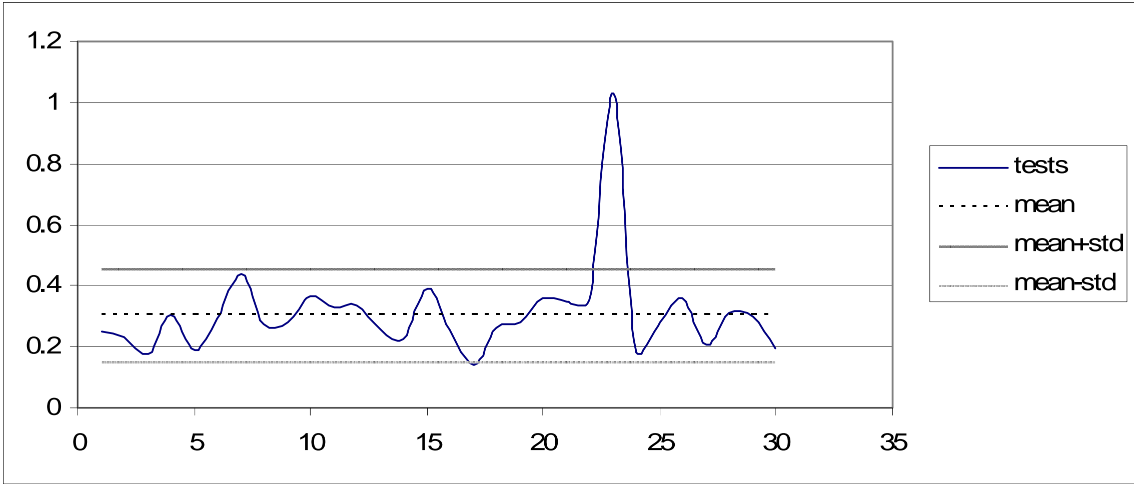

Figure 6.8- Tournament selection graphic for F3.

F4 = $f_4(x) = -\sum_{i=1}^{N} \sin(x_i) * \left(\sin\left(\frac{i * x_i^2}{\pi} \right) \right)^{2*m}$.

$N = 5$.

Range of $x_i = 0 \le x_i \le \pi$ $x(i) = 0 \;\; i = 1:N$.

$i = 1:5$ with a resolution factor $\Delta x_i = 0.01$ on each axis.

Global minimum of function = $\min f(x) = -4.687 \;\; x(i) = ? \;\; i = 1:N$.

Table 6.10- Test results of tournament selection for F4.

#TEST	#Generation	Best_min Fitness
1	738	-4.673854
2	182	-4.636954
3	657	-4.555921
4	741	-4.640059
5	833	-4.680653
6	476	-4.674481
7	912	-4.682179
8	904	-4.670561
9	893	-4.677593
10	526	-4.675248
11	339	-4.673780
12	358	-4.655159
13	968	-4.593417
14	320	-4.685186
15	358	-4.685059
16	753	-4.677592
17	785	-4.663129
18	391	-4.676722
19	96	-4.652464

20	525	-4.640640
21	238	-4.684059
22	608	-4.525021
23	66	-4.676359
24	588	-4.670075
25	124	-4.665690
26	322	-4.504623
27	204	-4.533488
28	250	-4.681997
29	337	-4.673471
30	48	-4.525344

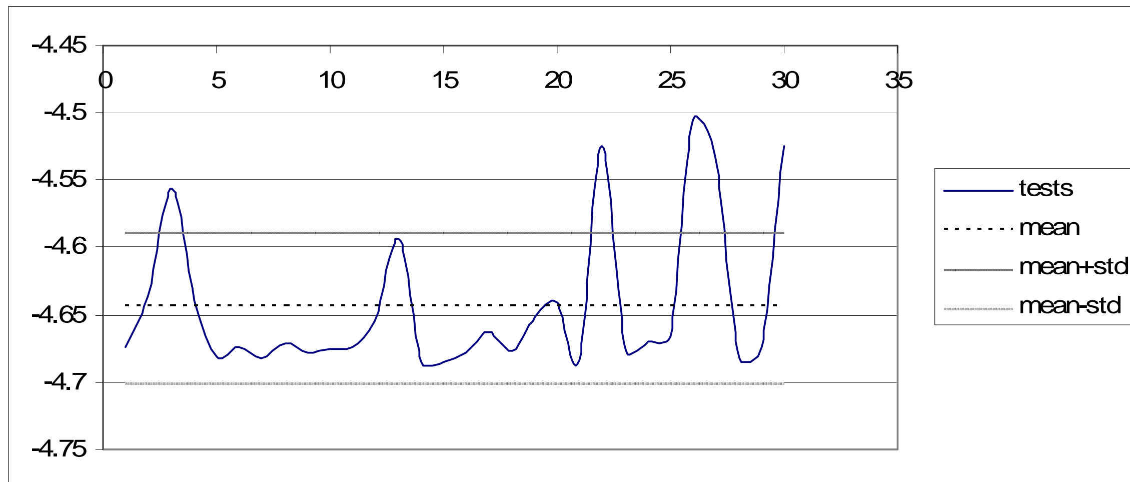

Figure 6.9- Tournament selection graphic for F4.

$$\textbf{F5} = f_5(x) = -\sum_{i=1}^{30} \frac{1}{\sum_{j=1}^{N} \left(x_i - a_{ij}\right)^2 + c_i}.$$

$N = 5$.

Range of x_i = $0 \le x_i \le 10$ $x(i) = 0$ $i = 1:N$.

$i = 1:5$ with a resolution factor of $\Delta_i = 0.01$ on each axis.

Global minimum of function = $f(x) = -10.40561$

$x_1 = 8.0249$ $x_2 = 9.1517$ $x_3 = 5.1139$ $x_4 = 7.6208$ $x_5 = 4.5640$

Table 6.11- Test results of tournament selection for F5.

#TEST	#Generation	Best_min Fitness
1	933	-1.826693
2	88	-1.775673
3	161	-1.826196
4	707	-1.599929
5	597	-2.539828
6	503	-2.202337
7	232	-2.199739
8	908	-1.826442
9	651	-2.549974
10	828	-1.823142
11	193	-1.290021
12	840	-1.826787
13	6	-2.124148
14	645	-2.233962
15	668	-2.200686
16	871	-1.827369
17	566	-1.824457
18	977	-2.663647

19	82	-2.550678
20	231	-1.827475
21	39	-4.849190
22	410	-1.828194
23	547	-1.824628
24	509	-1.888075
25	816	-2.201939
26	32	-2.131688
27	683	-2.204327
28	890	-1.824691
29	344	-2.667527
30	399	-2.198164

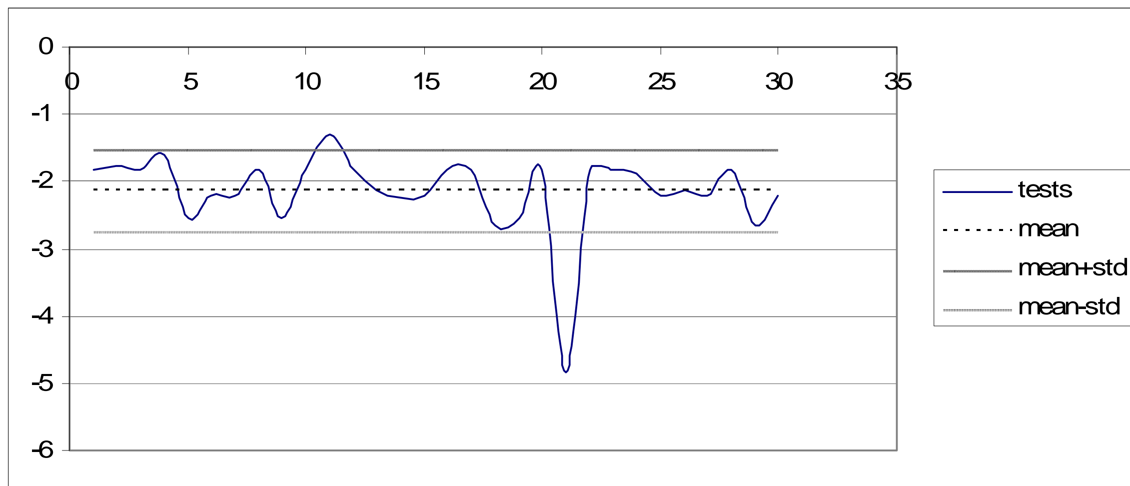

Figure 6.10- Tournament selection graphic for F5.

6.1.3- Comparison of Computational Test Results

Table 6.12- Convergence speed statistics for roulette-wheel selection.

ROULETTE-WHEEL SELECTION CONVERGENCE SPEED				
	Min	**Max**	**Mean**	**Std.Dev.**
F1	30	980	412	336
F2	4	903	432	248
F3	11	919	474	297
F4	108	932	438	267
F5	118	997	637	256

Table 6.13- Convergence speed statistics for tournament selection.

TOURNAMENT SELECTION CONVERGENCE SPEED				
	Min	**Max**	**Mean**	**Std.Dev.**
F1	1	943	236	332
F2	76	997	552	250
F3	107	996	459	212
F4	48	968	485	282
F5	6	977	512	309

Table 6.14- Comparison of convergence speed.

COMPRARISON of CONVERGENCE SPEED		
	ROULETTE-WHEEL	**TOURNAMENT**
F1		BETTER
F2	BETTER	
F3		BETTER
F4	BETTER	
F5		BETTER

For convergence speed;

Rosenbrock function (F1) tournament selection gave the better result than roulette-wheel selection.

De Jong 1 (F2) roulette-wheel selection gave the better result than tournament selection.

For Griewank's Function (F3) tournament selection gave the better result than roulette-wheel selection.

Michalewicz's Function (F4) roulette-wheel selection gave the better result than tournament selection.

Shekel's function (F5) tournament selection gave the better result than roulette-wheel selection.

Table 6.15- Roulette-wheel selection fitness statistics.

	ROULETTE-WHEEL SELECTION FITNESS						
	Optimal	**Best Value**	**Worst Value**	**Mean**	**σ Std deviation**	**Max = Mean+ σ**	**Min = Mean- σ**
F1	0	0.000305	0.062003	0.0118	0.013806	0.025606	-0.00201
F2	0	0.226367	4.204733	1.349426	0.879282	2.228708	0.470144
F3	0	2.507257	11.730000	6.971895	2.474827	9.446722	4.497068
F4	-4.687	-4.670503	-4.570916	-4.62373	0.025573	-4.598157	-4.6493
F5	-10.40561	-9.165722	-1.827188	-3.25448	2.015446	-1.239034	-5.26993

Table 6.16- Tournament selection fitness statistics.

	TOURNAMENT SELECTION FITNESS						
	Optimal	**Best Value**	**Worst Value**	**Mean**	**σ Std deviation**	**Max =Mean+ σ**	**Min =Mean- σ**
F1	0	0.000007	0.947822	0.071402	0.173673	0.2450751	-0.10227
F2	0	0.000326	0.010145	0.004875	0.002317	0.007192	0.002558
F3	0	0.143045	1.027333	0.306598	0.152835	0.459433	0.153763
F4	-4.687	-4.68519	-4.504623	-4.64369	0.055968	-4.587722	-4.69966
F5	-10.40561	-4.84919	-1.290021	-2.13859	0.607858	-1.530732	-2.74645

Table 6.17- Comparison of quality of global optimum.

COMPRARISON of Quality of Global Optimum		
	ROULETTE-WHEEL	**TOURNAMENT**
F1		BETTER
F2		BETTER
F3		BETTER
F4		BETTER
F5	BETTER	

For quality of global optimum;
Rosenbrock function (F1) tournament selection gave the better result than roulette-wheel selection.
De Jong 1 (F2) tournament selection gave the better result than roulette-wheel selection.
Griewank's Function (F3) tournament selection gave the better result than roulette-wheel selection.
Michalewicz's Function (F4) tournament selection gave the better result than roulette-wheel selection.
Shekel's function (F5) roulette-wheel gave the better result than tournament selection.

The below section analyzes why the tournament selection gave better results than roulette wheel according to selection intensity and selection probability.

6.2- Revision

Experiments were performed using genetic algorithm for five function optimization problems shown in Table 6.1. Each function has a prescribed search domain given in constraint column of the Table 6.1. The first function is Rosenbrock' function, which takes global minimum of function = $\min f(x) = 0$ at $x_1 = 1$ $x_2 = 1$. The second function is De Jong's sphere, which takes a global minimum of function = $\min f(x) = 0$ at

$x(i)=0$ $i=1:N$ =5. The third function is Griewank's function, which takes a global minimum of function = $\min f(x)=0$ at $x(i)=0$ $i=1:N$ =5. The forth function is Michalewicz's function, which takes a global minimum of function = $\min f(x)=-4.687$ $x(i)=?$ $i=1:N$ =5. The fifth function is Shekel's foxholes functions, which takes a global minimum of function = $f(x)=-10.40561$ $x_1=8.0249$ $x_2=9.1517$ $x_3=5.1139$ $x_4=7.6208$ $x_5=4.5640$.

For the each experiment, the population size is fixed with 50 individuals, for selection operation tournament and roulette-wheel selection is used, for crossover operation one point crossover technique (crossover probability = 0.8) and for mutation operation flip bit mutation (mutation probability = 0.05) is used.

Considering the tests results in Table 6.12 and Table 6.13, for first, third and fifth function, the tournament selection convergence faster to the global optimum solution than roulette-wheel selection. But for second and forth functions roulette-wheel selection convergence much faster to the optimal solution than tournament selection.

Also considering the test results in Table 6.15 and Table 6.16, for the first, second, third and forth function, the tournament selection's quality of optimal solution obtained is better than roulette-wheel selection. But for the fifth function the quality of solution obtained is better for the roulette-wheel selection than tournament selection.

CONCLUSION

Genetic Algorithms (GAs) are a popular class of iterative search techniques used to find good solutions to hard problems. GA evolves the population of individuals (solutions) to the global optimum by engaging the evolutionary operators: selection, mutation and crossover. The selection operator influences significantly on the speed of convergence to the global optimum and the quality of obtaining optimal solution.

The thesis investigated the two groups of selection schemes: proportionate-based selection strategy and ordinal-based selection strategy with the representative that is the roulette wheel selection and stochastic universal sampling selection operator (proportionate-based selection strategy) and binary tournament selection and linear ranking selection operator (ordinal-based selection strategy).

The various test cases were used to demonstrate the performance of GA with two selection operators described above.

Analysis of performance the selection schemes compared was done by using two criteria: the speed of convergence to the global optimum, and the quality of optimal solution obtained.

Quantitative analysis of the selection strategies showed that genetic algorithm with ordinal-based selection strategy converges faster than with proportionate-based selection schemes with comparative quality of optimal solution obtained. However, for some multimodal test functions the quality of the solution obtained is better for proportionate-based selection schemes than for ordinal-based one. It is because for multimodal functions the high level of selection intensity of ordinal-based selection strategy drives the GA to the local optimum which can be avoided by low level of selection intensity of proportionate-based selection schemes.

REFERENCES

[1]. N. Chaiyaratana and A. M. S. Zalzala (1997). Recent developments in evolutionary and genetic algorithms: Theory and applications, genetic algorithms in engineering systems. Innovations and applications, Conference publication no: 446.

[2]. Tobias Blickle and Lothar Thiele (1995). A comparison of selection schemes used in genetic algorithms. Computer engineering and communication laboratory Swish Federal Institute of Technology. Zurich Switzerland.

[3]. Goldberg D.E., Deb K. and Theirens (1993). Toward a better understanding of mixing in genetic algorithms. Journal of the Society of Instrument and Control Engineers.

[4]. Holland J.H. (1975). Adaptation in natural and artificial Systems. University of Michigan Press.

[5]. Baker J.E. (1985). Adaptive selection methods for genetic algorithms. Processing of an International Conference on Genetic Algorithms and Their Applications (pp.101-101). Hillsdale, NJ: Lawrence Erlbaum.

[6]. Grefenstette J.J. and Baker J.E. (1989). How genetic algorithms work: A critical look at implicit parallelism. Processing of the 3rd International Conference on Genetic Algorithms (pp.20-27). San Mateo, CA: Morgan Kaufmann.

[7]. D.S. Huang, Horace H.S.Ip, Zheru Chi and H.S. Wong (2003). Dilation method for finding close roots of polynomials based on constrained learning neural networks. Physics letters a, Vol.309, No.5-6 (pp.443-451).

[8]. Wael Mustafa (2003). Optimization of production systems using genetic algorithms. International Journal of Computational Intelligence and Applications, Vol. 3 (pp.233-248).

[9]. Brindle A. (1981). Genetic algorithms for function optimization. Unpublished doctoral dissertation. University of Alberta, Edmonton. Canada.

[10]. Zen and the art of genetic algorithms (1989). In J.D. Schaffer (Ed.). Proceedings of the 3rd International Conference on Genetic Algorithms (pp.80-85). San Mateo, CA: Morgan Kaufmann (Also TCGA Report 88003).

[11]. Mühlenbein H. & Schlierkamp-Voosen, D. (1993). Predictive models for the breeder genetic algorithm. 1st Continuous Parameter Optimization. Evolutionary computation 1 (pp.25-49).

[12]. Baker, J.E. (1987). Reducing bias and inefficiency in the selection algorithm. In J.J.Grefenstette(Ed.). Proceedings of the 2nd International Conference on Genetic Algorithms (pp.14-21). Hillsdale, NJ: Lawrance Erlbaum.

[13]. Davis L. (1991). Handbook of genetic algorithms, Van Nonstrand Reinhold.
New York.

[14]. Goldberg, D.E. (1989). Genetic algorithms in search optimization and machine learning. Addison Wesley, Reading, MA.

[15]. Oliver, J. M., Smith, D. J. and Holland, J. R. C. (1987). A study of permutation crossover operators on the traveling salesman problem. 2nd International Conference on Genetic Algorithms (pp. 224-230).

[16]. Michalewicz Z. (1992). Genetic algorithm + Data structures = Evolution programs, Springer Verlag.

[17]. J. Arabas, Z. Michalewicz and J. Mulawka Gavaps (1994). A genetic algorithm with varying population size. 1st IEEE Conference on Evolutionary Computation (pp.73-78). IEEE Press, Piscataway.

[18]. Brad L. Miller & David E. Goldberg (1996). Genetic algorithms, selection schemes and the varying effects of noise. International Conference on Evolutionary Computation.

[19]. Sywerda, G. (1989). Uniform crossover in genetic algorithms. 3rd International Conference on Genetic Algorithms (pp.2-9).

[20]. Spears W.M and De Jong K.A. (1994). On the virtues of parameterized uniform crossover. 4th International Conference on Genetic Algorithms.

[21]. Davis L. (1985). Applying algorithms to epistatic domains. International Joint Conference on Artificial Intelligence (pp.162-164).

[22]. K. A. De Jong. (1975). Analysis of the behavior of a class of genetic adaptive systems. Ph.D. dissertation. The University Of Michigan.

[23]. R.L. Haupt and S.E. Haupt (1998). Practical genetic algorithms. New York: John Wiley & Sons.

[24]. J. J. Grefenstette (1986). Optimization of control parameters for genetic algorithms. IEEE Transactions on Systems, Man, and Cybernetics (pp.128).

[25]. George H. Gates Jr., Laurance D.Merkle, Gary B.Lamont (1995). Simple genetic algorithm parameter selection for protein structure prediction. Department of Electric and Computer Engineering Graduate School of Engineering Air Force Institute of Technology.

[26]. Schaffer J.David. (1989). A study of control parameters affecting online performance of genetic algorithms for function optimization. 3rd International Conference on Genetic Algorithms

[27]. Bäck K. (1995). Genratized convergence models for tournament selection and (μ, λ) selection. 6th International Conference on Genetic Algorithms (pp.2-8). SanFrancisco, CA: Morgan Kaufmann.

[28]. Xin Yao (1997). Global optimization by evolutionary algorithms. Computational Intelligence Group, School of Computational Science University Collage, The University of New South Wales Australian Defence Force Academy, Canberra, ACT, Australia 2600. IEEE Computer Society Press.

[29]. Goldberg D.E. and Lingle R. (1985). Alleles, loci, and the TSP. 1st International Conference on Genetic Algorithms (pp.54-159).

[30]. Goldberg D. and Deb K. (1991). A comparative analysis of selection schemes used in genetic algorithms. In G. Rawlins, editor, Foundations of Genetic Algorithms, pp. 69-93, Sam Mateo, Morgan Kaufmann.

APPENDIX A

STOCHASTIC UNIVERSAL SAMPLING SELECTION TEST RESULTS

For F1 = $f_1(x) = 100*(x_1^2 - x_2)^2 + (1 - x_1)^2$.

Range of x_1 and $x_2 = -2.048 \le x_i \le 2.048$.

x_1, x_2 with resolution factor of $\Delta x_i = 0.001$.

Global minimum of function = $\min f(x) = 0$ at $x_1 = 1$ $x_2 = 1$.

Table A.1- Test results of stochastic universal sampling selection for F1.

#TEST	**#Generation**	**Best_min Fitness**
1	483	0.000413
2	290	0.006787
3	368	0.008196
4	578	0.014895
5	660	0.063488
6	979	0.011768
7	185	0.029804
8	559	0.01305
9	759	0.052128
10	75	0.002095
11	242	0.050462
12	518	0.000056
13	408	0.000899
14	247	0.013412
15	0	0.006471
16	11	0.012307

17	183	0.04091
18	576	0.000434
19	82	0.004121
20	837	0.004846
21	267	0.009367
22	428	0.000766
23	75	0.001824
24	160	0.04602
25	424	0.021923
26	760	0.023113
27	929	0.001495
28	106	0.000735
29	546	0.000517
39	571	0.002949

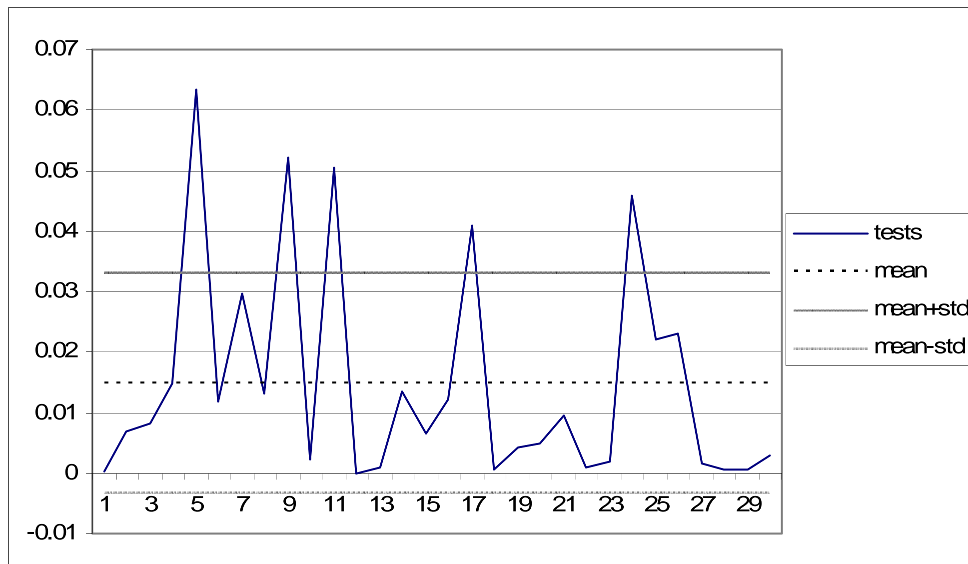

Figure A.1- Stochastic universal sampling selection graphic for F1.

For F2 = $f_2(x) = \sum_{i=1}^{N}(x_i)^2$.

$N = 5$.

Range of x_i = $-5.12 \le x_i \le 5.12$ $x(i) = 0$ $i = 1:N$.

$i = 1:5$ with a resolution factor $\Delta x_i = 0.01$ on each axis.

Global minimum of function: $\min f(x) = 0$ at $x(i) = 0$ $i = 1:N$.

Table A.2- Test results of stochastic universal sampling selection for F2.

#TEST	**#Generation**	**Best_min Fitness**
1	333	3.545847
2	470	0.706504
3	651	1.430992
4	634	1.593636
5	178	2.395201
6	996	0.340189
7	628	1.400806
8	506	0.757804
9	391	0.652599
10	835	2.640279
11	120	1.676197
12	916	1.601005
13	778	0.682658
14	410	1.090454

15	916	2.503011
16	684	0.676045
17	372	0.153224
18	682	1.573396
19	522	1.640127
20	0	2.565533
21	110	0.802692
22	754	2.202425
23	470	0.510121
24	849	0.660615
25	910	2.175372
26	743	2.367947
27	484	0.968015
28	218	1.446148
29	891	1.115503
30	139	0.68366

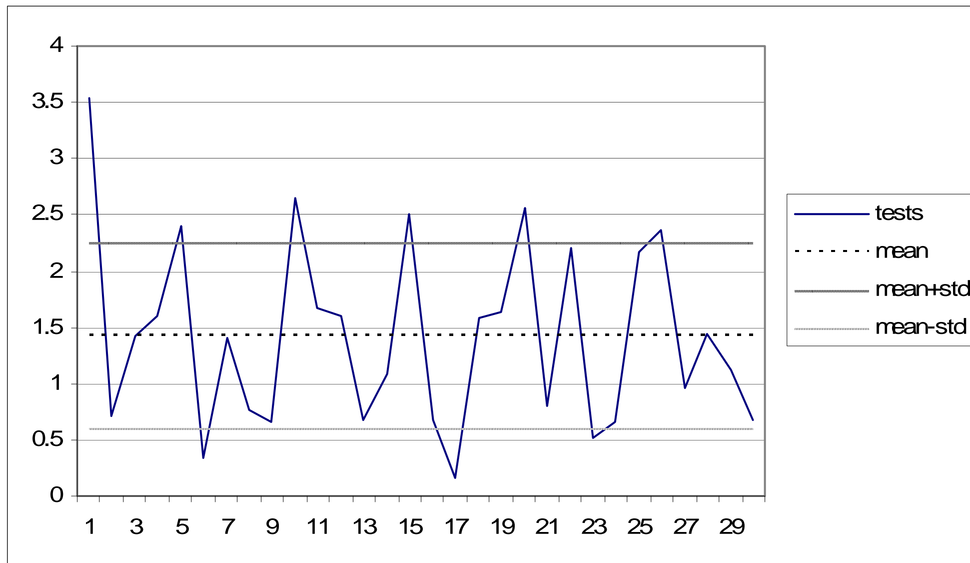

Figure A.2- Stochastic universal sampling selection graphic for F2.

For F3 = $f_3(x) = \frac{1}{4000}\sum_{i=1}^{N}(x_i - 100)^2 - \prod_{i=1}^{N}\cos\left(\frac{x_i}{\sqrt{i}}\right) + 1$.

$N = 5$.

Range of x_i = $-600 \le x_i \le 600$ $x(i) = 0$ $i = 1:N$.

$i = 1:5$ with a resolution factor $\Delta x_i = 0.01$ on each axis.

Global minimum of function: $\min f(x) = 0$ at $x(i) = 0$ $i = 1:N$.

Table A.3- Test results of stochastic universal sampling selection for F3.

#TEST	**#Generation**	**Best_min Fitness**
1	56	3.172907
2	19	3.389672
3	314	7.666584
4	432	4.384844
5	254	8.340647
6	117	9.293668
7	120	6.304651
8	39	5.554811
9	469	7.503999
10	929	9.360593
11	827	9.971512

12	781	5.495712
13	583	10.154668
14	796	10.171372
15	516	7.35035
16	940	8.082796
17	393	17.447573
18	824	10.38502
19	486	11.311725
20	650	5.324049
21	592	6.872097
22	432	10.882447
23	822	12.615409
24	56	5.466848
25	823	8.507217
26	319	5.449208
27	361	11.246358
28	965	7.981456
29	175	8.924101
30	156	9.577565

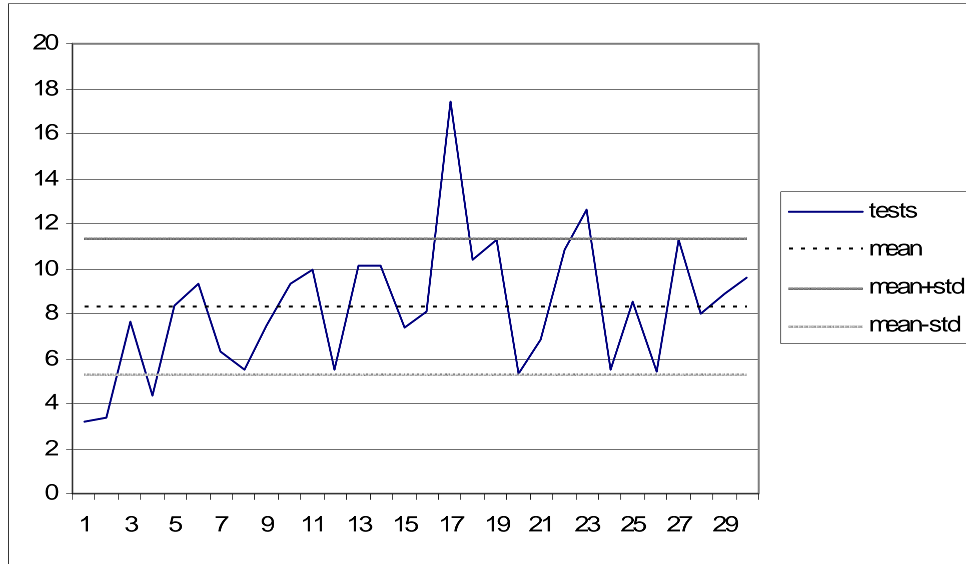

Figure A.3- Stochastic universal selection sampling graphic for F3.

For F4 = $f_4(x) = -\sum_{i=1}^{N} \sin(x_i) * \left(\sin\left(\frac{i * x_i^2}{\pi} \right) \right)^{2*m}$.

$N = 5$.

Range of $x_i = 0 \le x_i \le \pi \qquad x(i) = 0 \;\; i = 1:N$.

$i = 1:5$ with a resolution factor $\Delta x_i = 0.01$ on each axis.

Global minimum of function = $\min f(x) = -4.687 \;\; x(i) = ? \;\; i = 1:N$.

Table A.4- Test results of stochastic universal sampling selection for F4.

#TEST	#Generation	Best_min Fitness
1	461	-4.63001
2	866	-4.6223
3	608	-4.68107
4	942	-4.657022
5	630	-4.665945
6	539	-4.63699
7	728	-4.642744
8	763	-4.639535
9	805	-4.661801
10	593	-4.653302
11	822	-4.635121
12	79	-4.654263
13	873	-4.66182
14	544	-4.611335
15	691	-4.622484
16	374	-4.623354
17	262	-4.647264
18	233	-4.629636
19	820	-4.627598
20	493	-4.639965
21	641	-4.626991
22	589	-4.622513
23	839	-4.645216
24	744	-4.583519
25	348	-4.625952
26	982	-4.635473
27	987	-4.643882
28	239	-4.616704
29	784	-4.632202

30	148	-4.546722

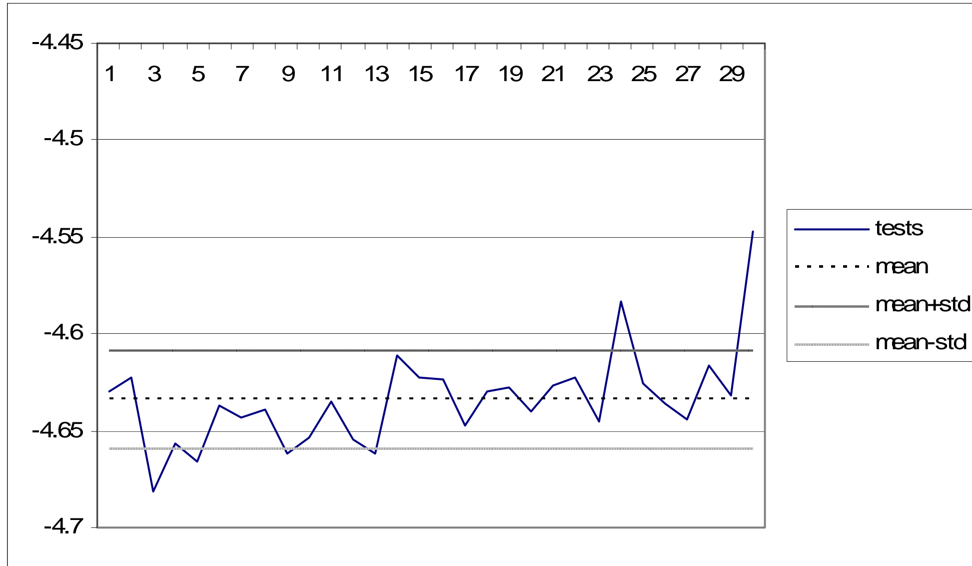

Figure A.4- Stochastic universal sampling selection graphic for F4.

For F5 $= f_5(x) = -\sum_{i=1}^{30} \frac{1}{\sum_{j=1}^{N} (x_i - a_{ij})^2 + c_i}$.

$N = 5$.

Range of x_i = $0 \le x_i \le 10$ $\quad x(i) = 0 \;\; i = 1 : N$.

$i = 1:5$ with a resolution factor of $\Delta_i = 0.01$ on each axis.

Global minimum of function = $f(x) = -10.40561$

$x_1 = 8.0249$ $x_2 = 9.1517$ $x_3 = 5.1139$ $x_4 = 7.6208$ $x_5 = 4.5640$

Table A.5- Test results of stochastic universal sampling selection for F5.

#TEST	#Generation	Best_min Fitness
1	293	-2.033499
2	779	-2.535636
3	39	-3.027679
4	748	-1.888267
5	606	-10.318615
6	823	-2.548636
7	361	-1.869241
8	120	-1.941765
9	985	-3.396333
10	736	-9.125092
11	266	-1.955783
12	313	-2.654759
13	67	-2.871051
14	404	-2.537512
15	800	-3.273476
16	275	-2.192412
17	213	-2.383385
18	571	-2.537207
19	942	-2.251596
20	264	-2.648869
21	259	-2.655948
22	663	-9.096451
23	522	-1.811075
24	66	-2.178304

25	105	-2.980146
26	234	-3.459007
27	892	-2.199427
28	44	-2.050634
29	565	-2.652656
30	776	-9.114754

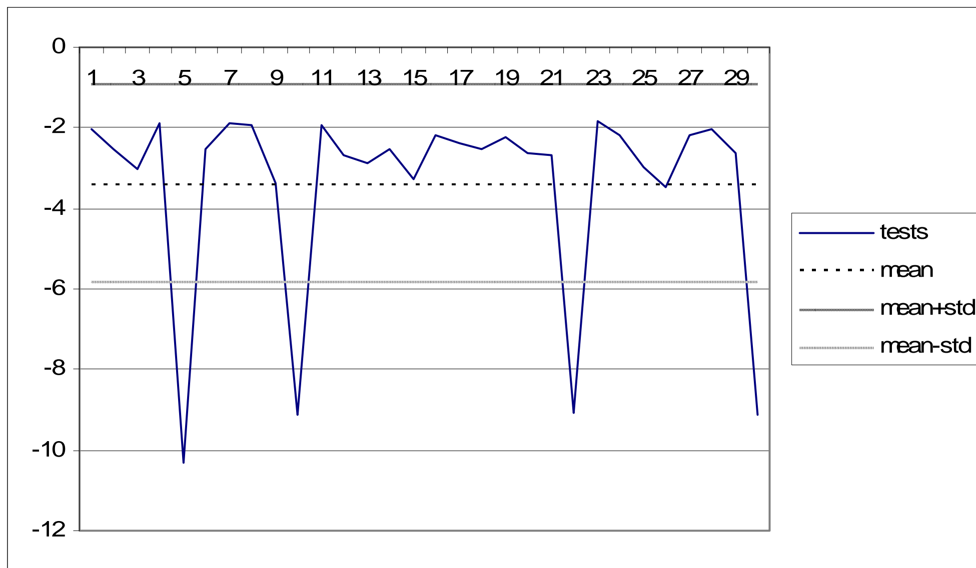

Figure A.5- Stochastic universal sampling selection graphic for F5.

Table A.6- Convergence speed statistics for stochastic universal sampling selection.

Function	**Best**	**Worst**	**Mean**	**Std.Dev.**
F1	518	660	410	277
F2	17	333	553	283
F3	56	393	475	306
F4	608	148	614	254
F5	606	522	458	301

Table A.7- Stochastic universal sampling selection fitness statistics.

	STOCHASTIC UNIVERSAL SAMPLING SELECTION						
Fnc.	**Optimal**	**Best Value**	**Worst Value**	**Mean**	**Std σ deviation**	**Max = Mean+ σ**	**Min = Mean- σ**
F1	0	0.000056	0.063488	0.014842	0.018144	0.032986	-0.003302
F2	0	0.153224	3.545876	1.4186	0.827653	2.246253	0.590947
F3	0	3.172907	17.44757	8.272995	2.989659	11.26265	5.283336
F4	-4.687	-4.68107	-4.54672	-4.63409	0.025146	-4.60894	-4.65924
F5	-10.40561	-10.31861	-1.81108	-3.40631	2.444945	-0.96137	-5.85126

APPENDIX B

GENETIC ALGORITHM SOURCE CODES

Definition.h **:** contains type definitions of data types.

```
#include <stdio.h>
#include <math.h>
#include <conio.h>
#include <stdlib.h>
#include <time.h>//for srand() function

const int MAXPOP=200;
const int MAXSTRING=100;
const int MAXPARMS=10;
const float PI=3.1415927;
const int MAXGEN=10000;
typedef bool allele; // Allele=bit position
typedef allele chromosome[MAXSTRING]; // String of bits

typedef struct ind_tag{
   chromosome chrom; // Genotype=bit string
   float x[MAXPARMS];// Phenotype = unsigned integer
   float fitness;    // Objective function value
   int parent1,parent2,xsite;  // Parents and cross pt
}individual;
typedef individual population[MAXPOP];

typedef struct parmparm_tag{    //parameters of parameter
   int lparm;  //length of parameter
   float parameter,maxparm,minparm; //parameter & range
}parmparm;
```

```
typedef parmparm parmspecs[MAXPARMS];
population oldpop, newpop; // Two non-overlapping populations
int pop_b,pop_w,popsize, lchrom, gen, maxgen; // Integer global variables
float pcross, pmutation, sumfitness;  // Real global variables
int nmutation, ncross;// Integer statistics
float avg, max, min;// Real statisticzs
//double stddev;/* std. deviation of population fitness */
//double sum_square;/* sum of square for std. calc */
//double square_sum;/* square of sum for std. calc */
parmspecs parms;
int nparms;
FILE *infile,*logfile;
int   cur_best,cur_worst,cur_genb,cur_genw,currentWorst;
float best,worst,currentWorstFitness;
float minsarray[MAXGEN],ia;
float maxsarray[MAXGEN],aa;
float maxmax,maxmin;
float minmax,minmin;
int gen_maxmax,gen_maxmin,gen_minmax,gen_minmin;
int tourneylist[MAXPOP],tourneypos,tourneysize;
int choices[MAXPOP],nremain;
int ranklist[MAXPOP];
float ranklist_probabilities[MAXPOP];
```

Generate.h **:** contains new population generation routine.

```
/* create a new generation through select, crossover and
mutation. Note: generation assumes an even-numbered popsize */
void generation(){
int j, mate1, mate2, jcross,i,k;
j=0;
        do{// select , crossover and mutation until newpop is filled
        mate1=select_ts(); //pick pair of mates
        mate2=select_ts();
                //Crossover and mutation-mutation embedded within crossover
    crossover(oldpop[mate1].chrom,oldpop[mate2].chrom,
        newpop[j].chrom,newpop[j+1].chrom,lchrom,ncross,nmutation,jcross,pcross,pmut
ation);
   //Decode string, evaluate fitness,&record parentage date on both children
        decode_parms(nparms,lchrom,newpop[j].chrom,parms);
        for(i=0;i<nparms;i++)
        newpop[j].x[i]=parms[i].parameter;
        newpop[j].fitness=objfunc(newpop[j].x,nparms);
        newpop[j].parent1=mate1;
        newpop[j].parent2=mate2;
        newpop[j].xsite=jcross;
        for(k=0;k<nparms;k++)
        newpop[j+1].x[k]=parms[k].parameter;
        newpop[j+1].fitness=objfunc(newpop[j+1].x,nparms);
        newpop[j+1].parent1=mate1;
        newpop[j+1].parent2=mate2;
        newpop[j+1]=jcross;
        j=j+2; // Increment population index
  }while(j<popsize);
}
```

Initialize.h **:** contains initdata(), initpop(), initreport() routines.

```
void initdata(); //Interactive data inquiry and setup
void initreport(); //Initial report
void initpop(); // Initialize population at random
void initialize(); //Initialization coordinator

void initdata(){
    int i;
    lchrom=0;
        if((infile=fopen("gaparameters.txt","r"))== NULL)
        {
            fprintf(logfile,"\n Cannot open gaparameters.txt file!\n");
            exit(1);
        }
        for(i=0;i<1;i++){
            fscanf(infile,"%d",&nparms);
            fscanf(infile,"%d",&popsize);
            fscanf(infile,"%d",&maxgen);
            fscanf(infile,"%f",&pcross);
            fscanf(infile,"%f",&pmutation);
        }
        if((infile=fopen("parameters.txt","r"))== NULL)
        {
            fprintf(logfile,"\n Cannot open parameters.txt file!\n");
            exit(1);
        }
        for(i=0;i<nparms;i++){
            fscanf(infile,"%d",&parms[i].lparm);
            fscanf(infile,"%f",&parms[i].minparm);
            fscanf(infile,"%f",&parms[i].maxparm);
            lchrom=lchrom+parms[i].lparm;
```

```
            }
        // Initialize random number generator
        randomize();
        // Initialize counters
        nmutation=0;
        ncross=0;
}
void initreport(){

        fprintf(logfile,"        SGA Parameters\n");
        fprintf(logfile,"        ---------------\n");
        fprintf(logfile,"  Number of Parameter(nparms)              =   %d\n",nparms);
        fprintf(logfile,"  Population size (popsize)           =   %d\n",popsize);
        fprintf(logfile,"  Chromosome length (lchrom)           =   %d\n",lchrom);
        fprintf(logfile,"  Maximum # of generation (maxgen)      =   %d\n",maxgen);
        fprintf(logfile,"  Crossover probability (pcross)        =   %6f\n",pcross);
        fprintf(logfile,"  Mutation probability (pmutation)      =   %6f\n\n",pmutation);
        for (int j=0;j<nparms;j++){
                //fprintf(logfile,"%d th parameter.  value > ",j); fprintf(logfile,"%f
\n",parms[j].parameter);
                fprintf(logfile,"%d th parm length > ",j); fprintf(logfile,"%d
\n",parms[j].lparm);
                fprintf(logfile,"%d th max.  value > ",j); fprintf(logfile,"%f
\n",parms[j].maxparm);
                fprintf(logfile,"%d th min.  value > ",j); fprintf(logfile,"%f
\n",parms[j].minparm);
        }
        fprintf(logfile,"        Initial Generation Statistics\n");
        fprintf(logfile,"        -----------------------------\n");
        fprintf(logfile,"  Initial population maximum fitness    =   %6f\n",max);
        fprintf(logfile,"  Initial population minimum fitness    =   %6f\n",min);
```

```
        fprintf(logfile,"  Initial population average fitness     =    %6f\n",avg);
        fprintf(logfile,"  Initial population sum of fitness      =    %6f\n",sumfitness);
}

void initpop(){
        int j, j1,i;
        for(j=0;j<popsize;j++){
         for(j1=0;j1<lchrom;j1++)
                oldpop[j].chrom[j1]=flip(0.5); // a fair coin toss
         decode_parms(nparms,lchrom,oldpop[j].chrom,parms);
         for(i=0;i<nparms;i++)
         oldpop[j].x[i+1]=parms[i].parameter;
         //oldpop[j].x=decode(oldpop[j].chrom,lchrom);
         //oldpop[j].fitness=objfunc1(oldpop[j].x);
         oldpop[j].fitness=objfunc(oldpop[j].x,nparms); // Evaluate initial fitness
         oldpop[j].parent1=0;
         oldpop[j].parent2=0;
         oldpop[j]=0;
      /* Initialize printout vars */
    }
}

void initialize(){
  initdata();
  initpop();
  statistics(popsize,max,min,sumfitness,oldpop);
  initreport();
}
```

Kernel.h : contains three operators Reproduction (selection), Crossover (crossover), Mutation (mutation).

```
int select_rws(int popsize, float sumfitness, population pop);
/* Select a singleindividual via roulette wheel selection */
void reset();
int select_ts(int popsize,population pop);
int select_sus(int popsize,float avg,population pop);
allele mutation(allele alleleval, float pmutation, int &nmutation);
/* Mutate an allele w/ pmutation, count number of mutations */
void crossover(chromosome parent1, chromosome parent2, chromosome child1,
   chromosome child2,int &lchrom,int &ncross,int &nmutation,int &jcross,
   float &pcross,float &pmutation);
/* Cross 2 parent strings, place in 2 child strings */
int select_sus(int popsize,float avg,population pop){
        int    j, k;
        float  pointer, sum;
  if (avg == 0) {
    for (j = 0; j < popsize; j++)
      choices[j] = j;
  }
  else {
    k = 0;
    pointer = random();
                                                                 //
    sum = 0.0;
    for (j = 0; j < popsize; j++) {
                    for (sum += (pop[j].fitness / avg); sum > pointer; pointer++) {
                      choices[k++] = j;
                    }
      if(k >= popsize)
        break;
```

```
    }
  }
  nremain = popsize - 1;

  int     jpick, slect;
  jpick = rnd(0, nremain);
  slect = choices[jpick];
  choices[nremain] = choices[nremain];
  nremain--;
  return (slect);
}
int select_ts(int popsize,population pop){
        int pick, winner, i;
        tourneysize=2;
    /* If remaining members not enough for a tournament, then reset list */
    if((popsize - tourneypos) < tourneysize){
                reset();
                tourneypos = 0;
    }
    /* Select tourneysize structures at random and conduct a tournament */

    winner=tourneylist[tourneypos];
        for(i=1; i<tourneysize; i++){
                pick=tourneylist[tourneypos];
                if(pop[pick].fitness < pop[winner].fitness)
    }
    /* Update tourneypos */
    tourneypos += tourneysize;
    return(winner);
}
void reset(){/* Shuffles the tourneylist at random */
```

```
    int i, rand1, rand2, temp;
    for(i=0; i<popsize; i++)
                    tourneylist[i] = i;
    for(i=0; i < popsize; i++){
        rand1=rnd(i,popsize-1);
        rand2=rnd(i,popsize-1);
        temp = tourneylist[rand1];
        tourneylist[rand1]=tourneylist[rand2];
        }
}

int select_rws(int popsize, float sumfitness, population pop){
            float rand, partsum; //Random point on wheel, partial sum
            int j; //population index
            partsum=0.0; j=0; //Zero out counter and accumulator
            rand=sumfitness*random(); //Wheel point calc. uses random number [0,1]
            do{      // Find wheel slot
                        partsum=partsum+pop[j].fitness;
                        j++;
                        }while(partsum>rand);
            return (j-1);  // Return individual number
}
allele mutation(allele alleleval, float pmutation, int &nmutation){
            bool mutate;
  mutate=flip(pmutation); //Flip the biased coin
  if(mutate){
              nmutation++;
      return !alleleval; //Change bit value
    }
  else  return alleleval; //No change
}
```

```
void crossover(chromosome parent1, chromosome parent2, chromosome child1,
  chromosome child2,int &lchrom,int &ncross,int &nmutation,int &jcross,
  float &pcross,float &pmutation)
{
int j;
 if(flip(pcross))    //Do crossover with pcross
   {
        jcross=rnd(1,lchrom-1);//Cross between 1 and lchrom-1
        ncross++;       //Increment crossover counter
   }
 else jcross=lchrom;
 // first exchange, 1 to 1 and 2 to 2
 for(j=0;j<jcross;j++)
   {
        child1[j]=mutation(parent1[j],pmutation,nmutation);
        child2[j]=mutation(parent2[j],pmutation,nmutation);
   }
 // second exchange, 1 to 2 and 2 to 1
 if(jcross!=lchrom)
   for(j=jcross;j<lchrom;j++)
     {
        child1[j]=mutation(parent2[j],pmutation,nmutation);
        child2[j]=mutation(parent1[j],pmutation,nmutation);
     }
}
```

ObjectFunction.h **:** contains objectfunction() and decode() routines.

```
float objfunc(float x[],int n); // fitness function-f(x)=...
float decode(chromosome chrom,int lbits);
/* Decode string as unsigned binary integer - true=1, false=0 */
float objfunc(float x[],int n){
                /* Put object function */
        }
float decode(chromosome chrom,int lbits){
        int j;
        float accum=0.0,powerof2=1.0;
        for(j=0;j<lbits;j++){
                if(chrom[j])
                        accum=accum+powerof2;
                powerof2=powerof2*2;
                }
         return accum;
}
```

Parameters.h : cotains extract_parm(), map_parm(), decode_parms() routines.

```
void extract_parm(chromosome chromfrom,chromosome chromto,int &jposition, int
&lchrom,int &lparm); //Extract a substring from a full string
float map_parm(float x,float maxparm,float minparm,float fullscale); /* Map an unsigned
binary integer to range [minparm,maxparm] */
void decode_parms(int &nparms,int &lchrom,chromosome chrom,
       parmspecs parms); // Decode parameter
void writech(chromosome chrom,int l);
void extract_parm(chromosome chromfrom,chromosome chromto,int &jposition,
     int &lchrom,int &lparm)
{
 int j,jtarget;
```

```
 j=0;
 if(jtarget>lchrom)
          jtarget=lchrom; // Clamp if excessive
 while(jposition<jtarget)
   {
         chromto[j]=chromfrom[jposition];
         ++jposition;
         j++;
   }
}

float map_parm(float x,float maxparm,float minparm,float fullscale){
         return (minparm+(maxparm-minparm)/fullscale*x);
}

void writech(chromosome chrom,int l){
         int j;
         printf("\n");
         for(j=0;j<l;j++)
   if(chrom[j])
                  printf("1");
   else printf("0");
}
void decode_parms(int &nparms,int &lchrom,chromosome chrom,parmspecs parms){
         int j,jposition;
         chromosome chromtemp; // Temporary string buffer
         j=0; //Parameter counter
         jposition=0; //String position counter
         do{
     if(parms[j].lparm>0){
                  extract_parm(chrom,chromtemp,lchrom,parms[j].lparm);
```

```
parms[j].parameter=map_parm(decode(chromtemp,parms[j].lparm),parms[j].maxparm,pa
rms[j].minparm,pow(2.0,parms[j].lparm)-1);
                    //writech(chrom,parms[j].lparm);printf("-");
                    //writech(chromtemp,parms[j].lparm);
                    //printf("\n:%f        :%f :%f         :%f
         :%f\n",parms[j].parameter,decode(chromtemp,parms[j].lparm),parms[j].maxparm,
parms[j].minparm,pow(2.0,parms[j].lparm)-1);
                    //printf("/%f  /%f
         /%f\n",parms[j].parameter,decode(chromtemp,parms[j].lparm),pow(2.0,parms[j].l
parm)-1);
          }
    j++;
   }
while(j<nparms);
}
```

Random.h **:** contains random number utility programs.

```
float oldrand[55]; //array of 55 random number
int jrand; //current random
void advance_random(); // create next batch of 55 random numbers
void warmup_random(float random_seed); // get random off and runnin
float random(); /* fetch a single random number between 0.0 and 1.0 -
Subtractive Method (see Knuth D.(1969), v.2 for details) */
bool flip(float probability);//flip a biased coin-true if heads
int rnd(int low,int high); //pick a random integer between low and high
void randomize(); //get seed number for random and start it up

void advance_random()
{
```

```
float new_random;
 for(int j1=0;j1<24;j1++)
   { new_random=oldrand[j1]-oldrand[j1+30];
     if(new_random<0.0) new_random=new_random+1.0;
     oldrand[j1]=new_random;
   }
 for(int j1=24;j1<55;j1++)
   {
          new_random=oldrand[j1]-oldrand[j1-23];
         if(new_random<0.0) new_random=new_random+1.0;
                  oldrand[j1]=new_random;
   }
}

void warmup_random(float random_seed)
{ int ii;
 float new_random,prev_random;
 oldrand[54]=random_seed;
 new_random=1.0e-06;
 prev_random=random_seed;
 for(int j1=0;j1<54;j1++)
   { ii=21*j1%55;
     oldrand[ii]=new_random;
     new_random=prev_random-new_random;
     if(new_random<0.0) new_random=new_random+1.0;
     prev_random=oldrand[ii];
   }
 advance_random();
 advance_random();
 advance_random();
 jrand=0;
```

```
}
float random()
{ jrand=jrand+1;
  if(jrand>55)
    { jrand=1;
      advance_random();
    }
  return oldrand[jrand];
}

bool flip(float probability)
{ if(probability==1.0) return true;
  else return (random()<=probability);
}
int rnd(int low,int high)
{ int i;
  if(low>=high) i=low;
  else{ i=(int)(random()*(high-low+1)+low);
        if(i>high) i=high;
      }
  return i;
}
void randomize(){
            srand((unsigned)time(NULL));
            double randomseed,high,low;
            high=1.0;low=0.0;
            double ra=(high-low);
            randomseed=low+double(ra*rand()/double(RAND_MAX+1.0));
            fprintf(logfile,"Auto random seed number:%f\n",randomseed);
    warmup_random(randomseed);
}
```

Reports.h : contains routines used to print a report from each cycle of genetic algorithm's operation.

```
void writechrom(chromosome chrom,int lchrom); /* Write a chromosome
    as a string of 1's (true's) and 0's (false's)   */
void report(int gen); // Write the population report

void writechrom(chromosome chrom,int lchrom)
{ int j;
  for(j=0;j<lchrom;j++)
   if(chrom[j]) fprintf(logfile,"1");
   else fprintf(logfile,"0");
}

void report(int gen)
{ int j;

  fprintf(logfile,"\n        Population Report  Generation  %d\n",gen);
  for(j=0;j<popsize;j++){
            fprintf(logfile,"\n population %2d>>",j);
            //writechrom(newpop[j].chrom,lchrom);
            for (int i = 0; i < nparms; i++){

                  fprintf (logfile," parm(%d) = %6f",i,newpop[j].x[i]);
                  }
            fprintf(logfile,"fitness: %6f",newpop[j].fitness);
      }

  // Generation statistics and accumulated values
  fprintf(logfile,"\nNote: Generation  %d  & Accumulated Statistics:\n",gen);
  fprintf(logfile,"max=%6f min=%6f avg=%6f  sum=%6f nmutation=%d
ncross=%d\n",max,min,avg,sumfitness,nmutation,ncross);
```

```
  //printf("gen:%d max=%.3f min=%.3f avg=%.3f sum=%.3f
\n",gen,max,min,avg,sumfitness);
  for (int i = 0; i < nparms; i++){
          fprintf (logfile,"parameter  ::%6f\n",newpop[popsize].x[i]);
          }
            fprintf(logfile,"gen:%d Best fitness::%f\n",cur_genb,best);
            //printf("gen:%d Best  fitness::%f   \n",cur_genb,best);

            //printf("gen:%d       Best  fitness max::%f \n",gen_maxmax,maxmax);
            //printf("gen:%d       Best  fitness min::%f  \n",gen_maxmin,maxmin);
            //printf("gen:%d       Worst fitness max::%f          \n",gen_minmax,minmax);
            //printf("gen:%d       Worst fitness min::%f\n\n",gen_minmin,minmin);
}
```

Statistics.h : contains the routine statistics(), which calculates population statistics for each generation.

```
void statistics(int popsize,float &max, float &avg, float &min,float &sumfitness,
population pop)
//Calculate population statistics
{
          int i,j,k,mem;
  //initialize
  sumfitness=pop[0].fitness;
  min=pop[0].fitness;
  max=pop[0].fitness;
  //Loop for max, min, sumfitness
  for(j=1;j<popsize;j++){
             sumfitness=sumfitness+pop[j].fitness; //Accumulate fitness sum
             //printf("%f  %f\n",pop[j].fitness,avg);
             if(pop[j].fitness>max){
```

```
            max=pop[j].fitness; //New max
        }

        if(pop[j].fitness<min){
            min=pop[j].fitness; //New min
        }
        /*
        if (pop[j].fitness > best){
            cur_genb=gen;//best fitness found generation
            cur_best = j;//best fitness found population
            //worst=min;//keep minimum fitness for best fistness
            pop[popsize].fitness = pop[j].fitness;
            best=pop[j].fitness;//keep best;
            for (i = 0; i < nparms; i++)
           pop[popsize].x[i] = pop[cur_best].x[i];//keep best fitness parameters
            }*/
}
//Calculate average
           avg=sumfitness/popsize;
           maxsarray[gen]=max;
           minsarray[gen]=min;
        if(maxsarray[gen]<maxmin){
            maxmin=maxsarray[gen];
            gen_maxmin=gen;
        }
        if(maxsarray[gen]>maxmax){
            maxmax=maxsarray[gen];
            gen_maxmax=gen;
        }
        if(minsarray[gen]>minmax){
            minmax=minsarray[gen];
```

```
                gen_minmax=gen;
        }
        if(minsarray[gen]<minmin){
                minmin=minsarray[gen];
                gen_minmin=gen;
        }
}
```

Main.cpp **:** contains the main SGA program loop, main().

```
// Main program of  Genetic Algorithm
#include "definitions.h"
#include "random.h"
#include "objectfunction.h"
#include "parameters.h"
#include "statistics.h"
#include "initialize.h"
#include "reports.h"
#include "kernel.h"
#include "generate.h"

void main(){
        if ((logfile = fopen("galog.txt","w"))==NULL){
                exit(1);
    }
        gen=0;   // Set things up
        initialize();
        maxmax=maxsarray[0];
        maxmin=maxsarray[0];
        minmax=minsarray[0];
        minmin=minsarray[0];
```

```
do{
gen++;
generation();
statistics(popsize,max,avg,min,sumfitness,newpop);
report(gen);
for(int i=0;i<popsize;i++)
oldpop[i]=newpop[i]; //advance the generation
}while(gen<maxgen);

printf("maxmax:%f    gen_max:%d\n",maxmax,gen_maxmax);
printf("maxmin:%f    gen_min:%d\n",maxmin,gen_maxmin);
printf("minmax:%f    gen_max:%d\n",minmax,gen_minmax);
printf("minmin:%f    gen_min:%d\n",minmin,gen_minmin);

getch();
}
```