Parameterless Genetic Algorithms: Review, Comparison and Improvement

Fady Draidi

a Thesis

in

the Department

of

Electrical and Computer Engineering

Presented in Partial Fulfillment of the Requirements

for the Degree of Master of Applied Science (Computer Engineering) at

Concordia University

Montreal, Quebec, Canada

April 2004

Your file Votre référence
ISBN: 0-612-91020-2
Our file Notre référence
ISBN: 0-612-91020-2

Abstract

This dissertation compares the performance of five existing Genetic Algorithms (GAs) that do not require the manual tuning of their parameters, and are thus called Parameterless Genetic Algorithms (pGAs). The five pGAs selected for evaluation span the three most important categories of Parameterless GAs: Deterministic, Adaptive and Self-Adaptive pGAs. The five test functions used to evaluate the performance of the pGAs include unimodal, multimodal and deceptive functions. We assess p erformance in terms of fitness, diversity, reliability, speed and memory load. *Surprisingly*, the simplest Parameterless GA tested proves to be the best overall performer. Last, but not least, we describe a new parameterless Genetic Algorithm (nGA), one that is easy to understand and implement, and which bests *all* five tested pGAs in terms of performance, particularly on hard and deceptive surfaces.

Acknowledgments

I am very grateful to the following people and institutions for their help and support during my dissertation work.

Firstly, I would like to express my fervent gratitude to Dr. Nawwaf Kharma for his guidance, invaluable suggestions and being a constant source of encouragement as my research supervisor. In addition to that, Dr. Kharma taught me to be more organized, to write better, to explain my ideas more clearly, and taught me that hard work pays off. I feel very fortunate to have had the opportunity to work with him.

I am also thankful the Hani Qadumi foundation, Amman, Jordan, for their support, as well as An-Najah National University, Nablus, Palestine, for giving me the opportunity to study abroad.

I would like to thank my colleagues Ashraf Nijim and Mohammad Daoud for their moral and occasional technical support.

Last, but certainly not least, I would like to thank my mother and father, sisters and brothers for their endless love and support.

Table of Contents

List of Tables

List of Figures

Chapter 1

Overview of Genetic Algorithms

1. Introduction

A Genetic Algorithm is a method of computation that simulates the mechanisms of natural selection. It is typically used to optimize functions that are intractable to solve or problems with large or unknown search spaces. The simple GA inspired by Holland [34] starts with a randomly generated population of individuals, each corresponding to a particular candidate solution to the problem at hand. Candidate individuals are allowed to evolve over a number of generations. The best individuals survive, mate and create offspring, the worst typically die or/and do not produce offspring. Typically evolving individuals over time leads to better populations.

This chapter is a review of GA theory, operations, representation, selection, reproduction, and mutation.

1.1 Genetic Algorithm Operation

This section provides an overview of the basic operation of a simple GA [34], including:

- Representation
- Fitness Evaluation
- Genetic Operators
- Parameters
- Initialization and Termination Criteria

1.1.1 Representation

We represent a possible solution to a problem in the form of a structure of variables. This structure is called a chromosome or individual. The set of values assigned to the decision variables represent a particular solution to the problem. A variable is called a gene and its value is called an allele. The set of possible solutions is called search space, and a particular solution in the population represents a point in that search space. In practice, the chromosome could be presented in various forms, including among others, strings, trees or graphs. In simple genetic algorithms, a string of Boolean

variables is used usually gray code. Gray code is an ordering of 2^n binary numbers such that only one bit changes from one entry to the next.

1.1.2 Fitness Evaluation

Each chromosome in the population is assigned a "fitness" value as a measure of how well the chromosome optimizes an objective function. The mapping that does that is called a fitness function. At each generation, the fitness value of each chromosome is calculated. The task of a GA is to find solutions that have high fitness values among the set of all possible solutions.

1.1.3 Genetic Operators

After the fitness of each individual is computed, a number of genetic operator (typically, two) and selection are applied to evolve new solutions.

Selection simulates the survival of the fittest. The fitter the chromosome the more likely it is to survive and reproduce. There are several selection techniques used in GAs. The one used by Holland is Roulette Wheel Selection (fitness-proportion selection).

As shown in figure 1, the idea behind roulette wheel selection is that each individual is given a chance to become a parent in proportion to its

fitness value. Roulette wheel selection can be implemented in the following way. Sort the individuals by fitness. Then, compute an aggregate fitness value (AF) for each individual in the sorted list, equal to the sum of the fitness values of all the individuals up to (and including) the target individual. Generate a random number r, between 0 and AF of the last individual in the list. Return the first population member with an AF that is greater than or equal to r.

The problem with roulette wheel selection is that one member can dominate all the others and get selected many times, and this leads to premature convergence. For that various fitness normalization techniques are used such as tournament selection. In tournament selection potential parents are selected randomly and a tournament is held to decide which of the individuals will be the parent. Tournament selection can be implemented by: Select two individuals at random. The individual with the highest fitness becomes the parent. Repeat to find a second parent. This is tournament selection with size two. Figure 1 illustrates the process of roulette selection using a population of 5 individuals.

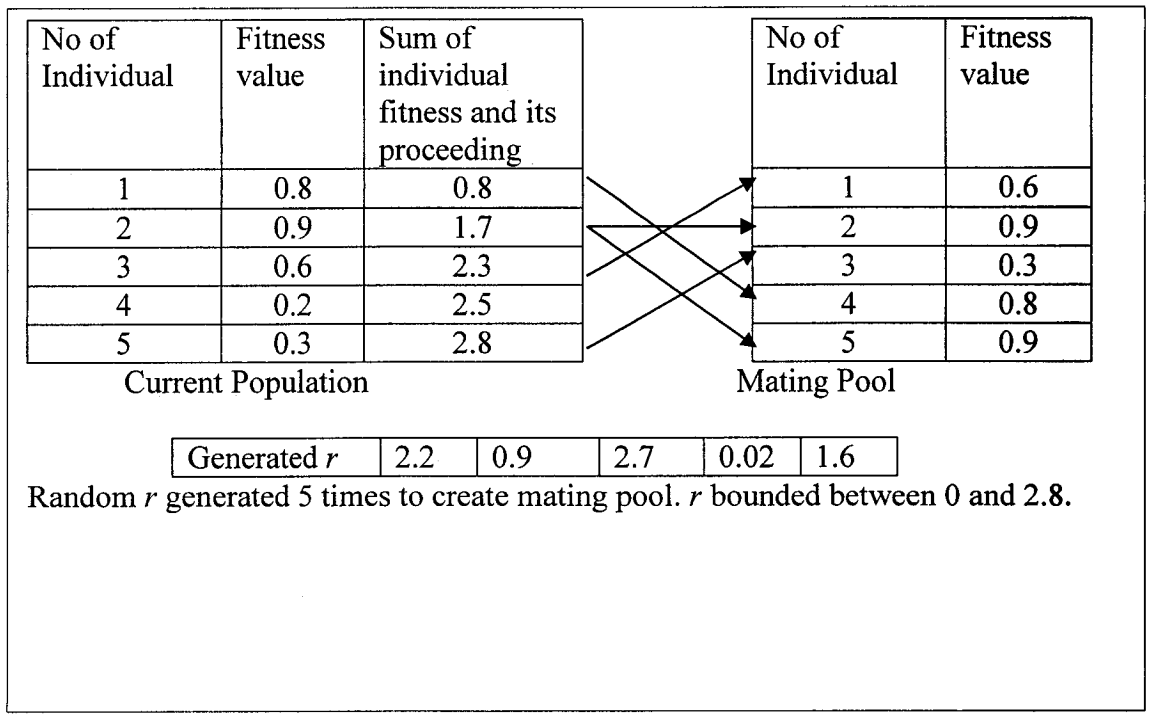


Figure 1: The Roulette Wheel Operator applied to a Population of 5 Individuals

Using selection will not result in the introduction of new chromosomes into the population. To generate new chromosomes, crossover and mutation are used. Crossover is inspired by the role of sexual reproduction in creating new and different individuals from two (or more) parents. The mechanism works by mixing the genes of one (good) solution with genes from other (also good) solutions. Two chromosomes with high fitness are selected randomly from the mating pool. These selected chromosomes are called parents. Then, two new chromosomes are created by pairing the parents. The new pair of chromosomes is called children.

There are many types of crossover ranging from single- and dual-point crossover to uniform crossover.

Figure 2 illustrates a commonly used crossover operator: single-point crossover. After two chromosomes are selected, a random variable r is chosen as a crossover point (here $r = 4$). The genes before r are taken from parent one and are copied into the first child and the genes after r from the second parent is copied into first child too. The second child is created in a similar manner – see figure 2. Crossover is applied with a high probability (>0.5) and is responsible for most of the search activity performed by a GA.

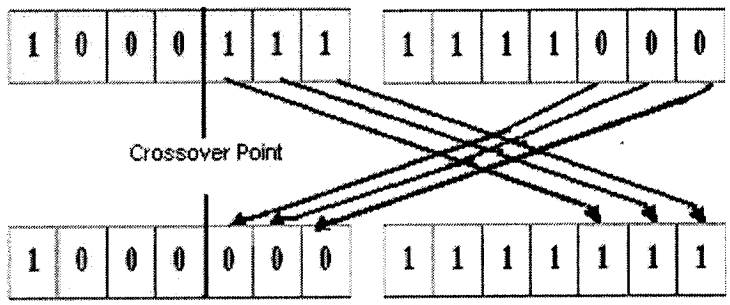


Figure 2: Single Point Crossover

Another commonly used crossover operator is uniform crossover. You randomly generate a template string of binary digits of equal length to that of the parent strings. Two children are created from two randomly chosen parents. The first child is created by copying the values of the first parent when the template value is 1 and from the second parent when the template

value is 0. The second child is created in a complementary matter as shown in figure 3.

Parent 1	1	0	1	1	1	0	1
Parent 2	1	1	0	0	1	1	0
Template	0	1	1	0	0	1	0

Child 1	1	0	1	0	1	0	0
Child 2	1	1	0	1	1	1	1

Figure 3: Uniform Crossover

Mutation is inspired by the role of mutation in natural evolution. With binary coded GAs, this means flipping a 0 bit to a 1 bit or vice versa. Mutation yields a new chromosome, which *mostly* less fit than its parent. Nevertheless, mutation used in order to prevent the GA from getting trapped in a local minimum. Mutation, hence, is often used as a means of diversifying converging population, but is usually used with a very low probability (<< 0.1). Figure 4 illustrates bit-flip mutation, as applied to binary-coded chromosomes.

Figure 4: Bit-Flip Mutation

1.1.4 Parameters

Before a GA is run, the user must specify some parameters. These include population size, crossover probability and mutation probability. These values are problem dependent. It is not an easy to specify the correct values of parameters for an optimization problem.

1.1.5 Initialization and Termination Criteria

A Genetic Algorithm starts by randomly generating a population of chromosomes; sometimes a percent of the initial population created using a number of good solutions.

After the initial population is created, a GA starts to evolve the population in order to find an optimal (or near-optimal) solution to the target problem. This evolution continues over a number of generations (using selection and the genetic operators) until a termination condition is met. There are several potential termination conditions, including: evolving an optimally fit individual and reaching a maximum number of generations.

1.2 Basic Genetic Algorithm Theory

This section reviews the basic theory of genetic algorithms, which was developed by John Holland. Holland's theory relies heavily on the notion that a good solution can be constructed by combining good pieces, called building blocks, from different solutions. Holland introduced the notion of schema to analyze the effects of the GA operators on these pieces (or sub-solutions). The results of his analysis are summarized in the Schema Theorem.

A schema is a similarity template that represents a set of solutions from the search space. In the context of binary alphabets, a schema is a string over the ternary alphabet {0,1,*}. The star (*) symbol represents a "don't care". For example, schema H = 1**0* represents all strings that have a 1 in the first position and a 0 in the fourth position. Strings 10101 and 11100 are members (or instances) of schema H, but 11111 and 00100 are not. Two important definitions are the *order* and the *defining* length of a schema. The order is the number of fixed-positions (ones and zeroes) in a schema. The defining length is the distance between the two outermost fixed positions. For example, schema H = 1**0* has order 2 and defining length 3. Holland quantified mathematically how the numbers of representatives of

a schema change when going from one generation to the next, and summarized it in the schema theorem:

$$m(H,t+1) \geq m(H,t)\frac{\bar{f}(H,t)}{\bar{f}(t)}\left(1-P_c\frac{\partial(H)}{\ell-1}-P_m \mathrm{O}(H)\right) \qquad (1)$$

where:

$m(H,t)$ is the number of instances of schema H at time t;

$\bar{f}(H,t)$ is the average fitness of the instances of schema H at time t;

$\bar{f}(t)$ is the average fitness of the population at time t;

P_c is the probability of crossover;

$\partial(H)$ is the defining length of schema H;

ℓ is the string length;

P_m is the probability of mutation;

$\mathrm{O}(H)$ is the order of schema H.

Holland's schema theorem is written assuming fitness-proportion selection, and single point crossover. The overall lesson of the schema theorem is that highly fit schemata that are not too disrupted by the variation operators, those that are of low order and of short defining length, tend to

grow from generation to generation. A more general version of the schema theorem can be written as:

$$M(H,t+1) \geq m(H,t)\Phi(H,t)(1-\varepsilon(H,t)) \tag{2}$$

The effect of selection is given by the reproduction ratio $\Phi(h,t)$ which is equal to $\frac{\bar{f}(H,t)}{\bar{f}(t)}$ and the effect of the variation operators is given by the disruption factor $\varepsilon(H,t)$. Overall, a schema can grow or decay according to the net growth factor defined by:

$$\Phi(H,t)\big(1-\varepsilon(H,t)\big) \tag{3}$$

In intuitive terms, this theorem says that the number of instances of a building block increase (decrease) when the observed fitness in the current schema H is above (below) the observed population average.

1.3 Criticisms of the Schema Theorem

The Schema Theorem has traditionally formed the basis for theoretical analyses of Genetic Algorithms. It provides insights into the nature of the evolutionary process. Holland's theory asserts that genetic algorithms work

by combining highly fit sub-solutions (or "building blocks"). Unfortunately, the simple genetic algorithm can only process well a small fraction of these building blocks, those whose genes are located close to each other in the chromosome. For that reason, the schema theory does not work in cases of "deception". Deceptive problems have two characteristics that cause genetic algorithm serious difficulties: the global solution is isolated, and information misleads the genetic algorithms to sub-optimal solutions. In this case, low-order schemata that are highly fit do not combine to form high-order schemata that are also highly fit.

Also, the Schema Theorem fails to predict the behaviour of small populations. For example, consider the two competing schemata: 1*** and 0***. Suppose that, on average, the strings belonging to 1*** have higher fitness than the strings belonging to 0***. Clearly, the genetic algorithms should prefer strings of the form 1***, but it can sometimes prefer 0*** because the population might be too small to properly sample the competing schemata. For all of these reasons (and more), the schema theorem has come under increasing criticism from serious GA researchers such as Vose [55], Grefenstette *et al.* [25] and Muhlenbein [39].

Two obvious limitations of the schema theorem restrict its usefulness. First: the schema theorem does not provide adequate characterization of the

genetic search. The inexactness of the inequality is such that if one were to try to use the schema theorem to predict the representation of a particular schema over multiple generations, the resulting predictions would in many cases be useless or misleading Vose [55]. Second: the observed fitness of a schema at time t can change dramatically as the population concentrates its new samples in more specialized sub-partitions of schema Grefenstette *et al.* [25].

Chapter 2

A Review of Parameter-less Genetic Algorithms

2.1 Introduction

The performance of a GA depends on a number of factors, such as candidate solution representation, and fitness evaluation and manipulation: *via* crossover and mutation. One of the main difficulties that a user faces when applying a genetic algorithm is to decide on an appropriate set of parameter values. Both crossover and mutation have parameters - probability of crossover P_c and probability of mutation P_m - that require initialization and adjustment. For a given problem, these parameters, as well as the size of the population of candidate solutions (S), require careful manual optimization, often done through trial and error. Several studies have shown that the GA can tolerate some variation in its parameter values without affecting much its overall performance. But even though a GA can robust in respect to some of its settings, that does not mean that a GA will perform optimally (or even near-optimally) regardless of the way it is set up.

Choosing a proper set of parameter values is not an easy task due to its dependency on many aspects of the particular problem being solved; such

as dimensionality and density of the search space and shape of the fitness surface. Many of these features are not available or computable prior to application of the GA; there are no formal guidelines for setting parameters up for any GA applied to an arbitrary problem. This diminishes the autonomy of GAs, and renders them much less attractive to potential users, such as engineers, that are not experts in GAs, and view them as nothing more than a tool for solution or optimization.

This chapter surveys the most important research efforts expended to date in pursuit of GAs that require no manual tuning of their parameters, and are thus called Parameter-less GAs (or pGAs). We divide them into two categories:

- Parameter Tuning;
- Parameter Control.

Parameter tuning involves finding good values for the parameters on a variety of test problems before the GA is run, and then using these values during the GA run. Parameter control includes several techniques that change or adapt the parameter values as the search progresses. Below, we review each category in detail.

2.2 Parameter Tuning

De Jong 1975

In 1975, De Jong [20] empirically investigated various combinations of parameter values using a set of five test functions. These functions included continuous and discontinuous functions, convex and non-convex surfaces, unimodal and multimodal functions, and deterministic and noisy functions. Since then, De Jong's functions have been used by researchers to test and compare various types of Genetic Algorithms.

De Jong tested the influence of four parameters: population size, crossover probability, mutation probability, and generation gap. Generation gap allowed him to study the effect of overlapping populations. De Jong did his study by applying a simple GA with roulette wheel selection, single-point crossover, and bit flip mutation. He observed that larger populations returned good results in the long-term. On the other hand, smaller populations responded faster, but sometimes converged prematurely. Mutation was needed to restore lost alleles and to explore new search areas, but its probability should be low; otherwise a GA turns to a random search engine.

Overall, the De Jong discovered a set of parameter values, which were good for the classes of test functions that he used. The following set of

parameters gave good performance: population sizes in the range 50-100, crossover probability of 0.6, and mutation probability of 0.001. These parameter values have been used often by researchers, and have thus become a kind of "standard" set. Unfortunately, this set of parameter values is not optimal for all application domains. Subsequent research has shown that using these settings blindly can be a serious mistake.

Grefenstette 1986

In 1986, Grefenstette [24] used a meta-GA to find a combination of parameter values using De Jong's test functions. The set of parameter values he found are: population size of 30, a crossover probability of 0.95, and a mutation probability 0.001.

The aforementioned studies suggested a low value for mutation probability, proposed double-digit values (< 100) for population size, and used high (> 0.5) values for crossover probability. Although none of these researchers were able to prove that their sets were optimal for every optimization task, many GA users viewed their results as sound empirically founded guidelines.

Schaffer *et al.* 1989

In 1989, Schaffer *et al.* [43] investigated various combinations of parameter settings on a set of test functions. They used De Jong's functions, but included five more functions. They used a simple GA with roulette wheel selection, two-point crossover and bit flip mutation. For each combination of parameter values, ten independent runs were performed. They observed a reverse relation between mutation probability and population size: high mutation probability is needed for smaller populations and *vice versa*. Also, they observed that the following set of parameter settings performed well: population sizes in the range of 20-30, crossover probability in the range of 0.75-0.95, and mutation probability in the range of 0.005-0.01.

2.3 Parameter control

In Parameter Control, one starts with certain initial parameter values; possibly the De Jong or Grefenstette's sets or some amalgamation thereof. These initial values are adjusted, during run-time, in a number of ways. The manner in which the values of the parameters are adapted at run-time is the basis of Eiben's classification. Eiben *et al.* [21] classified Parameter Control into three different sub-categories:

- Deterministic Control;
- Adaptive Control;
- Self-Adaptive Control.

Parameter adaptation techniques have two main advantages. First, the user does not have to specify in advance the values of the parameters. Second, time-dependent values for the parameters may actually be beneficial to the search. Below is a review of these parameter adaptation methods.

2.3.1 Deterministic Control

In this type of Parameter-less GAs, the values of the parameters are changed, during a run, according to a heuristic formula, which usually depends on time (i.e. number of generations or fitness evaluations).

Fogarty *et al.* 1989

Fogarty *et al.* [22] experimentally examined dynamical mutation probability control for genetic algorithms and he proposed to change the probability mutation in line with equation (4)- t is the generation number.

$$P_m = \frac{1}{240} + \frac{0.11375}{2^t} \tag{4}$$

The probability of mutation decreases exponentially over generations; a small constant value was added to prevent mutation from decreasing to zero in later generations. The researchers found that varying the probability of mutation over generations significantly improved the performance of the genetic algorithm. They examined the effect of applying this formula to a generational genetic algorithm with single-point crossover, a probability of crossover equal to 0.95.

Hesser *et al.* 1991

Hesser *et al.* [30] theoretically derived a general formula for probability of mutation using the current generation number, in addition to a number of constants (C_1, C_2, and C_3) used to customize the formula for different optimization problems. Unfortunately, these constants are hard to compute for some optimization problems. In equation (5) n is the population size, l is the length of a chromosome (in bits), and t is the index of the current generation.

$$P_m = \sqrt{\frac{C_1}{C_2}} \frac{\exp(-C_3 t / 2)}{n\sqrt{l}} \tag{5}$$

Muhlenbein 1992, and Back 1993

Independently of each other, Mouhlenbein [40] and Back [7] did a theoretical investigation of the effects of the mutation operator in a simple (1+1) evolutionary algorithm. The notation (1 + 1) is borrowed from evolutionary strategies (which are similar to GAs, but use real numbers and matching operators, instead of bit strings). A (1+1) GA is an algorithm that sees single parent chromosomes each producing a single child by means of mutation. Hence, the best of parent and child is passed to the next generation. Proportional selection is used and no crossover is used in their study. Both Muhlenbein and Back concluded that for a fixed mutation rate throughout the run, the optimal mutation probability in the case of a unimodal problem is $1/l$, where l is the chromosome length.

In other studies, Back [10] proposed a general formula for P_m, one that is a function of both generation number (t) and chromosome length (l). The formula is presented as equation (6); T is the maximum number of generations allowed in a GA run. Excellent experimental results where obtained for hard combinatorial optimization problems.

$$P_m = \left(2 + \frac{l-2}{T-1} t\right)^{-1} \tag{6}$$

The mutation probability value in all formulae presented above is handled as a global parameter, i.e. one mutation probability value P_m is valid for all chromosomes of the population. All formulae presented above are variations on a single theme presented symbolically by $1/t$, where t is the generation number. In this theme the probability of mutation is initially very high, but is quickly reduced to a low and reasonably stable value. This agrees with common sense, as most GAs go through a short and frantic period of locating areas of interest on the fitness surface, followed by a lengthy and deliberate exploration of those locales (mainly *via* crossover). Naturally, random search, and hence mutation, are ineffective methods of exploration of large spaces. This simple fact leads to the incorporation of $1/l$ (and variants) into many formulae for P_m – l is the length of the chromosome, which is a direct reflection of the dimensionality of the search space.

Not only is the need for manual tuning of P_m eliminated, but also the performance of GAs is much improved by the use of time-dependant formulae for P_m.

2.3.2 Adaptive Control

In this mode of parameter control, information fed-back from the GA is used to adjust the values of the GA parameters, during runtime. However, (as opposed to self-adaptive control) these parameters have the same values for all individuals in the population. This topic has been investigated since the early days of the evolutionary computation field.

Rechenberg 1973

Rechenberg [41] empirically introduced the 1/5 successful rule to (1+1) algorithms. He presented equation (7) as rule for determining how severely to mutate fixed-length real-valued vector representations using Gaussian noise:

$$\sigma = \begin{cases} \dfrac{\sigma}{c} & , \quad if \quad p > 1/5 \\ \sigma . c & , \quad if \quad p < 1/5 \\ \sigma & , \quad if \quad p = 1/5 \end{cases} \tag{7}$$

where p is the relative frequency of successful mutations, measured over each generation. A choice of $c = 0.817$ was theoretically derived by Schwefel [46]. After each generation, the ratio of the number of successes (successful mutations) to the total number of trials (mutations) is obtained. If

the ratio is greater than 1/5, increase the variance; if it is less than 1/5 decrease the variance.

Essentially, this heuristic codifies the valid association that if the ratio of successful mutations is larger than 1/5 then the distribution of offspring is too focused and should be broadened. Likewise, if the ratio of successful mutations is less than 1/5 then the distribution of offspring is spread over too large an area and should be more focused. This heuristic is useful in smooth multimodal environments of the type well studied by the Evolutionary Strategies community but would be less applicable in discontinuous or extremely irregular fitness surfaces.

Bryant *et al.* 1995

Bryant *et al.* [16] tried to adapt crossover and mutation probabilities in a steady-state genetic algorithm. They updated the probabilities of crossover and mutation (from initial values) according to the contribution of each operator to the generation of fit individuals. Bryant *et al* mechanism updates the operator probabilities after the generation of each new individual. To do this, a new data structure binary tree associates with each individual, and it maintains a queue that records the operators' contributions over most recent new individuals.

In Bryant's *et al.* algorithm two individuals are selected per one round. If these two individuals go under crossover one child will be created. If the children fitness greater than current population average, Bryant's mechanism assigns credit to crossover operator. It is called immediate operator credit. Mutation operator assigned credit too if it creates individuals with fitness greater than current average fitness. The mechanism records also the credit to the operator that generated the individual's parent. Therefore, Bryant *et al.* attaches operator tree with each individual. When the algorithm applies crossover or mutation, the operator copies entries from the parental operator tree into offspring's operator tree. When the GA generates an improved individual, Bryant's *et al.* mechanism scans the individual's operator tree to update the operator probability. Crossover probability p_r is updated according to the following formula:

$$P_r = \frac{C_r / N_{cr}}{C_r / N_{cr} + M_r / N_{mr}} \tag{8}$$

where C_r is the credit of the immediate crossover operator, N_{cr} is the total amount of crossover credit recorded in the tree, M_r is the credit of the immediate mutation probability, and N_{mr} is the total amount of mutation credit recorded in the tree. Mutation probability is $1\text{-}p_r$.

Bryant *et al.* tested their mechanism using a steady-state genetic algorithm that does not allow duplicate individuals in the population. Mutation and crossover each produced one offspring. This mechanism shows good performance in optimizing mathematical functions and minimizing tour length in the traveling salesman.

Hinterding *et al.* 1996

Hinterding *et al.* [32] ran three populations simultaneously. These populations had an initial size ratio of 1:2:4 or, in absolute terms: 50, 100 and 200. A mutation function uses the best fitness to update population sizes. Let us denote the best fitness found after a set number of function evaluations in population one as *P1*, population two as *P2*, and for population three as *P3*. Then the population sizes are updated according to two mechanisms. First if the best fitnesses converge:

If *P1* and *P3* have the same best fitness then size (*P1*) = size (*P1*)/2, and size (*P3*) = size (*P3*)*2

If *P1* and *P2* have the same best fitness then size (*P1*) = size (*P1*)/2

If *P2* and *P3* have the same best fitness then size (*P3*) = size (*P3*)*2

Second, where the best fitness values for *P1*, *P2*, and *P3* are distinct. The following set of rules are used to update the population sizes:

P1 P2 P3: move right

P1 P3 P2 or *P2 P1 P3*: compress left

P2 P3 P1 or *P3 P1 P2*: compress right

P3 P2 P1: move left

Theses cases are ordered by best fitness, smallest on the left, largest on the right. The adjustment operator 'move' and 'compress' are defined as:

Move right: size (*P1*) = size (*P2*); size (*P2*) = size (*P3*); and size (*P3*) = size (*P3*)*2

Move left: size (*P1*) = size (*P1*)/2; size (*P2*) = size (*P1*); and size (*P3*) = size (*P2*)

Compress left: size (*P1*) = (size (*P1*) +size (*P2*))/2; rest unaltered

Compress right: size (*P3*) = (Size (*P2*) +size (*P3*))/2; rest unaltered

Using these rules the researchers tried to maximize the performance of the middle population size. A steady-state genetic algorithm is used with tournament selection (with a tournament size of two). A bit string representation is used for chromosomes. Crossover and mutation are applied independently, i.e. new individuals are produced either through crossover or through mutation, but never by both. This mechanism returns good performance in optimizing multimodal functions.

Schlierkamp *et al.* 1996

Schlierkamp *et al.* [44] focused their efforts on adapting the size of the population. It is based on a competition between subpopulations. They simultaneously evolved a number of populations with different sizes. After each generation, the subpopulation with the best maximum fitness is stored in a quality record. After a specific number of generations (called evaluation interval), the population size of each group is modified according to its quality record. Normally, the size of the group with the best quality is increased, and the sizes of all other groups are decreased. In addition after a number of generations (called immigration interval) the global best individual is copied into the other groups. They used real-coded representation. Experimentally, they showed impressive results in optimizing difficult multimodal functions.

Thijssen 1997

Thijssen [14] used steady state genetic algorithms to adapt crossover probability and population size. He ran several subpopulations of equal size each with its own crossover operator. If one crossover is better than the other, then its subpopulation should increase in size according to a migration mechanism between subpopulation. This is explained below.

The average fitness f_i of every subpopulation is monitored after a set number of function evaluations. Second, the differences in relation to the subpopulation with the largest average fitness are calculated:

$$\Delta f_i = f_{\max} - f_i \tag{9}$$

where f_{max} is the average Fitness of the best subpopulation. Defined f_{min} is as the average fitness of the worst subpopulation. Then If $f_{max} = f_{min}$ there will be no migration, where. If $f_{max} > f_{min}$ there will be migration between subpopulations. Δf_i^- is defined to determine how much individuals should be taken from subpopulation i to the migration:

$$\Delta f_i^{'} = \frac{\Delta f_i}{f_{\max} - f_{\min}} \tag{10}$$

These Δf_i^- are multiplied with a constant c, which is a parameter of the mechanism. Then $\Delta f_i^- c$ multiplied by subpopulation i size is the number of individuals to be taken from subpopulation i to migration pool. Equation (10) gives subpopulation with a bad average fitness to migrate more individuals to the immigration pool than subpopulations with a better average fitness. Finally redistribute the individuals in migration pool back to the subpopulations in random.

Harik *et al.* 1999

Harik *et al.* [29] ran multiple populations simultaneously. The idea is to establish a race among populations of various sizes. Harik *et al.* allocates more time to those populations with higher maximum fitness, and firing new populations whenever older populations had drifted towards suboptimal (search) subspaces. See section 3.2.4 for details

Annunziato *et al.* 2000

Annunziato *et al.* [4] asserted that an individual's environment contains useful information that could be used as a basis for parameter tuning. They used a trip-partite scheme in which a new parameter (meeting probability) influences the likelihood of meeting between any two individuals, which (if they meet) can either mate or fight- see section 3.2.3 for details.

2.3.3 Self-adaptive Control

These GAs use parameter control methods that utilize information fed-back from the GA, during its run, to adjust the values of parameters attached to each and every individual in the population.

This technique was first used by Schwefel in an Evolutionary Strategy [46], where he tried to control the mutation step size. Each chromosome in

the population was combined with its own mutation variance, and this mutation variance is subjected to mutation and crossover, as was the rest of the chromosome.

Arabas *et al.* 1994

Arabas *et al.* [3] defined a new quantity called Remaining Life Time (or RLT) to use as part of a new mechanism for controlling population size. Every new individual is assigned a RLT variable. Each time a new individual is created, an RLT value is assigned to it using the formula below:

$$RLT(i) = \begin{cases} MinLT + \eta \dfrac{WorstFit - fitness(i)}{WorstFit - AvgFit} & if \quad fitness(i) \geq AvgFit \\ \dfrac{1}{2}(MinLT + MaxLT) + \eta \dfrac{AvgFit - fitness(i)}{AvgFit - BestFit} & if \quad fitness(i) < AvgFit \end{cases} \qquad (11)$$

$$\eta = \frac{1}{2}(MaxLT - MinLT)$$

$MinLT$, $MaxLT$ are minimum and maximum remaining life time, they are set to 1, and 11 respectively; $WorstFit$, $BestFit$ are the worst, and best individual fitness in the population; $AvgFit$ is the average fitness of the population; $fitness(i)$ is the fitness of the i^{th} chromosome.

After each generation, RLT for individuals are decremented by one except the RLT for the maximum fitness. Once the RLT of an individual reaches 0, it dies (is removed from population).

Srinivas *et al.* 1994

Srinivas *et al.* [50] varied P_c, and P_m to prevent premature convergence of the GA to local optimum. They used the relation between population average fitness f' and the maximum fitness value $f_{\max}$ as an indicator for population convergence. The difference $f_{\max} - f'$ decreases when the GA converges to any optimum (local or global). Just to be on the safe side, we can assume that when $f_{\max} - f'$ goes down, the GA is converging towards a local optimum, and as such both P_c and P_m should be increased. In addition, Srinivas *et al.* used the fitness value of each individual in the population to vary its own P_c and P_m: fitter individuals were less likely to be mutated or crossed-over. So in summary, each individual has it is own values of P_c and P_m based on a) its own fitness f and b) a population statistic $f_{\max} - f'$. The expressions that used for P_c and P_m are:

$$P_c = k_1(f_{\max} - f_m)/(f_{\max} - f') \qquad P_m = k_2(f_{\max} - f)/(f_{\max} - f') \tag{12}$$

f_m is fitness of the fitter of the two selected individuals for crossover, k_1 and k_2 are mechanism parameters limited to [0,1].

For an individual with less than average fitness, P_c and P_m are constrained to:

$$P_c = k_3 \qquad , f_m \leq f'$$
$$P_m = k_4 \qquad , f \leq f' \tag{13}$$

The values of k_1, k_2, k_3, and k_4 are 1.0, 0.5, 1.0, and 0.5, respectively. A generational GA was used with single-point crossover, and roulette wheel selection is used for selecting parents. Experimentally, they showed a good result for complex multimodal functions.

Back *et al.* 2000

Back *et al.* [13] extended Schwefel's [46] work in Evolution strategies to GAs. They tried to adapt all three parameters of evolution: P_m, Pc and *S*. At the end of each chromosome, extra bits are added to store both P_m and P_c. P_m is allowed to take values from [0.001, 0.25] and P_c from [0, 1]. See section 3.2.2 for details.

Chapter 3

Experimental Setup

3.1 Introduction

For the purpose of our research, we choose five frequently used GAs. Each one of them belongs to a category of Parameter Control GAs. These algorithms are: Traditional Steady-State Genetic Algorithm (TSSGA) [13], Self-Adaptive Mutation Crossover and Population Size for Genetic Algorithms (SAMXPGA) [13], Adaptive Evolutionary Algorithm via Reproduction and Competition (AEARC) [4], Adaptive Population Size Genetic Algorithm (APSGA) [29] and Deterministic Intelligent Mutation Rate Control Canonical Genetic Algorithm (DIMCCGA) [10]. To compare the performance of the five GAs, we examined the five parameterless GAs using the same test functions used in [13]. Namely De Jong [20], Rosenbrock [42], Ackley [1], Rastrigin [11], and a fully deceptive function [11]. This set of test functions have:

- Problems resistant to hill-climbing;

- Nonlinear non-separable problems;
- Scalable functions;
- A canonical form;
- A few unimodal functions;
- A few multi-modal functions of different complexity with many local optima;
- Multi-modal functions with irregularly arranged local optima;
- High-dimensional functions.

All these functions have 10 dimensions; we used a bitstring of length 200, 20 bits/variable except for f_5, which uses 6 bits/variable. Uniform crossover is used for producing new individuals. For selection, a tournament of size two is used. Simple bit-flip mutation is applied. The GA represents the values of the variables using a Grey code scheme. The GAs runs terminates when the (known) optimum is found or the maximum number of fitness evaluations is reached. The maximum number of evaluations is 500,000 for all test functions.

3.2 Test Algorithms

3.2.1 Traditional Steady-State Genetic Algorithm (TSSGA)

This algorithm is not a true parameter-less GA, but we make as one by fixing the values of its parameters, without any preparatory tuning via trial and error. It comes from Back *et al.* [13]. It uses a variable mutation probability Pm of $1/l$, a fixed crossover probability P_c of 0.9, a fixed population size S of 60. The genome (or chromosome) is made of 200 bits (20 per dimension), plus 20 bits used to encode both P_m and P_c: these are included - though obviously not used - to ensure a level playing field between this and the other parameter-less GAs. Uniform crossover is used. For selection, a tournament is used for selecting two parents of two new individuals, which in turn replace the worst couple of individuals in the current population. Simple bit-flip mutation is applied after crossover. The GA represents the values of the variables using a Grey code scheme.

3.2.2 Self-Adaptive Mutation Crossover and Population Size for Genetic Algorithms (SAMXPGA)

Back *et al.* [13] extended Schwefel's [46] work in Evolution strategies to GAs. They tried to adapt all three parameters of evolution: P_m, P_c and S. At

the end of each chromosome, extra bits are added to store both P_m and P_c. P_m is allowed to take values from [0.001, 0.25] and P_c from [0, 1].

Mutation takes place in two steps. First, the bits that encode the mutation rate are themselves mutated (using their own value as P_m). Second, the new mutation probability is used to mutate the remaining bits (those encoding the candidate solution and P_c). For reproduction, two chromosomes are selected *via* tournament selection. The bits that encode the crossover rate P_c are decoded, and a random number r (<1) is compared to P_c. If r is less than P_c, the selected chromosome is ready to mate. If both chromosomes are ready to mate, two children are created by uniform crossover, then mutated and inserted in the next generation. On the other hand, if both chromosomes are not ready to mate then two children are created *via* mutation (as described above). If only one of the selected chromosomes wants to mate but the other does not, then a child is created by mutating the chromosome that does not want to mate; the other chromosome is put on hold until a willing unmatched partner is found in a future round of reproduction. As to S, this is decided by a scheme similar to Arabas *et al.* [3]- mentioned in chapter 2. The initial population has $S = 60$.

3.2.3 Evolutionary Algorithm via Reproduction and Competition (AEARC)

Annunziato *et al.* [4] introduced a dynamic environment dominated by reproduction and competition among chromosomes. Various environmental mechanisms are responsible for adapting the GA parameters. The environment is constrained by a maximum population size (M_p) set before the GA run. The mode of interaction between the chromosomes of a given population is determined largely by a new parameter called population density (or meeting probability P_m). This is equal to the size of the current population size C_p divided by M_p.

If and when two chromosomes meet then they can engage in either one of a) sexual reproduction (crossover): to produce two offspring; b) competition: in which the fitter individual survives; or c) asexual reproduction: in which identical or/and mutated offspring are created. The probability of crossover is $P_r = 1 - P_m$, and the probability of competition is $P_c = 1 - P_r$. For each generation, every (primary) individual in the population gets an opportunity to interact with another randomly chosen (secondary) individual form the same population. A randomly generated number r (<1) is compared to the meeting probability P_m of the current population.

If r is less than P_m then interaction occurs between the two individuals. In that case, another randomly generated number r_1 (<1) is compared to P_r. If $r_1 < P_r$ then two children are created by uniform crossover and inserted in the next generation with their parents. When the population reaches its maximum allowed size, then reproduction gets destructive in the sense that children will replace their parents if they are fitter than their parents, or else the parents stay.

If, on the hand, r is not less than P_m, then a single child is created by mutating the primary individual and hence inserting the result into the next population with its parent. As in the case of crossover, when the population reaches its maximum size, then mutation operation will replace the original individual with the mutated one if the mutated one is fitter, or else leave the original one in place.

3.2.4 Adaptive Population Size Genetic Algorithm (APSGA)

Harik *et al.* [29] ran multiple populations simultaneously. The idea is to establish a race among populations of various sizes. The algorithm starts with a small population, then runs it for 3 generations, then it fire a second population and runs it for 1 generation, then runs population one for 3 more

generations, then population two for 1 generation, and so on as per the schedule shown in table 1.

A base 4 counter is used to determine which population should be run. This counter is incremented, and the position of the most significant digit that changes during the increment operation is noted. That position indicates which population should be run. If at any point in time, a larger population has an average fitness greater than that of a smaller population or if a population converges, then the GA gets rid of that population, and then resets the counter. Overall, population i is allowed to run 4 times more generations than population $i + 1$, and each new population that gets fired is twice the size of the previous population.

The idea behind this mechanism is to allocate more function evaluations to smaller populations. Consequently, the smaller populations are more likely to converge faster than the larger ones. That is, smaller populations get a head start at the beginning, but if they start to drift too much, they will be caught by a larger population. When that happens, the smaller populations become inactive.

Fixed crossover probability is used, no mutation, and proportional selection.

Table 1: Mechanics of Harik *et al.* GA

Counter base 4	Most significant digit changed	Action
0		
1	1	run 1 generation of population 1
2	1	run 1 generation of population 1
3	1	run 1 generation of population 1
10	2	run 1 generation of population 2
11	1	run 1 generation of population 1
12	1	run 1 generation of population 1
13	1	run 1 generation of population 1
20	2	run 1 generation of population 2
21	1	run 1 generation of population 1
22	1	run 1 generation of population 1
23	1	run 1 generation of population 1
30	2	run 1 generation of population 2
31	1	run 1 generation of population 1
32	1	run 1 generation of population 1
33	1	run 1 generation of population 1
100	3	run 1 generation of population 3
101	1	run 1 generation of population 1
.	.	.
.	.	.

3.2.5 Deterministic Intelligent Mutation Rate Control Canonical Genetic Algorithm (DIMCCGA)

This GA comes from Back *et al.* [10]. In this algorithm, crossover is not used, the size of the population is fixed at 60 and the probability of mutation is updated in line with equation (6) chapter 2. During every generation, one chromosome is selected to be parent by tournament selection. Then one child is created by mutation and the better of both parent and child survives to the next generation.

3.3 Test Functions

3.3.1 De Jong Function

$$f_1(\bar{x}) = \sum_{i=1}^{n} x_i^2 \tag{14}$$

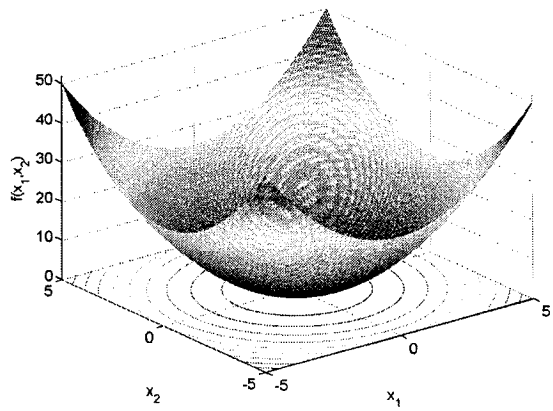


Figure 5: Fitness Surface of f_1

With n = 10; x_i , [-5.12, 5.12]. f_1 is continuous and unimodal. De Jong problem was designed to be very easy for a genetic algorithm. The function has global optimum at point zero.

3.3.2 Rosenbrock Function

$$f_2(\bar{x}) = \sum_{i=1}^{n-1} (100(x_i^2 - x_{i+1})^2 + (1 - x_i)^2) \tag{15}$$

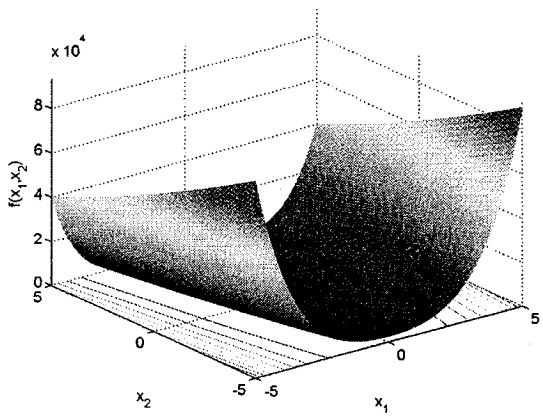


Figure 6: Fitness Surface of f_2

With n = 10; x_i , [-5.12, 5.12]. It is a standard test function in optimization which was proposed by Rosenbrock. f_2 has an apparently simple surface, but is quite hard to optimize due the nature of the local neighborhood of the global minimum. The function has global optima inside a long, narrow, parabolic shaped flat valley. To find the valley is trivial, however convergence to the global optima is difficult. The function has global optima for $x_i = 1, I = 1....n$.

3.3.3 Ackley Function

$$f_3(\bar{x}) = -20\exp\left(-0.2\sqrt{\frac{1}{n}\sum_{i=1}^{n} x_i^{\,2}}\right) - \exp\left(\frac{1}{n}\sum_{i=1}^{n}\cos(2\pi x_i)\right) + 20 + e \tag{16}$$

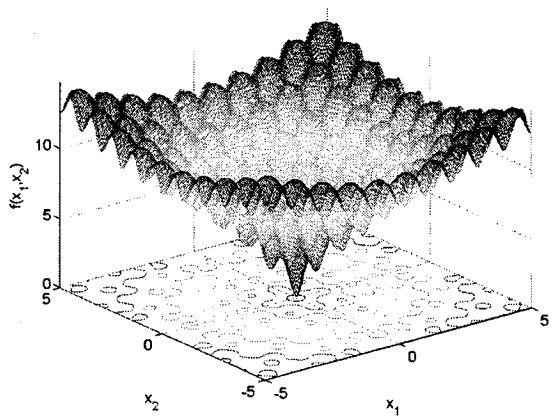


Figure 7: Fitness Surface of f_3

With n = 10; x_i , [-5.12, 5.12]. f_3 is a multimodal function with a global minimum located at the origin.

3.3.4 Rastrigin Function

$$f_4(\bar{x}) = 10n + \sum_{i=1}^{n} \left(x_i^{\,2} - 10\cos(2\pi x_i) \right) \tag{17}$$

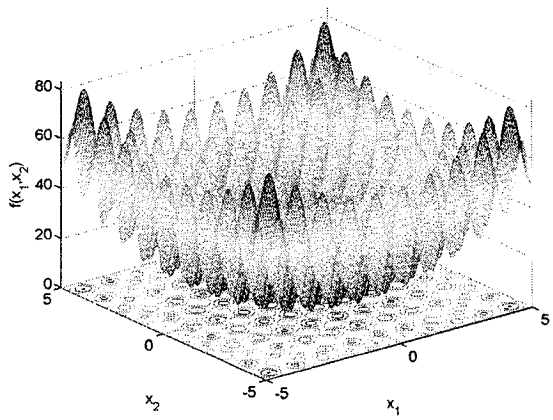


Figure 8: Fitness Surface of f_4

With n = 10; x_i, [-5.12, 5.12]. f_4 is another non-linear multimodal function. The main characteristic of this function is that, within the assigned variation range for the variables, it is characterized by 11^n local minima, and only one global minimum at point zero. It is therefore very easy for an optimizer t be trapped in local minima, which makes this function a suitable candidate to investigate the robustness of an optimization procedure.

3.3.5 Fully Deceptive Function

$$f_5(\bar{x}) = n - \sum_{i=1}^{n} \left\{ \begin{array}{lll} \frac{0.92}{4}(4 - x_i) & if & x_i \leq 4 \\ \frac{2.00}{4}(x_i - 4) & if & x_i > 4 \end{array} \right\} \qquad (18)$$

x_i *is the number of* 1's $\in$ *gene i*

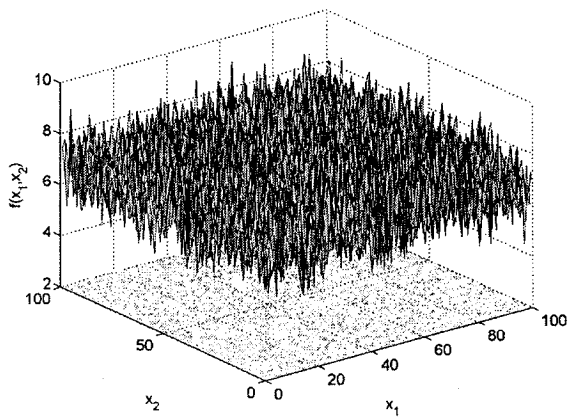


Figure 9: Fitness Surface of f_5

f_5 is the fully deceptive six-bit function. Deceptive problem was designed to push the genetic algorithm a way from the solution. The deceptive function used in our research is piecewise-linear functions of units. A unit x_i is defined as the number of 1s in a string and is considered to be the hamming distance of the string from the local optima. They depend only on the number of 1's in an individual and not on the positions of the 1's, In our case 10 units are used for f_5. Each unit contains 6 bits. Number of 1's bit in each

unit determines which part of equation (18) should be used. Deceptive function divides the search space into two peaks in the hamming space; one leads to the global optimum "all bits are 1s in the string" and the other to a local optimum "all bits are 0s in the string". F_5 is considered fully deceptive because all schemata with l-1 "l is string length" defined bits are misleading.

Chapter 4

Experimental Results

4.1 Introduction

Thirty runs were executed for every combination of test algorithm and test function. Hence, every result presented in the graphs and in the tables represents the average results of thirty runs. To compare the performance of the five different GAs (pGAs), several statistical measures and performance metrics are calculated.

4.2 Basic Statistics

The following statistics are defined:

- Percentage of Runs to Optimal Fitness;
- Average Number of Evaluations to Best Fitness and Coefficient of Variation (C.V.);
- Average Number of Evaluations to Near-Optimal Fitness;
- Average Best Fitness;

- Average Mean Fitness;
- Average Mean Population Size to Optimal Fitness;
- Average Maximum Population Size to Optimal Fitness;
- Average Mean Population Size to Near-Optimal Fitness;
- Average Maximum Population Size to Near-Optimal Fitness.

4.2.1 Percentage of Runs to Optimal Fitness

Each GA was run 30 times. This measure reflects the percentage of runs that were successful in converging to the optimal solution at or before 500 thousand (fitness function) evaluations.

4.2.2 Average Number of Evaluations to Best Fitness and Coefficient of Variation

Average number of evaluations is used as a measure for reliability. This measure represents the average number of evaluations that are required for a GA to achieve its best fitness value in a run. In cases where the best fitness is 1(fitness 1 is the fitness map of the global optimum), it serves as a measure of convergence velocity. Every run produces a different number of evaluations to best fitness. Therefore, Coefficient of Variation (C.V.) is calculated to measure the deviation between runs. Coefficient of Variation

(C.V.) is equal to the standard deviation of that set of evaluations, divided by the average.

4.2.3 Average Number of Evaluations to Near-Optimal Fitness

Near-Optimal fitness is defined as a fitness of 0.95. In cases where optimal fitness is not obtained, near-optimal fitness is the next best measure of convergence velocity. This measure is defined in the same way as the preceding measure, except that we substitute near-optimal for optimal.

4.2.4 Average Best Fitness and Standard Deviation (S.D.)

This is the average of the set of best fitness values achieved in all 30 GA runs. S.D. is standard deviation of that set. Naturally, this is a crucial measure; GAs that are able to achieve a best fitness of 1 (and reliably) are taken seriously; those that return best fitnesses of less than 1 (or 1 but inconsistently) are not as good.

4.2.5 Average Mean Fitness and Standard Deviation (S.D.)

This is the average of the set of average fitness values of population, achieved at the end of the 30 GA runs. S.D. is the standard deviation of that set.

4.2.6 Average Mean Population Size to Optimal Fitness

In a given run, the size of the population may differ from one generation to the next until (and after) the GA converges to the optimal value (if ever). In one run, the average size of all the populations preceding optimal convergence is called Average Population Size to Optimal Fitness (or APSOF). Every one of the 30 runs may return a value for APSOF. The average value for the set of APSOF values is the Average Mean Population Size to Optimal Fitness.

4.2.7 Average Maximum Population Size to Optimal Fitness

For each GA run, the largest population size prior to optimal convergence is stored in a set. The mean of that set is the average maximum population size to optimal fitness.

4.2.8 Average Mean Population Size to Near-Optimal Fitness

In a given run, the size of the population may differ from one generation to the next until (and after) the GA converges to the near-optimal value of 0.95 (if ever). In one run, the average size of all the populations preceding near-optimal convergence is called Average Population Size to Near-Optimal Fitness (or APSNOF). Every one of the 30 runs may return a value for

APSNOF. The average value for the set of APSNOF values is the Average Mean Population Size to Near-Optimal Fitness.

4.2.9 Average Maximum Population Size to Near-Optimal Fitness

For each GA run, the largest population size prior to near-optimal convergence is stored in a set. The mean of that set is the average maximum population size to near-optimal fitness.

Population size measures allow GA users to assess the memory requirements for a given GA. The smaller the size of the population required to get an optimally fit individual the better. This is because smaller populations require less memory. And, memory is a serious concern, still, if one is using large populations for real-world optimization and design problems.

The following set of tables reports the values for the various statistics described above for each one of the test algorithms as applied to all 5 of the test functions $f_1 - f_5$. Please note that, in the tables, *NA* stands for not available, and means that the associated GA failed to converge (on average) to an optimal or near-optimal fitness value. This was the case, for example, with all the test algorithms, when they were applied to f_5, which is the fully

deceptive function. Also, none of the GAs achieved optimal convergence on f_2; some, however, did manage to converge to near-optimal values.

Table 2: Results of Application of Test Algorithms on Test Function f_1

Function 1	TSSGA	SAMXPGA	AEARC	APSGA	DIMCCGA
Percentage of Runs to Optimal Fitness	100%	100%	100%	100%	100%
Ave. No. of Evaluations to Best Fitness	12,561	25,172	247,612	191,991	153,038
C.V.[1]	6.9	81.0	22.8	44.5	6.0
Ave. No. of Evaluations to Near-Optimal Fitness	3,540	2,160	26,520	21,760	42,960
Average Best Fitness	1	1	1	1	1
S.D.	0	0	0	0	0
Ave. Mean Fitness	0.9997	0.9605	0.353	0.9676	0.9041
S.D.	0.0009	0.049	0.0699	0.0156	0.0093
Ave. Mean Population Size to Optimal Fitness	60	8.8	76.9	122.2	60
Ave. Max. Population Size to Optimal Fitness	60	60	79.6	189.3	60
Ave. Mean Population Size to Near-Optimal Fitness	60	7.8	76.6	35.4	60
Ave. Max. Population Size to Near-Optimal Fitness	60	60	79.4	54.4	60

Table 3: Results of Application of Test Algorithms on Test Function f_2

Function 2	TSSGA	SAMXPGA	AEARC	APSGA	DIMCCGA
Percentage of Runs to Optimal Fitness	0%	0%	0%	0%	0%
Ave. No. of Evaluations to Best Fitness	500,000	500,000	500,000	500,000	500,000
C.V.[1]	0	0	0	0	0
Ave. No. of Evaluations to Near-Optimal Fitness	63660	217260	NA	NA	134160
Average Best Fitness	0.9879	0.9816	0.9482	0.9458	0.977
S.D.	0.014	0.0185	0.0044	0.0028	0.0077
Ave. Mean Fitness	0.98	0.7765	0.2518	0.9111	0.6387
S.D.	0.0127	0.2213	0.0648	0.0062	0.0168
Ave. Mean Population Size to Optimal Fitness	60	49.6	NA	NA	60
Ave. Max. Population Size to Optimal Fitness	60	182.1	NA	NA	60
Ave. Mean Population Size to Near-Optimal Fitness	60	27.9	NA	NA	60
Ave. Max. Population Size to Near-Optimal Fitness	60	85.4	NA	NA	60

Table 4: Results of Application of Test Algorithms on Test Function *f3*

Function 3	TSSGA	SAMXPGA	AEARC	APSGA	DIMCCGA
Percentage of Runs to Optimal Fitness	100%	100%	16.7%	100%	100%
Ave. No. of Evaluations to Best Fitness	20,304	44,547	491,031	197,959	252,547
C.V.[1]	48.2	61.5	4.6	41.5	5.3
Ave. No. of Evaluations to Near-Optimal Fitness	12,960	6,900	67,560	35,280	76,080
Average Best Fitness	1	1	0.9997	1	1
S.D.	0	0	0.0005	0	0
Ave. Mean Fitness	0.9979	0.9889	0.3305	0.9667	0.8629
S.D.	0.0042	0.0264	0.0638	0.0173	0.0145
Ave. Mean Population Size to Optimal Fitness	60	47.4	76.6	71.6	60
Ave. Max. Population Size to Optimal Fitness	60	87	79.7	115.2	60
Ave. Mean Population Size to Near-Optimal Fitness	60	6	76.6	37.4	60
Ave. Max. Population Size to Near-Optimal Fitness	60	60	79.5	58.7	60

Table 5: Results of Application of Test Algorithms on Test Function f_4

Function 4	TSSGA	SAMXPGA	AEARC	APSGA	DIMCCGA
Percentage of Runs to Optimal Fitness	100%	100%	23.33%	0%	53.33%
Ave. No. of Evaluations to Best Fitness	148,562	115,360	494,680	500,000	458,050
C.V.[1]	44.1	76.7	3.7	0	15.3
Ave. No. of Evaluations to Near-Optimal Fitness	248,580	121,800	NA	NA	NA
Average Best Fitness	1	1	0.8485	0.1233	0.8086
S.D.	0	0	0.2510	0.0298	0.2535
Ave. Mean Fitness	0.9973	0.9800	0.1966	0.1171	0.1257
S.D.	0.0056	0.0330	0.0967	0.0286	0.0092
Ave. Mean Population Size to Optimal Fitness	60	355.2	NA	NA	NA
Ave. Max. Population Size to Optimal Fitness	60	1318.2	NA	NA	NA
Ave. Mean Population Size to Near-Optimal Fitness	60	148	NA	NA	NA
Ave. Max. Population Size to Near-Optimal Fitness	60	568.5	NA	NA	NA

Table 6: Results of Application of Test Algorithms on Test Function f_5

Function 5	TSSGA	SAMXPGA	AEARC	APSGA	DIMCCGA
Percentage of Runs to Optimal Fitness	0%	0%	0%	0%	0%
Ave. No. of Evaluations to Best Fitness	500,000	500,000	500,000	500,000	500,000
C.V.[1]	0	0	0	0	0
Ave. No. of Evaluations to Near-Optimal Fitness	NA	NA	NA	NA	NA
Average Best Fitness	0.6197	0.6348	0.6055	0.6288	0.8066
S.D.	0.0352	0.0539	0.0318	0.0325	0.0811
Ave. Mean Fitness	0.6187	0.4645	0.3352	0.3762	0.5557
S.D.	0.0351	0.0919	0.0282	0.0109	0.1603
Ave. Mean Population Size to Optimal Fitness	NA	NA	NA	NA	NA
Ave. Max. Population Size to Optimal Fitness	NA	NA	NA	NA	NA
Ave. Mean Population Size to Near-Optimal Fitness	NA	NA	NA	NA	NA
Ave. Max. Population Size to Near-Optimal Fitness	NA	NA	NA	NA	NA

4.3 Evolution of Fitness and Diversity

Measuring fitness is easy when one is trying to optimize a mathematically described fitness surface. In our case, fitness is simply *1/ (1+ the value of the test function at a given point)*. This is the case because we are trying to find global minima rather than global maxima. The evolution of (maximum) fitness is the most important and most exhibited measure of dynamic

performance of a GA. However, a GA's ability to evolve optimally fit individuals much depends on the diversity of the population. The idea is that some level of diversity must be retained for as long as the GA is still searching for an optimal individual. Early loss of diversity leads to premature convergence, and late loss of diversity leads to stagnation near (but not at) the highest point in the fitness surface. Diversity, hence, is important, and measuring it, even more so.

We use entropy to measure population diversity in genetic algorithms. James *et al.* [36] used entropy to measure the convergence of the genetic algorithm. Jonathan *et al.* [37] used entropy to describe the dynamics of a number of genetic operations for some very simple problems.

In a typical genetic algorithm run, the initial population's neighborhood is very large, but as the optimization progresses, the neighborhood size shrinks to reflect the reduced uncertainty that we have in any given population member. In the best conceivable case, the global optimum is found by all population members and in this unique case, we have no uncertainty in any of our population members. Hence, in order to quantitatively assess the cumulative effect of uncertainty in our population, we can use entropy as a metric for population diversity.

While a continuous form of entropy measure exists, the discrete form is sufficient for estimating our metric. It is also simpler and far more intuitive to understand. We begin by reviewing that the uncertainty (u) associated with the value x_i for a random variable X is given by:

$$u(x_i) = \ln\left(\frac{1}{p(x_i)}\right) \tag{19}$$

Where P_i is the probability that $X=x_i$. The expected value of all of the entropy (H) associated with the random variable is given by the following expression:

$$H(X) = \sum_i p(x_i) \ln\left(\frac{1}{p(x_i)}\right) \tag{20}$$

If two random variables are distributed jointly, then the joint entropy H(X,Y) is given as:

$$H(X, Y) = \sum_j \left(\sum_i p(x_i, y_j) \ln\left(\frac{1}{p(x_i, y_j)}\right)\right) \tag{21}$$

In other words the joint uncertainty is computed from the joint distribution of the two random variables. Of course, if the random variables are independent, we have:

$$H(X, Y) = \sum_{j} \left(\sum_{i} p(x_i)\, p(y_j) \ln\left(\frac{1}{p(x_i)\, p(y_j)} \right) \right) \tag{22}$$

which is clearly:

$$H(X, Y) = H(X) + H(Y) \tag{23}$$

Finally we extend this result to an n-dimensional vector of independently distributed random variables. The entropy in this case is the sum of all of the individual entropies:

$$H(Z) = \sum_{j=1}^{n} H(X_i) \tag{24}$$

Finding the discrete entropy of a population on a fitness landscape can be done by first estimating the joint probability density function of the combined feature space (the domain of the fitness landscape). By first constructing a discrete partitioning scheme in the solution space, we are able to acquire density information about each of the resulting regions. From this, a frequency histogram can be tabulated and converted into a discrete probability density function. Under the assumption that the feature vectors are orthogonal in the solution space, we can assume that they are uncorrelated and have independent probability density functions. The discrete marginal probability density functions are then very easily tabulated

along each feature vector in the solution space giving us a means for computing the entropy for that particular subspace. The total entropy associated with the population distribution within the solution space on the fitness landscape is the joint uncertainty of all features of all population members. In this way, every member of the population will contribute to the overall uncertainty of the genetic algorithms ability to move the neighborhood of candidate solutions through feature space. It should be noted here that the base of the logarithms used in the computations here is unimportant, but is selected to be e for the purposes of comparison, as is partitioning scheme used for the initial probability density function estimation.

By measuring the final amount of diversity in a genetic algorithm's population through estimation of joint entropy, we should be able to make very clear statements about the consistency of its performance on a given fitness landscape type. If for a given fitness landscape, the resulting entropy is consistently high, then we know that the genetic algorithm that was used to move the population was not very effective on this type of surface. If the entropy is consistently low on a given fitness surface, then we know that the genetic algorithm that was used for the optimization was performing well. Having said this, we know that there are upper and lower bounds for the

metric. Of course the best case is where all of the population members find the same optimum point, resulting in no uncertainty in the population (H(X) =0). The worst case possible would be where all of the population members are uniformly spread across the fitness surface with equal probability. In the case of a uniform distribution of a given feature we have the following associated entropy, given $p(x_i) = 1/N$, where N the number of uniform partitions of the feature:

$$H(X) = \sum_{i=1}^{N} \frac{\ln(N)}{N} \tag{25}$$

$$H(X) = \ln(N) \tag{26}$$

Finally this gives us the upper bound on the joint entropy for the population:

$$H_u(Z) = \sum_{i=1}^{n} \ln(N) \tag{27}$$

$$H_u(Z) = n \ln(N) \tag{28}$$

where n is the number of features in the solution space. Our metric is now bounded:

$$0 \leq H(Z) \leq Hu(Z) \tag{29}$$

$$0 \leq \sum_{j=1}^{n} \left(\sum_{i=1}^{N} \mathrm{p}(x_{i,j}) \ln\left(\frac{1}{\mathrm{p}(x_{i,j})} \right) \right) \leq n \ln(N) \tag{30}$$

Summarizing, we propose to measure the resulting entropy in the population in order to judge its consistency and hence its effectiveness on a given fitness surface type. Computing the diversity of the population in this manner will give us a means for comparison through our proposed metric.

We present, below, paired plots for the evolution of both maximum fitness and entropy of population as a function of number of evaluations. Fitness/Entropy is represented on the vertical axis (using a linear scale), while the number of fitness evaluations is presented on the horizontal axis (using a logarithmic scale). Each GA was run 30 times and the plots represent average values for all the plots generated during all the runs. In all the plots, red stands for SAMXPGA; dark blue is for TSSGA; light green is for AEARC; magenta is for APSGA; and finally light blue is for DIMCCGA.

We make the following comments on the results below. Except for APSGA, entropy invariably starts at or about its highest value within the run, then either collapses relatively quickly (as is the case with TSSGA and

SAMXPGA), or much later (as is the case with DIMCCGA and AEARC). This is to be expected, as the first population is randomly generated.

In case of DIMCCGA mutation only used so it is like a random search. Therefore, high diversity is preserved in early stages until highly solutions dominated the population. That's why entropy collapses in much later stage. For AEARC Annunziato *et al.* [4] used a selection mechanism that gives equal opportunity for each individual in the population to be selected for reproduction. Weak individuals have a good opportunity to survive to the next generation. Therefore high diversity is preserved in the population which makes the entropy collapses in later stages of run. In contrast to the other GAs, the entropy of APSGA increases with time, except when applied to f_1. The reason is that when APSGA fails to optimally converge, it starts firing new populations in random and continues to so, until an optimally fit individual is evolved.

The manner in which the entropy of a given GA evolves: whether it starts from a low or high level; whether it decreases or increases initially; whether it collapses early or late; is characteristic of a given GA. It does not depend on the type of test function (being optimized). The entropy of two GAs (TSSGA and SAMXPGA) collapses well before the associated GA has approached near-optimal convergence. Though, TSSGA proves to be our

best performer, collapsing entropy is a concern in any GA: it is important that some measure of diversity is maintained until the global optimum is actually found.

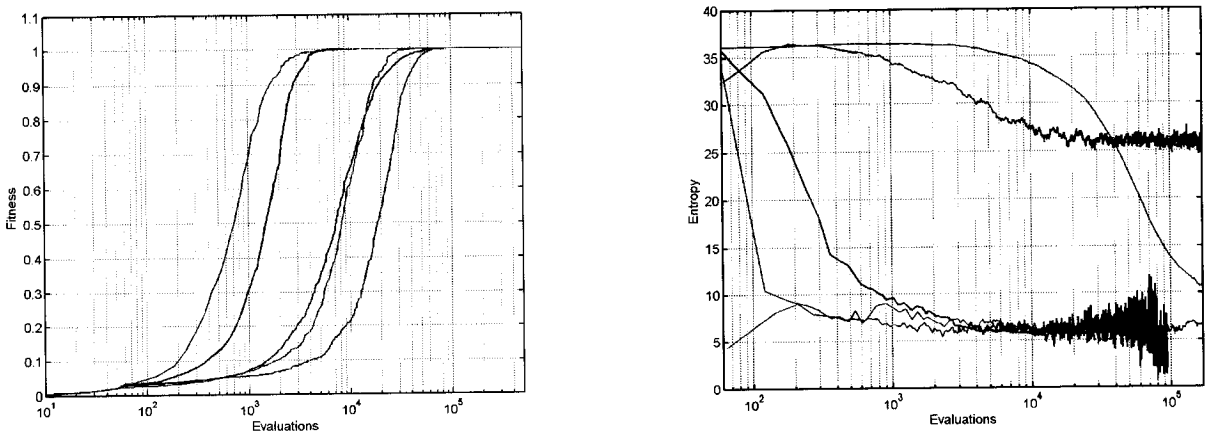


Figure 10. 10a Fitness (left) and 10b Entropy (right) for Test Algorithm f_1

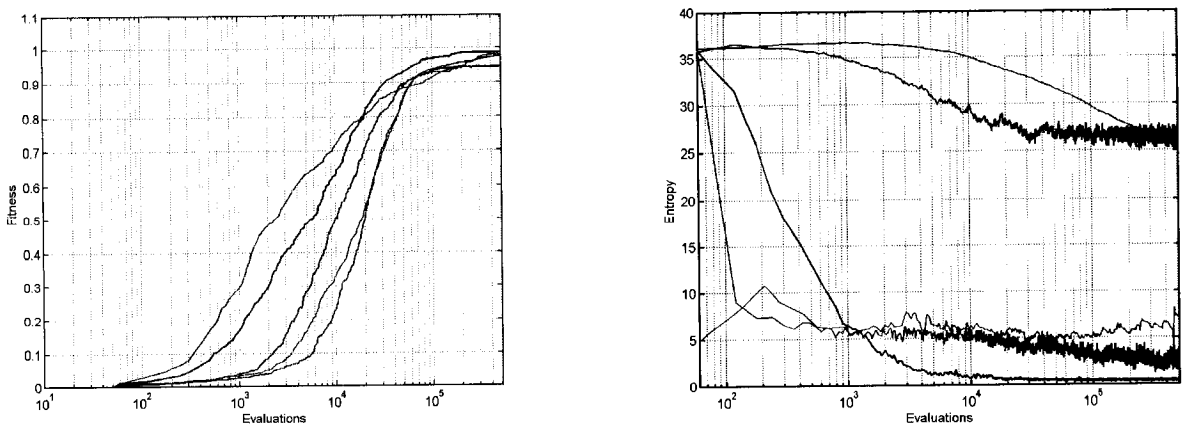


Figure 11. 11a Fitness (left) and 11b Entropy (right) for Test Algorithm f_2

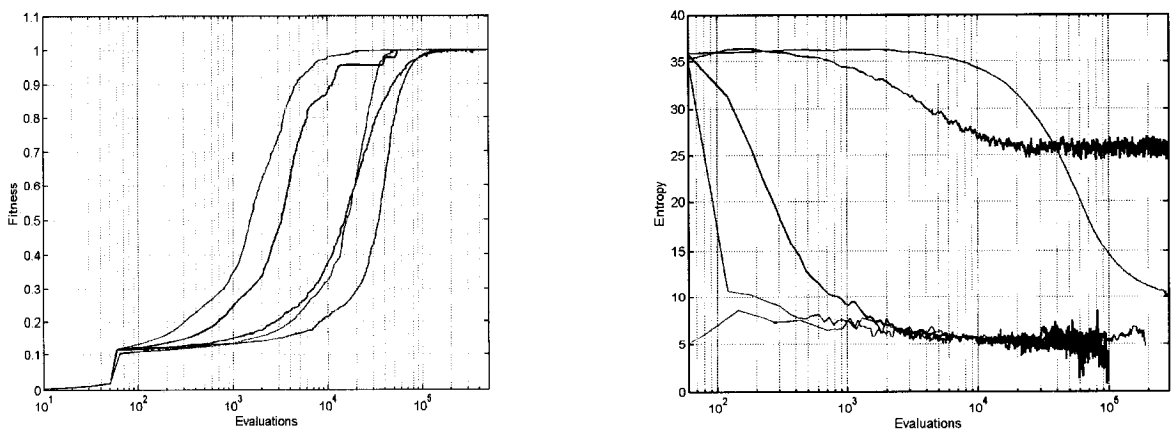


Figure 12. 12a Fitness (left) and 12b Entropy (right) for Test Algorithm f_3

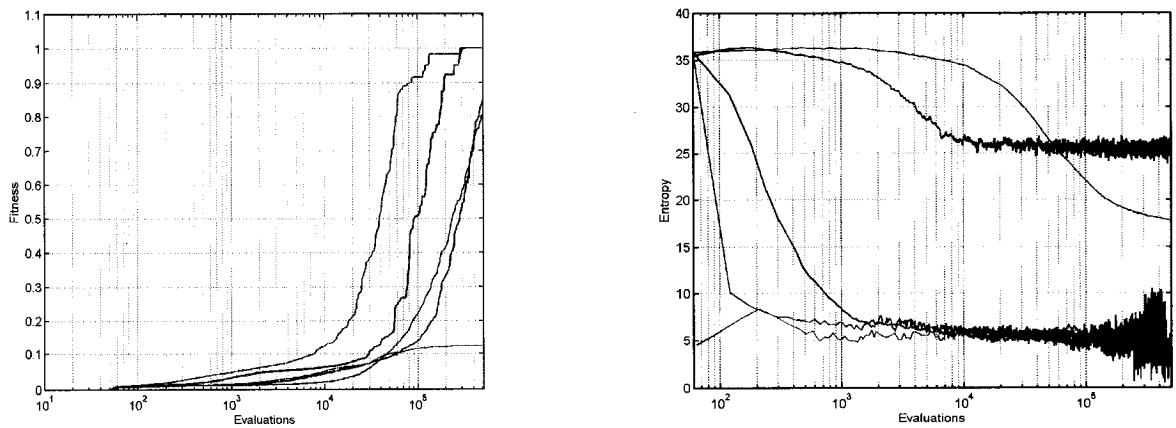


Figure 13. 13a Fitness (left) and 13b Entropy (right) for Test Algorithm f_4

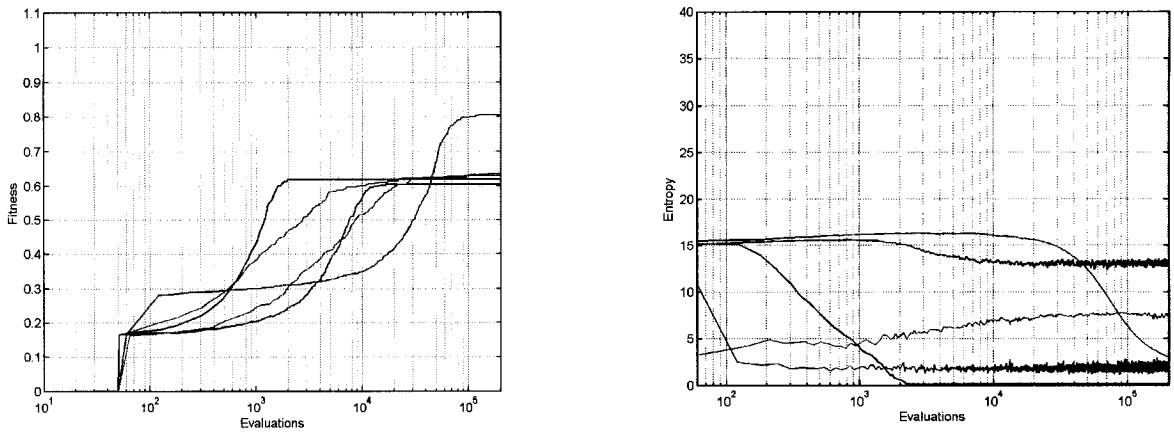


Figure 14. 14a Fitness (left) and 14b Entropy (right) for Test Algorithm f_5

Figure 10a compares the five pGAs on test function f_1. All the five pGAs reach an optimal solution within 500,000 evaluations. The fastest one is TSGA; it takes an average of 12,561 function evaluations in order to reach an optimal solution. The SAMXPGA takes an average of 2,160 function evaluations to reach a near-optimal fitness, while it takes an average of 25,172 function evaluations in order to reach the optimal solution. That is because SAMXPGA shows a good ability to find good parameter values in first few generations. However, once the algorithm comes close to convergence it takes much more time to adapt the parameter values. The third fastest pGA is DIMCCGA: it takes an average of 153,038 function evaluations to reach the optimal solution. While APSGA and AEARA take an average of 191,991 and 247,612 function evaluations, respectively, before reaching the global optimum.

Figure 11a compares the five pGAs on test function f_2. All the pGAs failed (in 30 runs) to reach the optimal solutions. Three pGAs: TSSGA, SAMXPGA and DIMCCGA reach near-optimal convergence and then stagnate at (or near) that level. AEARC and APSGA both stagnate before even reaching near-optimal levels.

Figure 12a shows a comparison of the five pGAs on test function f_3. TSSGA reached the optimal solution in an average of 20,304 function

evaluations. TSSGA is the fastest algorithms among all the five pGAs. Second in speed is SAMXPGA. It takes an average of 44,547 function evaluations to reach the optimal solution. In case of APSGA and DIMCCGA, they take an average of 197,959 and 252,547 function evaluations, respectively, in order to reach the optimal solution. AEARC reaches the optimal solution 5 times out of 30 runs.

Figure 13a shows a comparison of the five pGAs on test function f_4. Only SAMXPGA and TSSGA reached the optimal solution within 500,000 function evaluations, and they did so only 3 times. SAMXPGA is the fastest; it takes an average of 115,360 functions evaluations in order to reach the optimal solution. TSSGA takes 148,562 function evaluations. DIMCCGA and AEARC reach the optimal solutions 16 and 7 times (respectively) out of 30 runs. Finally, APSGA never reached the optimal solution in any one of the 30 runs.

In Figure 14a all the five pGAs did not reach the optimal solution on test function f_5. Function five is a fully deceptive function that leads the GAs away from the global optimum.

4.4 Metric and Rankings

4.4.1 Performance Metrics

In order to compare the performance of any number of parameter-less GA's – not just those used for in our research - one requires a standardized set of metrics that are

- Fair, i.e. algorithm- and platform-independent.
- Meaningful, i.e. reflect those properties of a GA that are relevant to the human user.

To ensure fairness, variables that may discriminate in favor of a particular type of GA must not be used. For example, number of generations is not an acceptable component of a formula measuring speed of convergence. This is so, because some GA's may use big populations and run them for a small number of generations, while other GA's may run for somewhat longer periods (in terms of generations) but using much smaller populations. On the other hand, time is not a good measure of speed, since different computers can run at radically different speeds. Algorithmic complexity is not either, because what actual users (mostly engineers) care about is not some idealized measure of complexity but a portable measure of

real speed. We use number of fitness evaluations. This and the other two measures are explained in detailed below.

As to being meaningful, the metrics we propose are: *reliability*, *speed* and *memory load*. GA's are often accused of being unreliable since they are non-deterministic and few things are proven about their performance – this situation is changing but slowly. What we can do, empirically, to remedy this situation, is attach a sound statistical measure of reliable performance to GA's. As to speed: speed is important for real-time near real-time and on-line applications. Speed is also important for off-line applications that operate within industrial and/or commercial environments. Finally, memory load determines if a computer with specific physical (and virtual) memory resources is able to run a given GA or not. If a given computer is slow then a GA will simply take longer to converge. However, if a GA requires more memory than is available then the computer will not be able to run the GA at all, unless additional memory is added. Together, speed and memory load are the two core measures of any computational algorithm. When-defining reliability, speed and memory load, we assume that each algorithm is run (at least) 30 times and that the maximum number of fitness evaluations allowed during one run is 500, 000.

A. Reliability

Reliability is *primarily* defined as the percentage of runs that were able to return an optimally fit individual (within 500,000 evaluations). Of course, it is possible for more than one GA to return 100% reliability. In order to differentiate between these GAs, we use C.V. as a further means of differentiation. The lower the C.V. is, for a given GA, the more reliable it is, and *vis versa*. Once all GAs with 100% reliability are ranked, those GAs with less than 100% reliability (and > 0%) are ranked simply according to their reliability percentage. To mark the fact that their reliability is less than 100%, an asterisk (*) is attached to their rankings. If a given GA completely fails to return any optimally fit individuals in all the runs (reliability = 0%), then it is given an *NA* (not applicable) designation, for such a GAs is totally unreliable.

For example, assume 3 of 5 parameter-less GAs (call them GA1, GA2 and GA5) return an optimally fit individual 100% of the time, with GA3 and GA4 optimally converging only 0% and 98% of the time, respectively. Further assume that the C.V.s for GA1, GA2 and GA5 are: 10, 14 and 64, respectively. This will lead to a ranking of 1, 2, *NA*, 4* and 3 for GA1-GA5.

B. Speed

Speed is *primarily* defined as the average number of fitness evaluations to optimal convergence. First, GAs with the same primary reliability of 100% is ranked according to the average number of evaluations executed up to optimal convergence. Then, those GAs that have, on average, reached a maximum fitness of 0.95 (or more, but less than 1) are ranked according to the average number of evaluations executed before 0.95 fitness was reached; to mark the fact that their convergence is near-optimal, an asterisk (*) is attached to their rankings. Any GA that, on average, failed to reach even the near-optimal fitness (of 0.95), is given an *NA* (not applicable) assignment.

For example, assume five GAs: GA1-GA5. GA1-GA4 all achieved positive reliability (>0%), but only GA1 and GA2 returned 100% reliability with the rest (GA3-GA4) achieving 85% and 33% reliability. Assume the number of evaluations required for optimal convergence in GA1 and GA2 were (on average) 42545 and 56010, respectively. Further assume that the number of evaluations required for near-optimal convergence in GA3 and GA4 were (on average) 65050 and 32545, respectively. Finally, the average maximum fitness of GA5 never reached 0.95. This leads to the following speed rankings for GA1-GA5: 1, 2, 4*, 3* and *NA*.

C. Memory Load

Memory load is *primarily* defined as the average size of the largest populations encountered during runs that resulted in optimal convergence (call that *ave-max*); populations that were evolved after the moment of optimal convergence are excluded. First, GAs with 100% reliability are ranked according to their *ave-max* values; GAs requiring smaller populations receive better rankings and visa versa. Then, those GAs that have, on average, reached a maximum fitness of 0.95 (or more, but less than 1) are ranked according to the average size of the largest populations encountered during runs that resulted in near-optimal convergence; to mark the fact that their convergence is near-optimal, an asterisk (*) is attached to their rankings. Finally, any GA that, on average, failed to reach even the near-optimal fitness (of 0.95), is given an *NA* (not applicable) assignment.

For example, assume five GAs: GA1-GA5. GA1 and GA2 both achieved 100% reliability. GA3 and GA4 only achieved near-optimal convergence, on average. Further assume that the largest size population (on average) encountered on or before optimal convergence was 45000 for GA1 and 32333 for GA2. On the other hand, the largest size population (on average) encountered on or before near-optimal convergence was 50231 for GA3 and 132133 for GA4. Finally, GA5 never achieved optimal or near-

optimal convergence. This leads to the following memory load rankings for GA1-GA5: 2, 1, 3*, 4* and *NA*.

4.4.2 Ranking tables

All the rankings presented in the following tables (table 7-11) are based on the application of the formulae for reliability, speed and memory load to the data provided in tables 2-6 (in section 4.2).

Table 7: Overall Rankings for Test Algorithms as applied to Test Function f_1

Metric	TSSGA	SAMXPGA	AEARC	APSGA	DIMCCGA
Reliability	2	5	3	4	1
Speed	1	2	5	4	3
Memory Load	2	1	4	5	2

Table 8: Overall Rankings for Test Algorithms as applied to Test Function f_2

Metric	TSSGA	SAMXPGA	AEARC	APSGA	DIMCCGA
Reliability	NA	NA	NA	NA	NA
Speed	1*	3*	NA	NA	2*
Memory Load	2*	1*	NA	NA	2*

Table 9: Overall Rankings for Test Algorithms as applied to Test Function f_3

Metric	TSSGA	SAMXPGA	AEARC	APSGA	DIMCCGA
Reliability	3	4	5*	2	1
Speed	1	2	5*	3	4
Memory Load	1	3	4*	3	1

Table 10: Overall Rankings for Test Algorithms as applied to Test Function f_4

Metric	TSSGA	SAMXPGA	AEARC	APSGA	DIMCCGA
Reliability	1	2	4*	NA	3*
Speed	2	1	NA	NA	NA
Memory Load	1	2	NA	NA	NA

Table 11: Overall Rankings for Test Algorithms as applied to Test Function f_5

Metric	TSSGA	SAMXPGA	AEARC	APSGA	DIMCCGA
Reliability	NA	NA	NA	NA	NA
Speed	NA	NA	NA	NA	NA
Memory Load	NA	NA	NA	NA	NA

Tables 7-11 show that the simplest Parameter-less GA (TSSGA) is the best overall performer. SAMXPGA is the fastest pGA for multi-modal fitness surfaces i.e. function f_4.

Chapter 5

A New Parameter-less Genetic Algorithm

5.1 Introduction

Setting the parameters of a genetic algorithm is not trivial task. It is needs a lot of knowledge and care from the user. User is interested in solving a problem. He is not interested in tuning the population size, crossover rate, mutation rate, or any other GA technicalities. He would like to have a black-box algorithm, and simply press a start button. In chapter two we see several technique used by researcher to get rid off tuning GA parameters. In chapter 4 we compared the performance of five parameter-less GAs, which were implemented 'as is' without any attempt at improvement. In this chapter, we present a new parameter-less GA (nGA), which adds a number of features to the classic Simple GA (SGA) [34] in order to achieve much better results. A number of elaborations were implemented in order to boost the performance of the simple GA and simultaneously make it into a parameter-less GA. These are:

- Stagnation-Triggered-Mutation (STM);

- Reverse Traversal- Phenotopic (RTP) and Genotopic (RTG);
- Non-Linear Fitness Amplification (NLA).

5.2 Stagnation-Triggered-Mutation (STM)

The idea behind STM is simple: older individuals stuck at a sub-optimal points on the fitness surface for a long time need to be given some kind of 'push' (e.g. mutation) to reach a new potentially more promising position on the surface. This feature helps GAs deal with fitness functions that are hard (and hence take long) to optimize, such as multi-modal functions (e.g test functions f_3 and f_4).

Attached to each chromosome are two numbers; a mutation probability (p_m), and a new quantity, Life Time (or LT), which measures the number of generations passed since the chromosome was last modified (via crossover or mutation). Initially, P_m is equal to $1/l$, where l is number of bits in the rest of the chromosome. In later generations, every chromosome that passes through (probabilistic) crossover and/or mutation is tested to see if it is identical to any of its parents. If it is, then its P_m is multiplied by its LT (and its LT is incremented by 1). If, on the other hand, this chromosome is altered (via crossover or/and mutation) then its P_m is reset to $1/l$ and its LT is

reset to 1. Also, the population size could increases due to the STM mechanism. This increases the number of individuals in the stagnated area. The population increases until reach upper bound defined in section 5.4.

5.3 Reverse Traversal (RT), Phenotopic (RTP) and Genotopic (RTG)

Phenotopic Reverse Traversal deals with fitness surfaces that tend to drive the majority of the population toward one (or more) local maxima, and away from the global maximum like in figure 15. SGA drives most population members towards local optima rather than the global optimum. To avoid that case we should force some individuals to traverse towards minima in order to find the global optimum. RTP does this by getting a portion of the population to traverse the fitness surface against the gradient, i.e. towards minima rather than maxima. This also has the side effect of producing a more diverse population than simple fitness-proportional selection. 20% of the next generation is produced by RTP. RTP selects individuals via fitness proportional selection, but instead of selecting those individuals with the greatest fitness, RTP selects those with the lowest fitness. Then these individuals go through probabilistic crossover/mutation.

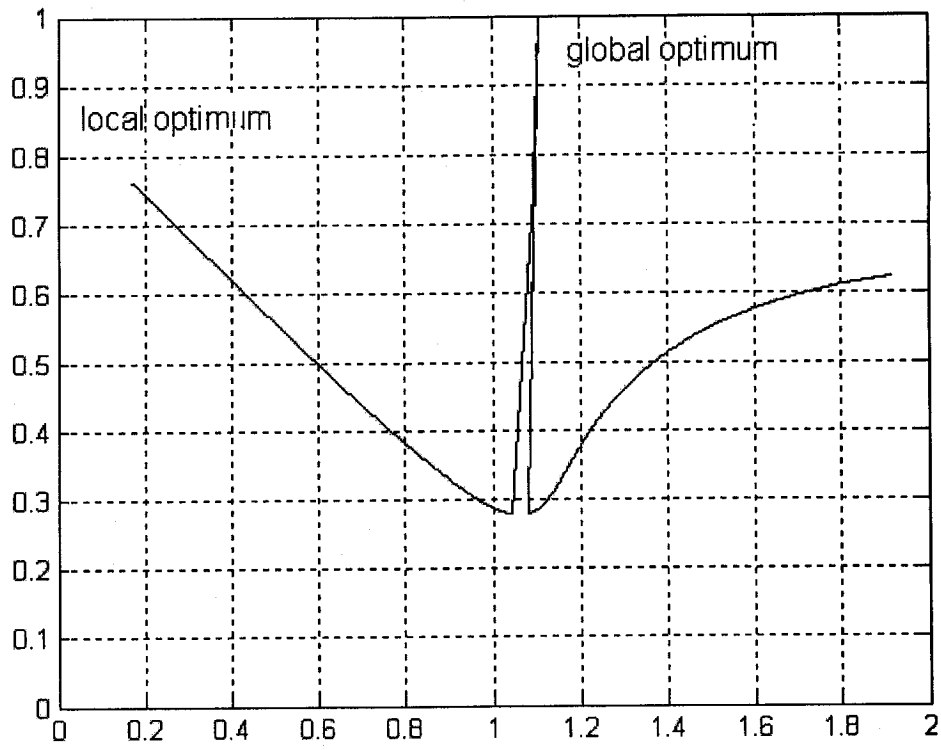


Figure 15. Fitness surface with global optimum in the minimum area of the surface.

On the other hand, Genotopic Reverse Traversal (RTG) deals with deceptive fitness surfaces (e.g. test function f_5). 20% of the next generation is produced by RTG. RTG selects individuals via proportional selection. Applies the genetic operator's crossover and mutation. In case of crossover, it inverts the bit string (turning 1's to 0's and 0's to 1's) for one individual. Then it crosses over the inverted individual with the other individual whose bits are not inverted. The created offspring are added to the next generation after mutation is applied. This simple trick was the main factor behind the

100% reliability figure returned by the nGA for the fully deceptive function f_5.

5.4 Non-Linear Fitness Amplification (NLA)

This enhancement of the SGA is designed to deal with situations where the population converges to a rather flat neighborhood of a global optimum. In such cases, it is important that the selection mechanism becomes very sensitive to slight variations in the gradient of the fitness surface.

The way NLA works is straightforward: once the average fitness of the population exceeds 0.9, the fitness is scaled using the following formula (31); f' is the scaled fitness, f is the original unscaled fitness and c is a constant (that we set to 100).

$$f' = \frac{1}{c(1-f)+1} \tag{31}$$

The nGA introduces the three main new features explained above, but also uses a fixed probability of crossover equal = 0.7 (~ De Jong [20] empirically determined value), and implements elitism at 10%.

To determine the minimum size of the population, a (pre-run) large population of 1000 individuals is created and the fitness of each individual is

computed. Hence, the standard deviation of fitness of the population is computed (call that *SDfitness*) and used in equation 32 (below). The size of the initial population is set to *LowBound*; but the population is allowed to grow to as much as double that value (as a result of STM). Constant ***k*** is set to 3; the probability of failure (α) is set to 0.05; and sensitivity (*d*) to 0.005- see [29] for more detailed information about equation 32.

$$LowBound = -2^{k-1} \ln(\alpha) \frac{SDfitnes}{d} \tag{32}$$

5.5 Experiments and Results

This section presents computer simulations of nGA. Six controlled experiments are executed the results are compared to those of the simple GA. For each test problem, 30 independent runs were performed in order to get results with statistical significance. The experiments executed are:

- Investigate the effect of stagnation-triggered mutation on test function f_2, test function f_3 and test function f_4;
- Investigate the effect of phenotopic reverse traversal on test function f_2, test function f_3 and test function f_4;
- Investigate the combined use of stagnation-triggered mutation and phenotopic reverse traversal on test function f_2, test function f_3 and test function f_4;
- Investigate the effect genotopic reverse traversal on test function f_5;
- Investigate the effect of non-linear fitness amplification on test function f_2;
- Investigate the combined use of all new features of nGA on test functions $f_1 - f_5$.

10 dimensions are used for each test functions, at 20 bits/variable, which makes 200 bits long. Uniform crossover, bit flip mutation, and proportional selection were used in all the experiments. A fixed crossover rate equal to 0.7 was used in all the experiments. In the nGA, we designed the lower bound and upper bound of population size according to equation 32. The simple GA used exactly the same parameters values detailed above, except for the population size, which was fixed at 60.

Table 12 illustrates the result of SGA on the five test functions. These results are meant to act as a means of comparing the performance of the nGA to the simple GA. The test functions used are those of section 3.3. The measures used are those of section 4.2. 'NA' stands for not applicable and means that the GA was not able to achieve optimal and/or near-optimal convergence.

Table 12: Results of Application SGA on Test Functions $f_1 - f_5$

SGA	F_1	F_2	F_3	F_4	F_5
Percentage of Runs to Optimal Fitness	100%	100%	100%	100%	0%
Ave. No. of Evaluations to Best Fitness	12,534	500,000	25,324	182,922	500,000
C.V.[1]	11.28	0	11.78	52.61	0
Ave. No. of Evaluations to Near-Optimal Fitness	4,260	51,522	8,386	18,051	NA
Average Best Fitness	1	0.9734	1	1	0.6778
S.D.	0	0.035	0	0	0.0474
Ave. Mean Fitness	0.8952	0.5110	0.8572	0.7495	0.5084
S.D.	0.04	0.09	0.39	0.064	0.0245
Ave. Mean Population Size to Optimal Fitness	60	NA	60	60	NA
Ave. Max. Population Size to Optimal Fitness	60	NA	60	60	NA
Ave. Mean Population Size to Near-Optimal Fitness	60	60	60	60	NA
Ave. Max. Population Size to Near-Optimal Fitness	60	60	60	60	NA

As shown in table 12 the SGA return 100% on functions f_1, f_3, and f_4. f_1 is easy to optimize, but f_3 and f_4 are multimodal functions. SGA failed to find the global optimum for test function f_5; it also returned 0% on test function f_2.

Figure 16 represents paired plots on applying SGA on test functions f_1 – f_5. Fig. 16a presents maximum fitness of population as a function of number of evaluations, and fig. 16b presents entropy of population as a

function of number of evaluations. In all the plots presented below, Fitness/Entropy is represented on the vertical axis (using a linear scale), while the number of fitness evaluations is presented on the horizontal axis (using a logarithmic scale). Each function was run 30 times and the plots represent average values for all the plots generated during all the runs. In figure 16 red stands for result obtained on test function f_2; dark blue is for test function f_1; light green is for test function f_3; magenta is for test function f_5; and finally black is for test function f_4.

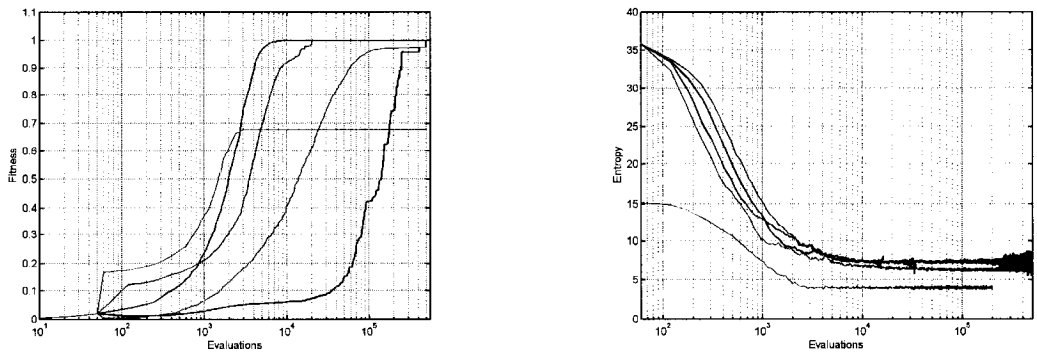


Figure 16. 16a Fitness (left) and 16b Entropy (right) of SGA

SGA starts at its highest value within the run, and then collapses relatively quickly.

5.5.1 The Effect of Stagnation Trigger Mutation on Test Functions f_2, - f_4

To see the performance of STM mechanism on solving hard problems, we add STM only to SGA and apply the resulting algorithm to test functions f_3, f_2 and f_4. Table 13 contains the results.

Table 13: Results of Application STM on Test Functions $f_2 - f_4$

STM	F_2	F_3	F_4
Percentage of Runs to Optimal Fitness	100%	100%	100%
Ave. No. of Evaluations to Best Fitness	226,610	32,439	106,630
C.V.[1]	30.5	14.23	41.84
Ave. No. of Evaluations to Near-Optimal Fitness	8,100	10,719	18,345
Average Best Fitness	1	1	1
S.D.	0	0	0
Ave. Mean Fitness	0.6770	0.8029	0.6846
S.D.	0.02	0.0344	0.0441
Ave. Mean Population Size to Optimal Fitness	112.2	111.3	158
Ave. Max. Population Size to Optimal Fitness	112.2	111.3	158
Ave. Mean Population Size to Near-Optimal Fitness	107	78.4	158
Ave. Max. Population Size to Near-Optimal Fitness	107	78.4	158

STM return 100% for test function f_2, while SGA returns 0% on the same test function. Also STM speeds up the performance when applied to hard problems like f_4: it takes it an average of 106,630 function evaluations

to reach the optimal solution, while SGA needs 182,922 function evaluations.

Paired plots of maximum fitness/entropy of population as a function of function evaluations are presented in figure 17. In figure 17, the solid line refers to test function f_2, the dashed line refers to function f_3, and dotted line refers to function f_4.

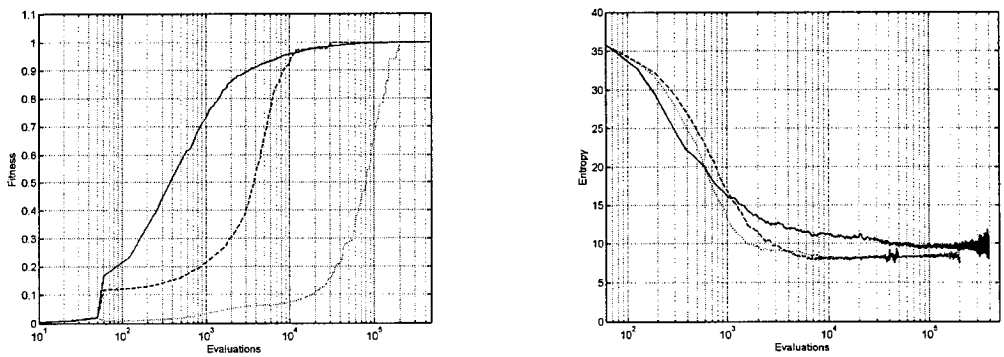


Figure 17. 17a Fitness (left) and 17b Entropy (right) of STM

5.5.2 The Effects of Phenotopic Reverse Traversal on Test Functions f_2, $-f_4$

The results on applying RTP on test functions $f_2 - f_4$ are shown in table 14. The performance is not too different from SGA on the same test functions.

Table 14: Results of Application RTP on Test Functions $f_2 - f_4$

SGA	F_2	F_3	F_4
Percentage of Runs to Optimal Fitness	0%	100%	100%
Ave. No. of Evaluations to Best Fitness	500,000	30,862	139,764
C.V.[1]	0	11.73	48.38
Ave. No. of Evaluations to Near-Optimal Fitness	123,060	14,100	18,678
Average Best Fitness	0.9782	1	1
S.D.	0.0306	0	0
Ave. Mean Fitness	0.4327	0.8186	0.7126
S.D.	0.0967	0.0603	0.0858
Ave. Mean Population Size to Optimal Fitness	102	101	147
Ave. Max. Population Size to Optimal Fitness	102	111	147
Ave. Mean Population Size to Near-Optimal Fitness	100	73.4	147
Ave. Max. Population Size to Near-Optimal Fitness	100	73.4	147

Figure 18 presents paired plots of maximum fitness/entropy of population as function of function evaluations. The lines refer to the same functions as the previous experiment.

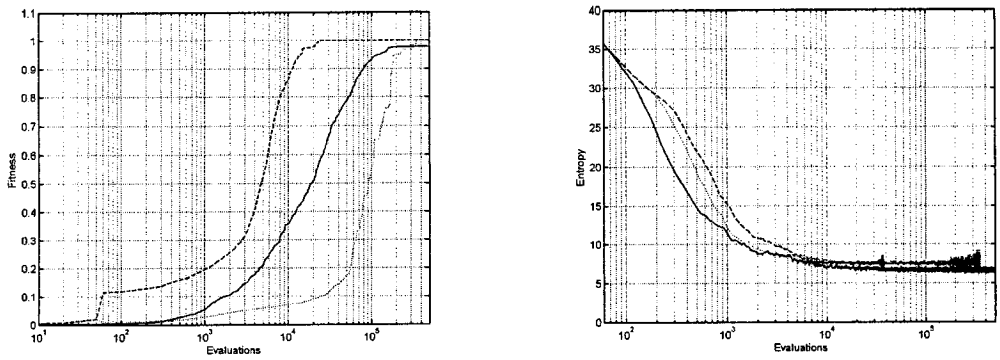


Figure 18. 18a Fitness (left) and 18b Entropy (right) of RTP

5.5.3 The Combined Use of Stagnation Trigger Mutation and Phenotopic Reverse Traversal on Test Functions f_2, $-f_4$

Table 15 and figure 19 illustrate that STM and RTP combined, enhanced the speed on test function f_4, but the speed for test function f_3 and test function f_2 is decreased. However, 100% reliability was returned.

Table 15: Results of Application STM & RTP on Test Functions $f_2 - f_4$

STM & RTP	F_2	F_3	F_4
Percentage of Runs to Optimal Fitness	100%	100%	100%
Ave. No. of Evaluations to Best Fitness	272,040	40,157	96,479
C.V.[1]	23.46	9.98	37.15
Ave. No. of Evaluations to Near-Optimal Fitness	11,504	11,397	14,688
Average Best Fitness	1	1	1
S.D.	0	0	0
Ave. Mean Fitness	0.6559	0.7520	0.6368
S.D.	0.0399	0.0506	0.0643
Ave. Mean Population Size to Optimal Fitness	107	131	157
Ave. Max. Population Size to Optimal Fitness	110	141	157
Ave. Mean Population Size to Near-Optimal Fitness	103	100	157
Ave. Max. Population Size to Near-Optimal Fitness	104	100	157

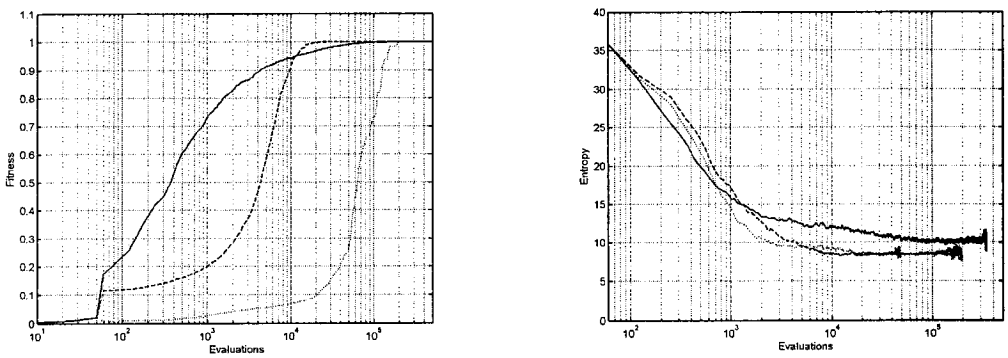


Figure 19. 19a Fitness (left) and 19b Entropy (right) of RTP & STM

5.5.4 The Effects of Genotopic Reverse Traversal on Test Function f_5.

Applying RTG on function f_5 returned results illustrated in table 16 and figure 20. RTG is designed to deal with deceptive functions, and as shown, it actually works as it returned 100% reliability.

Table 16: Results of Application RTG on Test Functions f_5

STM & RTP	F_5
Percentage of Runs to Optimal Fitness	100%
Ave. No. of Evaluations to Best Fitness	17,674
C.V.[1]	29.21
Ave. No. of Evaluations to Near-Optimal Fitness	8,340
Average Best Fitness	1
S.D.	0
Ave. Mean Fitness	0.6521
S.D.	0.0487
Ave. Mean Population Size to Optimal Fitness	73
Ave. Max. Population Size to Optimal Fitness	73
Ave. Mean Population Size to Near-Optimal Fitness	65
Ave. Max. Population Size to Near-Optimal Fitness	67

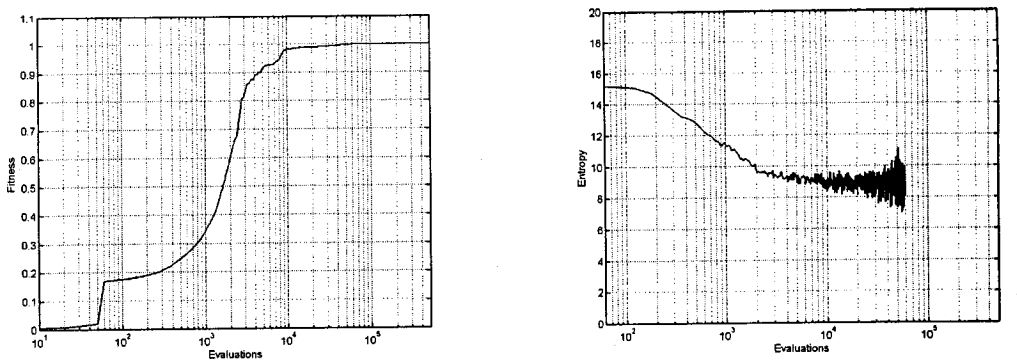


Figure 20. 20a Fitness (left) and 20b Entropy (right) of RTG

5.5.5 The Effects of Non-Linear Fitness Amplification on Test Function f_2

Applying NLA on test function f_2 retuned 100% reliability. In speed it is the fastest. It returned the global optimum in an average of 181,870 function evaluation; it takes an average of 226,610 using STM. Table 17 and figure 21 present the results of applying NLA on test function f_2.

Table 17: Results of Application NLA on Test Functions f_2

NLA	F_2
Percentage of Runs to Optimal Fitness	100%
Ave. No. of Evaluations to Best Fitness	181,870
C.V.[1]	29.85
Ave. No. of Evaluations to Near-Optimal Fitness	9,480
Average Best Fitness	1
S.D.	0
Ave. Mean Fitness	0.7850
S.D.	0.0415
Ave. Mean Population Size to Optimal Fitness	117
Ave. Max. Population Size to Optimal Fitness	120
Ave. Mean Population Size to Near-Optimal Fitness	110
Ave. Max. Population Size to Near-Optimal Fitness	111

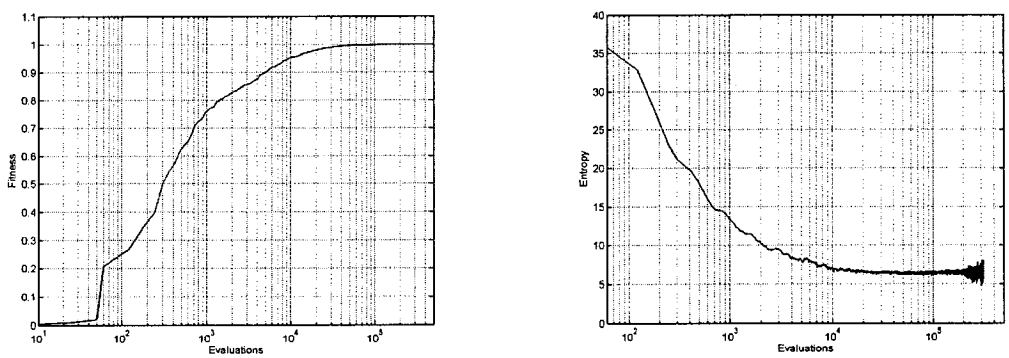


Figure 21. 21a Fitness (left) and 21b Entropy (right) of RTG

5.5.6 The Combined Use of All New Features of nGA on Test Functions $f_1 - f_5$

The results of applying the nGA to the five test functions are shown below.

Table 18: Results of Application nGA on Test Functions $f_1 - f_5$

nGA	F_1	F_2	F_3	F_4	F_5
Percentage of Runs to Optimal Fitness	100%	100%	100%	100%	100%
Ave. No. of Evaluations to Best Fitness	14,345	145,980	28,873	94,640	10,413
C.V.[1]	18.85	18.34	16.8	40.85	40.64
Ave. No. of Evaluations to Near-Optimal Fitness	4,912	15,512	9,175	90,465	6,793
Average Best Fitness	1	1	1	1	1
S.D.	0	0	0	0	0
Ave. Mean Fitness	0.7784	0.51	0.7307	0.6	0.5241
S.D.	0.0423	0.0252	0.03	0.04	0.0339
Ave. Mean Population Size to Optimal Fitness	111.78	132.2	121.3	178	114.7
Ave. Max. Population Size to Optimal Fitness	111.78	132.2	121.3	178	114.7
Ave. Mean Population Size to Near-Optimal Fitness	77	117	78.4	178	98.2
Ave. Max. Population Size to Near-Optimal Fitness	77	117	78.4	178	98.2

In figures 22a and 22b below: blue stands for f_1, red stands for f_2, green stands for f_3, black stands for f_4 and magenta for f_5. The nGA was run 30 times per test function and the figures used to plot the curves represent average values (over the 30 runs) of both fitness and diversity (entropy).

Figure 22a demonstrates the evolution of fitness. For four out of the five test functions, nGA's behavior is exemplary: it succeeds in converging

by about 10^4 fitness evaluations; this speed of convergence is not achieved by any of the evaluated (older) pGAs, except in the case of function 1 (see figure 10a), which is the easiest to optimize. The only exception is function f_4, which is the hardest multi-modal test function used. Indeed, the nGA performs better on the deceptive surface of function f_5 than on function f_4, which is a testimony to the power of the anti-deceptive measures (Reverse Traversal of both colors) included in the nGA.

Figure 22b, on the other hand, demonstrates that the nGA maintains a high degree of diversity (Entropy >= 10) throughout evolution- a positive feature of any GA. As seen in figures 10b – 14b, only one of the five pGAs (AEARC) consistently achieved a higher level of diversity at convergence than nGA. This is due to AEARC's use of special means of selection that do not involve fitness-proportional selection or tournaments: but sexual and asexual reproduction, as well as competition between individuals randomly selected from the current population.

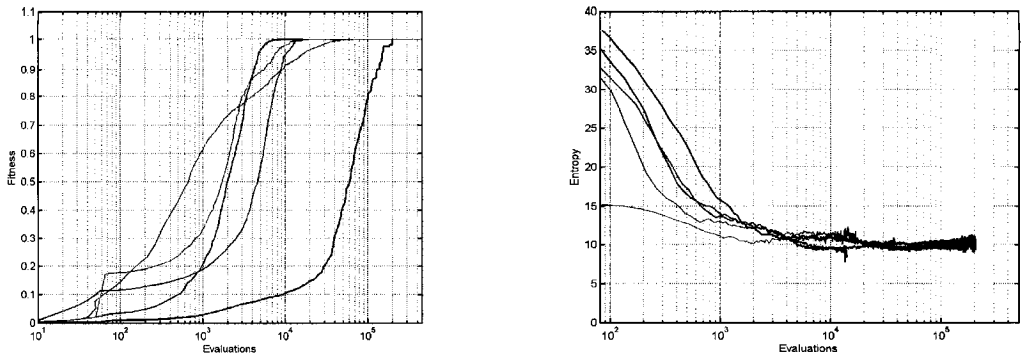


Figure 22. 22a Fitness (left) and 22b Entropy (right) of nGA

Chapter 6

Case Study

6.1 Introduction

Researchers usually test genetic algorithms on artificial problems, and that is precisely what we have done in chapters 4 and 5 when testing the parameter-less GAs. Artificial problems are useful for testing GAs due to a number of reasons:

- We can create problems of different sizes;
- We can create problems with varying degrees of difficulty;
- We can study boundary cases of GA performance.

The goal of this chapter is to show how the nGA can be applied to real-world problems. For purpose of illustration, we will be applying the nGA to the problem of optimizing the parameters of various edge detectors. We intend this as a case study, but similar design principles may be applied to other problems.

6.2 Edge Detectors

Edge detectors are a collection of very important local image pre-processing methods used to locate changes in the intensity function of an image. Edges are pixels where the brightness function changes abruptly. Figure 23 shows a sketch of a continuous domain, one-dimensional ramp edge modeled as a ramp increase in image amplitude from low to high intensity, or *vice versa*. The edge is characterized by its height, slope angle, and horizontal coordinate of the slope midpoint. An edge exists if the edge height is greater than a specified value. An ideal edge detector should produce an edge indication localized to a single pixel located at the midpoint of the slope.

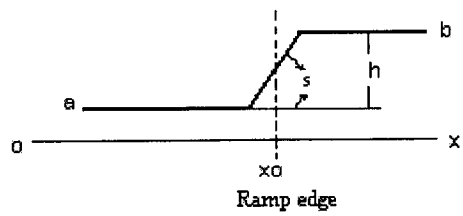


Figure 23. One –dimensional, Continuous Domain Edge

There are two generic approaches to the detection of edges in an image:

- Model Fitting;
- Differential Detection.

Model Fitting edge detection involves fitting of a local region of pixel values to a model of the edge. If the fit is sufficiently close, an edge is said to exist, and its assigned parameters are those of the appropriate model. With Differential Detection, spatial processing is performed on an original image *f(j,k)* to produce a differential image *g(j,k)* with accentuated spatial amplitude changes. Next, a differential detection operation is executed to determine the pixel locations of significant differentials. There are two major classes of differential edge detection:

- First-order derivative;
- Second-order derivative.

For the first-order derivative, some form of spatial first-order differentiation is performed, and resulting edge gradient is compared to a threshold value as shown in Figure 24. An edge is judged present if the gradient exceeds the threshold. For the second order derivative, an edge is judged present if there is a significant spatial change in the polarity of the second derivative.

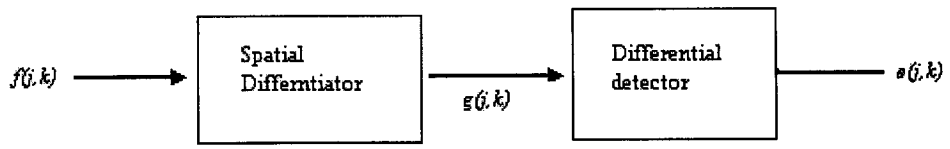


Figure 24. Differential Edge Detection

6.2.1 First-Order Derivative Edge detection

First-order derivative edge detection involves generation of gradient in two orthogonal directions in an image. Figure 25 describes the generation of an edge gradient $g(j,k)$ in the discrete domain in terms of a horizontal gradient $g_h(j,k)$ and a vertical gradient $g_v(j,k)$. The spatial gradient amplitude is given by:

$$g(j,k) = \sqrt[2]{g_h(j,k)^2 + g_v(j,k)^2} \tag{33}$$

The simplest method of discrete gradient is to form the running difference of pixels along rows and columns of the image. The row gradient is defined as

$$g_h(j,k) = f(j,k) - (j,k-1) \tag{34}$$

and the columns gradient is

$$g_v(j,k) = f(j,k) - f(j+1,k) \tag{35}$$

For ramp edges, the running difference edge detector cannot localize the edge to a single pixel. Therefore, Robert introduced two 2x2 filters to be convolved with the image to produce the gradient. Also Sobel uses two 3x3 filters to evaluated the gradient.

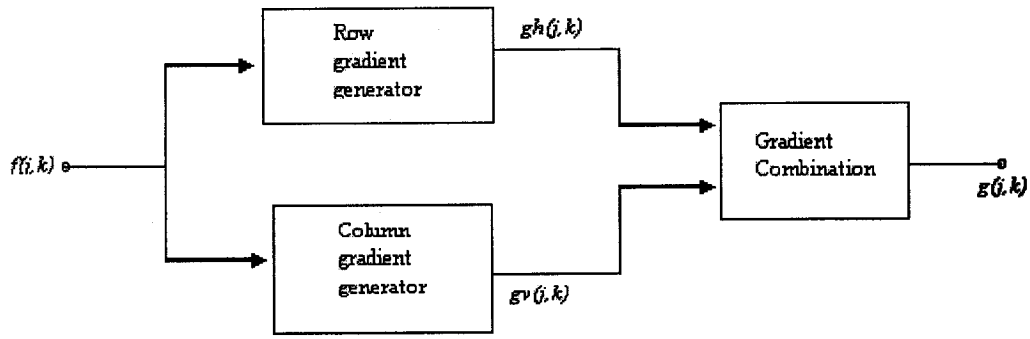


Figure 25. Orthogonal Gradient Generation

6.2.1 Second-Order Derivative Edge detection

An edge is marked if a significant spatial change occurs in the second derivative. The Laplace operator is a very popular operator approximating the second derivative.

6.3 Experimental Setup

For our application, we chose three edge operators frequently used for edge detection: the Sobel operator [53] and the Robert operator [53] (of the firs-

order derivative edge detection class of operators), and the Laplacian operator [53] (of the second-order derivative edge detection class of operators).

The Sobel operator uses two 3x3 filters to evaluate the gradient. These two filters are presented in section 6.4.2. We will consider the filter's coefficients in section 6.4.2 as the standard setting for the Sobel operator. Two filters of size 2x2 are used for the Robert operators. The standard settings of the Robert filter's coefficients are presented in section 6.4.1. With respect to the Laplacian operator, one filter of size 3x3 is used. The standard setting of the Laplacian filter coefficients are also presented in 6.4.3.

6.3.1 Chromosome Representation

All the filter coefficients have a domain interval of [-10, 10]. The nGA represents the values of the filter coefficients using Grey code, at 20 bits/coefficient. For each edge operator, 5 independent runs were performed. Then the run that returned the best edge results is taken. The nGA runs terminate when the maximum fitness reaches 0.99, or a maximum number of 500,000 fitness evaluations is exceeded- whichever comes first.

6.3.2 Fitness Evaluation

For fitness calculation, chromosomes are converted back to filter coefficients and applied to the training image. A resultant edge image results and is called the "binary image". This image is compared to the image with highlighted (ideal) edges to calculate the fitness value for similarity. The ideal edges (of the training image) are marked by hand. The input edge feature image and the resultant edge image formed by each chromosome in the population are compared on a pixel-by-pixel basis. Two calculations are made based on those edges identified in the input edge feature image, but not in the resultant edge image, and *vice-versa.*

- Underdetection (P_{ef}) is the number of edge pixels not detected in the resultant edge image divided by the total number of edge pixels in the input edge feature.
- Overdetection (P_{nf}) is the number of non-edge pixels detected in the resultant edge image divided by the total number of non-edge pixels in the input edge feature image.

We need to minimize both of these values simultaneously, and this is achieved by maximizing the following equation:

$$f = \frac{1}{1+(P_{nf}+P_{ef})} \tag{36}$$

The schematic diagram of the nGA, as it is used in optimizing edge operators, is shown in figure 26. The input of the algorithm consists of two images. One is a simple image and the other is a corresponding edge image, such as those shown in figure 27. These two images are used as the input to the GA training algorithm, in order to optimize the coefficients of the filters. The evolved filters or filter are then applied to different images to test their edge detection performance. The results are illustrated for each edge operator.

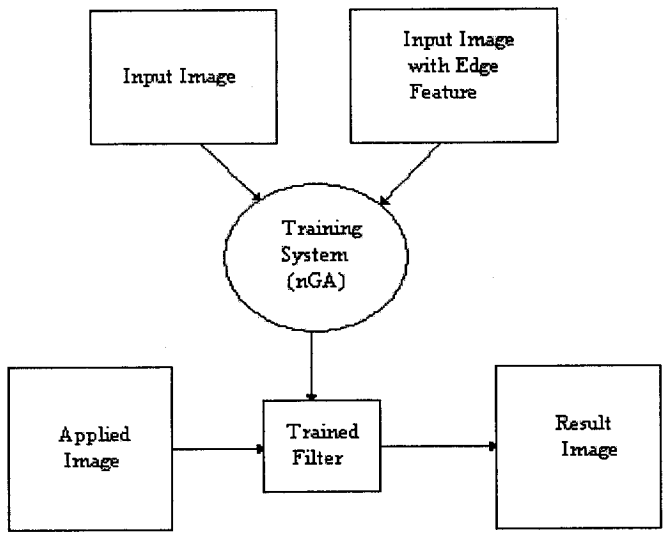


Figure 26. Block Diagram of nGA in Optimizing Edge Operators

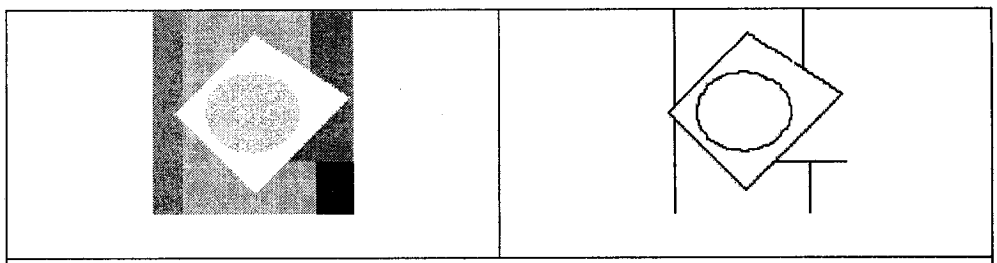


Figure 27. 27a (left) Input Image, 27b (right) Input Edge Image

6.4 Results

6.4.1 Robert Operator

The Roberts operator is one of the oldest operators. It is very easy to compute as it uses only a 2x2 neighborhood of the current pixel. Its convolution filters are:

$$h_1 = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \quad , \quad h_2 = \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix} \tag{37}$$

The magnitude of the edge is computed as:

$$g(i,j) = |f(j,k) - f(j+1,k+1)| + |f(j,k+1) - f(j+1,k)| \tag{38}$$

nGA is used to optimize the coefficients of the Robert filters. First of all, the eight coefficients of the edge detection filters are converted into bit format and combined to form a single chromosome. We use a chromosome of length 160, at 20 bits per coefficient. For fitness calculation,

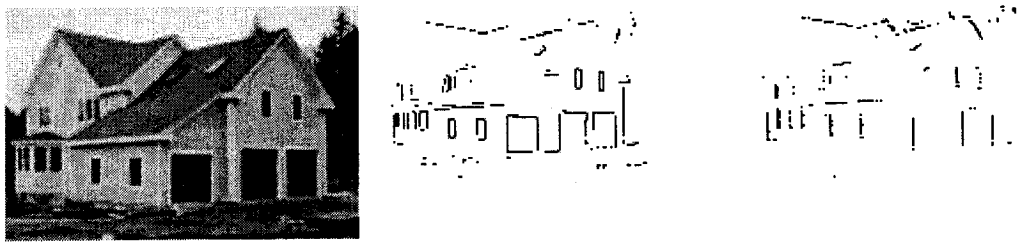
chromosomes are converted back to filter coefficients and applied to the training image. The best filter coefficients obtained within the 5 runs are:

$$h_{1o} = \begin{bmatrix} -1.13803 & 8.0421 \\ -4.3939 & 1.45732 \end{bmatrix} \quad , \quad h_{2o} = \begin{bmatrix} 3.61579 & -1.50714 \\ -1.03438 & 0.341025 \end{bmatrix} \tag{39}$$

These values are obtained after 11,527 fitness evaluations. Figures 28 and 29 show the edge images resulting from the application of the standard Robert filter and the nGA-optimized Robert filter, respectively.

Original Image Standard Robert Filter nGA-Optimized Robert Filter

Figure 28. Lina Edge Image Results of the Application of Robert Operator



Original Image Standard Robert Filter nGA-Optimized Robert Filter

Figure 29. House Edge Image Results of the Application of Robert Operator

6.4.2 Sobel Operator

The Sobel edge detection method used two 2-dimensional filters of size 3x3 to process vertical edges and horizontal edges separately. Its vertical and horizontal filters are:

$$h = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \quad , \quad v = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \tag{40}$$

nGA is used to optimize the coefficients of the Sobel operators. First of all, the eighteen coefficients of the edge filters are converted into bit format and combined to form a single chromosome. We used a chromosome of length 360, at 20 bits per coefficient. For fitness calculation, chromosomes are converted back to filter coefficients and applied to the original image. The best filters coefficients are obtained after 49,108 fitness evaluations. Those filters are:

$$h_o = \begin{bmatrix} -5.57252 & -2.43282 & -5.84549 \\ 7.02083 & -7.12869 & 8.44736 \\ -6.40888 & 8.22819 & -1.41591 \end{bmatrix} \quad , \quad v_o = \begin{bmatrix} -6.16537 & 8.8022 & 8.48327 \\ -7.82224 & 0.177584 & 8.24062 \\ -4.69886 & 1.22192 & -3.32546 \end{bmatrix} \tag{41}$$

The resultant edge images of the application of those filters on real images are illustrated in figures 30 and 31. The nGA-optimized Sobel filter retuned better edge images than the standard Sobel filter.

Original Image Standard Sobel Filter nGA-Optimized Sobel Filter

Figure 30. Lina Edge Image Results of the Application of Sobel Operator

Original Image Standard Sobel Filters nGA-Optimized Sobel Filters

Figure 31. House Edge Image Results of the Application of Sobel Operator

6.4.3 Laplace Operator

The edge Laplacian of an Image function *f(x,y)* in continuous domain is defined as:

$$g(x,y) = -\nabla^2\{f(x,y)\}$$
$$where \tag{42}$$
$$\nabla^2 = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}$$

The Laplacian *g(x,y)* is zero if *f(x,y)* is constant or changing linearly in amplitude. The zero crossing of *g(x, y)* indicates the presence of an edge. In the discrete domain, the simplest approximation to the continuous Laplacian is to compute the difference of slopes along each axis by the convolution operation

$$g(j,k) = f(j,k) \Theta h(j,k) \tag{43}$$

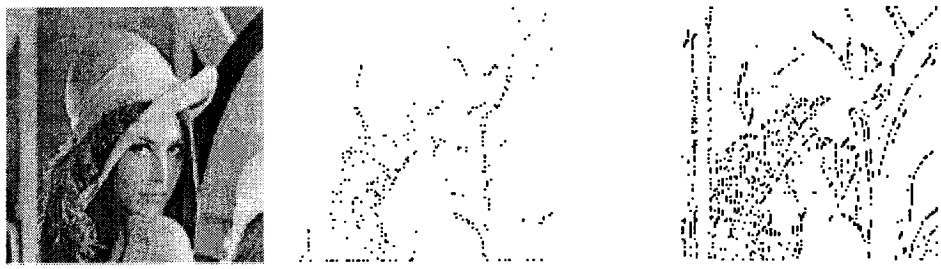
with

$$h = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 8 & 1 \\ 1 & 1 & 1 \end{bmatrix} \tag{44}$$

Figures 32 and 33 show the resultant edge images of the application of the standard Laplacian filter and nGA-optimized Laplacian Filter, respectively. nGa-optimized Laplacian filters coefficients are:

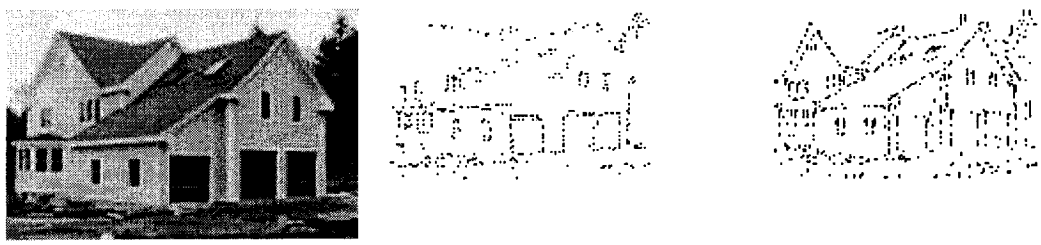
$$h_o = \begin{bmatrix} 2.09386 & -2.23645 & -0.81153 \\ 7.88854 & 0.187292 & -7.35468 \\ 1.05395 & 1.44857 & -2.183491 \end{bmatrix} \quad (45)$$

nGA returned these values in 41,973 fitness evaluations.



Original Image Standard Laplace Filter nGA-Optimized Laplace Filter

Figure 32. Lina Edge Image Results of the Application of Laplace Operator



Original Image Standard Laplace Filter nGA-Optimized Laplace Filter

Figure 33. House Edge Image Results of the Application of Laplace Operator

Chapter 7

Summary and Conclusions

6.1 Summary

This dissertation represents an attempt at establishing standardized means of empirical evaluation and comparison of Parameter-less Genetic Algorithms. We started by pointing out some pitfalls of genetic algorithms (chapter 1). Chapter one looked at GAs from an application perspective and realized that users of genetic algorithms usually have to do a lot of tuning with coding, operators, and parameters in order to have success with the GA. Second, we present a review and classification of parameter-less GA's (chapter 2). Following that, we execute a thorough evaluation of five commonly used pGA's (chapter 3 and 4). In order to do that:

- We ran extensive simulations of the genetic algorithms on five carefully chosen test functions (section 3.2), and presented the results in tables (section 4.2) and graphs (section 4.3);
- We proposed and used three fair and meaningful measures of GA performance (section 4.4).

- We ranked the five test algorithms, in terms of reliability, speed and memory load, in a clear and immediately usable fashion (section 4.4.2).

Finally we developed an algorithm that took the task of setting critical parameters of the GA away from the user (chapter 5). The nGA was implemented and tested on five carefully chosen test functions (section 3.2). Also the nGA tested on real application.

6.2 Conclusions

The most significant contributions of this study are:

- A standardized means of measuring and comparing parameter-less Genetic Algorithms using three metrics;
- An extensive empirical evaluation and ranking of five parameter-less GA's;
- A mathematically founded measure of the diversity of a population.
- Review and classification of all types of parameter-less GA's.
- Developed a new more reliable parameter-less GA and tested it.

The new metrics we proposed are applicable to any parameter-less GA; they are also platform-independent. Having them would facilitate the process of comparing many GA's without having to repeat other people's work. They are also meaningful, in that they allow GA users to choose those GA's that are most reliable, fastest, or require the least amount of memory.

The five pGA's we evaluated displayed much variance in performance, and as such we were able to make well-founded recommendations about which of them should be used, and when. Finally, the review chapter provides a good first reference for researchers that wish to familiarize themselves with the field of parameter-less GA's.

Finally, we developed a new parameter-less GA, one that was born out of the problems encountered with the five tested pGAs. Our main goals in proposing the nGA is to:

- Build a more reliable pGA (which is proven by the results of Table 18).
- To have a reliable pGA, we add a small number of easily realizable amendments to the simple GA.

It is our hope that given our success here, people would be more willing to adopt pGA as a common tool of optimization, rather than normal GAs, which require quite a bit of manual tuning by domain expert.

7.3 Future Research

- **Integration of a Variable Crossover Rate in the nGA**. The new GA introduced in this dissertation uses a fixed value for the crossover rate.

References

[1] Ackley, D. H., A connectionist machine for genetic hill climbing, *Kluwer, Boston, 1987.*

[2] Lex, R. and Bennett, A., Modeling the dynamics of a steady state genetic algorithm, *ISIS research, University of Southampton, 1997.*

[3] Arabas, J., Michalewicz, Z. and Mulawka, J., GAVaPS- A genetic algorithm with varying population size, *Proceedings of the First IEEE Conference on Evolutionary Computation, IEEE Press, 1994.*

[4] Annunziato, M. and Pizzuti, S., Adaptive parameterization of evolutionary algorithms driven by reproduction and competition, *Proceedings of (ESIT2000), PP 246-256, Achen, Germany, 1999.*

[5] Back, T., Self-Adaptation in genetic algorithms, *In F. J. Varela and P. Bourgine, editor, Proceedings of the First European Conference on Artificial Life, PP 263-271, The MIT Press, Cambridge, MA, 1992.*

[6] Back, T., The interaction of mutation rate, selection, and self-adaptation within a genetic algorithm, *In R. Manner and B. Manderick, editor, Parallel Problem solving from Nature, PP 85-94, Elsevier Amsterdam, 1992.*

[7] Back, T., Optimal mutation rates in genetic search. *In Forrest, S., editor, Proceedings the Fifth International Conference on Genetic Algorithms PP 2-8, San Mateo, Ca, Morgan Kufmann, 1993.*

[8] Back, T., and Schwefel, H.-P., An overview of evolutionary algorithms for parameter optimization, *Evolutionary Computation, Vol. 1, No. 1, PP 1-23, 1993.*

[9] Back, T., and Schwefel, H.-P., Evolution strategies I: Variants and their computational implementation. In winter, *G., Perisux, J., Galan, M., and Cuesta, P., editor, Genetic Algorithms in Engineering and Computer Science (Chapter 6, PP 11-126), Chiechester, John Wiley and Sons, 1995.*

[10] Back, T., and Schutz M., Intelligent mutation rate control in canonical genetic algorithm, *Proceedings of the International Symposium on Methodologies for Intelligent Systems, PP 158-167, 1996.*

[11] Back, T., Evolutionary Algorithms in theory and practice, *Oxford University Press, 1996.*

[12] Back, T., and Michalewicz, Z., Test landscapes, *In Back, T., Fogel, D.B., and Michalewicz, Z., editor, Handbook of Evolutionary Computation, (Chapter B2.7, PP 14-20), Institute of Physics Publishing and Oxford University Press, New York, 1997.*

[13] Back, T., Eiben, A. E. and Van derVaart, N. A., An empirical study on GAs "without parameters", *In Schenauer, M., Deb, K., Rudolph, G., Yao, X., Lutton, E.,Merelo, J. J., and Schwefel, H-P., editor, Parallel Problem Solving from Nature PPSN V, Lecture Notes in Computer Science Vol. 1917, PP 315-324, 2000.*

[14] Thijssen, B. A., Adaptive genetic algorithms with multiple subpopulations and multiple parents, *master thesis, 1997.*

[15] Freisleben, B., Metaevolutionary approaches, *handbook of evolutionary computation, publishing Ltd and Oxford University press, 1997.*

[16] Bryant, A., and Julstrom, What have you done for me lately? Adapting operator probabilities in a steady-state genetic algorithm, *Proceedings of the Sixth International Conference on Genetic Algorithms, PP 81-87, Morgan Kufmann, 1995.*

[17] Wook, C. and Ramakrishna, R. S., Elitism-based compact genetic algorithms, *IEEE Transactions on Evolutionary Computation, 2003*

[18] Davis, L., Adapting operator probabilities in genetic algorithms, *In Schaffer, J. D., editor, Proceedings of the Third International Conference on Genetic Algorithms, PP 16-69, San Mateo, Ca, Morgan Kaufman, 1989.*

[19] Deb, K., Deceptive Landscape, *In Back, T., Fogel, D.B. and Michalewicz, Z., editors, Handbook of Evolutionary Computation, Institute of Physics Publishing and Oxford University Press, New York, 1997.*

[20] De Jong, K. A., An analysis of the behavior of a class of genetic adaptive systems, *Doctoral dissertation, University of Michigan, Ann Arbor, University Microfilms No 76-9381, 1975.*

[21] Eiben, A. E., Hinterding, R. and Michalewicz, Z., Parameter control in evolutionary algorithms, *IEEE Transactions on Evolutionary Computation, Vol. 3, No. 2, PP 124-41, 1999.*

[22] Fogarty, T. and Terence, C., Varying the probability of mutation in the genetic algorithm, *Proceedings of the Third International Conference on Genetic algorithms, PP 104-109, Morgan Kufmann, 1989.*

[23] Gabriela, O., Inman, H. and Hilary, B., On recombination and optimal mutation rates, *Proceedings of Genetic and Evolutionary Computation Conference (GECCO-99), PP 488-495, Morgan Kaufmann, San Francisco, CA, 1999.*

[24] Grefenstette, J. J., Optimization of control parameters for genetic algorithms, *In Sage, A. P., editor, IEEE Transactions on Systems, Man,*

and Cybernetics, Volume SMC-16-1, PP 122-128, New York, IEEE, 1986.

[25] Grefenstette, J. J. and Baker, J. E., How genetic algorithms work: a critical look at implicit, parallelism, *In J. D. Schaffer, editor, Proceedings of the Third International Conference on Genetic Algorithms, PP 20–27, San Mateo, CA. Morgan Kaufmann, 1989.*

[26] Grefenstette, J. J., Deception considered harmful, *technical report in Navy Center for applied research in Artificial Intelligence, Washington, 1992.*

[27] Goldberg, D. E., Genetic algorithms in search, optimization, and machine learning, *Addison Wesley Publishing Company, Inc, 1989.*

[28] Goldberg, D. E., Sizing populations for serial and parallel genetic algorithms, *Proceedings of the Third International Conference on Genetic Algorithms and Their applications, PP 70-79, Morgan Kaufmann, 1989.*

[29] Harik, G. R. and Lobo, F. G., A genetic algorithm, *Banzhaf, W., Daida, J., Eiben, A. E., Garzon, M. H., Honavar, V., Jakiela, M. and Smith, R. E., editor, Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-99), PP 258–267, San Francisco, CA, Morgan Kaufmann, 1999.*

[30] Hesser, J. and Manner, R., Towards an optimal mutation probability in genetic algorithms, *Proceedings of the First Parallel Problem Solving from Nature, PP 23-32, Springer, 1991.*

[31] Hinterding, R., Gaussian mutation and self-adaptation for numeric genetic algorithms, *In Proceedings of IEEE International Conference on Evolutionary Computation, PP 384-389, 1995.*

[32] Hinterding, R., Michalewicz, Z. and Peachy, T. C., Self-Adaptive genetic algorithm for numeric functions, *Proceedings of the Fourth International Conference on Parallel Problem Solving from Nature, PP 420-429, in Lecture Notes from Computer Science, Springer Verlag, 1996.*

[33] Hoffmeister, F. and Back, T., Genetic algorithms and Evolution strategies: Similarities and differences, *In Shwefel, H-P. and Manner. R., Parallel Problem Solving from Nature- PPSN 1, in Lecture Note in Computer Science, Vol. 496, Springer Verlag, Berlin, 1991.*

[34] Holland, J. H., Adaptation in natural and artificial systems, *Ann Arbor, MI, University of Michigan Press, 1975.*

[35] Holland, J. H., Building Blocks, Cohort genetic algorithms, and hyper plane-defined functions, *Evolutionary Computation Vol. 8, No. 4, PP 373-391, 2000.*

[36] James, C. and Bean, Dynamics of the Multiple Choice Genetic Algorithm with Random Mating, *Proceedings of the First International Conference on Metaheuristics, University of Michigan, 1997.*

[37] Shapiro, J. and Prugel-Bennett, A., Maximum entropy analysis of genetic algorithm operators, *Lecture Notes in Computer Science, 993, PP 14-24, 1995.*

[38] Mitchell, M., Introduction to genetic algorithms, *MIT Press, Cambridge, Massachusetts, London, England, 1997.*

[39] Muhlenbein, H., Evolution in time and space-the parallel genetic algorithm, *In G. Rawlins, editor, Foundations of Genetic Algorithms, PP 316–338, Morgan Kaufmann, San Mateo, CA, 1991.*

[40] Muhlenbein, H., How genetic algorithms really work: I. Mutation and Hill climbing, *Parallel Problem Solving from Nature- PPSN II, PP 15-25, 1992.*

[41] Rechenberg, I., Evolutions strategie: Optimierung technischer systeme nach prinzipien der biologischen evolution, *Frommann, 1973.*

[42] Rosenbrock, H. H., An Automatic method for finding the greatest or least value of a function, *The Computer Journal, Vol. 3, No. 3, PP 175-184, 1960.*

[43] Schaffer, J. D., Caruana, R. A., Eshelman, L. J. and Das, R., A study of control parameters affecting online performance of genetic algorithms for function optimization, *Proceedings of the Third International Conference on Genetic Algorithms and Their Applications, PP 51-60, Morgan Kaufmann, 1989.*

[44] Schlierkamp-Voosen, D. and Muhlenbein, H., Adaptation of population sizes by competing subpopulations, *Proceedings of International Conference on Evolutionary Computation (ICEC'96), Negoya, Japan, PP 330-335, 1996.*

[45] Schwefel, H-P., Numerische optimierung von computer-modellen mittels der evolutionsstrategie, *Volume 26 of Interdisciplinary systems research,. Birkhauser, Basel, 1997.*

[46] Schwefel, H-P., Evolutionary and optimum seeking, S*ixth – Generational Computer Technology Series, whiley, new york, 1995.*

[47] Schwefel, H-P., Collective phenomena in evolutionary system, *In Preprints of the 31st Annual Meeting of the International Society for General System Research, Budapest, Vol. 2, PP 1025-1033, 1987.*

[48] Smith, J. and Fogarty, T., Self-Adaptation of mutation rates in a steady state genetic algorithm, *Proceedings of the Third IEEE Conference on Evolutionary Computation, IEEE Press, 1996.*

[49] Smith, R., Adaptively resizing populations: An algorithm and analysis, *Proceedings of the Fifth International Conference on Genetic Algorithms, P 653 Morgan Kaufmann, 1989.*

[50] Srinivas, M. and Patniak, L. M., Adaptive Probabilities of crossover and mutation in genetic algorithms, *IEEE Transactions on Systems, Man and Cybernetics, Vol. 24, No. 4, PP 17-26, 1994.*

[51] Baluja, S. and Caruana, R., Removing genetics from the standard genetic algorithm, *Proceedings of the Twelfth International Conference on Machine Learning, CA, 1995.*

[52] Torn, A. and Zilinskas, A., Global Optimization, *Lecture Note in Computer Science, Vol. 350, Springer Verlarg, Barlin, 1989.*

[53] Vaclav, H., Sonka, M. and Boyle, B., Image Processing Analysis and machine Vision, *Published by Chapman and Hall, London, Glasgow, First Edition (chapter 4, pp 76-86), 1993.*

[54] Van der Vaart, N. A. L., Towards Totally self-adjusting genetic algorithms: Let nature sort out, *Master Thesis, Leiden University, 1999.*

[55] Vose, M. D., Generalizing the notion of schema in genetic algorithms. *Artificial Intelligence 50(3), 385–396, 199.*

[56] Whitley, L. D., Mathias, K. E., Rana, S. and Dzubera, J., Building better test functions, *In Eshelman, L.J., editor, Proceedings of the Sixth*

International Conference on Genetic Algorithms, PP 239-246, Morgan Kaufmann, San Francisco, California, 1995.