GENETIC ALGORITHMS WORKING IN DYNAMIC ENVIRONMENTS

GENETIC ALGORITHMS WORKING IN DYNAMIC ENVIRONMENTS

By

BIEKEZHATI DILIMULATI, B. Sc.

A Thesis
Submitted to the School of Graduate Studies
in Partial Fulfillment of the Requirements
for the Degree
Master of Science

McMaster University

MASTER OF SCIENCE (2006)	McMaster University
(Computer Science)	Hamilton, Ontario

TITLE:	Genetic Algorithms Working in Dynamic Environments
AUTHOR:	Biekezhati Dilimulati, B.Sc. (Xinjiang University, China)
SUPERVISOR:	Dr. Ivan Bruha
NUMBER OF PAGES:	x, 72

ABSTRACT

Genetic Algorithms (GAs) are search methods based on principles of natural selection and genetics. GAs attempt to find good solutions to the problem at hand by manipulating a population of candidate solutions.

Each member of the population is typically represented by a single chromosome, the chromosome encodes a solution to the problem, the initial population is generated randomly, GAs are often used as optimizers, and the fitness of an individual is typically the value of the objective function at the point represented by the chromosome. The individuals with better performance are selected as parents of the next generation. GAs create new individuals using simple randomized operators that resemble crossover and mutation in natural organisms. The new solutions are evaluated with the fitness function, and the cycle of selection, recombination, and mutation is repeated until a user defined termination criterion is satisfied.

In the real world, we always encounter the problems that need to be solved in a changing environment. This means that our algorithm needs to be dynamic or even adaptive to the changing environment.

In this thesis, we will mainly deal with the adaptive GAs that have a new genetic operator called *transformation* instead of traditional crossover.

In our study, we use a dynamic problem generator to create a dynamically changing landscape and study the behavior of transformation based GA in different parameter settings, such as: transformation rate, mutation rate, segment replacement rate.

ACKNOWLEDGMENTS

I would like to give my sincere thanks to my supervisor, Dr. Ivan Bruha, for his supervision and support in the accomplishment of this work. Thanks also go to the Department of Computing and Software for their support.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Genetic Algorithms (GAs) are mainly used to solve optimization problems, where optimization is defined as the process of adjusting the inputs to or characteristics of a device, mathematical process, or experiment to find the minimum or maximum output (result) [19]. The input consists of variables; the process or function is known as the cost function, objective function, or fitness function; and the output is the cost or fitness. If output is the cost then optimization becomes minimization, if output is the fitness then optimization becomes maximization.

Searching the cost surface (all possible function values is also called *fitness landscape*) for the minimum cost is the most common problem in optimization routines. There are different kinds of optimization methods. Before discussing the methodology of GA, we give a brief introduction to these optimization methods.

1.1. Exhaustive Search

The brute force approach to optimization looks at a sufficiently fine sampling of the cost function to find the global minimum [12]. This exhaustive search requires an extremely large number of cost function evaluations to find the optimum. For example, consider finding the minimum of the function:

$$f(x,y)=2\,x\sin(5x)+y\sin(3y) \tag{1.1}$$

where $x\in[0,10]$ and $y\in[0,10]$

With sampling fine enough, exhaustive searches do not get stuck in local minima and work for either continuous or discontinuous variables. However, they take an extremely long time to find the optimal global. A three dimensional plot and contour plot of (1.1) are given in Figure 1.1 and Figure 1.2.

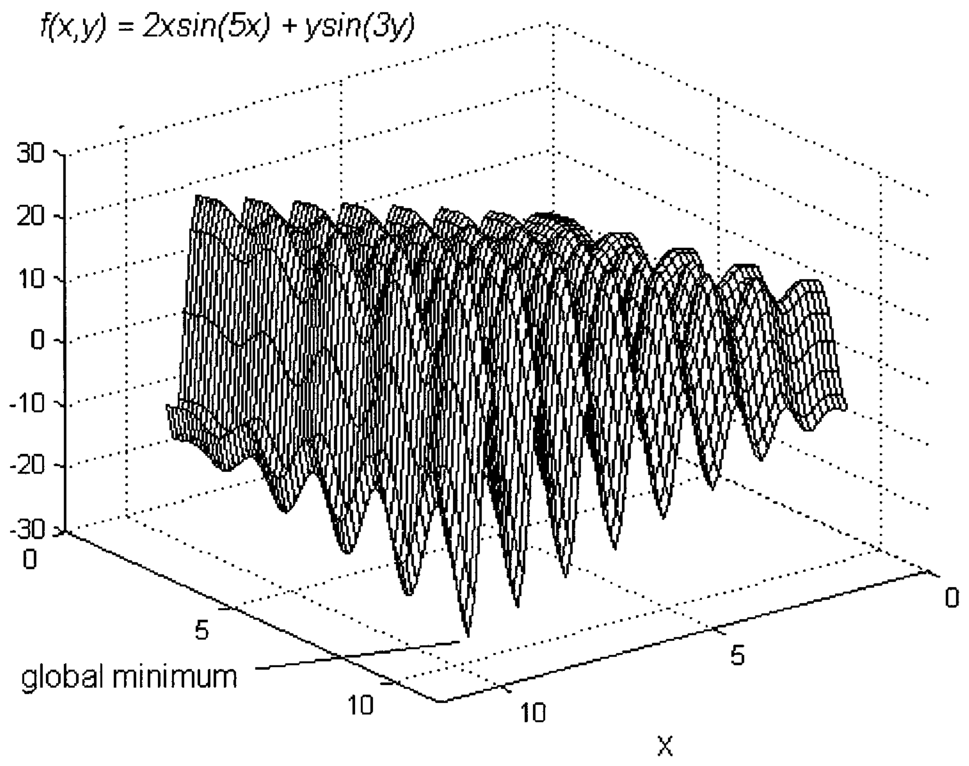

Figure 1.1 Three-dimensional plot of $f(x,y)$=2 x sin(5x) + y sin(3y)

We want to optimize the function (1.1) with some required precision: suppose three decimal places for the variables' values are desirable. To achieve such precision each domain X = $\{x| x \in[0,10]\}$ and Y = $\{y| y \in[0,10]\}$ should be cut into $(10 - 0)\cdot 10^3$ equal size ranges, so it needs $10^4 \cdot 10^4 = 10^8$ functional evaluations. The minimum is -29.4359 on point (9.743, 9.959); it took 47.54 seconds on a Pentium4 2.8G, 512MB computer to find out this minimum; apparently this is too much work for such a simple problem.

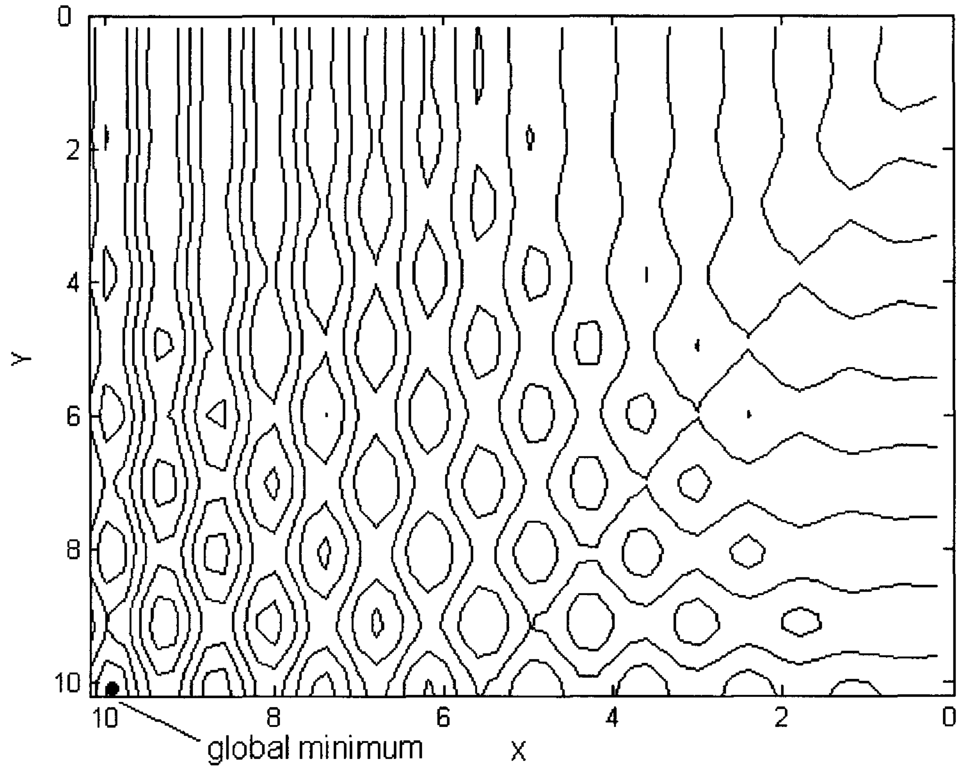

Figure 1.2 Contour plot of *f*(*x*,*y*)=2 *x* sin(5*x*) + *y* sin(3*y*)

1.2. Analytical Optimization

This is the calculus-based optimization method; the search process can be simplified to a single variable for a moment, and then an extremum is found by setting the first derivative of a cost function to zero and solving for the variable value [12]. If the second derivative is greater than zero, the extremum is a minimum, and conversely, if the second derivative is less than zero, the extremum is a maximum. One way to find the extrema of a function of two or more variables is to take the gradient of the function and set it equal to zero, $\nabla f(x,y)=0$. For example, taking the gradient of equation (1.1) resulted in

$$\frac{\partial f}{\partial x} = 2\sin(5x) + 10x\cos(5x) = 0, \qquad 0 \le x \le 10 \tag{1.2}$$

and

$$\frac{\partial f}{\partial y} = \sin(3y) + 3y\cos(3y) = 0, \qquad 0 \le y \le 10 \tag{1.3}$$

Next these equations are solved for their, x and y which is a family of lines. Extrema occur at the intersection of these lines. Finally, the Laplacian of the function is calculated.

$$\frac{\partial^2 f}{\partial x^2} = 20\cos 5x - 50x\sin 5x, \qquad 0 \le x \le 10 \tag{1.4}$$

and

$$\frac{\partial^2 f}{\partial y^2} = 6\cos 3y - 9y\sin 3y, \qquad 0 \le y \le 10 \tag{1.5}$$

The roots are minima when $\nabla^2 f(x,y) > 0$. Unfortunately, this process does not give a clue as to which of the minima is a global minimum. This approach is mathematically elegant compared to the exhaustive or random searches. But it requires continuous functions with analytical derivatives.

1.3. Line Minimization Methods

This algorithm begins at some random point on the cost surface, chooses a direction to move, then moves in that direction until the cost function begins to increase. Next the procedure is repeated in another direction [12].

A very simple approach to line minimization is the coordinate search method. It starts at an arbitrary point on the cost surface, and then does a line minimization along the axis of one of the variables. Next, it selects another variable and performs another line minimization along that axis. This process continues until a line minimization is carried out along each of the variables. Figure 1.3 shows the possible path that this algorithm might take on quadratic cost surface. In general this method is slow. Figure 1.4 shows the flowchart of typical line search algorithms.

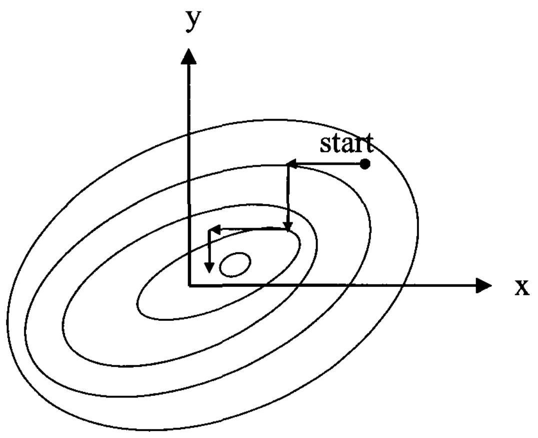

Figure 1.3 Possible path that the coordinate search might take on a quadratic cost surface

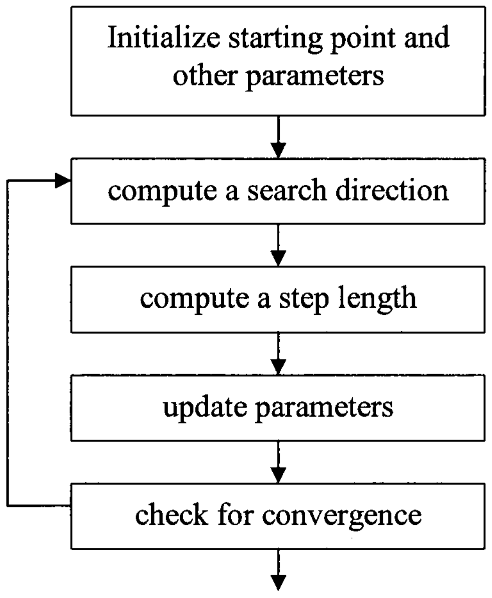
Initialize starting point and other parameters
compute a search direction
compute a step length
update parameters
check for convergence

Figure 1.4 Flowchart for a typical line search algorithm

1.4 Natural Optimization Methods

Natural optimization methods generate new points in the search space by applying operators to current points and statistically moving toward more optimal places in the search space [12]. They rely on an intelligent search of a large but finite solution space using statistical methods. Here we briefly introduce some natural optimization methods:

1.4.1 Simulated Annealing

This method simulates the annealing process in which a substance is heated above its melting temperature and then gradually cooled to produce the crystalline lattice, which minimizes its energy probability distribution. This crystalline lattice, composed of millions of atoms perfectly aligned, is a beautiful example of nature finding an optimal structure. However, quickly cooling or quenching the liquid retards the crystal formation, and the substance becomes an amorphous mass with a higher than optimum energy state. The key to crystal formation is carefully controlling the rate of change of temperature.

The algorithmic analogue to this process begins with a random guess of the cost function variable values. Heating means randomly modifying the variable values. Higher heat implies greater random fluctuations. The cost function returns the output associated with a set of variables. If the output decreases, then the new variable set replaces the old variable set. If the output increases, then the output is accepted provided that

$$r \le e^{\frac{f(p_{old})-f(p_{new})}{T}} \tag{1.6}$$

where r is a uniform random number (a random number in the interval (0,1)), T is a variable analogous to temperature, f is the cost function, p_{old} is old variable set, and p_{new} is new variable set. Otherwise the new variable set is rejected. Thus, even if a variable set leads to a worse cost, it can be accepted with a certain probability. The new variable set is found by taking a random step from the old variable set

$$p_{new} = d^* p_{old} \tag{1.7}$$

The variable d is either uniformly or normally distributed about p_{old}. This control variable sets the step size so that, at the beginning of the process, the algorithm is forced to make large changes in variable values. At times the changes move the algorithm away from the optimum, which forces the algorithm to explore new regions of variable space. After a certain number of iterations, the new variable sets no longer lead to lower costs. At this point, the values of T and d decrease by a certain percent and the algorithm repeats. The algorithm stops when $T \approx 0$. The decrease in T is known as the cooling schedule. The temperature is usually lowered slowly so that the algorithm has a chance to find the correct valley before trying to get to the lowest point in the valley.

1.4.2 Ant Colony Optimization (ACO)

In the real world, ants (initially) wander randomly, and upon finding food they return to their colony while laying down pheromone (chemical) trails. If other ants find such a path, they are likely not to keep traveling at random, but to instead follow the trail, returning and reinforcing it if they eventually find food.

Over time, however, the pheromone trail starts to evaporate, thus reducing its attractive strength. The more time it takes for an ant to travel down the path and back again, the more time the pheromones have to evaporate. A short path, by comparison, gets marched over faster, and thus the pheromone density remains high as it is laid on the path as fast as it can evaporate.

Thus, when one ant finds a good (i.e., short) path from the colony to a food source, other ants are more likely to follow that path, and positive feedback eventually leaves all the ants following a single path. The idea of the ant colony algorithm is to mimic this behaviour with "simulated ants" walking around the graph representing the problem to solve.

Ant colony optimization algorithms have been used to produce near-optimal solutions to the traveling salesman problem.

1.5. Genetic Algorithm

Genetic Algorithms (GAs) are search methods based on principles of natural selection and genetics. GAs attempt to find good solutions to the problem at hand by manipulating a population of candidate solutions.

Each member of the population is typically represented by a single *chromosome*-like data structure, which can be as simple as a string of zeroes and ones, or as complex as a computer program. The chromosome encodes a solution to the problem, in GAs chromosomes are also called *strings*, *individuals,* or *objects*. The initial population of individuals or set of solutions is generated randomly, unless some good solutions are known, or there is a heuristic to generate good solutions to the problem. In the latter case, a portion of the population is still generated randomly to ensure that there is some diversity in the population.

The individuals are evaluated to determine how well they solve the problem with an *evaluation function* (*objective function* or *fitness function*), which is unique to each problem, and must be supplied by the user of the algorithm. In particular, GAs are often used as optimizers, and the fitness of an individual is typically the value of the objective function at the point represented by the chromosome. The individuals with better performance (the individuals that have higher fitness value or lower cost value) are selected as parents of the next generation. GAs create new individuals by using simple randomized operators that resemble crossover and mutation in natural organisms. The new solutions are evaluated with the evaluation function, and the cycle of selection, recombination, and mutation is repeated until a user defined termination criterion is satisfied.

GAs are controlled by several inputs, such as the size of the population, and the rates that control how often crossover and mutation are used. GAs are not guaranteed to converge to the optimal solution, but a careful manipulation of the input parameters (or operators) and choosing a representation that is adequate to the problem increase the chances of success. In simple GAs, the operator set is usually fixed, but in a real world, we always encounter problems that need to be solved in a changing environment. That means our algorithm has to be dynamic or even adaptive to the changing environment.

There are different ways of making Genetic algorithms dynamic or adaptive to its environment; one alternative is to let genetic operators (such as crossover, mutation) be free; an other alternative is to use variable-length individuals, with more complicated strings than binary strings and variable population size.

In this thesis we will mainly deal with the adaptive GAs that have new genetic operator called *transformation* [24] instead of the traditional crossover. In our study we use a *dynamic problem generator* [21] to create a dynamically changing landscape and study the behavior of a transformation-based GA in different parameter settings, such as: transformation rate, mutation rate.

CHAPTER2

GENETIC ALGORITHMS

2.1 Introduction to Genetic Algorithm

The concept of Genetic Algorithms (GAs) has been developed by John Holland [14], and his colleagues. GA is an optimization and search technique based on the principles of genetics and natural selection. A GA encodes a potential solution to a specific problem on a simple chromosome-like data structure; this data structure is known as a *chromosome* or *individual.* At first, a GA randomly creates a population of individuals (potential solutions) and GA operators are applied to these individuals to find the best solution or evolve to a state that maximizes the "fitness" of the individuals (i.e., minimizes the cost function).

GAs use a vocabulary borrowed from natural genetics. So we need a bit of Biological background on heredity at the cellular level. A *gene* is the basic unit of heredity. An organism's genes are carried on one of a pair of *chromosomes* in the form of deoxyribonucleic acid (DNA). Each cell of the organism contains the same number of chromosomes. Genes often occur with two functional forms, each representing a different characteristic. Each of these forms is known as *allele.* For instance, a human may carry one allele for brown eyes and another for blue eyes. The combination of alleles on the chromosomes determines the traits of the individual. The trait actually observed is the *phenotype*, but the actual combination of alleles is the *genotype.*

We thus talk about *individuals* (*chromosomes, genotypes, strings*) in a population, and *genes* (*features, characters*).

Each chromosome represents a potential solution to problem (the meaning of a particular chromosome, i.e., its phenotype, is defined externally by the user); an evolution process run on a population of chromosomes corresponds to a search through a space of potential solutions.

The population undergoes a simulated evolution: at each generation the relatively "good" solutions reproduce, while the relatively "bad" solutions die. To distinguish between different solutions we use an *objective* (*evaluation*) function which plays the role of an environment [19]. Usually, the evaluation function is called a *fitness function* if the individuals with a maximum evaluation is desired, or called a *cost function* if the individuals with a minimum evaluation is desired. We can easily change a maximizing problem into a minimizing problem by simply changing the sign of the evaluation function.

2.2 The Advantages of Genetic Algorithms

Genetic Algorithms are a class of general purpose (domain independent) search methods which strike a remarkable balance between exploration and exploitation of the search space. GAs can be used in image processing, numerical function optimization, combinatorial optimization, and machine learning [4], [5]. GAs outperform other searching optimization methods when solving very complex problems such as combinatorial optimization problems and highly constrained engineering problems. GAs belong to the class of probabilistic algorithms, yet they are very different from random algorithms as they combine elements of directed and stochastic search. Because of this, GAs are also more robust than existing directed search methods.

Some of the advantages of GA are [12]:

- Optimizes variables with extremely complex cost surfaces(they can jump out of a local minimum)
- Optimizes with continuous or discrete variables
- Does not require derivative information
- Simultaneously searches from a wide sampling of the cost surface
- Deals with a large number of variables
- Provides a list of optimum variables, not just a single solution
- Works with numerically generated data, experimental data, or analytical functions.

GAs have been quite successfully applied to optimization problems like wire routing, scheduling, adaptive control, game playing, cognitive modeling, transportation problems, traveling salesman problems, optimal control problems, etc.

2.3 A Simple Genetic Algorithm

A genetic algorithm for a particular problem must have the following five components:

- A genetic representation for potential solutions to the problem,

- A way to create an initial population of potential solutions,
- An evaluation (fitness) function that rates the solutions in terms of their "fitness"
- Genetic operators that alter the composition of children,
- Values for various parameters that the genetic algorithm uses (population size, crossover rate, mutation rate, etc.)

The process of a simple genetic algorithm is shown as a flowchart in Figure 2.1. We explain every process in this flowchart with an implementation of a simple genetic algorithm.

Since a genetic algorithm originated in binary coding and is easy to implement, let us take a binary genetic algorithm for example. Suppose we have a string of length of eight that consists of 1's and 0's, and we want it to have most 1's in it.

2.2.1 Representation

To solve this problem as a GA, we first consider the representation of the potential solution; from the definition of our problem it is clear that an eight bit binary string is a perfect representation of the solutions.

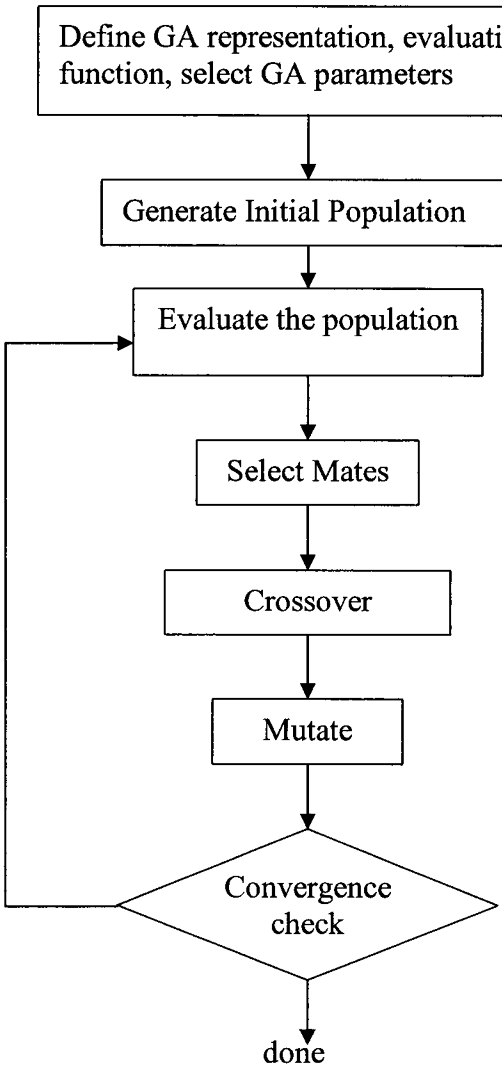
Define GA representation, evaluation function, select GA parameters
Generate Initial Population
Evaluate the population
Select Mates
Crossover
Mutate
Convergence check
done

Figure 2.1 Flowchart of a simple GA

2.2.2 Evaluation function

Since we want the string to have most 1's, we can define the evaluation function as the number of 1's in the string, (*Note*: the value of this function is not the value of the binary number, for this evaluation function $f(10000000) = f(00000001) = 1$, it does not care about the position of 1's).

2.2.3 Initial Population

The initialization process is very simple: we create a population of chromosomes, where each chromosome is a binary vector of eight bits. All eight bits for each chromosome are initialized randomly. In this example we set population at four, and we keep the population size fixed, during the whole process. Our initial population is:

s_1= (1 1 0 0 1 0 1 0)

s_2= (0 1 0 0 0 0 1 1)

s_3= (0 1 1 1 0 1 0 1)

s_4= (0 0 1 0 0 0 1 0)

The evaluation function *eval*( ) evaluates them as following:

eval(s_1)=f(1 1 0 0 1 0 1 0) = 4

eval(s_2)=f(0 1 0 0 0 0 1 1) = 3

eval(s_3)=f(0 1 1 1 0 1 0 1) = 5

eval(s_4)=f(0 0 1 0 0 0 1 0) = 2

The chromosome s_3 is the best of the four chromosomes since it has the highest fitness value among them.

2.2.4 Selection

In this step we choose better individuals (usually we set a number N_{keep} as the number of individuals to be chosen) for reproduction, so hopefully they will produce even better offspring, the new offspring will replace the "unfit" individuals.

There are many selection methods [12]:

1. *Pairing from top to bottom*

   Sort the individuals according to their fitness and start at the top of the list and pair the chromosomes two at a time. So the first one mates with second one, third one mates with fourth one, and so on. In this example the top to down list is s_3, s_1, s_2, s_4, if we set N_{keep}=2, then s_3, s_1 will be chosen for reproduction.

2. *Random Paring*

   This approach uses a uniform random number generator to select chromosomes. Parent chromosomes can be selected as:

$$ma = \lceil N_{keep} * rand(\,) \rceil \tag{2.1}$$

$$pa = \lceil N_{keep} * rand(\,) \rceil \tag{2.2}$$

   where *rand*( ) is the random number generator, and $\lceil\ \rceil$ is the ceiling operator. This method is very easy to apply, but it is not fast due to the uniform random selection.

3. *Tournament selection*

   Randomly picks a small subset of chromosomes and the chromosome with the highest fitness becomes a parent. This method is good for large population size, where we can choose tournament size according to our needs; if the tournament size is bigger, weak individuals have a smaller chance to be selected. In our example if we divide the population into two subsets, $\{s_1, s_2\}$, and $\{s_3, s_4\}$, then the best chromosomes s_1 and s_3 in these two subsets will be chosen as parents.

4. *Roulette wheel weighting*

The probabilities assigned to the chromosomes are proportional to their fitness (or inversely proportional to the their cost) . A random number determines which chromosome is to be selected.

Roulette wheel weighting and tournament selection are used in most GAs, in this example we use roulette wheel weighting.

We construct roulette wheel as follows (see Table 2.1):

- Calculate the fitness value $f(s_i)$ for each chromosome s_i (i=1, 2, ... *pop_size*), where *pop_size* is the number chromosomes in the population (third column in Table 2.1).
- Find the total fitness of the population (third column in Table 2.1).

$$F = \sum_{i=1}^{pop\_size} f(s_i) \tag{2.3}$$

- Calculate the probability of a selection p_i for each chromosome s_i (i=1, 2, ... *pop_size*) (fourth column in Table 2.1):

$$p_i = f(s_i)/F \tag{2.4}$$

- Calculate a cumulative probability q_i for each chromosome s_i (i=1, 2, ... *pop_size*) (fifth column in Table 2.1):

$$q_i = \sum_{j=1}^{i} p_j \tag{2.5}$$

Table 2.1 demonstrates the construction of a roulette wheel table.

The selection process is based on spinning the roulette wheel *pop_size* times; each time we select a single chromosome for a new population in the following way:

- Generate a uniform random (float) number r from the range [0..1]
- If $r<q_1$ then select the first chromosome (s_1); otherwise select the *i-th* chromosome s_i ($2\leq i \leq pop\_size$) such that $q_{i-1}< r <q_i$.

Table 2.1 Constructing roulette wheel table

Chrom. No.	Initial Population (Randomly Generated)	Fitness value $f(s_i)$	$p_i = f(s_i)/F$	$q_i = \sum_{j=1}^{i} p_j$
1	1 1 0 0 1 0 1 0	4	0.308	0.308
2	0 1 0 0 0 0 1 1	3	0.231	0.539
3	0 1 1 1 0 1 0 1	5	0.384	0.923
4	0 0 1 0 0 0 0 0	1	0.077	1.000
Sum		13	1.000	
Max		5	0.385	
Average		3.25	0.25	

An example of roulette wheel selection is shown in Table 2.2. The roulette wheel weighting of chromosomes is shown in Figure 2.2.

Table 2.2 Roulette wheel weighting selection

random number r	range	selected chromosome
0.726	$q_2\leq r < q_3$	s_3
0.157	$r < q_1$	s_1
0.869	$q_2\leq r < q_3$	s_3
0.324	$q_1\leq r < q_2$	s_2

As a result we get s_1,s_2, s_3 as parents, among them s_3 is chosen twice.

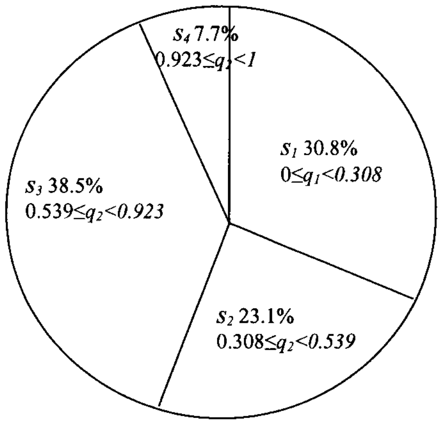

Figure 2.2 Roulette wheel weighting

From this figure we can see that the higher the fitness of a chromosome, the higher the chance of being selected for mating.

2.2.5 Crossover

Crossover is the creation of one or more offspring from the parents selected in the paring process. In single point crossover, a crossover point is selected randomly between the first and last bits of the parent's chromosomes. Table 2.3 shows the pairing and crossover process for the problem at hand, where " | " is the crossover point.

Table 2.3 Process of single point crossover

Parent Chromosome	Crossover	Offspring
s_1	*1 1 0 0 1*|*0 1 0*	*1 1 0 0 1* **1 0 1**
s_3	**0 1 1 1 0|1 0 1**	**0 1 1 1 0** *0 1 0*
s_2	*0 1 0 0 0 0*| *1 1*	*0 1 0 0 0 0* **0 1**
s_3	**0 1 1 1 0 1| 0 1**	**0 1 1 1 0 1** *1 1*

First parent passes its binary code to the left of that crossover point to first offspring. In a like manner, second parent passes its binary code to the left of the same crossover point to second offspring. Next, the binary code of the right of the crossover point of first parent goes to second offspring and second parent passes its code to first offspring. Consequently the offspring contain portions of the binary codes of both parents.

2.2.6 Mutations

Mutation is performed on a bit-by-bit basis. It randomly alters the bits in the chromosome with the small probability p_m. Every bit (in all chromosomes in the whole population) has an equal chance to undergo mutation, i.e., change from 0 to 1 or vice versa. Increasing the number of mutations increases the algorithm's freedom to explore outside the current region of variable space. We proceed in the following way.

For each chromosome in the current population and for each bit within the chromosome:

- Generate a random number r from the range [0..1];
- If $r < p_m$, mutate the bit.

Let us set $p_m = 0.1$, now population size is 4, every chromosome has eight bits,

$0.1*4*8 \approx 3$, so about three bits will be mutated. The mutated bits in Table 2.4 are shown in bold, italics.

The other way of performing mutation is to think of a whole population as a two dimensional array $pop[N_{pop}, N_{bits}]$ (where N_{pop} is population size, N_{bits} is the length of each chromosome) in which each row represents a chromosome, and each column represents a gene. Thus a random number generator creates pairs of random integers (*mrow*, *mcol*) that correspond to the rows and columns of the mutated bits.

Table 2.4 Mutating the individuals (bit by bit mutation)

Population after mating	Population after mutation	New fitness value
1 1 0 0 1 1 0 1	1 1 *1* 0 1 1 0 1	6
0 1 1 1 0 0 1 0	0 1 1 1 0 0 *0* 0	3
0 1 0 0 0 0 0 1	0 1 0 *1* 0 0 0 1	3
0 1 1 1 0 1 1 1	0 1 1 1 0 1 1 1	6

We can use following computer code to find the rows and columns of the mutated bits.

$nmut = \text{round}(N_{pop} * N_{bits} * p_m)$ // number of mutations

$nrow = \text{round}(\text{rand}(1, p_m) * N_{pop} + 1$ // row of the bits to be mutated

$ncol = \text{round}(\text{rand}(1, p_m) * N_{bits})$ //column of the bits to be mutated

$$nmut = \text{round}(N_{pop} * N_{bits} * p_m) = \text{round}(4*8*0.1) = 3 \tag{2.6}$$

So, mutations occur three times, the following pairs were randomly selected.

$nrow = [\ 2\ \ 3\ \ 4]$ $ncol = [\ 6\ \ 2\ \ 1]$

Table 2.5 Mutating the individuals (by generating row, column pairs)

Population after mating	Population after mutation	New fitness value
1 1 0 0 1 1 0 1	1 1 0 0 1 1 0 1	5
0 1 1 1 0 0 1 0	0 1 1 1 0 *1* 1 0	5
0 1 0 0 0 0 0 1	0 *0* 0 0 0 0 0 1	1
0 1 1 1 0 1 1 1	*1* 1 1 1 0 1 1 1	7

2.2.7 The Next Generation

After the mutation takes place, the cost associated with the offspring and mutated chromosomes are calculated (third column in Table 2.4 and Table 2.5). The process described is iterated. For our example, the population at the end of next generations is shown in Table 2.6, Table 2.7, Table 2.8, and Table 2.9 (We took the population in Table 2.5 as the starting population of the second generation).

Table 2.6 Population at the end of second generation

Population after mating	Population after mutation	New fitness value
1 1 1 1 0 1 1 1	1 1 1 1 0 1 1 1	7
1 1 0 0 1 1 0 1	1 1 ***1*** 0 1 1 0 1	5
0 1 1 1 0 1 1 1	0 1 1 ***0*** 0 1 1 1	5
1 1 1 1 0 1 1 0	1 1 1 1 0 1 1 ***1***	7

Table 2.7 Population at the end of third generation

Population after mating	Population after mutation	New fitness value
1 1 1 1 0 1 0 1	1 1 ***0*** 1 0 1 0 1	5
1 1 1 0 1 1 1 1	1 1 1 0 1 1 1 1	7
0 1 1 1 0 1 1 1	0 1 ***0*** 1 0 1 1 1	4
1 1 1 0 0 1 1 1	1 1 1 0 ***1*** 1 1 1	7

Table 2.8 Population at the end of fourth generation

Population after mating	Population after mutation	New fitness value
1 1 0 1 1 1 1 1	1 1 0 1 1 1 1 1	7
1 1 1 0 0 1 0 1	1 1 1 0 ***1*** 1 0 1	6
0 1 0 0 1 1 1 1	0 1 ***1*** 0 1 1 1 1	5
1 1 1 1 0 1 1 1	1 1 1 1 0 1 ***0*** 1	6

2.2.8 Convergence

The number of generations evolve depends on whether an acceptable solution is reached or a set number of iterations is exceeded. After a while all the chromosomes and associated costs would become the same if it were not for mutations. At this point the algorithm should be stopped.

Most genetic algorithms keep track of the population statistics in the form of a population mean and minimum cost. As shown in Table 2.9, for our example after five generations the global maximum fitness value is found to be 8. This maximum fitness was found in

$$\underset{\substack{\text{fitness evaluation}\\ \text{per generation}}}{4} \quad * \quad \underset{\text{generations}}{5} = 20 \tag{2.7}$$

fitness function evaluations or checking 20/(256*256) = 0.03% of the population. Figure 2.3 shows a plot of the algorithm convergence in terms of the best and average fitness of each generation.

Table 2.9 Population at the end of fifth generation

Population after mating	Population after mutation	New fitness value
1 1 0 0 1 1 0 1	1 1 0 1 1 1 1 1	5
1 1 1 1 1 1 1 1	1 1 1 0 *1* 1 0 1	8
0 1 1 1 0 1 0 1	0 1 *1* 0 1 1 1 1	5
1 1 1 0 1 1 1 1	1 1 1 1 0 1 *0* 1	7

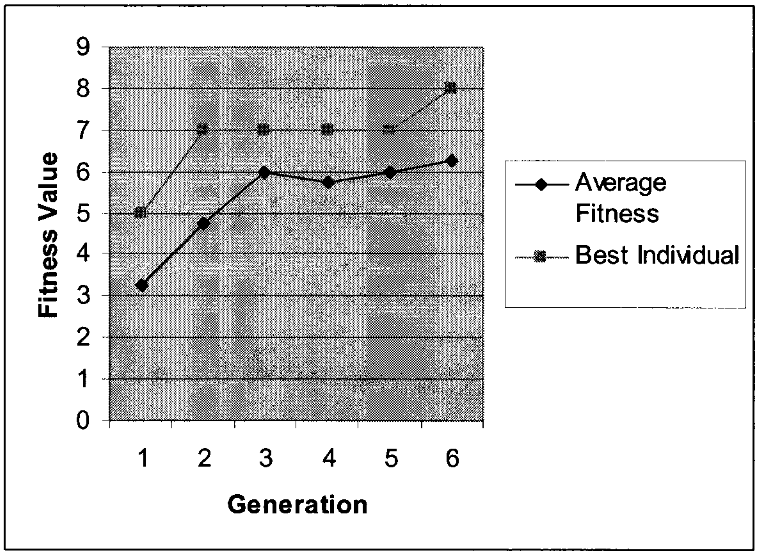

Figure 2.3 Graph of average and best fitness value of each generation.

From figure 2.3 we can see that at the end of the fifth generation, the algorithm is able find the best fitness value of 8. The average fitness value is also increased from initial value of 3.25 to final value of 6.25.

CHAPTER 3

GENETIC ALGORITHMS IN DYNAMIC ENVIRONMENTS

The genetic algorithms we have discussed above, work in stationary environments, which means that the fitness function does not change during the evolution process; a problem with a fitness (objective) function that changes over time is referred to as having a "dynamic fitness landscape". A variety of engineering, economic, and information technology problems require systems that adapt to changes over time. Examples of problems where environmental changes could cause the fitness landscape to be dynamic include: target recognition, where the sensor performance varies based on environmental conditions; scheduling problems, where available resources vary over time; financial trading models, where market conditions can change abruptly; investment portfolio evaluation, where the assessment of investment risk varies over time; and data mining, where the contents of the database are continuously updated. These types of problems may experience simple dynamics, where the fitness peaks that represent the optimal problem solution drift slowly from one value to the next, or complicated dynamics, where the fitness peaks change more dramatically, with current peaks being destroyed and new, remote peaks arising from valleys.

3.1 Previous Research

In recent years there has been significant research in making genetic algorithms work efficiently in dynamic environments [6], [10], [11], [7], [3], [18], most of this research could be grouped into one of these categories:

1. *Increasing diversity after change*

   The GA is run in a standard fashion, but as soon as a change in the environment has been detected, explicit actions are taken to increase diversity and thus to facilitate the shift to the new optimum. Cobb [6] has proposed a simple adaptive mutation mechanism called *triggered hypermutation* to deal with continuously changing environments. Cobb's approach is to monitor the quality of the best performers in the population over time. When this measure declines, it is a plausible indicator that the environment has changed. Hypermutation then essentially restarts the search from scratch. The most attractive feature of this approach is that it is adaptive; for example, it emulates a standard GA in a stationary environment.

2. *Maintaining diversity throughout the run*

   Convergence is avoided all the time and it is hoped that a spread-out population can adapt to changes more easily. Grefenstette [10] introduced the method of random immigrants where in every generation, the population is partly replaced by randomly generated individuals in every generation. As opposed to strong mutations, random immigrants only affect part of the population. Thus it introduces diversity without disrupting the ongoing search process.

3. *Memory based approaches*

   The GA is supplied with a memory to be able to recall useful information from past generations, which seems especially useful when the optimum repeatedly returns to previous locations. Obviously, strategies with a memory may be especially beneficial in periodically changing environments, when there are repeated occurrences of a small set of situations. Additionally, redundant representations may slow down convergence and favor diversity, memory may be provided in two general ways: implicitly by using redundant representations, or explicitly by introducing an extra memory and formulating strategies to store in and retrieve solutions from it.

4. *Multi-population Approaches*

Multiple subpopulations are used, some to track known local optima, some to search for new optima. A general problem with memory is that the stored information, like the location of peaks found, becomes obsolete as the environment changes. One possibility to reduce this problem is to maintain small subpopulations in several promising areas of the search space which can track the peaks as they move and change, thus acting as self-adaptive memory. Morrison [20] introduced the concept of using *sentinels*, which have the following definition and attributes:

- Sentinels constitute a subset of the population that is uniformly distributed through the search space upon initialization.
- Sentinels are regular members of the population for selection and crossover operations but are stationary and are not, themselves, replaced or mutated.

By using sentinels to continuously sample the same points in the search space, we can guarantee that there are always some individuals that spread evenly throughout the search space. Whenever a fitness landscape shift occurs, the part of the population that has started to converge near a found peak may suddenly find itself at lower fitness. At this point, sentinels in other parts of the search space will get an increased opportunity to mate and create offspring that are spread throughout the search space, immediately increasing the dispersion of the population.

Since sentinels remain stationary in the search space across multiple generations, they can be used to provide a more informative picture of the dynamics of the fitness landscape. Specially, they are able to retain memory of previous fitness values at their search-space location. If they are provided with some limited ability to communicate among themselves, they could derive information about the type and extent of any detected fitness landscape changes. With this information, it may be possible to improve performance by altering the sentinel behavior.

Most of the adaptive GAs are trying to improve the diversity of the population so that the change in the fitness landscape can be detected by some individuals, importing random immigrants and placing sentinels are good examples of this.

3.2 Transformation-based Genetic Algorithm

Transformation [24] is a biologically inspired genetic operator that, when incorporated into the Genetic Algorithm can promote diversity in the population; in nature this operator occurs in colonies of bacteria.

Usually, transformation consists in the transfer of small pieces of cellular DNA between organisms. These pieces of DNA, or *gene segments* [24] are extracted from the environment and added to recipient cells.

After that, there are two possibilities, failure or success, known technically as *restriction* [24] and *recombination* [24]. Restriction is the destruction of the incoming foreign DNA, since those bacteria assume that foreign DNA is more likely to come from an enemy, such as a virus. In this case transformation fails. Recombination is the physical incorporation of some of the incoming DNA into the bacterial chromosome. If this happens, genes from the assimilated gene segment replace some of the host cell's genetic information and bacteria are permanently transformed. Once integrated in the chromosome, the DNA segment is able to survive.

In some ways transformation-based GA (TGA) has similarity with the random immigrants method, when using TGA, we start with an initial population of individuals and an initial *pool of gene segments* [24] (also called *gene segment pool* or *segment pool*, is a set of gene segments that are used in transformation of selected individuals; these gene segments act like foreign DNA pieces in bacterial transformation), both created at random. In each generation, we select individuals to be transformed and we apply transformation using the gene segments in the segment pool, then if necessary we also apply mutation. After that, the segment pool is updated using the individuals from the old population to create part of the new segments, and the rest of the segments are created at random; from the above description we can see that in TGA the crossover operator in standard GA is replaced by transformation operator. Figure 3.1 shows the flowchart of TGA.

After selecting individuals, we use the transformation mechanism to produce new individuals. To transform an individual we randomly select a segment from the segment pool, and also randomly choose a point of transformation in the selected individual. The segment is incorporated in the genome of the individual (chromosome), replacing the genes after the transformation point. It should be noted that the chromosome is seen as a circle. Figure 3.2 and Figure 3.3 illustrate this transformation mechanism.

The transformation process can be divided into the following steps:

1. *Select an individual*

   In this study we used the roulette wheel selection method to select an individual. Because of the randomness of the roulette wheel selection method, it is possible that one individual with a high fitness value can be selected several times. This method is one of the most common selection methods in GAs, the mechanism of roulette wheel selection method was discussed in chapter 2.

2. *Select a gene segment*

   In this step we randomly select a gene segment to replace the genes in the selected individual.

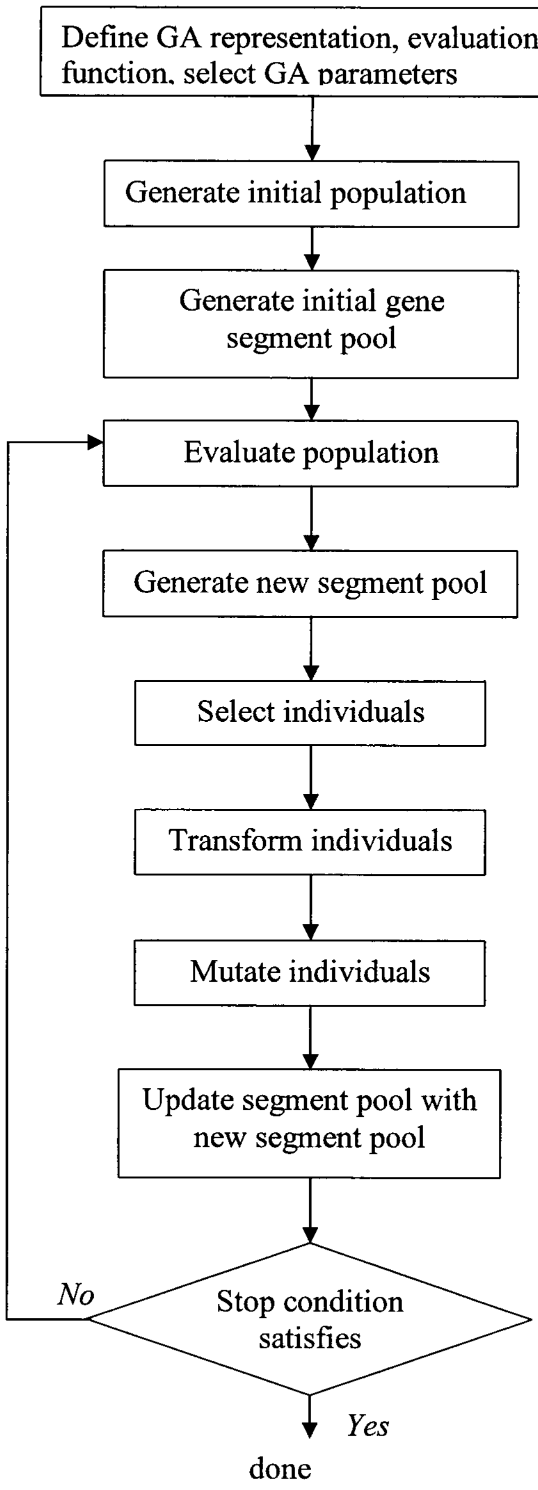
Define GA representation, evaluation function, select GA parameters
Generate initial population
Generate initial gene segment pool
Evaluate population
Generate new segment pool
Select individuals
Transform individuals
Mutate individuals
Update segment pool with new segment pool
Stop condition satisfies
No
Yes
done

Figure 3.1 Flowchart of transformation-based genetic algorithm

3. Choose transformation point

The transformation point (*TransPoint*) is also chosen randomly, which is an integer number between 0 and *LENGTH-1* (*LENGTH* is the number of genes in a chromosome). The number of genes in a segment (*SEGLEN*) is less than the number of genes in a chromosome (*LENGTH*).

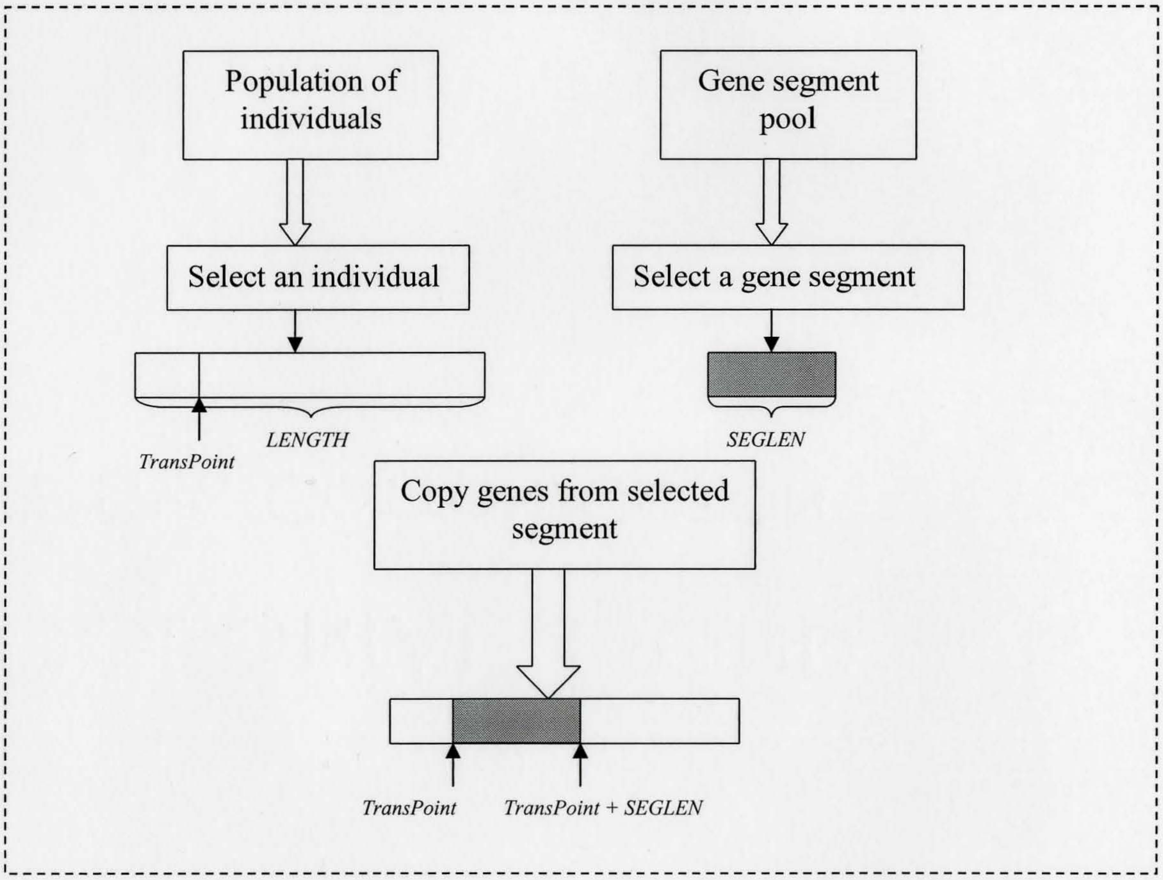

Figure 3.2 Transformation mechanism (gene segment lies in the middle of the chromosome)

4. Copy genes from selected segment

The transformation point is chosen randomly so we should consider two possible situations separately:

- If *LENGTH* ≥ *TransPoint* + *SEGLEN*, this means the genes to be replaced lie in the middle of the chromosome. In this case the genes in the gene segment are copied into the chromosome as a whole (see Figure 3.2).
- If *LENGTH* < *TransPoint* + *SEGLEN*, this means the genes to be replaced lie at the two ends of the chromosome. In this case the (*LENGTH* – *TransPoint*) genes at beginning of the gene segment are copied to the end of the chromosome, then, the rest of the (*SEGLEN* – (*LENGTH* – *TransPoint*)) genes are copied to the beginning of the chromosome (see Figure 3.3).

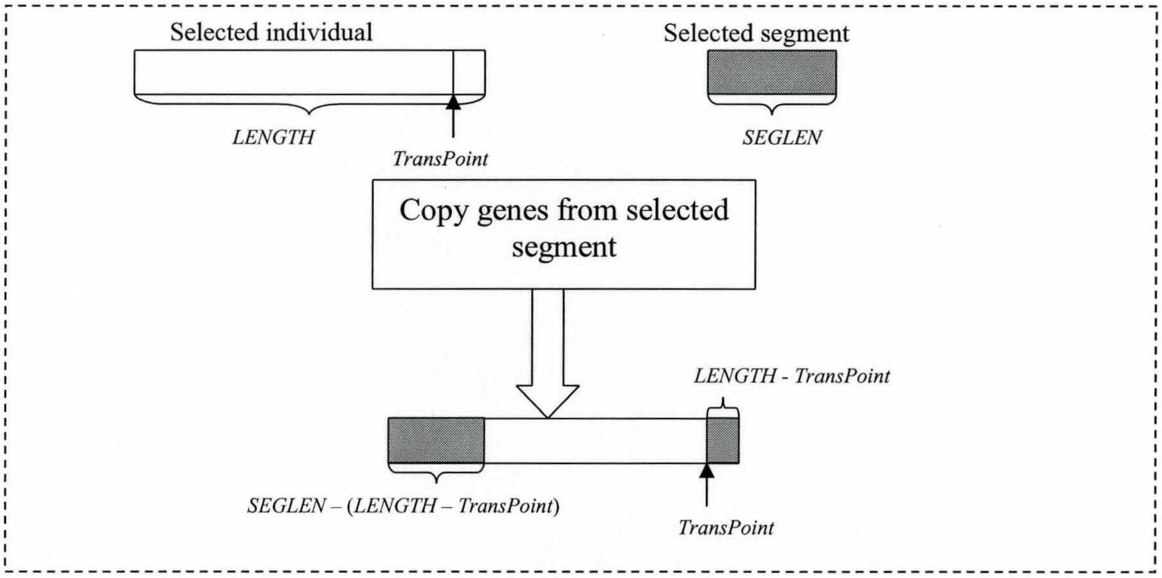

Figure 3.3 Transformation mechanism (gene segment lies in the two ends of the chromosome)

5. *Replace an old individual with the new one*

   The last step is to add this new individual into the current population. Because the population size is fixed, we must replace an old individual with this newly generated individual. In this study, we choose the worst individual (the individual that has the lowest fitness value) to be replaced.

The performance of TGA highly depends on the parameters' setting. The parameters include *transformation rate* (percentage of population to be transformed), *segment replacement rate* (the percentage of gene segments that are updated using the genetic information of the individual's of previous population), *segment length*, and *mutation rate*.

CHAPTER 4

DYNAMIC PROBLEM GENERATOR

Researchers who have studied the GAs in dynamic environments have developed different dynamic test functions; this study introduces the dynamic problem generator (DPG) that was developed by Morrison and De Jong [19]. The reason why we chose this dynamic problem generator is that it provides easy methods to reproduce a wide variety of interesting dynamic test problems for use in GA research.

The process of generating a dynamic problem can be divided into two steps: first construct the shape of the fitness landscape and then change the landscape according to the user specified settings.

4.1 Morphology of the Fitness Landscape

The basic morphology of the landscape is the "field of cones" of different heights and different slopes that are randomly scattered across the landscape [19]. The static function can be specified for any number of dimensions. In the 2-dimensional case we have:

$$f(X,Y) = \max_{i=1,N}[H_i - R_i * \sqrt{(X-X_i)^2 + (Y-Y_i)^2}] \tag{4.1}$$

where: N– specifies the number of cones in the environment,

(X_i, Y_i) – independently specifies the location of each cone,

H_i – the height of each cone,

R_i – the slope of each cone (tangent value of the base angle).

We choose a highest value for a given point using the *max* function, so that the cone that has a higher value "dominates" the cone that has a lower value at the given point. As we can see in Figure 4.1, there are two cones located at x_1, and x_2. For a given point x, we have two values $f_1(x)$, and $f_2(x)$; in this case $f_1(x) > f_2(x)$,

$$f(x) = max(f_1(x), f_2(x)) = f_1(x) \tag{4.2}$$

so we take $f_1(x)$ as a function value for the point x.

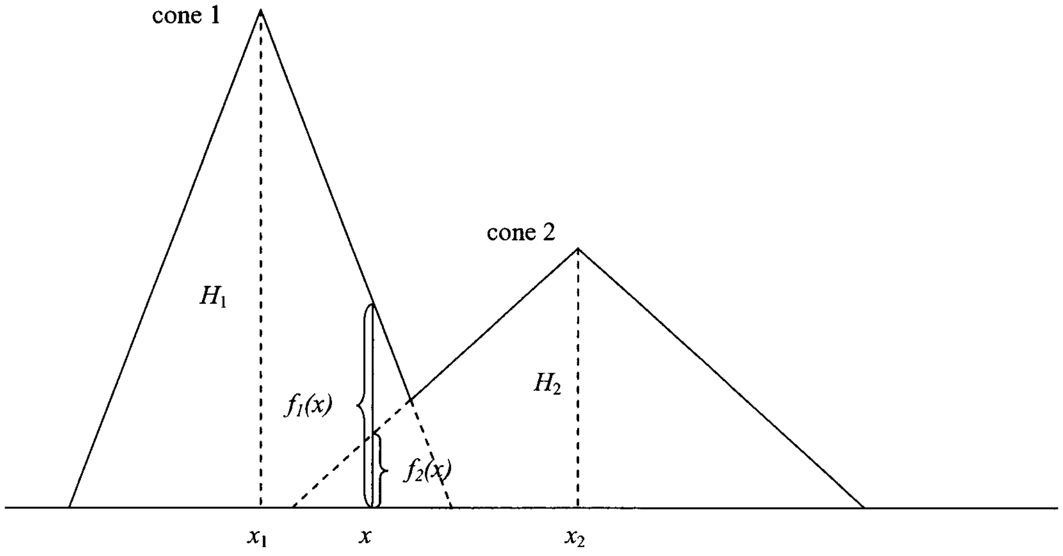

Figure 4.1 Illustration of calculating the function value for a given point in the cone field landscape

When the generator is called each time, it produces a randomly generated landscape of this type in which random feature values for each cone are assigned based on user-specified ranges:

$$\begin{aligned} H_i &\in [Hbase, Hbase + Hrange] \\ R_i &\in [Rbase, Rbase + Rrange] \end{aligned} \tag{4.3}$$

$$\begin{aligned} x_i &\in [Xbase, Xbase + Xrange] \\ y_i &\in [Ybase, Ybase + Yrange] \end{aligned} \tag{4.4}$$

To generate wide range of static problems of varying complexity, one needs only specify the parameters:

- *N* (the number of peaks),
- *Hbase* (the minimum cone height),
- *Hrange* (the range of allowed cone heights),
- *Rbase* (the minimum value of slope control variable),
- *Rrange* (the allowed range for the slope control),
- *Xbase* (the minimum value of *x* coordinates of peaks),
- *Xrange* (the allowed range for peaks to move in *x* direction. Peaks move between *Xbase* and *Xbase+Xrange*),
- *Ybase* (the minimum value of *y* coordinates of peaks),
- *Yrange* (the allowed range for peaks to move in *y* direction. Peaks move between *Ybase* and *Ybase+Yrange*),

Figure 4.2 and 4.3 show two randomly generated landscapes.

4.2 Dynamics of the Fitness Landscape

In this dynamic problem generator, the features of the fitness landscape change in discrete step sizes. To control the generation of a variety of different step sizes the following function was used:

$$Y_i = A * Y_{(i-1)} * (1 - Y_{(i-1)}) \tag{4.5}$$

where: A is a constant specified by the user, Y_i is the value at iteration i.

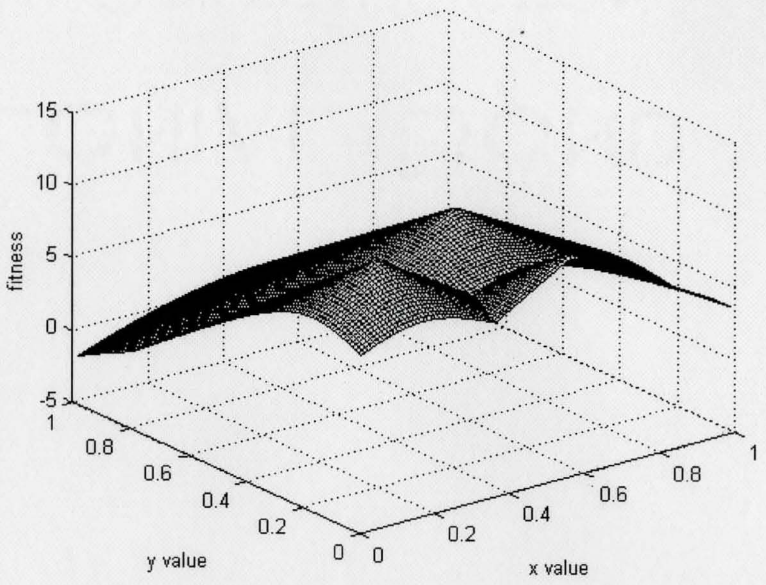

Figure 4.2 Randomly generated fitness landscape with *N* = 3, *Hbase* = 10, *Hrange* = 2, *Rbase* = 15, *Rrange* = 2

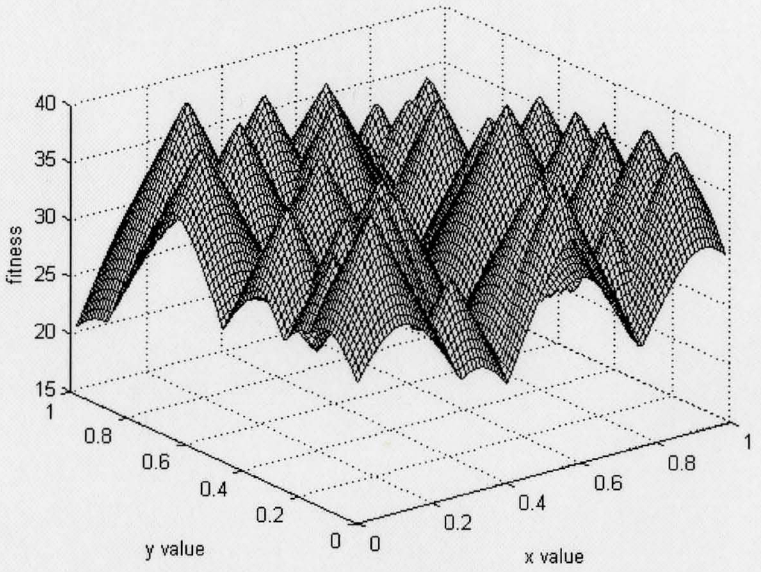

Figure 4.3 Randomly generated fitness landscape with *N* = 50, *Hbase* = 30, *Hrange* = 10, *Rbase* = 70, *Rrange* = 15

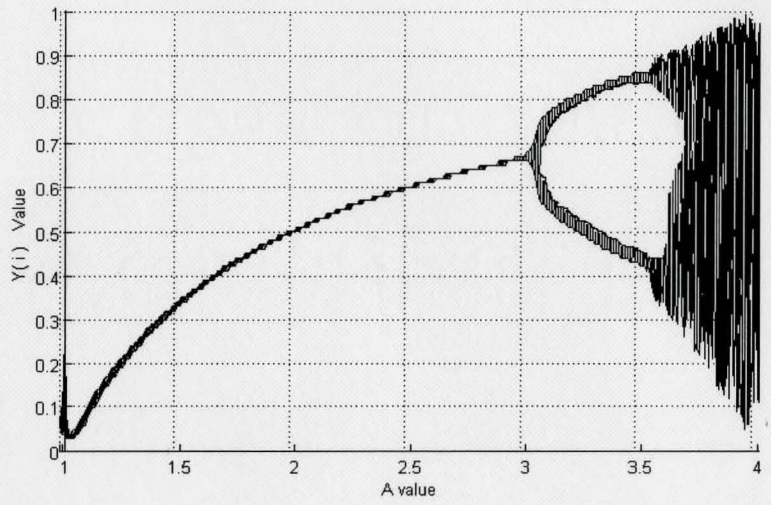

Figure 4.4 Graph of dynamics control function $Y_i = A * Y_{(i-1)} * (1 - Y_{(i-1)})$

From Figure 4.4 we can see that for the values of *A* between 1.0 and 3.0, *Y* is constant; the values of *A* between 3.1 and 3.6 generate two different *Y* values. If the value of *A* is bigger than 3.6, *Y* is a random number between 0 and 1.0. In this study we used *A* value of 3.3.

What remains then is to map the range of *Y* values produced into appropriately scaled step sizes for the particular dynamic feature. This is accomplished by scaling the *Y* values to keep the step sizes less than 0.5 of the user-range to add to or subtract from the current parameter value. For example, for an individual cone that is increasing dynamically in height, first the current height is computed as a percentage of maximum height:

$$Hpct = H/(Hbase + Hrange) \tag{4.6}$$

The current *Y* value is then scaled by a user-supplied height scaling factor and added to the *Hpct*:

$$Hpct = Hpct + Y * Hscale \tag{4.7}$$

If this is less than the 100% of the valid range, then the new height value is computed from the percentage value. If it is greater that 100%, then the new value is computed as:

$$Hbase + (100\% - (Hpct - 100\%)) * Hrange \tag{4.8}$$

and the step change sign is reversed. This causes the movement to "bounce" off of the limits of the search space. The sign remains reversed for each iteration until the minimum value of the range is reached, at which point it is reversed again.

To illustrate the dynamics of this landscape let us look at the following figures. Figure 4.5 is the initial figure that is generated with N = 3, $Hbase$ = 10, $Hrange$ = 4, $Rbase$ = 15, and $Rrange$ = 4.

Through specifying the A values and scaled step sizes for different features we can:

1. *Change peak heights*

   By specifying A value and *Hscale* (scaled step size for height change) value for peak heights, we can change peak heights randomly, we can also choose which peak to be changed.

2. *Change slopes (cone shapes)*

   Like changing the peak heights, we can also change cone slopes by providing appropriate A value and *Rscale* (scaled step size for slope change) value for cone slope dynamics control function. Figure 4.6 is the landscape that we got from the initial landscape in Figure 4.5, after randomly changing the peak heights and cone slopes at the same time. We can see that the cones became wider and the cone at the front became the highest one.

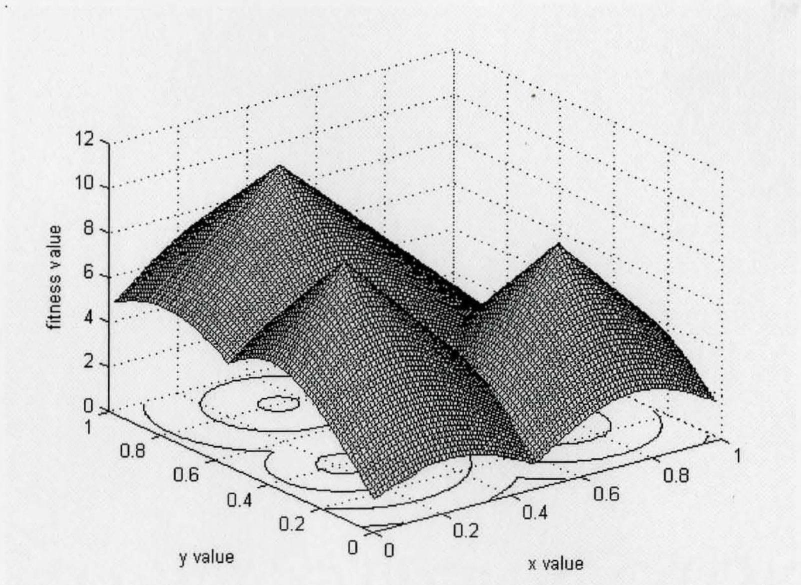

Figure 4.5 Initially generated landscape with *N* = 3, *Hbase* = 10, *Hrange* = 4, *Rbase* = 15, *Rrange* = 4

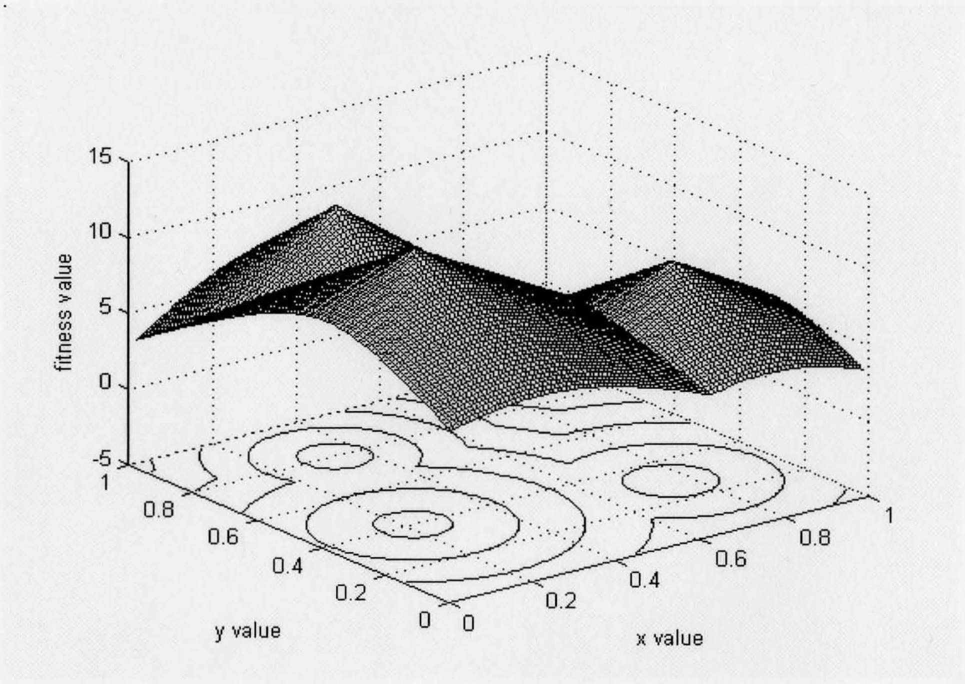

Figure 4.6 After changing the peak heights and slopes of the landscape in Figure 4.5 with *N* = 3, *Hbase* = 10, *Hrange* = 4, *Rbase* = 15, *Rrange* = 4

3. *Move the cones in one or multiple dimensions*

Another dynamics is that the cones can be moved in any direction and we can also control the speed (step size) of the movement, by specifying the *A* value and scaled step size value for movement change of the movement control function. Figure 4.7 and Figure 4.8 are the landscapes that we got from the initial landscape in Figure 4.5, by moving peaks randomly in *x* axis and *y* axis respectively. If we have a *N* dimensional landscape then we can also move the peaks in all dimensions at the same time.

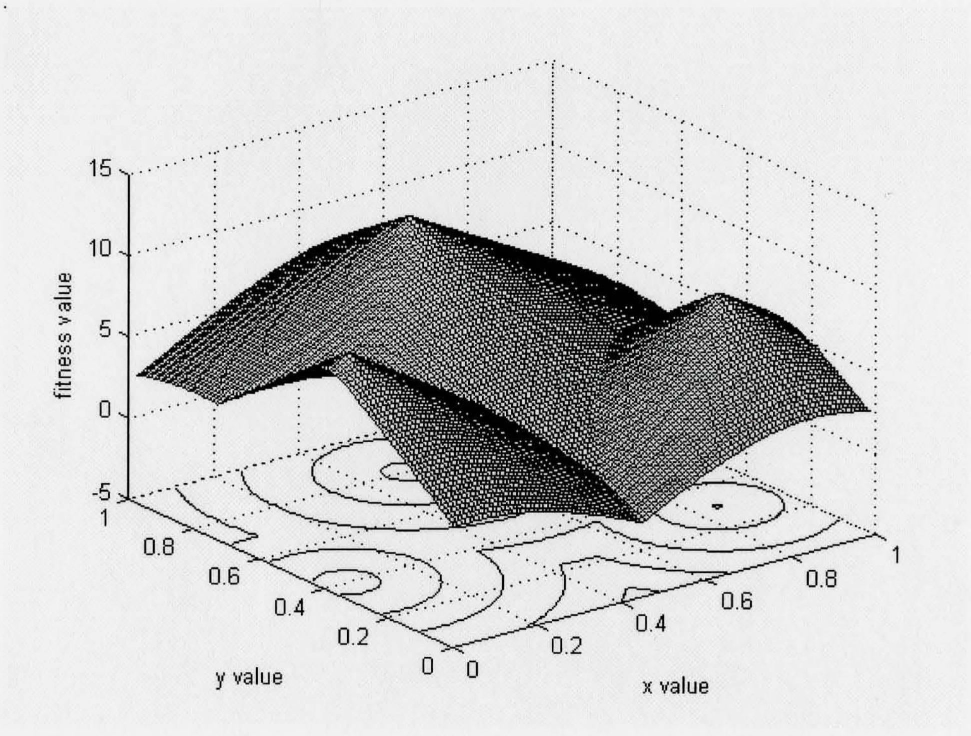

Figure 4.7 After moving the peaks of the landscape in Figure 4.5 along *x* axis randomly, with *N* = 3, *Hbase* = 10, *Hrange* = 4, *Rbase* = 15, *Rrange* = 4

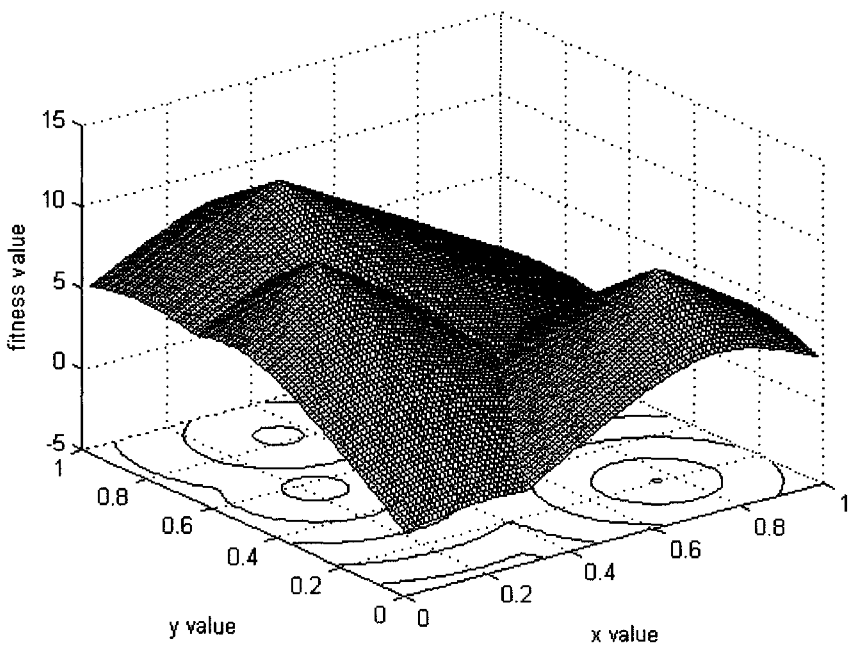

Figure 4.8 After moving the peaks of the landscape in Figure 4.5 along *y* axis randomly, with *N* = 3, *Hbase* = 10, *Hrange* = 4, *Rbase* = 15, *Rrange* = 4

To summarize, we can set the dynamics of the landscape as we want; by applying all the dynamics simultaneously, we can get a complex changing landscape.

CHAPTER 5

IMPLEMENTATION AND EXPERIMENTAL RESULTS

In the previous chapters, we have discussed the mechanism of TGA and dynamic problem generator (DPG). In this chapter, first we introduce their implementation: as we have said before, in TGA, the transformation operator replaces the crossover operator in standard GA; for comparison purposes, we also provided crossover procedure so that the user can decide whether to use TGA or standard GA. Secondly, we discuss our experimental results that we got by running TGA in different dynamically changing environments with different parameters' setting.

5.1 Implementation

In the implementation of GAs, one of the first things to consider is the representation of the potential solutions to the problem (definition of chromosome structure). The dynamic test problem generator [19] we used produces dynamic landscape in multiple dimensions, and our goal is to search for the best point which has the highest fitness value. So the representation of a potential solution should be the coordinate of a point in multiple dimensional space. We used binary representation to encode the coordinates, because it is convenient to perform transformation by using bitwise operations. The user can set the number of dimensions and the number of bits that are used to represent the coordinate in one dimension. From this, we can calculate the length of a chromosome: for example, if number of dimensions is *N* number of bits in one dimension is *LENGTH*, then, length of a chromosome is *N*LENGTH.* It should be noted that *LENGTH* should be the multiple of 8, or else we have to make extra effort to get the correct coordinate value. In this study *LENGTH* is set to 8.

The flowchart of the entire program is given in Figure 5.1, from this we can see that it is very similar to the flowchart in Figure 3.1. Now we give a brief description of every block.

1. *Define GA representation, TGA, DPG parameters, and global variables*

   We have discussed the GA representation above. TGA parameters and global variables are defined in header file. TGA parameters are:

   - transformation rate (the percentage of individuals to be transformed in the population, it is a global constant named *TRANS_RATE*),
   - mutation rate (probability of mutation, it is a global constant named *MUT_RATE*),
   - segment replacement rate (percentage of the segments to be generated from previous population, it is a global constant named *SEG_REPL_RATE*),
   - population size (number of individuals in the population, in this study we keep the population size fixed; once the user sets the population size, it will not change throughout all the generations. It is a global constant named *POPSIZE*),
   - number of segments (the number of segments is also fixed in this study, it is a global constant named *NUMBER_SEG*).

The DGP parameters are:

- number of dimensions (it is a global constant named *NUMBER_DIM*),
- number of peaks (it is a global constant named *NUMBER_PEAKS*),
- minimum height of the peaks (it is a global constant named *Hbase*),
- range of the peak height change (it is a global constant named *Hrange*, the peak heights will oscillate between *Hbase* and *Hbase* + *Hrange*),
- minimum slope value (it is a global constant named *Rbase*),
- range of the cone slope change (it is a global constant named *Rrange*, the cone slope values will oscillate between *Rbase* and *Rbase* + *Rrange*),

Global variable for TGA are:

- two dimensional array *pop*[*POPSIZE*][*NUMBER_DIM*] is used to store all the individuals in the population,
- two dimensional array *seg*[*NUMBER_SEG*][*NUMBER_DIM*] is used to store the gene segments, and *newseg*[*NUMBER_SEG*][*NUMBER_DIM*] is used to store newly generated segment pool, it will replace *seg*[*NUMBER_SEG*][*NUMBER_DIM*] at the end of each generation.

Global variable for DPG are:

- *x*[*NUMBER_PEAKS*][*NUMBER_DIM*] is the coordinate of every peak in the fitness landscape,
- *H*[*NUMBER_PEAKS*] specifies the height of every peak,
- *R*[*NUMBER_PEAKS*] specifies the slope of every cone.

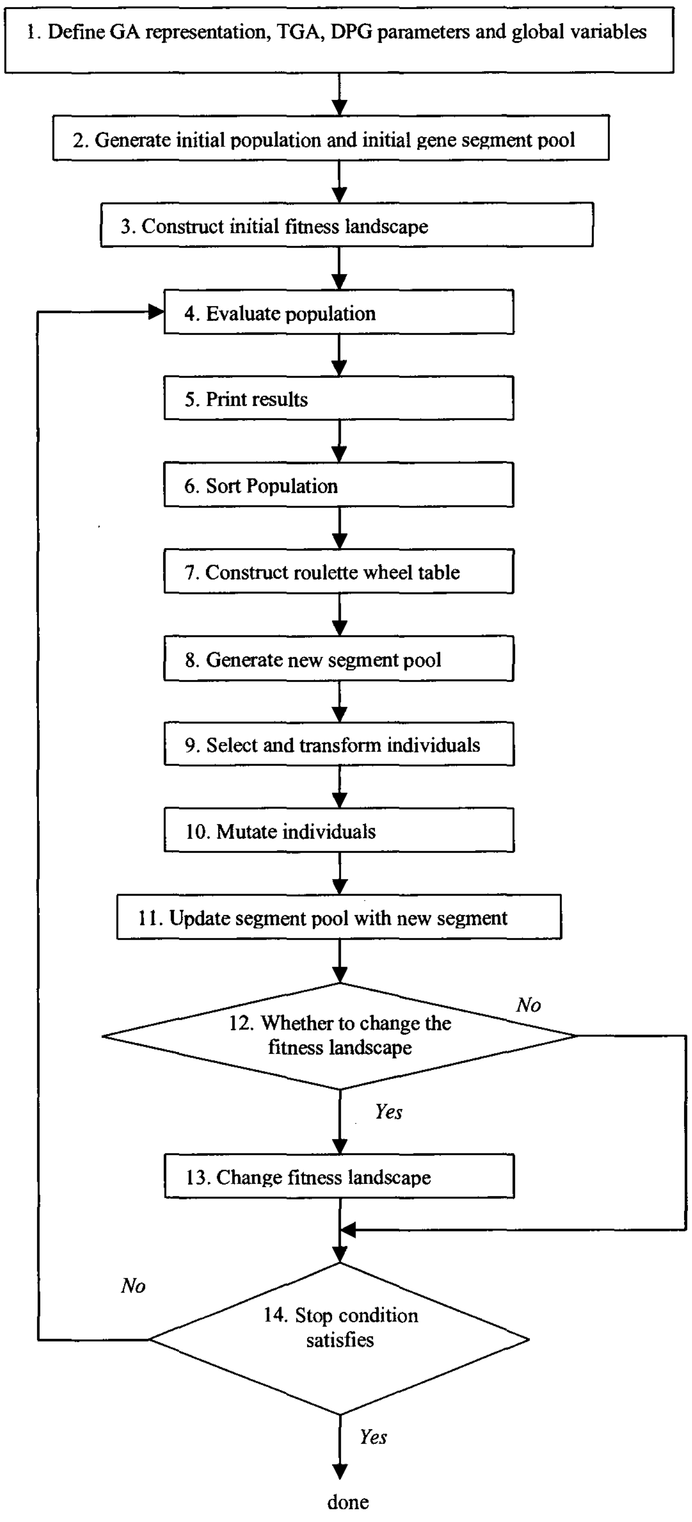
1. Define GA representation, TGA, DPG parameters and global variables
2. Generate initial population and initial gene segment pool
3. Construct initial fitness landscape
4. Evaluate population
5. Print results
6. Sort Population
7. Construct roulette wheel table
8. Generate new segment pool
9. Select and transform individuals
10. Mutate individuals
11. Update segment pool with new segment
12. Whether to change the fitness landscape
No
Yes
13. Change fitness landscape
No
14. Stop condition satisfies
Yes
done

Figure 5.1 Flowchart of TGA program

2. *Generate initial population and initial gene segment pool*

   This initialization is carried out by a procedure named *tga_Init*( ), it is a straightforward procedure that randomly generates every individual in the population and every gene segment in the segment pool, stores the generated values in *pop*[ ][ ] and *seg*[ ][ ]respectively.

3. *Construct initial fitness landscape*

   dpg_init( ) procedure constructs the initial fitness landscape, it randomly generates the feature values in given range for every peak. It stores these values in global variables: *x*[ ][ ], *H*[ ][ ], and *R*[ ][ ]. The landscape features include peak location, heights, and slope.

4. *Evaluate population*

   During the evolution process, we evaluate the population once in every generation. This evaluation is accomplished by procedure *tga_Evaluate*(*pop*). Procedure *tga_Evaluate*(*pop*[ ][ ]) calls function *dgp_eval*(*pop*[*i*]) to calculate the fitness of every individual, function *dpg_eval*(*pop*[*i*]) returns the fitness value to which that individual corresponds. After getting the fitness value from *dpg_eval*(*pop*[*i*]), *tga_Evaluate*(*pop*[ ][ ]) stores the evaluated fitness values in global array *eval*[ ].

5. *Print results*

   User can modify this procedure to print out what he needs, in this study we printed out the best and average fitness value of current population.

6. *Sort Population*

   We need to sort the population according the fitness of the individuals in order to replace the poorly fitted individuals with the new ones. The procedure *tga_SortPop*(*eval*[ ], *index*[ ]) accomplishes this task, it reads the fitness values from array *eval*[ ], and stores the sorted indices in array *index*[ ].

7. *Construct roulette wheel table*

   The mechanism of roulette wheel selection and the construction of roulette wheel table have been discussed in Chapter 2. The procedure *tga_ConsRltWhl*(*eval*[ ], *qi*[ ]) works in the same way. It reads fitness values from *eval*[ ], and stores the roulette wheel table (accumulative selection probabilities) in *qi*[ ].

8. *Generate new segment pool*

   In TGA we update the segment pool with the new segments. Some of the new segments are generated from individuals in the old population and the rest of them generated randomly. The procedure *tga_GenSegPool*(*pop*[ ][ ], *qi*[ ], *newseg*[ ][ ]) generates a new segment pool and stores it in array *newseg*[ ][ ], this new segment pool is used to update the current segment pool at the end of the iteration.

9. *Select and Transform individuals*

   We incorporated the implementation of selection and transformation. The procedure *tga_Transform*(*pop*[ ][ ], *seg*[ ][ ] , *qi*[ ], *index*[ ]) carries out both selection and transformation, the flowchart of this procedure is given in Figure 5.2. It first calculates the number of individuals to be transformed (block 9.1 in Figure 5.2), and then the rest of the process is carried out in iteration loop.

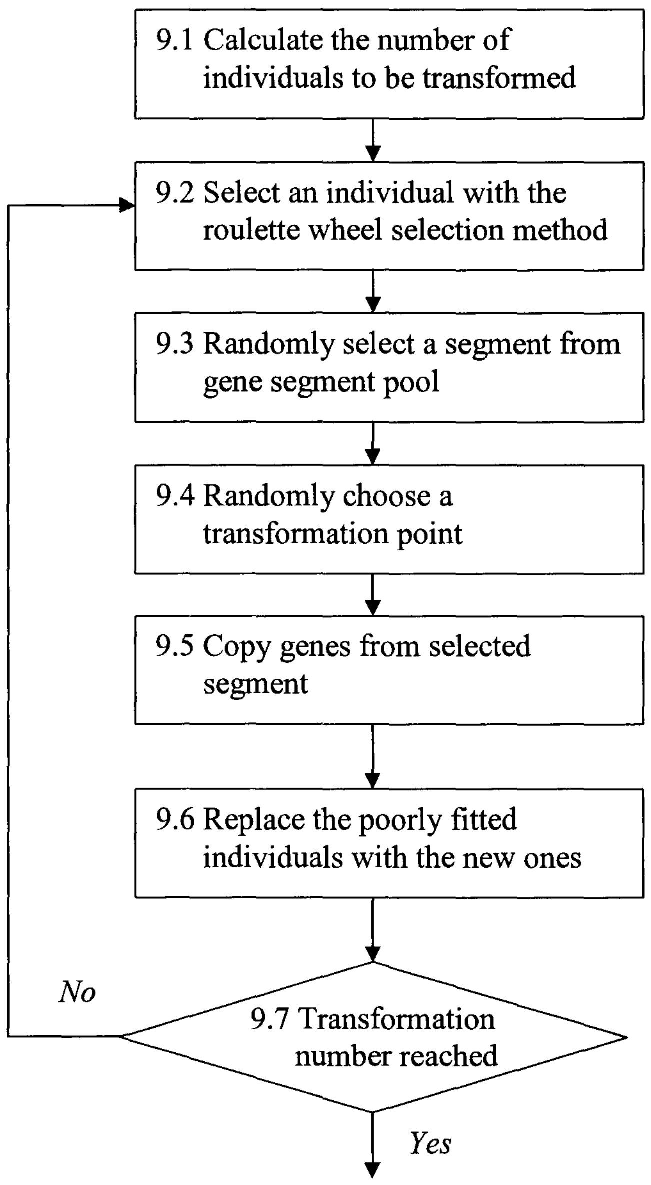
9.1 Calculate the number of individuals to be transformed
9.2 Select an individual with the roulette wheel selection method
9.3 Randomly select a segment from gene segment pool
9.4 Randomly choose a transformation point
9.5 Copy genes from selected segment
9.6 Replace the poorly fitted individuals with the new ones
No
9.7 Transformation number reached
Yes

Figure 5.2 Flowchart exhibiting the procedure of selection and transformation (block 9 of the flowchart on Figure 5.1)

In every loop, it selects an individual from the population with roulette wheel selection (block 9.2 in Figure 5.2), and randomly selects a segment from the segment pool (block 9.3 in Figure 5.2), then, randomly selects the transformation point (block 9.4 in Figure 5.2). As we have discussed before, there are two cases to consider depending on the value of the transformation point. It copies the genes from selected gene segment according to the transformation point (block 9.5 in Figure 5.2), then, replaces a poorly fitted individual (the individual that has a lowest fitness value) from the old population with the new individual (block 9.6 in Figure 5.2). This procedure gets individuals from *pop*[ ][ ] and segments from *seg*[ ][ ]. It uses roulette wheel table *qi*[ ] for selection purposes, and uses *index*[ ] when replacing old individuals. From the above description we can see that this procedure updates the population *pop*[ ][ ] every time it transforms an individual.

10. Mutate individuals

In binary GAs mutation is simply flipping the bits. The procedure *tga_Mutate*(*pop*[ ][ ]) carries out the mutation. It first calculates number of bits to be mutated, then, it randomly chooses which bit of which individual to be mutated. After selecting the bit it simply flips that bit. This procedure also updates the population every time it mutates an individual.

11. Update segment pool with new segment pool

We have already generated a new segment pool that part of its segments was generated from the individuals of the old population. To update current segment pool we just copy all the segments from new segment pool into current segment pool. The procedure *tga_UpdateSegPool*(*seg*[ ], *newseg*[ ]) updates *seg*[ ] with *newseg*[ ].

12. *Condition: whether to change the fitness landscape*

    In this study, the fitness landscape changes once in several generations. Between two changes the fitness landscape is kept static. The global variable *CHANGE_GAP* represents the number of generations between two consequent changes. We used *CHANGE_GAP* to set the frequency of the landscape change.

13. *Change fitness landscape*

    We have discussed the dynamics of the landscape in Chapter 4. The following procedures change the landscape:

    - *dpg_chg_C*( ) moves the peaks in a given axis,
    - *dpg_chg_H*( ) changes the peak heights,
    - *dpg_chg_R*( ) changes the peak slopes.

    We can choose which procedure to call according to the landscape change we need. By using all of above procedures we can get a very complicatedly changing landscape.

14. *Stop condition satisfies*

    In this study, the stop condition is to check whether the maximum number of generations is reached or not.

Above we have introduced the implementation of TGA. In the next section we will discuss the results of the experimentations we performed on this TGA.

5.2 Experimental Results

We first carried out a comparative study between TGA and other common GAs, including *standard genetic algorithm* (SGA) [19] which we have introduced in Chapter 2, and *triggered* h*ypermutation-based genetic algorithm* (HGA) [6] that was introduced in section 3.1. Secondly we tested the TGA performance in different parameter settings.

There are different kinds of measurements that are used to compare the performance of the GA. Most common measurements are *online performance* [2] and *offline performance* [2].

Online performance is an average of all individual's fitness value on the entire run.

Offline performance is the average of the best fitness values on the entire run.

In this study, we used offline performance as a measurement to compare the efficiency of different GAs.

5.2.1 TGA and SGA Performance in Static Landscape

Before comparing these genetic algorithms in a dynamic environment, we compared them in a static environment where the fitness landscape was kept static throughout all the generations.

The comparison result of offline performance (the average of the best fitness values on the entire run) of TGA and SGA is shown in figure 5.3. We repeated the test for 20 times and took the average of offline performances. For both algorithms the fitness values were calculated with function (4.1), *pop_size* (population size) was set to 20, *gen_num* (number of generations) was set to 100, *max_height* (highest fitness value can be found, or in other words, the height of the highest peak in the fitness landscape) was set to

1873.00. SGA parameters were set as: *cross_rate* (crossover rate) = 0.7, *mut_rate* (mutation rate) = 0.01; TGA parameters were set as: *trans_rate* (transformation rate) = 0.6, *seg_repl_rate* (segment replacement rate) = 0.5, *mut_rate* = 0.001. We can see from this figure that in static landscape TGA performs about 5% better than SGA.

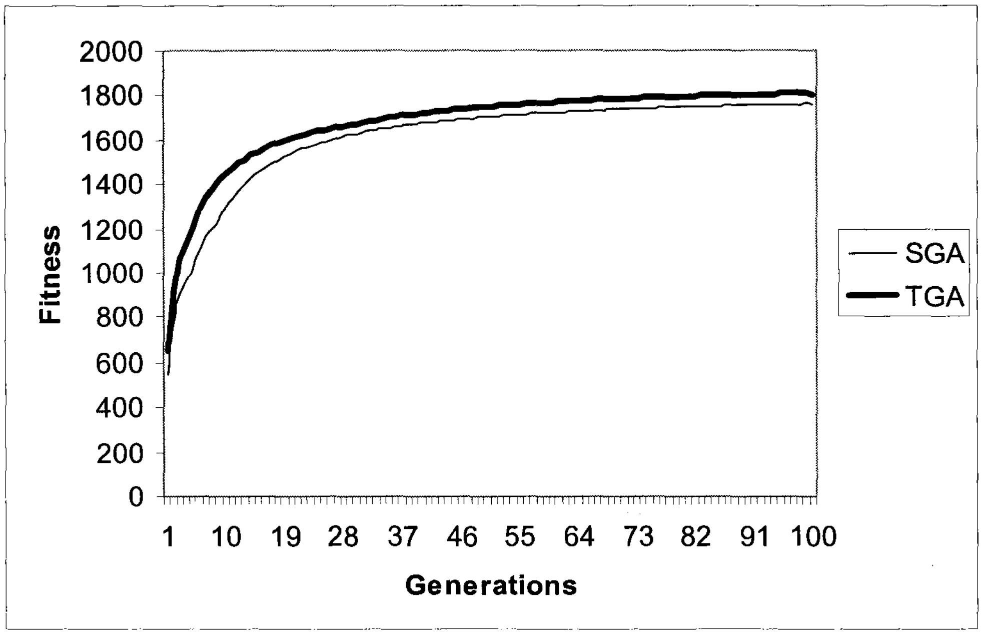

Figure 5.3 Offline performances of TGA and SGA in static fitness landscape

We also tested TGA and SGA performance in different combinations of population sizes and numbers of generations. This time we compared the highest fitness value. For every combination of population size and number of generations, we repeated the test for 20 times and took the average of these 20 highest fitness values. The fitness values were calculated with function (4.1); the results are shown in Table 5.1. Numbers in the second row are the population sizes; numbers in the second column are the numbers of generations. Every value in bold is the average of highest fitness values that were found in the corresponding population size and number of generations. The best values among them are shown in italic and underlined.

Table 5.1 The highest fitness values found in different population sizes and different number of generations (best values are in italic and underlined)

		TGA			**SGA**		
pop_size →		20	50	100	20	50	100
***Gen_num* (Number of generations)**	10	**1648.24**	**1765.38**	**1824.87**	**1622.58**	**1795.00**	**1829.95**
	20	**1781.68**	**1824.87**	***<u>1869.23</u>***	**1613.81**	**1824.87**	**1838.97**
	50	**1838.9**	**1836.45**	***<u>1870.13</u>***	**1614.26**	**1827.34**	**1857.78**
	100	**1857.78**	**1857.78**	***<u>1873.00</u>***	**1625.56**	**1835.74**	***<u>1873.00</u>***

In this test, the *max_height* = 1873.00. SGA parameters were set as: *cross_rate* = 0.7, *mut_rate* = 0.01; TGA parameters were set as: *trans_rate* = 0.6, *seg_repl_rate* = 0.5, *mut_rate* = 0.001.

From this table we can see that TGA performs about 3% better (we took the average difference of TGA and SGA values in the table) than SGA in these parameter settings. For both algorithms the bigger the population size the better the performance is. The reason for this is that when the population size increases the diversity of the population also increases, so there is greater chance of placing the individuals around an optimal peak. We can also see from Table 5.1 that when the number of generations increases the performance also improves. This is because genetic algorithm has enough time to find optimal values when the number of generations increases.

We did not compare TGA with HGA in a static environment, because HGA works in exactly same manner as SGA in a static environment. So from SGA performance, we can also conclude the same HGA performance.

5.2.2 TGA, HGA, and SGA Performance in Dynamic Landscape

In this study, we mainly focused on the TGA performance in a dynamic environment. Researchers have intensively studied the GAs working in static environment for over 30 years, but GAs working in dynamic environments are still new fields.

As we have stated before, we used dynamic test problem generators to create dynamic landscapes for our study. To better test the performance of algorithms, we fixed the peak heights so that the best fitness value is the same throughout the entire generations. The dynamics we applied was: moving the peak locations, and changing the peak slopes randomly. Thus we can still get a changing landscape.

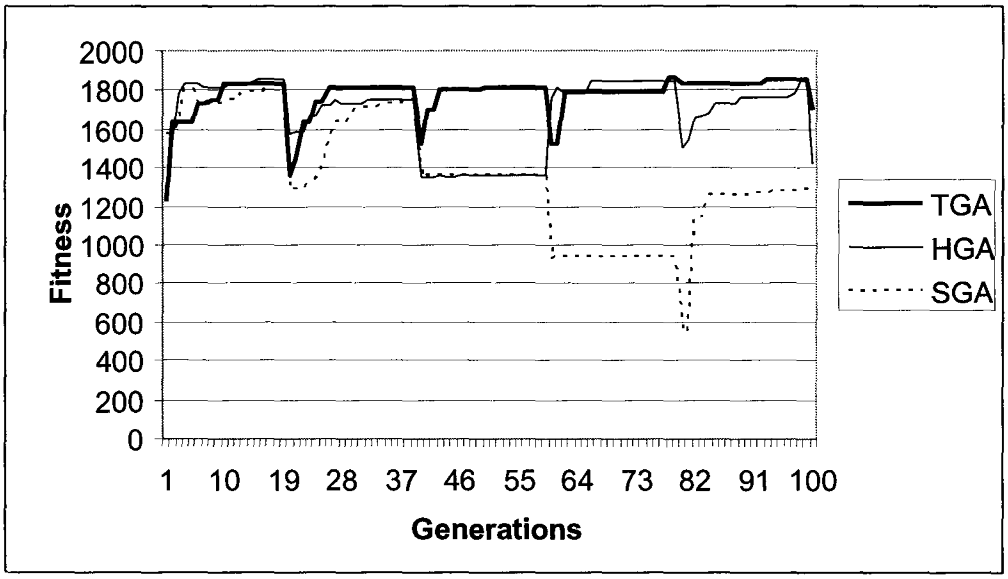

Figure 5.4 Highest fitness values of TGA, HGA, and SGA in a dynamic landscape where the landscape changes every 20 generations

We tested the three algorithms TGA, HGA, and SGA in an environment where landscape changes every 20 generations; *max_step_scale* (the percentage of maximum step size of landscape movement) is set to 30%. This time we also compared the highest fitness values; the fitness values were calculated with function (4.1). We repeated the test for 20 times and calculated the average of highest fitness values. The test results are shown in Figure 5.4. For all algorithms *pop_size* = 50, *gen_num* = 100, and, *max_height* = 1873.00. SGA parameters were set as: *cross_rate* = 0.7, *mut_rate* = 0.01; TGA parameters were set as: *trans_rate* = 0.6, *seg_repl_rate* = 0.5, *mut_rate* = 0.001; HGA parameters were set as: *cross_rate* = 0.5, *mut_rate* = 0.01, *hyper_mut_rate* (hypermutation rate) = 0.2.

From this figure we can see that every time landscape changes, the performances of algorithms declines. We can also see that TGA is always able to find a higher fitness value quickly, HGA also performs well. But SGA behaves poorly in this dynamic landscape.

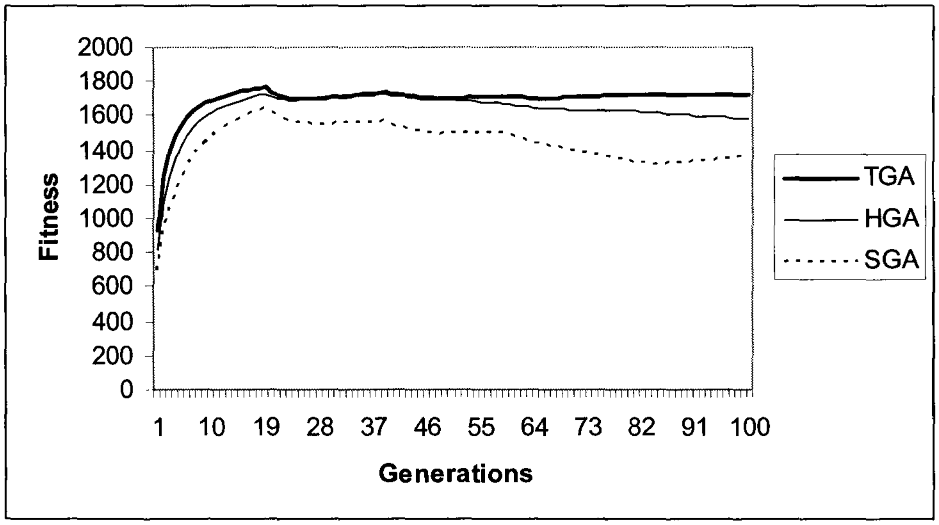

Figure 5.5 Offline performances of TGA, HGA, and SGA in dynamic landscape where the landscape changes every 20 generations

We also compared the offline performance (the average of the best fitness values on the entire run) of these three algorithms in the same parameter settings as in Figure 5.4. We repeated the test 20 times and took the average of the offline performances. The result is shown in Figure 5.5.

It is very noticeable from these two figures that SGA behaves poorly in this dynamic landscape. The reason for this is that before the change of the landscape, individuals tend to converge around the optimal peak, but after the landscape change, these individuals find themselves in lower fitness. In the following generations SGA still generates new individuals around a previous region which is now in a low fitness area. That means SGA lacks *population diversity* (the extent to which individuals spread evenly throughout the search space) which is vital in dynamic environments. In SGA the main genetic operators are crossover and mutation. The individuals created through crossover are most probably in the same region as their parents. The only mechanism that can increase the population diversity is mutation, but in SGA the mutation rate is usually very small. So SGA has a smaller chance of finding optimal peak in dynamic environment.

From Figure 5.4 and Figure 5.5 we can also see that, HGA performs almost the same as TGA and far better than SGA. The reason is that HGA keeps track of the population fitness in every generation. If it finds some significant decline in population fitness that means the landscape has changed. In this case HGA dramatically increases the mutation rate and consequently increases the population diversity.

Like in section 5.2.1, we also tested these algorithms in different combinations of population sizes and landscape change durations. This time we compared the highest fitness value; the fitness values were calculated with function (4.1). For every combination of population size and landscape change durations, we repeated the test for 20 times and took the average of these 20 highest fitness values. The results are shown in Table 5.2. Numbers in the second row are the population sizes; numbers in the second column are the landscape change durations. Every value in bold is the average highest fitness value that was found in the corresponding population size and landscape change duration. The best values among them are shown in italic and underlined. For all algorithms *gen_num* = 100, *max_height* = 1873.00, *max_step_scale* = 30%. SGA parameters were set as: *cross_rate* = 0.7, *mut_rate* = 0.01; TGA parameters were set as:

trans_rate = 0.6, *seg_repl_rate* = 0.5, *mut_rate* = 0.001; HGA parameters were set as: *cross_rate* = 0.5, *mut_rate* = 0.01, *hyper_mut_rate* = 0.2.

Table 5.2 The highest fitness values found in different population sizes and different landscape change durations (LCHD) (best values are in italic and underlined)

		TGA			HGA			SGA		
pop_size →		20	50	100	20	50	100	20	50	100
LCHD	10	**1765.10**	**1791.25**	**1850.52**	**1742.35**	**1801.23**	**1836.73**	**1672.12**	**1783.25**	**1810.34**
	20	**1796.90**	**1831.94**	**1855.16**	**1752.36**	**1840.63**	**1857.54**	**1705.21**	**1802.74**	**1830.62**
	50	**1842.56**	**1857.78**	***1873.00***	**1821.45**	**1865.25**	***1873.00***	**1814.57**	**1837.25**	**18.67.42**

From this table we can see that TGA performs about 2% better (we took the average difference of TGA and SGA values in the table) than SGA, and, performs almost same as HGA in these parameter settings. For all three algorithms, the bigger the population size, the better the performance is. The reason for this is same as we have explained before. We can also see from Table 5.2 that when the landscape change duration increases the performance also improves. This is because when the landscape is kept static for longer period of time, genetic algorithm has more time to find higher fitness values.

5.2.3 TGA performance in different parameter settings

In this section we will discuss the TGA performance in different parameter settings such as, transformation rate, segment replacement rate and mutation rate.

1. TGA performance in different transformation rate

In this test we compared the offline performance (the average of the best fitness values on the entire run) of TGA in two different dynamics; the fitness values were calculated with function (4.1). In Figure 5.6 the landscape changes every 20 generations, while in Figure 5.7 the landscape changes every 50 generations. In both tests, we repeated the tests for 20 times and calculated the average of the offline performances. In this test, *gen_num* = 100, *max_height* = 1873.00, *max_step_scale* = 30%. TGA parameters were set as: *seg_repl_rate* = 0.5, *mut_rate* = 0.001.

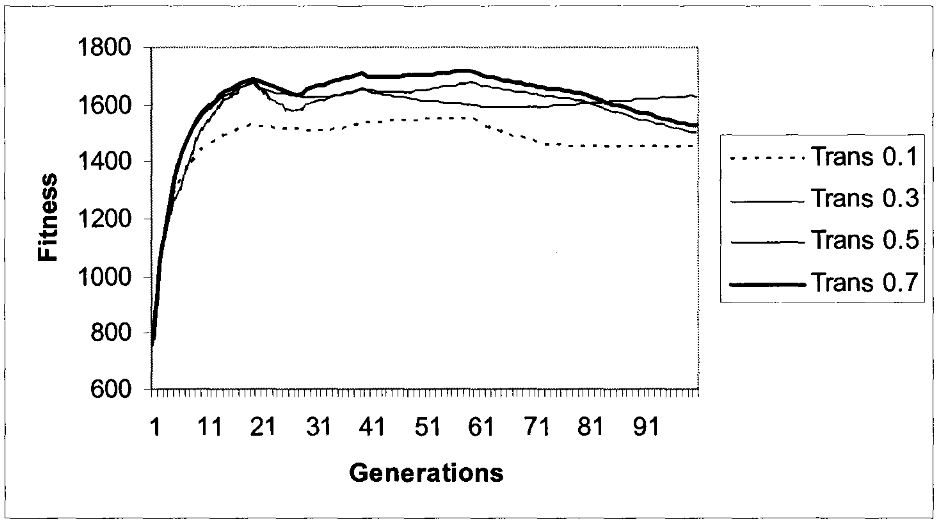

Figure 5.6 Offline performance of TGA with different transformation rates (Trans) in a landscape that changes every 20 generations

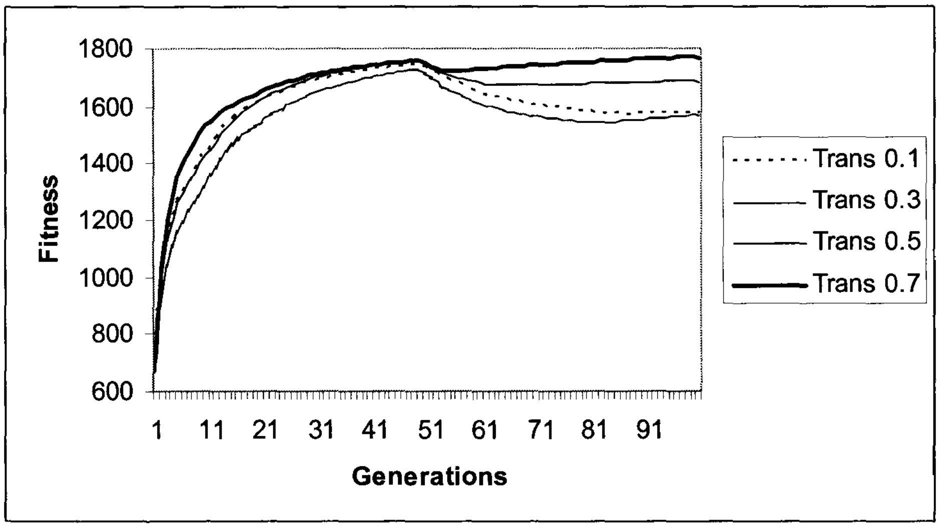

Figure 5.7 Offline performance of TGA with different transformation rates (Trans) in a landscape that changes every 50 generations

From these two figures, we can see that in both cases the TGA with the transformation rate of 0.7 performs better. The reason for this is that when the transformation rate is small, only small portion of the population is transformed; thus the number of newly generated individuals is not big enough to increase the population diversity. When transformation rate is increased to 0.7, the number of new individuals is increased, and, consequently population diversity is also increased. But transformation rate cannot be too high. If it is too high, then, some individuals with higher fitness value will also be transformed. This may cause the destruction of these individuals' chromosomes, thus the overall performance of the algorithm will decrease.

2. *TGA performance in different segment replacement rate*

In this experiment we tested the offline performances of TGA in different combinations of segment replacement rates and landscape change durations; the fitness values were calculated with function (4.1). We also repeated the test for 20 times, and, took the average of the offline performances. The highest values are shown in italic and underlined.

The results are shown in Table 5.3. In this test, *pop_size* = 50, *gen_num* = 100, *max_height* = 1873.00, *max_step_scale* =30%. TGA parameters were set as: *trans_rate* = 0.7, *mut_rate* = 0.001.

Table 5.3 Offline performance of TGA in different segment replacement rates and different landscape change durations (LCHD) (best values are in italic and underlined)

	LCHD =10	LCHD =20	LCHD =50	LCHD =100
seg_repl_rate = 0.1	**1507.41**	**1566.26**	**1631.99**	**1682.25**
seg_repl_rate = 0.2	**1527.42**	**1583.00**	**1641.45**	***<u>1700.45</u>***
seg_repl_rate = 0.3	**1549.31**	**1588.26**	***<u>1683.25</u>***	***<u>1708.45</u>***
seg_repl_rate = 0.5	**1513.71**	**1570.31**	**1622.51**	***<u>1710.73</u>***
seg_repl_rate = 0.7	**1524.96**	**1564.39**	**1626.33**	**1645.3**

From this table we can see that, segment replacement rate of 0.3 is good for all dynamics. We think the reason for this is that when segment replacement rate is 0.3, only 30% of the segments generated from old population and the rest of them generated randomly. This large number of randomly generated gene segments increases the population diversity. Consequently the algorithm behaves well in dynamic environments. If the segment replacement rate is too small, then almost all the gene segments are generated randomly. In this case it is possible that some segments with better genes may be replaced by randomly generated segments, thus decreasing the performance of TGA.

3. *TGA performance in different mutation rate*

Mutation rate is one of most important genetic operators in standard genetic algorithms. In this study we found that even mutation can be replaced by transformation in TGA.

In this experiment, we also compared the offline performances of TGA in different combinations of mutation rates and landscape change durations; the fitness values were calculated with function (4.1). We repeated the test for 20 times, and, took the average of the offline performances. The highest values are shown in italic and underlined.

The results are shown in Table 5.4. In this test, *pop_size* = 50, *gen_num* = 100, *max_height* = 1873.00, *max_step_scale* =30%. TGA parameters were set as: *trans_rate* = 0.7, *set_repl_rate* = 0.3.

Table 5.4 Offline performance of TGA in different mutation rates and different landscape change duration (LCHD) (best values are in italic and underlined)

	LCHD =10	LCHD =20	LCHD =50	LCHD =100
mut_rate = 0.0	**1522.42**	**1576.56**	**1639.64**	***<u>1689.47</u>***
mut_rate = 0.001	**1530.45**	**1578.65**	**1637.46**	***<u>1680.53</u>***
mut_rate = 0.005	**1531.96**	**1580.83**	**1621.53**	**1650.25**
mut_rate = 0.01	**1526.42**	**1576.56**	**1624.94**	**1641.48**
mut_rate = 0.05	**1491.22**	**1528.63**	**1570.64**	**1603.43**
mut_rate = 0.1	**1485.25**	**1435.52**	**1490.28**	**1510.31**

From this table we can see that TGA behaves better if there is no mutation or mutation rate is very small. When the mutation rate increases TGA performance decreases. We think that unnecessarily changing the genes may result the destruction of good chromosomes, thus decreasing the algorithm performance.

CHAPTER 6

CONCLUSION

6.1 Conclusion

In this thesis we have presented a genetic algorithm that uses new biologically inspired genetic operator called transformation. We used this operator as an alternative to crossover. In transformation-based genetic algorithm (TGA), an individual is generated from a single parent and a gene segment. This differs from other GAs that use crossover.

We carried out a series of experiments on the performance of TGA. We used offline performance as a measurement of algorithm performance.

We learned from these experiments that:

- TGA with a higher transformation rate of 0.7 performs better. The reason for this is that higher transformation rate of 0.7 causes more new individuals to be generated; these new individuals replace the poor individuals in the old population, so it increases the overall performance of the algorithm. But transformation rate cannot be too high. If it is too high, then, some individuals with higher fitness value will also be transformed. This may cause the destruction of these individuals' chromosomes, thus the overall performance of the algorithm will decrease.
- Smaller segment replacement rate is preferred in all dynamics. We think the reason for this is that if segment replacement rate is small, then only small

part of the segments is generated from old population and rest of them generated randomly. This large number of randomly generated gene segments increases the population diversity. Consequently the algorithm behaves well in dynamic environments.

- In this study we found that even mutation can be replaced by transformation in TGA. The reason is that mutation is used to increase the diversity of the population. In TGA, randomly generated gene segments can increase the diversity of the population, so by setting proper segment replacement rate, mutation can be replaced by transformation.

We mainly focused on methodology and implementation of general-purpose GAs rather than carrying out a huge set of experimentation.

6.2 Contributions

In the completion of this work:

- We designed and implemented the new version of TGA. In the design of TGA (Simoes, and Costa [24]), we made some modifications to the algorithm when replacing the old individuals with new individuals.
- In order to test the performance of TGA, we used dynamic problem generator (Morrison, [21]) that generates a wide variety of dynamic environments, and we incorporated the implementation of TGA with the dynamic problem generator.
- We compared the TGA performance with other GAs such as standard GA (SGA) [19], and triggered hypermutation-based GA (HGA) [6]. We also studied the TGA performance in different parameter settings. From these

experiments we concluded some characteristics of TGA performance in dynamic environments.

6.3 Future work

We have compared TGA performance with other GAs, and carried out some experiments on TGA. There is still much to study on TGA.

Future work may consist of:

- Study the TGA with variable length gene segments. Because in some problems variable length chromosomes are preferred, in this case TGA with variable length gene segments may perform better.
- Number of segments in the segment pool is also an important factor in TGA. The relation between population size and segments pool size need to be studied. In our implementation of TGA, the individuals are selected by using roulette wheel selection method, while segments are selected randomly. The selection method of segments may need more study also.
- In our experiments the fitness landscape changes abruptly once in several generations. There are also some cases that the fitness landscape changes slowly in every generation. TGA performance and parameter settings need to be studied in these environments.

REFERENCES

[1] Affenzeller, M., and Wagner, S. "Offspring selection: A New Self-Adaptive Selection Scheme for Genetic Algorithms." *In Adaptive and Natural Computing Algorithms*, pp. 218-221. Springer, 2005.

[2] Annunziato, M., and Pizzuti, S. "Adaptive Parameterization of Evolutionary Algorithms Driven by Reproduction and Competition," *in Proceedings of ESIT'2000*, Aachen, Germany, 2000.

[3] Branke J. "Evolutionary Approaches to Dynamic Optimization Problems – Updated Survey." *In: GECCO workshop on Evolutionary Algorithms for Dynamic Optimization Problems*, pp. 27–30, 2001.

[4] Bruha, I., Kralik, P. "Embedding a Genetic Algorithm in Attribute-based Rule-Inducing Learning", *Soft Computing (SOCO-99), Symposium ICSC* pp. 631-635 (International Computer Science Conventions, Canada), Genova, Italy, 1999.

[5] Bruha, I., Kralik, P., Berka, P. "Genetic Learner: Discretization and Fuzzification of Numerical Attributes", *Intelligent Data Analysis Journal*, vol. 4, pp. 445-460, 2000.

[6] Cobb, H. G. "An Investigation Into the Use of Hypermutation as an Adaptive Operator in Genetic Algorithms Having Continuous, Time-Dependent Nonstationary Environment." *NRL Memorandum Report 6760*, 1990.

[7] Cobb, H.G., and Grefenstette, J. J. "Genetic Algorithms for Tracking Changing Environments." *Proceedings of the Fifth International Conference on Genetic Algorithms*, pp.523-530, 1993.

[8] Garrido, S., and Moreno, L. "Learning Adaptive Parameters with Restricted Genetic Optimization Method.", *Proceedings of the 6th International Work-Conference on Artificial and Natural Neural Networks: Connectionist Models of Neurons, Learning Processes and Artificial Intelligence-Part I* , pp.612-620, Springer-Verlag London, UK, 2001.

[9] Goldberg, D.E. *Genetic Algorithms in Search, Optimization, and Machine Learning*, Addison-Wesley Publishing Company Inc. 1989.

[10] Grefenstette, J. J. "Genetic algorithms for changing environments." In R. Manner and B. Manderick (Eds.), *Parallel Problem Solving from Nature, 2*, pp.137–144, Elsevier Science, 1992.

[11] Grefenstette, J. J. "Evolvability in dynamic fitness landscapes: A Genetic Algorithm Approach," *in Proceedings of the Congress on Evolutionary Computation*, vol. 3, pp.2031–2038.Piscataway, NJ: IEEE Press, 1999.

[12] Haupt, R.L., and Haupt, S.E. *Practical Genetic Algorithms*, second edition, John Wiley & Sons, Inc., 2004.

[13] Hercock, R. G. *Applied Evolutionary Algorithms in Java*, Springer, 2003.

[14] Holland, J.H. *Adaptation in Natural and Artificial Systems,* Ann Arbor: The University of Michigan Press, 1975.

[15] Hong, T.P., Wang, H.S., Lin, W.Y., and Lee, W.Y. "Evolution of Appropriate Crossover and Mutation Operators in a Genetic Process." *Applied Intelligence* Vol.16 No. 1, pp. 7–17, 2002.

[16] Jeong, I.K., Lee J.J., "Adaptive Simulated Annealing Genetic Algorithm for System Identification", *Engineering Applications of Artificial Intelligence*, vol. 9, No.5, pp. 523-532, 1996.

[17] Jin, Y., and Sendhoff, B. "Constructing Dynamic Optimization Test Problems Using the Multi-objective Optimization Concept.", *Lecture Notes in Computer Science*, pp. 525 – 536, Springer, 2004.

[18] Lund, H.H. "Adaptive Approaches Towards Better GA Performance in Dynamic Fitness Landscapes." *Technical Report DAIMI PB-487*, DAIMI, Aarhus University, 1994.

[19] Michalewicz, Z. *Genetic Algorithms + Data Structures = Evolution Programs* Springer, Verlag New York, third edition, 1996.

[20] Morrison, R.W. *Designing Evolutionary Algorithms for Dynamic Environments,* Springer, 2004.

[21] Morrison, R. W., and De Jong, K. A. " A Test Problem Generator for Nonstationary Environments." *Proceedings of Congress on Evolutionary Computation,* pp. 2047 – 2053, 1999.

[22] Murata, Y., Shibata, N., Yasumoto, K. and Ito, M. "Agent Oriented Self Adaptive Genetic Algorithm.", *Proceedings of IASTED International Conference on Communications and Computer Networks (CCN2002)*, pp.348-353, 2002.

[23] Smith, J., and Fogarty, T. C.. "Self Adaptation of Mutation Rates in a Steady State Genetic Algorithm."*,In Proceedings of the Third IEEE International Conference on Evolutionary Computing*, pp. 318–323. IEEE Press, 1996.

[24] Simoes, A., and Costa, E. "On Biologically Inspired Genetic Operators: Transformation in the Standard Genetic Algorithm.", *Proceedings of the Genetic and Evolutionary Computation Cenference,GECCO-2001*, pp. 584-591, W. B. Langdon et alli (Eds.) San Francisco, USA, 7-11 July, CA: Morgan Kaufmann Publishers, 2002.

[25] Simoes, A., and Costa, E. "Parametric Study to Enhance Genetic Algorithm's Performance when Using Transformation", *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO'02),* W. B. Langdon et alli (Eds.), Morgan Kaufmann Publishers, New York, 9-13 July, 2002.

[26] Yuan, B., and Gallagher, M. "A Hybrid Approach to Parameter Tuning in Genetic Algorithms". *In Proceedings of the 2005 Congress on Evolutionary Computation, IEEE*, pp. 1096-1103, Edinburgh, UK, 2005.

[27] Wikipedia, the free encyclopedia, http://en.wikipedia.org/wiki/Genetic_Algorithm, 2006.

[28] rEvolutionary Engineering, http://www.rEvolutionaryEngineering.com, 2006.