VAASAN YLIOPISTO

GHODRAT MOGHADAMPOUR

Genetic Algorithms, Parameter Control and Function Optimization: A New Approach

ACTA WASAENSIA

No. 160

Computer Science 6

UNIVERSITAS WASAENSIS 2006

Reviewers	Professor Olli Nevalainen Department of Information Technology University of Turku FI-20014 Turku Finland
	Professor Jyrki Nummenmaa Department of Computer Sciences University of Tampere FI-33014 Tampere Finland

I wonder whether knowledge can ever refine humanity so profoundly for a society in which each individual lives with dignity, in harmony with the society and the nature, and realizes that life is too precious to be wasted with hatred.

Preface

I did it! Yes, I just did it! I realized one of my greatest dreams. A dream, which occasionally turned to a nightmare for not being realized, and hope for its realization made me often very emotional. Like any other special event it feels like this should have happened much earlier too. Nevertheless, circumstances drove me to take possibly the most difficult path to get this dream realized. However, after all it now feels good. It actually feels even better. In fact, it feels far beyond better! It feels so freeing and relaxing and lets my thoughts fly high!

I still remember those days in the past and myself sitting all alone in my simple room reading books to escape the bitter reality. Those days I was just dreaming of getting my high school diploma. Since those days, I have gone through numerous exhaustive problems, which would tear a weaker person mentally apart. However, no problem has been able to diminish my love toward science. Nothing has been able to suffocate my natural desire for knowing more. Perhaps since childhood I have also been inspired by the Persian scientist, who in the deathbed asked his friend the solution of a problem.

Doing this research has also put extra challenges to me. I searched and searched for a community with which I could collaborate and possibly would get some support. Unfortunately, I found none! However, I did not give up; with self-confident and determinism I started developing my thoughts all alone and came up with new ideas.

Finnish society has offered me various precious possibilities, and I am deeply grateful for that. Many exciting elements in Finnish culture have impressed me genuinely. However, life in the new society has been far from a fairy tale; everything must have been built from scratch and the trust of each person in the social milieu should have been won. Moreover, occasionally you wish the society would take you more openly without cynicism. Anyhow, my observations suggest that open mind with a touch of happy nature help gravely with the survival process, and hope, determinism and hard work with overcoming problems!

Acknowledgement

I would like to thank my official supervisors, Professor Matti Linna and Professor Seppo Pynnönen for their comments, suggestions, time and all efforts to get this work ready. Professor Matti Linna has been involved in many efforts to forward this thesis. I would also like to thank Dean, Professor Ilkka Virtanen for his positive attitude and letting the reviewing process start as soon as possible. Their efforts and flexibility helped this work become ready now. I would also like to thank Rector, Professor Matti Jakobsson for his sympathy, support and the interest he has shown toward my work.

I would also like to thank the reviewers of this thesis Professor Olli Nevalainen and Professor Jyrki Nummenmaa for their valuable comments. I would like to particularly thank Professor Olli Nevalainen who did much more than a reviewer. With detailed comments and suggestions, he helped me to improve this work much more. I also met him a few years ago and had a discussion around my research.

I would also like to thank Dr. Zbigniew Michalewicz and Dr. William Spears and others in the ECT Digest Newsgroup who replied to my e-mails and provided me with some very useful information.

I would also thank the following foundations for their financial support: Vaasa University Foundation (Vaasan Yliopistosäätiö), Vaasa Technical Academy (Vaasan Teknillinen Seura), Ostrobothnia Chamber of Commerce (Pohjanmaan Kauppakamari) and Finnish Institute for Middle-East (Suomen Lähi-Idän Instituutti). I have done this research during my work at Vaasa Polytechnic and naturally have used the facilities, which I have been provided with through my work. Therefore, I would also like to express my appreciation to Vaasa Polytechnic.

I do thank my mother too. Without the love and care I got from her in the childhood, I would have not been able to surpass all problems throughout my life. I thank my father for his support, and my brother, Khosrow for his courageous acts, which made it possible for me to continue my studies.

I also do thank my lovely daughter Julia Nilufar, who has always been a friendly company for me, and with whom I have shared my thoughts. Hopefully I have been a symbol of a caring father for her.

I thank the other members of my family, relatives, colleagues and friends, particularly friends from the gym for the nice time we have had together almost everyday. Indeed, I do thank each and every person, who has had any sympathy for me and has wished my good, no matter whether we have ever talked or not. Thank you all and I wish you happiness.

Ghodrat Moghadampour
Vaasa April 4, 2006

Contents

Abstract

Moghadampour, Ghodrat (2006). Genetic Algorithms, Parameter Control and Function Optimization: A New Approach. *Acta Wasaensis* No 160, 196 p.

Solving hard optimization problems requires even more efficient implementations of evolutionary algorithms. In addition, difficult questions related to parameter control dilemma in genetic algorithms challenge researchers. New methods to implement genetic algorithms, their operators and parameter control have been developed in the present work. A new parameterless genetic algorithm for numerical optimization problems is presented. The algorithm uses binary representation of the problem, but utilizes real value operations efficiently to implement binary operations, and also to avoid problems and artifacts associated with binary representation. The algorithm implements and heavily utilizes the idea of elitism. The genetic algorithm offers: 1) an efficient and robust method for solving numerical optimization problems, 2) new approaches to implement initialization, breeding and survivor selection, 3) new approaches to implement parameter control in an implicit self-adaptive manner. Male and female chromosomes are presented for the population initialization and crossover operators. The algorithm utilizes the information from the distribution of the fitness value population to select individuals for breeding. During the selection phase, no individual is replaced by a worse offspring. On the other hand, offspring do not need to be better than their parents to be accepted as population members. Any offspring better than the worst individual in the population is accepted as a new population member. Except for the population size, no fixed operator rate is used. Instead, the operator rates are dynamically determined during the run as a function of the population size. Moreover, different operations are iterated as long as they produce better offspring. Several new operators are developed and classical operators are implemented in new ways. Operators used here are: higher quartile crossover, higher quartile variable crossover, higher quartile variable replacement, random building block, integer mutation, decimal mutation, elitist crossover, elitist variable crossover, elitist integer mutation and elitist decimal mutation. The performance of the algorithm was tested and reported in detail on 44 different widely used test functions. The performance of the algorithm was compared to several well-known algorithms including classic genetic algorithm, evolutionary algorithm, differential evolution (DE) and particle swarm (PS) on 36 different test cases. The algorithm presented here proved to be superior to all other algorithms mentioned above.

Ghodrat Moghadampour. Department of Technology and Communication. Vaasa Polytechnic. Wolffintie 30, 65200, Vaasa, Finland.

Keywords: genetic algorithms, parameter control, self-adaptive parameter control, optimization problems, algorithm design, distributional decimal binary genetic algorithm

1. Introduction

In the real world, there are numerous hard problems, which cannot be solved with conventional techniques within reasonable time. Some of these problems are optimization problems. Conventional techniques require rigid assumptions, like convexity, linearity, differentiability, explicitly defined objectives and so on. Moreover, in most cases the only available information regarding the object function is its value. Often, but not always we get away with simplifications such as linearization, convexication and so on. In addition conventional techniques lack generality and new problem solutions require new implementations. (Krink 2005.)

There is an apparent need for developing more efficient algorithms for solving optimization problems. Therefore, the main motivation of this research is to develop an optimization algorithm and experiment with new techniques, which would increase the efficiency of the algorithm. In the following, the general definition of optimization problems and their solution principles are provided.

An optimization problem is a computational problem in which the objective is to find the best possible solution. This objective can be expressed in a more formal way as: find a solution x in the n dimensional search space S, which has the optimum (minimum or maximum) value of the objective function f. In other words:

$$\text{optimize}(x)\,, \tag{1}$$

where $x = (x_1, ..., x_n) \in S \subseteq R^n$ and $\text{lower}(i) \le x_i \le \text{upper}(i),\ 1 \le i \le n$. Here $\text{lower}(i)$ stands for the lower bound and $\text{upper}(i)$ stands for the upper bound of the valid values for variable x_i.

By definition, a point x^* is a local minimum of a function f if there exists some $\varepsilon > 0$ such that $f(x^*) \le f(x)$ for all x with $\left|x - x^*\right| < \varepsilon$. Here $\left|x - x^*\right|$ stands for the distance between x and x^*. On the other hand, a global minimum is a point x^* for which

$f(x^*) \le f(x)$ for all $x \in R^n$. Similarly, a point $x^\#$ is a local maximum of a function f if there exists some $\varepsilon > 0$ such that $f(x^\#) \ge f(x)$ for all x with $\left| x - x^\# \right| < \varepsilon$. A global maximum is a point $x^\#$ for which $f(x^\#) \ge f(x)$ for all $x \in R^n$. (Krink 2005.)

An algorithm is any well-defined computational procedure that takes a set of values as input and produces a set of values as output (Cormen, Leiserson, Rivest & Stein 2003). An algorithm is a method for solving a class of problems on a computer. The complexity of an algorithm is the cost, e.g. running time and storage, of using the algorithm to solve one of these problems. Computing is a time taking process. Computational complexity is the study of the amount of computation effort necessary in order to perform certain kinds of computations. The time complexity of a calculation is measured by expressing the running time of the calculation as a function of some measure of the amount of data needed to describe the problem to the computer. If the running time is at most a polynomial function of the amount of input data, then the calculation is considered easy and otherwise hard. (Wilf 1994, Cormen et al. 2003.)

Complexity is the intrinsic minimum amount of resources, for instance, memory, time, messages, etc., needed to solve a problem or execute an algorithm (Black 2005). The complexity of an optimization problem is determined by its size and the computational effort needed to solve it. Problems, which have a known polynomial algorithm are said to be in class P. Problems for which there exists a non-deterministic algorithm running in polynomial time form the class of NP problems. The core of the NP class consists of so-called NP-hard problems which are really decision or recognition problems, e.g. instead of asking for the optimal-length tour of a traveling sales person (TSP), we look for a tour of length less than L. An algorithm for the recognition version of a problem can be employed to solve the optimization version. According to this theory, if a problem X is such that every problem belonging to the class NP is polynomially transformable to X, we say that X is NP-hard. If in addition problem X itself belongs to NP, X is said to be NP-complete. (Reeves 1993, Cormen et al. 2003.)

There is no known algorithm that can solve all instances of an NP-hard problem in polynomial time, but for some NP-hard problems it is possible to write an algorithm that approximates a solution to the problem in polynomial time (Krink 2005).

NP-hard optimization problems cannot be efficiently solved in an exact way, unless P=NP. Thus, if we want to solve an NP-hard optimization problem by means of an efficient algorithm within polynomial time, we have to accept that the algorithm does not always return an optimal solution but rather an approximate one. (Ausiello, Crescenzi, Gambosi, Kann, Marchetti-Spaccamela & Protasi 2003.)

Solution approaches for optimization problems are classified into three categories: *exact algorithms*, *approximation algorithms* and *heuristic algorithms*. By definition, exact algorithms solve the given optimization problem to optimality; they guarantee to find the globally optimal solution for every instance of the problem. Exact algorithms can be implemented in different ways, of which the most trivial one is *exhaustive search*.

By definition, an approximation algorithm is an algorithm to solve an optimization problem that runs in polynomial time in the length of the input and outputs a solution that is guaranteed to be close to the optimal solution, i.e. a solution in the feasible region with the minimum (or maximum) value of the objective function. The execution time of a computation, $m(n)$ is polynomial when it is no more than a polynomial function of the problem size, n. A function is polynomial if it is the sum of some constants times powers of the argument: $\mathrm{f}(x) = \sum_{i=0}^{k} c_i . x_i^p$. (Black 2005.)

The quality of an approximate solution can be expressed by relative error. Given an optimization problem X, for any instance x of X and for any feasible solution y of x, the relative error of y with respect to x is defined as (Ausiello et al. 2003.):

$$E(x, y) = \frac{|m^*(x) - m(x, y)|}{\max\{m^*(x), m(x, y)\}}, \qquad (2)$$

where $m^*(x)$ denotes the measure of an optimal solution of instance x and $m(x,y)$ denotes the measure of solution y. The relative error is equal to 0 when the solution obtained is optimal, and becomes close to 1 when the approximate solution is very poor. Given an optimization problem X and an approximation algorithm A for X, A is an ε-approximate algorithm for X if, given any input instance x of X, the relative error of the approximate solution $A(x)$ provided by algorithm A is bounded by ε, that is (Ausiello et al. 2003.):

$$E(x, A(x)) \le \varepsilon . \tag{3}$$

"A heuristic is a technique which seeks good (i.e. near-optimal) solutions at a reasonable computational cost without being able to guarantee either feasibility or optimality, or even in many cases to state how close to optimality a particular feasible solution is" (Reeves 1993). An algorithm is called *heuristic* if it "normally" provides near optimal results for hard optimization problems in a reasonable time, without any guarantee on the feasibility of the problem and the quality of the obtained solution. Heuristics are either *metaheuristics*, modifiable to a broad variety of problems, or *problem-specific heuristics* developed for a specific problem area.

A heuristic algorithm can be *constructive* or *improvement* method. A constructive heuristic method starts usually from an "empty solution" and fixes its components successively in either a *deterministic* or a *probabilistic* way. While the solution found by a deterministic heuristic algorithm remains the same from run to run, repeated runs of a probabilistic deterministic algorithm can yield different final solutions. Improvement heuristics start from a feasible solution and iteratively improve that by local reorganizations in its structure. Some classes of heuristic algorithms are further classified to *point-based*, which usually maintain and improve a single feasible solution during their operation, and *population-based*, which maintain a set of feasible solutions and manipulate it.

Basic *local search* methods (Papadimitriou & Steiglitz 1982; Reinelt 1994; Reeves 1995; Arts & Lenstra 1997; Magyar 2000), *simulated annealing* (Aarts & Laarhoven

1987; Aarts & Korst 1989; Reeves 1995; Magyar 2000) and *tabu search* (Glover 1989; Glover 1990; Magyar 2000) are examples of metaheuristic approaches.

The origins of evolutionary computation can be tracked to the late 1950s, but the field remained unknown to the broader scientific community for almost three decades due to the lack of available powerful computer platforms and also some methodological shortcomings of the early approaches (Bäck, Hammel & Schwefel 1997). There has been an increasing interest in imitating living beings to develop powerful algorithms for difficult optimization problems since the 1960s (Gen & Cheng 2000). Problem solving systems based on principles of evolution and hereditary, evolution programs (EPs), generally maintain a population of potential solutions, they have some selection process based on fitness of individuals, and some "genetic" operators. The evolution can be considered as a general optimization process, which deals with highly complex problems. They emulate Darwinian evolution and Mendelian inheritance and optimize the solution by competition and alteration of candidate solutions (Krink 2005). In early evolution programs a floating-point representation was used and mutation was the only recombination operator (Michalewicz 1996). These evolution programs have been applied to optimization problems with continuously changeable parameters, but quite later they were extended for discrete problems (Bäck, Hoffmeister & Schwefel 1991; Herdy 1991; Michalewicz 1996).

Evolutionary algorithms include genetic algorithms (GA), genetic programming (GP), evolutionary programming (EP), evolution strategies (ES) and differential evolution (DE). Related heuristics include simulated annealing (SA), hill-climbing, guided local search (GLS), tabu search (TS), ant system (AS) and particle swarms (PS). (Krink 2005.)

The best known evolution programs include *genetic algorithms*, *evolutionary programming*, *evolution strategies*, and *genetic programming*. Moreover, there are many other evolutionary hybrid systems which incorporate various features of the above paradigms. Differences between various evolutionary techniques are hidden on a lower level; various evolutionary techniques use just different chromosomal representations and appropriate sets of more or less genetic operators (Michalewicz

1996). The common denominator in evolutionary algorithms is that these algorithms deal with a population of individuals, which are selected and altered in an iterative process (Krink 2005).

Evolutionary algorithms can be distinguished from local search techniques in two aspects: 1) they are based on the idea of competition and recombination of candidate solutions in a population, 2) n repeated executions of a local search is different than execution of a population-based heuristic with n individuals. (Krink 2005.)

In an evolutionary algorithm 1) problems are described by a set of parameters, 2) parameters are interpreted as a set of artificial genes, 3) genes are considered as blueprints of individuals and 4) evolution is applied to individuals (Fogel, Owens & Walsh 1966; Rechenberg 1973; Holland 1975; Krink 2005). It is generally accepted that any evolutionary algorithm must have five basic components: 1) a genetic representation of a number of solutions to the problem, 2) a way to create an initial population of solutions, 3) an evaluation function for rating solutions in terms of their “fitness”, 4) “genetic” operators that alter the genetic composition of offspring during reproduction, 5) values for the parameters, e.g. population size, probabilities of applying genetic operators (Michalewicz 1996).

G. Rudolph describes the archetype of evolution strategies in (Rudolph 2000). He explains how the problem of minimizing the total drag of three-dimensional slender bodies in a turbulent flow was the motivation to solve the analytically intractable form design problem with the help of some kind of robot. The robot should perform the necessary experiments by iteratively manipulating a flexible model positioned at the outlet of a wind tunnel. The iterative search strategy, switching to small random changes that were only accepted in the case of improvements, the interpretation of binomially distributed changes as mutations and of the decision to step back or not as selection was the seed for all further developments leading to evolution strategies.

The earliest evolution strategies were based on a population consisting of a single individual only and a mutation operator. The individual was represented as a pair of float-valued vectors, i.e. $v = (x, \sigma)$, where x represents a point in the search space and σ is a vector of standard deviations. Mutation is then realized by replacing x by $x^{t+1} = x^t + N(0, \sigma)$, where $N(0, \sigma)$ is a vector of independent random Gaussian numbers with a mean of zero and standard deviation σ. The offspring x^{t+1} is accepted as a new member of the population if and only if it has better fitness and satisfies all possible constraints of the problem. (Michalewicz 1996.)

Some types of evolution based systems are: 1) a class of evolution strategies which imitate the principles of natural evolution for parameter optimization problems (Rechenberg 1973; Schwefel 1981), 2) Fogel's evolutionary programming which is a technique for searching through a space of small finite-state machines (Fogel, Owens & Walsh 1966), 3) Golver's scatter search techniques which maintain a population of reference points and generate offspring by weighted linear combinations (Glover 1977), 4) Holland's genetic algorithms (GAs) (Holland 1975), and 5) Koza's proposed genetic programming to search for the most fit computer program to solve a particular problem (Koza 1992; Michalewicz 1996; Bäck et al. 1997; Banzhaf, Nordin, Keller & Francone 1998).

The evolution program maintains a population P of n individuals $P(t) = \{x_1^t, ..., x_n^t\}$, where each individual, implemented as some data structure S, represents a potential solution to the problem at hand. At each iteration the "fitness" of all individuals are measured, some individuals are altered by means of "genetic" operators, e.g. mutation: $(m_i : S \rightarrow S')$ and crossover: $(c_j : S_1 \times \cdots S_n \rightarrow S')$, and fitter individuals are selected to form a new population at the next iteration. It is hoped that after a number of generations the best individual represents a near-optimum solution to the problem at hand. (Michalewicz 1996.)

Evolution strategies (ESs) are algorithms used to solve parameter optimization problems. They imitate the principles of natural evolution (Schwefel 1981; Bäck, Hoffmeister & Schwefel 1991; Michalewicz 1996). They adopt a special hill-climbing procedure with self-adapting step sizes σ and inclination angles θ. The major similarity between ESs and GAs are: 1) both maintain populations of potential solutions and 2) both select fitter individuals based on the selection principles.

The differences between ESs and GAs are hidden on the lower level. The first difference is that ESs operate on floating-point vectors and the alteration process may include other "genetic" operators, whereas classical GAs operate on binary vectors. However, the structure of GAs has also been modified for specified applications. These modifications include e.g. variable length strings, richer structures than binary strings, and modified genetic operators. The second difference lies in the selection process itself: in a single generation of the ES, μ parents, by means of recombination and mutation, generate λ offspring constituting an intermediate population of $\mu+\lambda$ individuals, from which λ less fit individuals are removed to create a population of μ individuals. (Michalewicz 1996.)

According to the no-free-lunch (NFL) theorem (Wolpert & Macready 1995; Schwefel 1995) there is no single algorithm for solving all, e.g. optimization problems that is on average superior to any competitor. However, the practical implication is not so dramatic. Those problems, for which no search algorithm is superior, are mostly irrelevant random problems with no correlation between neighbored points. The implication of NFL for EAs is that no set of parameters for an EA is superior on all problems (Krink 2005).

EAs behave better than other methods with respect to solving a specific class of problems while they behave worse for other problem classes. Classical optimization problems are more efficient in solving linear, quadratic, strongly convex, unimodal, separable and many other special problems. On the other hand, EAs do not give up so early when discontinuous, nondifferentiable, multimodal, noisy and otherwise unconventional response surfaces are involved. EAs show inefficiency on the classes of simple problems, but the effectiveness or robustness of them extends to a broader field of applications. (Schwefel 2000.)

As it can be concluded from the introduction, there is an apparent challenge for developing new more efficient evolutionary algorithms which solve hard optimization problems with required precision and take reasonable time and resources. Genetic algorithms and evolutionary strategies both offer interesting techniques. The interesting question is whether it is possible to combine techniques from GAs and ESs to form an efficient genetic algorithm for optimization problems. However, efficiency of a genetic algorithm depends on many factors in its implementation. Some of these factors are problem representation, types of operators and the number of times each operator is implemented, i.e. operator rates. Thus, the main hypothesis of this research can be formulated as:

Hypothesis (Efficiency Boosting Hypothesis*):* *The efficiency of binary genetic algorithms for numerical optimization problems can be improved by 1) introducing chromosomes of the opposite sex in the initialization and crossover operation, 2) injecting better random building blocks to chromosomes, 3) replacing all worse building blocks in a chromosome with the better one from the same chromosome, and 4) fine tuning candidate solutions by limiting the magnitude of changes imposed by genetic operators within the bounds determined by the magnitude of candidate solutions. Also selection for breeding can be successfully implemented straightly based on the population fitness distribution, and operator rates can be defined as a function of the population size.*

This thesis can be divided into three main parts: literature survey, new genetic algorithm description and experimentation. The literature survey is divided into four parts. Chapter 1 offers an introduction to the optimization problems and evolutionary algorithms. Chapters 2 and 3 provide a detailed description of genetic algorithms, their theory and implementation. Chapter 4 covers parameter control techniques in genetic algorithms. Chapter 5 describes the developed algorithm and operators in detail. Chapter 6 covers the experimentation and test results on several widely used benchmark functions. Chapter 7 provides a summary of the results of test runs, and comparison data with some other well-known algorithms. In Chapter 8 main conclusions are drawn.

2. Genetic Algorithms

There is no generally accepted rigorous definition of "genetic algorithm" in the evolutionary-computation community that differentiates GAs from other evolutionary computation methods. GAs have two essential components: survival of the fittest and variation (Goldberg 2000). The idea of a genetic algorithm can be understood as the intelligent exploitation of a random search (Reeves 1993). Most often GAs have at least the following elements in common: populations of chromosomes, selection according to fitness, crossover to produce new offspring, and random mutation of new offspring. Holland's fourth element of GAs—inversion—is rarely used in today's implementations, and its advantages are not well established (Mitchell 1998).

Genetic algorithms borrow a vocabulary from natural genetics. There are a few synonyms for members of a population; *individuals*, *genotypes*, *structures*, *strings* or *chromosomes*. It is assumed that organisms in GAs are *haploid*, i.e. one-chromosome individuals. Units which chromosomes are made of are called *genes*, *features*, *characters*, or *decoders*. These are arranged in linear succession; every gene controls the inheritance of one or several characters. Each place, i.e. *loci*, of the chromosome is reserved for gene of certain character. Each gene may be in several states, i.e. *alleles*, also called feature values. An evolution process run on a population of genotypes aims at finding the optimal solution, i.e. *phenotype* of a chromosome, for the problem at hand.

During this evolution process, exploring the search space may seem conflicting with exploiting the best solutions. GAs strike a remarkable balance between exploration and exploitation of the search space. However, for instance, hill climbing exploits the best solution at the cost of exploring the search space, and random search explores the search space at the cost of exploitations of the promising regions of the space (Michalewicz 1996).

The simplest form of GA involves three types of operators: selection, single point crossover, and mutation. A simple GA works as follows (Mitchell 1998):

1. Start with a randomly generated population of *n* *l*-bit strings (chromosomes)
2. Calculate the fitness *f(x)* of each bit string *x* in the population
3. Repeat the following steps until *n* offspring have been created:
    a. Select a pair of parent bit strings from the current population, the probability of selection being an increasing function of fitness. Selection is done “with replacement”, meaning that the same chromosome can be selected more than once to become a parent.
    b. With probability P_c (crossover probability or crossover rate) cross over the pair at a randomly chosen point (chosen with uniform probability) to form two offspring. If no crossover takes place, form two offspring that are exact copies of their respective parents. In multi-point crossover the crossover rate for a pair of parents is the number of points at which a crossover takes place.
    c. Mutate the two offspring at each locus with probability p_m (the mutation probability or mutation rate), and place the resulting bit strings in the new population
4. Replace the current population with the new population
5. Go to step 2.

In each *run,* the GA is typically iterated for 50 to 500 *generations* (iterations). Each run often produces one or more highly fit chromosomes in the population. Due to the large role of randomness in each run, two runs with different random-number seeds will generally produce different detailed behaviors. Researchers often report these behaviors statistically; e.g., the best fitness found in a run and the generation at which the individual with the best fitness was discovered, averaged over many different runs of the GA on the same problem. The success of the GA algorithm depends greatly on the population size and probabilities of crossover and mutation (Mitchell 1998). It seems that the major factor behind the failure and success of genetic algorithm is domain independence (Michalewicz 1996).

Because of domain independence of GAs, they are unable to deal with nontrivial constraints. A considerable question in designing a chromosome representation is the implementation of constraints on solutions—problem-specific knowledge. In evolution programming different representation structures have different characteristics of suitability for constraint representation. This complicates the problem even more; any problem would require careful analysis for defining appropriate representation and meaningful genetic operators. The issue is selecting “the best” chromosomal representation of solutions together with meaningful genetic operators to satisfy all constraints imposed by the problem (Michalewicz 1996).

A GA will have a good chance of being competitive with or even surpassing methods that do not use domain-specific knowledge in their search procedure, in the following situations (Mitchell 1998): 1) the search space is large, 2) the search space is known not to be perfectly smooth and unimodal, 3) the search space is not well understood, 4) the fitness function is noisy, 5) the task does not require a global optimum to be found.

Otherwise, a GA may converge on a local optimum in a rather small search space. A GA will be surpassed by a gradient-ascent algorithm such as steepest-ascent hill climbing in a smooth or unimodal search space. In addition, domain-specific heuristics can often be designated to outperform GAs in exploiting well-known search spaces (Mitchell 1998).

Genetic algorithms are not generally the best way to solve every problem, but they produce stunning results compared to the traditional minimization approaches since they 1) optimize with continuous or discrete variables, 2) do not require derivative information, 3) simultaneously search from a wide sampling of the cost surface, 4) deal with a large number of variables, 5) are well suited for parallel computers, 6) optimize variables with extremely complex cost surfaces, i.e. they can jump out of a local minimum, 7) provide a list of optimum solutions, not just a single solution, 8) may encode the variables so that the optimization is done with the encoded variables, 9) work with numerically generated data, experimental data or analytical functions (Haupt et al. 2004).

Although the GA theory provides some explanation for why the algorithm converges to the sought optimal point, practical applications do not always follow the theory due to:

1) the coding of the problem often makes GA operate in a space different from the actual problem space, 2) the number of iterations is limited, and 3) the population size is limited (Michalewicz 1996).

Hybridization of genetic algorithms with local search methods and other tailored optimization heuristic is essential when aiming at high-quality solutions. GAs can be hybridized by: 1) departing from classical, bit-string genetic algorithms towards more complex systems, involving the appropriate data structures (Mitchell 1998), 2) incorporating the positive features of the other search algorithm in the hybrid algorithm, and 3) creating crossover and mutation operators for the new type of encoding by analogy with bit string crossover and mutation operators (Davis 1991; Michalewicz 1996).

Hybridization of a GA by incorporating local optimization can be divided in two basic ways (Magyar 2000): 1) Lamarckian evolution (parallel genetic hill-climbing, genetic local search); local optimization on the members of the population is performed every now and then, and individuals are replaced by the locally optimal solutions, 2) Baldwinian strategy; the fitness values of the chromosomes are evaluated by a local optimizing algorithm of their locally optimized variants, but the chromosome itself remains unchanged.

Lamarckian theory is based on two observations: 1) use and disuse – individuals lose characteristics they do not require (or use) and develop characteristics that are useful, 2) inheritance of acquired traits – individuals inherit the traits of their ancestors. Baldwinian theory states that organisms can pass on learned abilities to their offspring. (Wikipedia 2005.)

3. Implementing Genetic Algorithms

There is no rigorous theory on when a GA outperforms other search methods. The performance of a GA depends very much on the details of the implementation such as the method for encoding candidate solutions, the operators, the parameter strings, and the particular criterion for success. Local search in numerical applications seems to be difficult for GAs. GAs are suggested to be used to perform the initial search, before turning local search using domain specific knowledge to guide the local search (Holland 1975; Michalewicz 1996).

GAs are probably the most generally applicable search heuristics for mixed numerical and discrete problems. The knowledge about the problem should be incorporated in the problem representation, i.e. in the encoding and fitness function and also in the operators, like mutation and crossover operators. For continuous numerical problems, the real number encoding should be adopted. Both uniform and non-uniform variance Gaussian mutation and arithmetic crossover should be used. The general scheme with real number mutation is to add a random vector $m = (m_1, m_2, ..., m_n)$ to solution candidate x such that m_i is scaled according to the domain of x_i and $x' = x + m$. In the Gaussian mutation the entries of m are normal distributed by $N(0, \sigma)$. In the uniform mutation the entries of m are uniformly distributed by $U(-\sigma, +\sigma)$. In the uniform variance mutation σ is kept constant while in the non-uniform variance mutation σ is changing over time. In the arithmetic crossover an offspring x' is created from a number of parents usually with $w \sim U(0,1)$, like $x' = wx_1 + (1-w)x_2$ for two parents. Tournament selection of size two or ranking selection, or in some cases preferably the steady-state selection should be used. Also elitism of 1-10% with minimum of one individual should be applied in order to get an efficient GA implementation. (Krink 2005.)

For integer and mixed numerical problems the binary encoding with Gray encoding with fast bit flip mutation and uniform or 1-point crossover are recommended. Here also the tournament selection of size two or ranking selection or in some cases preferably steady-state selection and elitism of 1-10% with minimum of one individual are recommended. (Krink 2005.)

Like natural genetic systems, individual structures are not the focus of attention for GAs. After identifying high performance regions of the search by a GA, it may be useful to invoke a local search routine to optimize the members of the final population (Grefenstette 1987; Michalewicz 1996).

The local search requires the utilization of schemata of higher order and longer defining length than those suggested by the Schema Theorem (Holland 1975; Altenberg 1995; Michalewicz 1996). A schema is built by introducing a *don't care* symbol (*) into the alphabet of genes, like 1**0100* and represents all strings, which match it on all positions other than *. The *order* of a schema defines its speciality and is calculated by subtracting the number of *don't care* symbols from the length of the string. The *defining length* of a schema defines the compactness of information contained in it and is the distance between the first and the last fixed string positions. For problems whose domains of parameters are unlimited, the number of parameters is quite large, and high precision is required, the binary solution vectors should be long. For such problems the performance of GAs has been reported to be quite poor. (Michalewicz 1996.)

3.1 Problem Encoding

Correct problem encoding is essential for genetic algorithms and it affects the efficiency. Any encoding technique should fulfill the following criteria: 1) nonredundancy, 2) legality, 3) completeness, Lamarckian property, 5) causality. Nonredundancy means that the mapping between the encoding and solutions must be one-to-one. Legality means that any permutation of an encoding corresponds to a solution. The completeness means that any solution has a corresponding encoding. The Lamarckian property means that the meaning of alleles for a gene is not context dependent. Generally, any encoding technique is expected to have Lamarckian property so that offspring can inherit goodness from parents. Causality means that small variations on the genotype space due to mutation imply small variations in the phenotype space. Search processes that do not destroy the neighborhood structure are said to exhibit strong causality. Weak causality describes the case where small changes in the genotype space correspond to later changes in the phenotype space, and vice versa. Genetic algorithms work on the genotype space (coding space) and the phenotype space (solution space) alternatively. The genetic operators work on the genotype space and the evaluation and selection operators work on the phenotype space. The mapping from the genotype space to the phenotype space influence the performance of genetic algorithms significantly. Natural selection is the link between chromosomes and the performance of decoded solutions. (Gen & Cheng 2000.)

The encoding methods can be classified according to what kind of symbols is used as the alleles of a gene. The encoding methods can be divided to the following classes: 1) binary encoding for binary, integer and real numbers, 2) real number encoding for real numbers only, 3) integer or literal permutation encoding, 4) general data structure encoding. Binary encoding forces the user to define precision for each variable and it can be directly used for both integer and binary problems. On the other hand, the binary encoding requires a decoding function and it might cause decoding anomalies (Krink 2005). The real number encoding has been widely confirmed to work better than binary encoding for optimizations and constrained optimizations problems and actually considered the best one for solving such kind of problems (Gen et al. 2000).

3.1.1 Binary Encodings

Typically, GA applications use fixed-length, fixed-order bit strings to encode candidate solutions. Due to the earlier work of Holland and his students bit strings are the most common encodings. Moreover, the binary alphabet offers the maximum number of schemata per bit of information of any coding (Goldberg 1989; Michalewicz 1996; Bäck et al. 1997). It also facilitates theoretical analysis and allows elegant genetic operators. Much of the existing GA theory is based on the assumption of fixed-length, fixed-order binary encodings. Heuristics about appropriate parameter settings, e.g. for crossover and mutation rates, have also generally been developed in the context of binary encodings. However, binary encodings are unnatural and unwieldy for problems such as evolving weights for neural networks, and they are prone to rather arbitrary orderings.

3.1.2 Multiple-Character and Real-Valued Encodings

It is most natural for many applications to use an alphabet of multiple characters or real numbers to form chromosomes. Even though Holland's schema-counting argument seems to imply that GAs should exhibit worse performance on multiple-character encodings than on binary encodings, several empirical comparisons have shown better performance for multiple-character or real-valued encodings (Janikow & Michalewicz 1991; Wright 1991; Mitchell 1998). The performance depends very much on the kind of problem and on the details of the GA being used. Moreover, there are no rigorous guidelines for predicting which encoding will work best (Michalewicz 1996). Experimentation indicated that floating-point representation was faster, more consistent from run to run, and provided a higher precision especially with large domains where binary coding would require prohibitively long representation. Furthermore, incorporating floating-point operators, which enhance the accuracy, can be more easily designed. Some more reasons for superiority of floating-point representation are given in (Goldberg 1990; Michalewicz 1996): 1) one gene corresponds to one variable, 2) Hamming cliffs, i.e. the significant differences between binary representations of close numbers, like 15 (01111) and 16 (10000), and other artifacts of mutation operating on bit strings treated as unsigned binary strings are avoided and 3) fewer generations are needed for population conformity.

3.1.3 Tree Encodings

Tree encodings allow the search space to be open-ended, i.e. a tree of any size could be formed via crossover and mutation. On the other hand, this open-endedness may lead to some pitfalls. The formation of more structured, hierarchical candidate solutions can be prevented when trees grow large in an uncontrolled way. Moreover, the resulting large trees can be very difficult to understand and to simplify. So far, there are only very nascent attempts to extend the GA theory to tree encodings (Tackett 1994; O'Reilly & Oppacher 1995; Mitchell 1998).

3.2 Selection Methods

Selection is one of the main operators used in evolutionary algorithms. It does not create any new solution; instead, it selects relatively good solutions from a population and deletes the worse solutions. Thus, the primary objective of the selection is said to emphasize better solutions in a population. The selection operator is a mix of two different concepts; reproduction and selection. *Reproduction* is referred to when one or more copies of a good solution is reproduced. *Selection* is referred to when multiple copies of a solution are placed in a population by deleting some inferior solution (Kalyanmoy 2000). Darwinian natural selection is the principle behind genetic algorithms and provides the driving force for them. Genetic search may terminate with too much pressure due to favoring better solutions too much, and may be too slow with too little pressure due to not favoring better solutions enough (Gen et al. 2000).

After encoding, the second step is to determine the way the GA performs selection—that is, how to emphasize the fitter individuals in the population, and how many offspring each individual will create. The purpose of selection is to drive the population towards increasingly better solutions by weeding out bad solutions. (Krink 2005.)

During the selection process the following actions take place: 1) the objective function is mapped to fitness 2) a probability distribution based on the fitness values of the population is created 3) samples from this distribution are drawn (Grefenstette 2000).

Selection methods include 1) tournament selection, 2) proportional (roulette-wheel) selection, 3) ranking selection, 4) steady-state selection and manual (interactive) selection (Krink 2005). Most common selection types are: 1) roulette wheel selection, 2) $(\mu - \lambda)$-selection, 3) tournament selection, 4) steady-state reproduction, 5) ranking and scaling, 6) sharing (Gen et al. 2000).

The selection procedures can be divided into *dynamics* and *static* methods (Bäck & Hoffmeister 1991; Michalewicz 1996). Static selection requires that the selection probabilities remain constant between generations (cf. rank selection), whereas in a dynamic selection (cf. proportional selection) these probabilities change.

The selection procedures may also be divided into *extinctive*; selection probability of an individual may be zero, and *preservative*, which requires a non-zero selection probability for each individual. Extinctive selection methods are further divided into *left* and *right* selections. In the *left extinctive selection,* premature population convergence due to super individuals is avoided by preventing the reproduction of the best individuals. In the *right extinctive selection,* the best individuals are not prevented form reproduction. Moreover, in a *pure* selection method individuals are allowed to reproduce in only one generation regardless of their fitness (Michalewicz 1996).

Implementations of genetic algorithms are prone to converge prematurely. Researches aimed at eliminating this danger are mostly related to: a) the magnitude and kind of errors introduced by the selection mechanism, and b) the characteristics of the function itself (Michalewicz 1996). Some strategies for combating premature convergence are (Eshelman & Schaffer 1991; Michalewicz 1996): 1) incest prevention, 2) using segmented crossover and shuffle crossover, and 3) detecting duplicate strings in the population. Incest prevention might prohibit an individual from mating with itself, with its parents, with its children, with its siblings, and so on. Segmented crossover is a variant of the multi-point crossover, which allows the number of crossover points to vary. The number of crossover points is replaced by a segment switch rate, which specifies the probability that a segment will end at any point in the string. Shuffle crossover is independent of the number of crossover points. It randomly shuffles the bit positions of the two strings in tandem, 2) crosses the strings and unshuffles the strings.

Two primary factors in genetic search are *population diversity* and *selective pressure*. These are in some sense just other variations of exploration of the search space versus exploitation of optimal solutions. Parameters used for tuning a GA actually affect the balance between the selective pressure and population diversity. A high selective pressure means intensive exploitation of better solutions and the loss of population diversity. On the other hand, a low selective pressure means intensive exploration of the search space and high population diversity at the cost of exploiting optimal solutions (Whitely 1989; Michalewicz 1996).

Striking a balance between the population diversity and the selective pressure seems important in the evolution process of the genetic search. Strong selective pressure would reduce diversity needed for further variation and progress; sub-optimal highly fit individuals would take over the population and cause the premature convergence of the search. On the other hand, a weak selective pressure would result in too slow evolution and make the search ineffective (Michalewicz 1996).

Several variations of the simple selection are presented in (De Jong 1975; Michalewicz 1996). The *elitist model* enforces preserving the best chromosome. It lets a certain proportion of best individuals, the elite in the population be untouched by the operators. It is very useful for all selection schemes and is included in steady-state selection. The elitist model prevents the best found candidate solution from accidentally getting lost and preserves the best found solution(s) as fix points to create offspring in their vicinity (Krink 2005).

The *expected value model* reduces the stochastic errors of the selection routine by introducing a count for each chromosome v. This count is initially set to the ratio of the fitness of the chromosome over the average fitness of the population, i.e. $f(v)/\bar{f}$ and then decreased by 0.5 or 1 when the chromosome is selected for crossover or mutation respectively. A chromosome count below zero would imply that the chromosome is not available for selection any more. Another significant selection method is *stochastic universal sampling*, which uses a single wheel spin, which is spun with a number of equally spaced markers equal to the population size as opposed to a single marker (Baker 1987; Michalewicz 1996).

In *selection-seduction, approach* individuals are allowed to choose their mates: individuals are selected based on their fitness, however, at the time of breeding an individual is allowed to select its mate based on its own preferences, which are formulated in terms of phenotypic characteristics and can constitute a part of the phenotype. (Ronald 1995; Michalewicz 1996).

It is believed that the common cause of premature convergence is the presence of individuals, which are much better than the average individuals of the population, i.e. *super individuals*. Such individuals have a large number of offspring, prevent other individuals from contributing any offspring in next generations, and can cause a rapid convergence to local optimum. This, in turn, is the basis for other sampling methods introducing artificial weights: chromosomes are selected proportionally to their rank rather than actual evaluation values (Baker 1985; Whitely 1989; Michalewicz 1996).

A modified genetic algorithm (modGA) with a different selection method with respect to the classical genetic algorithm is presented in (Michalewicz 1996). In modGA r (not necessarily distinct) chromosomes with fitness values over the average are selected for reproduction, r distinct ones with fitness value below the average are selected to die, and the remaining ones, neutral strings, are copied to the next generation. Depending on the number of selected distinct strings to reproduce and die, the number of neutral strings in a generation is at least $n-2r$ and at most $n-r$, where n is the population size.

In order to protect stronger chromosomes, the new population $P(t+1)$ of modGA is formed from $P(t)$, according to the stochastic universal sampling method, in the following way (Michalewicz 1996):

Step 1: Select r parents from $P(t)$ and mark each copy of selected chromosome as applicable to exactly one fixed genetic operation.
Step 2: Copy $n-r$ distinct chromosomes from $P(t)$ to $P(t+1)$.
Step 3: Let r parents breed to produce exactly r offspring.
Step 4: Insert these r new offspring into population $P(t+1)$.

There are some advantages of this selection method over others mentioned earlier: 1) an individual with a fitness value above average has a good chance to be a member of both

r and $n-r$ elements, 2) to provide a uniform treatment of used genetic operators, they are applied on whole individuals as opposed to individual bits.

The population size is utilized better in modGA; exact multiple copies of the same chromosome are not kept in the new populations. Therefore, it is unlikely that the population represents a decreasing number of unique chromosomes. The similarity between modGAs, steady state GAs (SSGA) and classifier systems is that they may allow only a few individuals to change within each generation. In breeder GA (BGA), (Mühlenbein & Schlierkamp-Vosen 1993; Michalewicz 1996) r best individuals are selected and mated randomly until the number of offspring is equal to the size of the population, so that offspring generation replaces the parent population and the best individual found so far remains in the population.

3.2.1 Fitness-Proportionate Selection

As in Holland's original GA, in fitness-proportionate selection, the number of times an individual is selected to reproduce ("expected value") is its fitness value divided by the average fitness of the population (Mitchell 1998). The most common method for implementing fitness-proportionate selection is "roulette wheel". Each individual is assigned a slice of a circular "roulette wheel", the size of the slice being proportional to the fitness of the individual. The wheel is spun as many times as the number of individuals, *N*. On each spin, the individual under the wheel's marker is selected to be as a member of parents for the next generation. This method can be implemented as follows (Mitchell 1998):

- Sum the total expected value of individuals in the population. Let this sum be *T*.
- Repeat *N* times:
- Choose a random integer *r* between 0 and *T*.
- Loop through the individuals in the population, summing the expected values, until the sum is greater than or equal to *r*. The individual whose expected value puts the sum over this limit is one selected.

In other words, a slot of the interval $[0,1]$ is assigned to each individual according to the ratio of its fitness value over the sum of fitness values of the population individuals and individuals are iteratively selected by the slot that holds a random number in $U(0,1)$. (Krink 2005)

The problem with this stochastic method is that with the relatively small populations typically used in GAs, the actual number of allocated offspring to each individual is often far form its expected value. Moreover, an extremely unlikely series of spins of the roulette wheel could even allocate all offspring to the worst individual in the population (Mitchell 1998).

To minimize the range of possible actual selection frequencies the SUS (stochastic universal sampling) method was proposed (Baker 1987; Mitchell 1998). In SUS, rather than spin the roulette wheel *N* times to select *N* parents, the wheel is spun once—but with *N* equally spaced pointers, which are used to select the *N* parents at one time. However, SUS does not solve the major problems with fitness-proportionate selection, i.e. early in the search, the fitness variance in the population is high and a small number of individuals are much fitter than the others. (Mitchell 1998)

3.2.2 Sigma Scaling

In "sigma scaling" (Forrest 1985; Mitchell 1998) selection is kept relatively independent of the fitness variance in the population over the course of the run. When the standard deviation of the fitness is high, the fitter individuals will not be allocated too many offspring. Likewise, when the population is more converged, the standard deviation is typically low; the fitter individuals will stand out more. An individual's expected value is a function of its fitness, the population mean, and the population standard deviation. For example, the following is a sigma scaling function:

$$\mathrm{ExpVal}(i,t) = \begin{cases} 1 + \dfrac{f(i) - \bar{f}(t)}{2\sigma(t)} & \text{if } \sigma(t) \neq 0 \\ 1.0 & \text{if } \sigma(\mathrm{t}) = 0 \end{cases}, \tag{4}$$

where $\mathrm{ExpVal}(i,t)$ is the expected value of individual i at time t, $f(i)$ is the fitness of i, $\bar{f}(t)$ is the average fitness of the population at time t, and $\sigma(t)$ is the standard deviation of the population fitness at time t. For instance, for an individual, which has the fitness of two standard deviations above the population mean fitness value, i.e. $f(i)=\bar{f}(t)+2\sigma$, the expected offspring will be 2.

3.2.3 Elitism

Elitism (De Jong 1975; Mitchell 1998) is an addition to many selection methods, which can significantly improve GA's performance. It forces GA to retain some number of the best individuals intact at each generation. These individuals are not mutated nor crossed over, but used for reproduction (Mitchell 1998).

Normally individuals in a population are expected to have a lifetime of one generation. However, when elitist strategies are employed, individuals with higher fitness values are expected to have a longer lifetime than on generation. Elitist strategies link the lifetimes of individuals to their fitness values and keep good solutions in the population longer than one generation. The elitist strategies are deemed necessary when genetic algorithms are used as function optimizers and the goal is to find a global optimal solution. However, elitist strategies tend to make the search more exploitative rather than explorative and may not work for problems with multiple optimal solutions (Sarma & De Jong 2000)

3.2.4 Boltzmann Selection

The Boltzmann selection mechanisms use principles from simulated annealing to control the selection pressure in evolutionary algorithm thermodynamically. The mechanics of the recombination and neighborhood operators are critical to the generation of the proper temporal population distributions. It is often impossible to separate the selection from the rest of the EA when Boltzmann selection is employed. (Mahfoud 2000).

The idea of Boltzmann selection is to have different amounts of selection pressure at different times in a run, e.g. letting selection occur early on run slowly while maintaining a lot of variation in the population, and stronger later to heavily emphasize highly fit individuals. More accurately, in Boltzmann selection, a continuously varying "temperature" controls the selection pressure according to a preset schedule. The temperature starts out high, which means every individual has some probability of reproducing. The temperature is gradually lowered, thereby allowing the GA to focus on to the best part of the search space while maintaining the "appropriate" degree of diversity (Mitchell 1998). A typical implementation of the Boltzmann selection is to assign each individual i an expected value:

$$\mathrm{ExpVal}(i,t) = \frac{e^{f(i)}/T}{\left\langle e^{f(i)}/T \right\rangle_t}, \tag{5}$$

where T is temperature and $\langle\ \rangle_t$ denotes the average over the population at time t. Experiments have shown that as T decreases, the difference in $\mathrm{ExpVal}(i,t)$ between high and low fitness value increases, however, the hope is that this happens gradually over the course of the search (Mitchell 1998).

The idea behind Boltzmann selection is competition between current solution i and alternative solution j in which i wins with the following logistic probability:

$$\frac{1}{1+e^{(f_i - f_j)/T}}, \tag{6}$$

where T is temperature and f_i is the energy, cost or objective function value of solution i in case of minimization. The Boltzmann selection can be implemented in slightly different ways, but all variations essentially accomplish the same thing when iterated; at fixed T, given a sufficient number of Boltzmann trials, a Boltzmann distribution arises among the winning solutions over time. (Mahfoud 2000)

3.2.5 Rank Selection

Rank-based selection or ranking means that the probability of selection is determined only by rank ordering of the fitness of the individuals within the current population

(Grefenstette 2000). The purpose of rank selection is to prevent too quick convergence. In rank selection, the individuals in the population are ranked according to the fitness, and the expected value of each individual depends on its rank rather than on its absolute fitness (Baker 1985; Mitchell 1998).

The idea with rank selection is to sort the individuals by their fitness values, assign a linearly decreasing probability according to the rank and select the parents as in proportional selection. (Krink 2005.)

The fitness values are not scaled in this method, since absolute differences in fitness are obscured. Discarding absolute fitness information can, on one hand, prevent leading to convergence problems, and, on the other hand, prevent knowing how far fitter an individual is from its nearest competitor. Rank selection reduces the selection pressure when the fitness variance is high, and keeps up selection pressure when the fitness variance is low: the ratio of expected values of individuals ranked *i* and *i+1* will be the same independent of their absolute fitness difference (Mitchell 1998).

There are many methods to assign a number of offspring based on ranking. For instance, as the upper bound for the expected number of offspring, a user defined value MAX is taken and a linear curve through MAX is taken such that the area under the curve equals the population size. In this way, the difference between the expected numbers of offspring for "adjacent" individuals can be easily determined, e.g. if $MAX = 2$ and population size is 50, the difference between expected numbers of offspring between adjacent individuals would be 0.04. (Baker 1985; Michalewicz 1996).

Another possibility is to rank each individual in the population in increasing order of fitness, from 1 to N. The user chooses the expected value Max of the individual with rank N, with $Max \geq 0$. The expected value of each individual i at time t is given by:

$$\mathrm{ExpVal}(i,t) = Min + (Max - Min)\frac{rank(i,t) - 1}{N - 1}, \tag{7}$$

where Min is the expected value of the individual with rank 1. Given the constraints $Max \geq 0$, and since population size stays constant, we have:

$$\sum_i \mathrm{ExpVal}(i,t) = \frac{N}{2}(Min + Max) + Max - Min = N\,. \tag{8}$$

This implies that $1 \leq Max \leq 2$ and $Min = 2 - Max$.

The increased preservation of diversity resulting from ranking leads to more successful search than the quick convergence resulting from fitness proportionate selection in many cases. However, in rank selection lowering the selection pressure means that the GA will in some cases be slower in finding highly fit individuals. There are various ranking schemes, such as exponential rather than linear. In ranking schemes, SUS methods can be used for choosing parents after expected values have been assigned (Mitchell 1998).

The rank selection can also be implemented so that for a population of size n a user defined parameter q is taken and a linear function, like:

$$prob(rank) = q - (rank - 1)r \tag{9}$$

or a nonlinear function, like:

$$prob(rank) = q(1-q)^{rank-1} \tag{10}$$

is defined. In both functions $rank = 1$ means the best individual and $rank = n$ the worst one. Both functions return the probability of an individual ranked in position $rank$ to be selected in a single selection, and both schemas allow the users to influence the selective pressure of the algorithm.

Rank selection methods can improve GA behavior in some cases, i.e. they prevent scaling problems, they control better the selective pressure and coupled with one-at-a-time reproduction, they give the search a greater focus (Baker 1987; Whitely 1989; Michalewicz 1996). On the other hand, they have some apparent drawbacks: 1) they put responsibility on the user to decide when to use these mechanisms 2) they ignore the information about the relative evaluations of different chromosomes 3) regardless of the magnitude of the problem they treat all cases uniformly 4) they violate Schema Theorem (Michalewicz 1996).

3.2.6 Tournament Selection

In tournament selection, a group of q individuals is randomly drawn from the population with or without replacement. This group takes part in a tournament, where a winning individual is determined by its fitness value. The best individual is usually chosen deterministically though occasionally a stochastic selection may be made. (Blickle 2000.)

In tournament selection, two individuals are chosen at random from the population. A parameter P, e.g. 0.75, and a random number r between 0 and 1, are then chosen. If $r < P$, the fitter of the two individuals is selected to be a parent; otherwise the less fit individual is selected. The two are then returned to the original population and can be selected again. (Mitchell 1998.)

The tournament selection combines the idea of ranking in a very interesting and efficient way: in a single iteration some number k, typically 2, of individuals are selected and the best ones are passed to the next generation. This process is repeated as many times as the size of population. For minimization problems the selection process may be flavored with simulated annealing by considering the Boltzmann selection method, where two elements, i and j compete with each other and the winner is determined according to:

$$\mathrm{P} = \frac{1}{1 + e^{\frac{f(i)-f(j)}{T}}}, \tag{11}$$

where T is temperature and $f(i)$ and $f(j)$ are values of the objective function for elements i and j respectively. The fitter individual is selected if a uniform random number is smaller than P. (Michalewicz 1996).

Selection pressure changes in tournament selection in the similar way as in rank selection, but it is computationally more efficient and more amenable to parallel implementation (Mitchell 1998).

3.2.7 Steady-State Selection

A typical GA is “generational”, i.e. at each generation the population consists of almost all new offspring formed by parents in the previous generation. In contrast, in steady-state selection, only a small number of the least fit individuals are replaced by offspring resulting from crossover and mutation of the fittest individuals. Steady-state GAs are often used in evolving systems in which incremental learning is important and not only one individual, but all members of the population collectively solve the problem at hand, e.g. classifier systems (Holland 1986; Mitchell 1998).

In steady-state selection, a small number λ (often less than 5) of offspring are created in each iteration and inserted into the population. All individuals are then sorted according to their fitness values and the worst λ individuals are discarded. (Krink 2005)

3.2.8 Interactive Selection

The idea with manual or interactive selection is that the user selects individuals. This approach does not require any fitness function and lets human intuition and judgment be incorporated in the selection phase. However, this selection approach might be slow, subjective and require expert knowledge. (Krink 2005.)

3.2.9 Sharing Selection

Sharing selection techniques were first introduced for multimodal function optimization in (Goldberg & Richardson 1987) and are used to maintain the diversity of population.

The idea with sharing is to reduce the fitness of individuals that have highly similar members within the population. This rewards individuals that uniquely exploit areas of the domain, while discouraging highly similar individuals in a domain. This causes population diversity pressure, which helps maintain population members at local optima.

The shared fitness of an individual i is given by $f_{sh,i} = \frac{f_i}{m_i}$, where f_i is the raw fitness of the individual, and m_i is the niche count, which defines the amount of overlap (sharing) of the individual i with the rest of the population. The niche count is calculated by summing a sharing function over all members of the population: $m_i = \sum_{j=1}^{N} sh(d_{i,j})$. The distance $d_{i,j}$ represents the distance between individual i and individual j in the population determined by the similarity metric:

$$sh(d_{i,j}) = \begin{cases} 1 - \left(\dfrac{d_{i,j}}{\sigma_{sh}} \right)^{\alpha_{sh}} & \text{if } d_{i,j} < \sigma_{sh}, \\ 0 & \text{otherwise.} \end{cases} \tag{12}$$

Each member of the population is considered to be the center of a niche, and its shared fitness value is reduced for every other member of a population whose distance is less than a niche radius σ_{sh} from that individual (Miller & Shaw 1996).

A sharing function is used to determine the degradation of an individual's fitness due to a neighbor at some distance. Degradation determines the reproduction probability of individuals so that reproduction is restrained in a crowd peak while other individuals are encouraged to give offspring (Gen et al. 2000).

3.3 Genetic Operators

For any evolutionary computation technique, the representation of an individual in the population and the set of operators used to alter its genetic code constitute probably the two most important components of the system. Therefore, a representation of object variables must be chosen along with the appropriate evolutionary computation operators. The reverse is also true; operators must match the representation. Data might be represented in different formats: binary strings, real-valued vectors, permutations, finite-state machines, parse trees and so on. Decision on what genetic operators to use

greatly depends on the encoding strategy of the GA. For each representation, several operators might be employed. (Michalewicz 2000.)
The most commonly used genetic operators are crossover and mutation. These operators are implemented in different ways for binary and real-valued representations. In the following, these operators are described in more details.

3.3.1 Crossover

Crossover is the main distinguishing feature of a GA. The simplest form of crossover is single-point: a single crossover position is chosen randomly and the parts of the two parents after the crossover position are exchanged to form two new individuals (offspring). The idea is to recombine building blocks (schemas) on different strings. However, single-point crossover has some shortcomings. For instance, segments exchanged in the single-point crossover always contain the endpoints of the strings, it treats endpoints preferentially, and cannot combine all possible schemas. For example, it cannot combine instances of 11*****1 and ****11** to form an instance of 11**11*1 (Mitchell 1998). Moreover, the single-point crossover suffers from "positional bias" (Eshelman, Caruana & Schaffer 1989; Mitchell 1998): the location of the bits in the chromosome determines the schemas that can be created or destroyed by crossover.

Consequently, schemas with long defining lengths are likely to be destroyed under single-point crossover. The assumption in single-point crossover is that short, low-order schemas are the functional building blocks of strings, but the problem is that the optimal ordering of bits is not known in advance (Mitchell 1998). Moreover, there may not be any way to put all functionally related bits close together on a string, since some particular bits might be crucial in more than one schema. Furthermore, the tendency of single-point crossover to keep short schemas intact can lead to the preservation of so-called hitchhiker bits. These are bits that are not part of a desired schema, but by being close on the string, hitchhike along with the reproduced beneficial schema. (Eshelman, Caruana & Schaffer 1989; Mitchell 1998).

In two-point crossover, two positions are chosen at random and the segments between them are exchanged. Two-point crossover reduces positional bias and endpoint effect, it is less likely to disrupt schemas with large defining lengths, and it can combine more schemas than single-point crossover (Mitchell 1998). Two-point crossover has also its own shortcomings; it cannot combine all schemas.

Multipoint-crossover has also been implemented, e.g. in one method, the number of crossover points for each parent is chosen from a Poisson distribution whose mean is a function of the length of the chromosome. Another method of implementing multipoint-crossover is the "parameterized uniform crossover" in which each bit is exchanged with probability p, typically $0.5 \leq p \leq 0.8$ (Spears & De Jong 1991; Mitchell 1998). In parameterized uniform crossover, any schemas contained at different positions in the parents can potentially be recombined in the offspring; there is no positional bias. This implies that uniform crossover can be highly disruptive of any schema and may prevent coadapted alleles from ever forming in the population. (Mitchell 1998)

There has been some successful experimentation with a crossover method, which adapts the distribution of its crossover points by the same process of survival of the fittest and recombination (Schaffer et al. 1987; Michalewicz 1996). This was done by inserting into the string representation special marks, which keep track of the sites in the string where crossover occurred. The hope was that if a particular site produces poor offspring, the site dies off and vice versa.

Spears (1991) considered a combination of two crossovers methods, the one-point and uniform crossover method by extending a chromosomal representation by additional bit. There has also been some experimentation with other crossovers: segmented crossover and shuffle crossover (Eshelman, Caruana & Schaffer 1989; Michalewicz 1996). Segmented crossover, a variant of the multi-point, allows the number of crossover points to vary. The fixed number of crossover points and segments (obtained after dividing a chromosome into pieces on crossover points) are replaced by a segment switch rate, which specifies the probability that a segment will end at any point in the string. The shuffle crossover is an auxiliary mechanism, which is independent of the number of the crossover points. It 1) randomly shuffles the bit positions of the two strings in tandem, 2) exchanges segments between crossover points, and 3) unshuffles the string (Michalewicz 1996). In gene pool recombination, genes are randomly picked from the gene pool defined by the selected parents.

There is no definite guidance on when to use which variant of crossover. The success or failure of a particular crossover operator depends on the particular fitness function, encoding, and other details of GA. Actually, it is a very important open problem to fully understand interactions between particular fitness function, encoding, crossover and other details of a GA. Commonly, either two-point crossover or parameterized uniform crossover has been used with the probability of occurrence $p \approx 0.7 - 0.8$ (Mitchell 1998).

Generally, it is assumed that crossover is able to recombine highly fit schemas. However, there is even some doubt on the usefulness of crossover, e.g. in schema analysis of GA, crossover might be considered as a "macro-mutation" operator that simply allows for large jumps in the search space (Mitchell 1998).

3.3.2 Mutation

The common mutation operator used in canonical genetic algorithms to manipulate binary strings $a = (a_1,...a_\ell) \in I = \{0,1\}^\ell$ of fixed length ℓ was originally introduced by Holland (1975) for general finite individual spaces $I = A_1 \times ... A_\ell$, where $A_i = \{\alpha_{i_1},...,\alpha_{i_{k_i}}\}$. By this definition, the mutation operator proceeds by:

i. determining the position $i_1,...,i_h (i_j \in \{1,...,l\})$ to undergo mutation by a uniform random choice, where each position has the same small probability p_m of undergoing mutation, independently of what happens at other position
ii. forming the new vector $a_i' = (a_1,...,a_{i_1-1}, a_{i_1}', a_{i_1+1},...,a_{i_h-1}, a_{i_h}', a_{i_h+1},...a_\ell)$, where $a_i' \in A_i$ is drawn uniformly at random from the set of admissible values at position i.

The original value a_i at a position undergoing mutation is not excluded from the random choice of $a_i' \in A_i$. This implies that although the position is chosen for mutation, the corresponding value might not change at all. (Bäck, Fogel, Whitley & Angeline 2000.)

Crossover is commonly viewed as the major instrument of variation and innovation in GAs, with mutation, playing a background role, insuring the population against permanent fixation at any particular locus (Mitchell 1998; Bäck, Fogel, Whitley & Angeline 2000). Mutation and crossover have the same ability for "disruption" of existing schemas, but crossover is a more robust constructor of new schemas (Spears 1993; Mitchell 1998). The power of mutation is claimed to be underestimated in traditional GA, since experimentation has shown that in many cases a hill-climbing strategy works better than a GA with crossover (Mühlenbein 1992; Mitchell 1998).

While recombination involves more than one parent, mutation generally refers to the creation of a new solution form one and only one parent. Given a real-valued representation where each element in a population is an $n-$dimensional vector $x \in \Re^n$, there are many methods for creating new offspring using mutation. The general form of mutation can be written as:

$$x' = m(x), \tag{13}$$

where x is the parent vector, m is the mutation function and x' is the resulting offspring vector. The more common form of mutation generated an offspring vector:

$$x' = x + M, \tag{14}$$

where the mutation M is a random variable. M has often zero mean such that $E(x') = x$; the expected difference between a parent and its offspring is zero. (Bäck, Fogel, Whitley & Angeline 2000.)

Some forms of evolutionary algorithms apply mutation operators to a population of strings without using recombination, while other algorithms may combine the use of mutation with recombination. Any form of mutation applied to a permutation must yield a string, which also presents a permutation. Most mutation operators for permutations are related to operators, which have also been used in neighborhood local search strategies. (Whitley 2000). Some other variations of the mutation operator for more specific problems have been introduced in Chapter 32 in (Bäck et al. 2000).

It is not a choice between crossover and mutation but rather the balance among crossover, mutation, selection, details of fitness function and the encoding. Moreover, the relative usefulness of crossover and mutation change over the course of a run. However, all these remain to be elucidated precisely (Mitchell 1998).

3.3.3 Other Operators and Mating Strategies

In addition to common crossover and mutation there are some other operators used in GAs including inversion, gene doubling and several operators for preserving diversity in the population. For instance, a “crowding” operator has been used in (De Jong 1975; Mitchell 1998) to prevent too many similar individuals (“crowds”) from being in the population at the same time. This operator replaces an existing individual by a newly formed and most similar offspring.

The same result can be accomplished by using an explicit “fitness sharing” function (Goldberg & Smith 1987; Mitchell 1998) whose idea is to decrease each individual’s fitness by an explicit increasing function of the presence of other similar population members. In some cases, this operator induces appropriate “speciation”, allowing the population members to converge on several peaks in the fitness landscape (Goldberg et al. 1987; Mitchell 1998). However, the same effect could be obtained without the presence of an explicit sharing function (Smith, Forrest & Perelson 1993; Mitchell 1998).

Diversity in the population can also be promoted by putting restrictions on mating. For instance, distinct “species” tend to be formed if only sufficiently similar individuals are allowed to mate (Deb & Goldberg 1989; Mitchell 1998). Another attempt to keep the entire population as diverse as possible is disallowing mating between too similar individuals, "incest" (Eshelman 1991; Eshelman & Schaffer 1991; Mitchell 1998). Another solution is to use a "sexual selection" procedure; allowing mating only between individuals having the same "mating tags"—parts of the chromosome that identify prospective mates to one another. These tags, in principle, would also evolve to implement appropriate restrictions on new prospective mates (Holland 1975; Booker 1985; Mitchell 1998).

Another solution is to restrict mating spatially. The population evolves on a spatial lattice, and individuals are likely to mate only with individuals in their spatial neighborhoods. Such a scheme would help preserve diversity by maintaining spatially isolated species, with innovations largely occurring at the boundaries between species (Hillis 1992; Mitchell 1998).

3.4 Genetic Local Search

The basic idea with local search is to iteratively refine a candidate solution by local search of promising neighbors. Local search heuristics can be divided to: 1) deterministic local search, 2) stochastic local search, 3) simulated annealing, 4) tabu search and guided local search. In deterministic local search, a starting point x is first selected and evaluated and then the search in the neighborhood of x is repeated until either a better point x' has been found or all neighboring points have been checked. In stochastic local search, first, a starting point x is selected and then the search for a better neighbor value is iterated for a number n of times. In each iteration, a value x' in the neighborhood of x is selected and evaluated. However, the new point will replace the starting point only if a uniformly generated random number $U(0,1)$ fulfills the following criteria:

$$U(0,1) \leq \frac{1}{1+e^{\frac{eval(x')-eval(x)}{T}}}. \tag{15}$$

Here T is a user define constant. Simulated annealing is a stochastic local search with decreasing value of T. The tabu search is a stochastic local search that accepts inferior neighbor points, but it prevents the revisiting of previously examined solutions. It keeps a memory of recently changed attributes of the candidate solutions, which may not be changed for a certain amount of time. Inferior solutions are selected if better solutions are "tabu". (Krink 2005.)

Genetic algorithms and local search heuristics can be combined for solving optimization problems. Typically, a hybrid genetic algorithm involves incorporating local optimization as an add-on extra to a canonical genetic algorithm. Here local optimization is applied to each newly generated offspring to move it to a local optimum before injecting it into population. With this hybrid approach, the genetic search performs global exploration among the population, whereas local search performs local exploitation around chromosomes. These complementary methods often outperform either method operating alone (Gen et al. 2000).

Two common forms of genetic local search feature 1) Lamarckian and 2) Baldwin effects (Whitely, Gordan & Mathias 1994; Gen et al. 2000). Both approaches use the metaphor that an individual learns (hill climbs) during its lifetime (generation). In the Lamarckian approach, the resulting individual is put back into the population after hill climbing. In the Baldwinian approach, the genotype remains unchanged but the fitness is changed. Experimentations on some test problems have shown that the Baldwinian search strategy can sometimes converge to global optimum when the Lamarckian strategy converges to a local optimum using the same form of local search. However, experiments showed that the Baldwinian strategy is much slower than the Lamarckian strategy (Gen et al. 2000).

4. Control Parameters in Genetic Search

Evolutionary algorithms are affected by more parameters than optimization methods typically. This is at the same time a source of their robustness as well as a source of frustration in designing them (Michalewicz & Fogel 2004). Adaptation can be used not only for finding solutions to a given problem, but also for tuning genetic algorithms to the particular problem (Gen et al. 2000).

Adaptation can be applied to problems as well as to evolutionary processes. In the first case adaptation modifies some components of genetic algorithms to provide an appropriate form of the algorithm, which meets the nature of the given problem. These components could be any of representation, crossover, mutation and selection. In the second case, adaptation suggests a way to tune the parameters of the changing configuration of genetic algorithms while solving the problem (Gen et al. 2000).

Some of such parameters are: 1) population size and structure, like subpopulations, 2) genome representation (floating point, binary, parse tree, matrix), precision and length, 3) crossover type (arithmetic, n-point, etc.), the number of crossover points and probability, 4) mutation type (uniform, Gaussian, etc.), mutation variance and probability, 5) selection type (tournament, proportional, etc.), tournament size. (Krink 2005.)

The challenge is that optimal parameters of an EA are problem dependent and there is a large set of possible EA settings. The No-Free-Lunch theorem implies that no set of parameters for an EA is superior on all problems (Krink 2005). Finding the right parameter values is a time-consuming task and it has been the subject of many researches.

The main criteria for classifying parameter setting methods are: 1) what is changed, 2) how the change is made. The first criterion refers to the components of the evolutionary algorithm and consists of six categories: 1) representation, 2) evaluation function, 3) variation operators (mutation and recombination), 4) selection, 5) replacement and 6) population. The second criterion refers to the parameter setting methods, which can be divided to three main types: 1) deterministic (or fixed) parameter control (also called parameter tuning) in which the parameter-altering transformations takes no input variables related to the progress of search method, 2) adaptive (also called explicitly adaptive) parameter control in which there is some form of feedback from the search, 3) self-adaptive (also called implicitly adaptive) parameter control in which the parameters to be adapted are encoded into the chromosomes and undergo mutation and recombination (Eiben, Hinterding & Michalewicz 1999).

Parameter setting can be classified to 1) absolute and 2) empirical. It can also be classified to 1) uncoupled and 2) tightly-coupled. However, the uncoupled/absolute category encompasses deterministic and adaptive control, whereas the tightly-coupled/empirical category corresponds to self-adaptive control (Angeline 1995; Spears 1995; Eiben et al. 1999).

Figure 1 shows the division of parameter control techniques. The main criterion is whether parameter setting happens before the run or during the run. Methods used for setting parameters before the run, i.e. parameter tuning methods can use constant parameter values or functions to set these parameters beforehand. Methods used for setting parameters during the run, i.e. parameter control methods can be divided to deterministic, adaptive, self-adaptive and population-structure-based techniques. In the following, each of these techniques is described in detail.

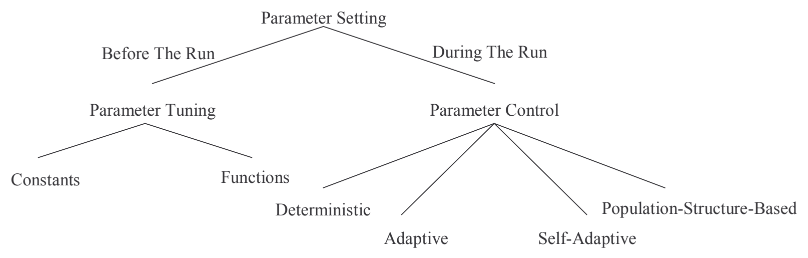

Figure 1. Taxonomy for parameter setting methods.

4.1 Parameter Tuning

Parameter tuning refers to the approach that amounts to finding good fixed values for the parameters before the run of the algorithm. In this approach, an attempt is made to find the optimal and general set of parameters applicable to a wide range of optimization problems. The parameter values may be either constant or calculated by a simple parameterized function of the generation counter (Ursem 2003). Two main approaches explained below were tried to do this.

4.1.1 Pre-Defined Parameter Values

The simplest possible method for parameter tuning is to use constant values. The interesting observation is that fixing a large part of parameters will not decrease the performance of EA significantly (Ursem 2003). A considerable effort has been put into finding parameter values for a traditional GA (bit-representation, one-point crossover, bit-flip mutation and roulette wheel selection), which were good for a number of numeric test problems (De Jong 1975; Eiben et al. 1999). As a result, the following parameters are considered the best values for the test functions: population size: 50, probability of crossover: 0.6, probability of mutation: 0.001, generation gap: 100%, scaling window: $W = \infty$ and selection strategy: elitist.

Generation gap defines the percentage of the population to be replaced during each generation. The remainder of the population is chosen at random to survive intact. Generation gap refers to the amount of overlap between parents and offspring. In an overlapping model parents and offspring compete for survival; the selection pool for deletion consists of both parents and their offspring. In a nonoverlapping model, parents are always replaced by offspring (Sarma et al. 2000).

Scaling is used to avoid problems with fitness-proportionate selection by scaling the fitness function to the worst individual in the population. This means that the measure of fitness used in calculating the probability for selecting an individual is not the individual's absolute fitness, but its fitness relative to the worst individual in the population (Eshelman 2000; Grefenstette 2000). *Scaling window* is a parameter used to control the search with respect to the characteristic of the function being optimized. For a minimization function the evaluation function *eval* usually returns $eval(x) = F - f(x)$, where F is a constant such that $F > f(x)$ for all x. The scaling window W allows the user to control how often the constant F is updated. If $W > 0$, F is set to the greatest value of $f(x)$ which has occurred in the last W generations. A value $W = 0$ indicates an infinite window, i.e. $F = \max\{f(x)\}$ over all evaluations (Michalewicz 1996).

A good starting point for the bit-flip mutation operation in binary encoding is $p_m = 1/L$, where L is the length of the chromosome. This mutation rate has also been proven optimal for the sphere problem (Bäck 1993; Bremermann, Rogson & Salaff 1996; Ursem 2003). Since $1/L$ corresponds to flipping one bit per genome on average, it is used as a lower bound for mutation rate.

Schaffer et al. recommend a range of $p_m \in [0.005, 0.01]$ for the (Schaffer, Caruana, Eshelman & Das 1989; Ursem 2003). For real-value encoding the mutation rate is usually $p_m \in [0.6, 0.9]$ and the crossover rate is $p_c \in [0.7, 1.0]$. In tournament selection, tournament size of two has often given the best results. A much higher tournament size imposes too strong selection pressure, which results to overall premature convergence. Unless the fitness evaluation is very time-consuming the population size should typically be set to more than 50. (Ursem 2003)

However, the main disadvantages of non-adaptive control techniques are that they require a significant amount of manual tuning due to the stochastic nature of the algorithms and the parameter values are problem dependent. In addition, parameters may be correlated and not independent of each other, which leads to a combinatorial explosion regarding the number of settings to try. Manual tuning of parameters implies searching the whole universe of evolutionary algorithms. (Ursem 2003)

4.1.2 Parameter Definition by Meta-algorithm

The search process in an EA typically shifts from highly explorative in the beginning of the run to fine-tuning towards the end. This leads to the need for keeping the variance in Gaussian mutation high in the beginning and decreasing it to perform the fine-tuning at the end of the run (Ursem 2003). Grefenstette used a GA as a meta-algorithm to optimize values for the same parameters for both on-line and off-line performance of the algorithm (Grefenstette 1986). On-line performance refers to monitoring the best solution in each generation, while off-line performance takes all solutions in the population into account (Eiben et al. 1999). The best set of parameters to optimize the on-line performance of the GA was found to be: population size: 30, probability of crossover: 0.95, probability of mutation: 0.01, generation gap: 100%, scaling window: $W = 1$ and selection strategy: elitist. Respectively the best set of parameters to optimize the off-line performance of the GA was found to be: population size: 80, probability of crossover: 0.45, probability of mutation: 0.01, generation gap: 90%, scaling window: $W = 1$ and selection strategy: non-elitist.

Any static set of parameters seems to be generally inappropriate and against the contemporary view of an EA, which acknowledges that specific problem or problem types require specific EA setup for satisfactory performance (Bäck, Fogel & Michalewicz 1997; Eiben et al. 1999).

A general drawback for such efforts is that runs of an EA are intrinsically dynamic and adaptive; hence, different values of parameters might be optimal at different stages of the evolutionary process (Davis 1989; Hesser & Männer 1991; Syswerda 1991; Bäck 1992 a; Bäck 1992 b; Bäck 93; Soule & Foster 1997; Eiben et al. 1999). The optimal parameters are not constant during the run of EAs (Ursem 2003). From a theoretical point of view the problem with general optimal parameters is that, no search algorithm is superior on all problems (Wolpert et al. 1995; Ursem 2003). Therefore, the use of rigid parameters that do not change their values during the run is in contrast to this spirit.

4.2 Parameter Control

Parameter control refers to the approach, which amounts to starting a run with initial parameter values, which are changed during the run. An evolutionary algorithm is normally used not only to solve a problem but also to adapt the same algorithm to the particular problem. Tuning parameters during the run can be implemented mainly in three different ways: 1) using a heuristic feedback mechanism allowing one to base changes on triggers different from elapsing time, such as population diversity measures, relative improvement, absolute solution quality, etc. 2) incorporating parameters into the chromosomes, which lets the evolution mechanism entirely take care of changes (Eiben et al. 1999), 3) using the population structure and spatial position of individuals to determine their parameters (Ursem 2003).

Since all components of the algorithms have parameters, which need to be set to reasonable values, manual tuning is impossible to avoid completely even in adaptive control techniques. Furthermore, most advanced parameter control techniques also have parameters. However, adaptive parameter control seems to be still worthy due to the following facts: 1) the performance of parameter-varying algorithms is better, 2) since the control technique is usually more robust with respect to parameter sensitivity tuning; the control method is often easier than tuning the actual parameters (Ursem 2003).

Parameter control methods can be further categorized based on *how* the mechanism of change works and *what* component of the EA is affected by the mechanism. Typically, the following components of an EA are changed: 1) individual representation, 2) evaluation function, 3) variation operators and their probabilities, 4) selection operator like parent selection or mating selection, 5) replacement operator like survival selection or environmental selection and 6) characteristics of the population like size, topology, and so on (Michalewicz et al. 2004).

Still each component can be parameterized and the number of parameters is arbitrary. Two major classification criteria are (Eiben et al. 1999):

1. Evolutionary algorithms can be considered as a whole without dividing attention to its different components, e.g. mutation, recombination, selection etc., levels of adaptation and type of update rules. Three levels of adaptation: population, individual and component, together with two types of update mechanisms: absolute and empirical rules can be considered (Angeline 1995; Eiben et al. 1999). Absolute rules determine how modifications should be made beforehand. However, empirical update rules modify parameter values by self-adaptation, namely competition among them.
2. The above classification can be extended by considering the environment level of adaptation categories and ignoring what parts of the EA are adapted. This lets divide types of update mechanisms into deterministic, adaptive and self-adaptive (Hinterding, Michalewicz & Eiben 1997).

Deterministic, also called fixed, parameter control implies that the value of a strategy parameter is altered by some deterministic rule, such as a time-varying schedule, without using any feedback from the search.

Adaptive, also called explicitly adaptive, parameter control makes use of some form of feedback from the search in order to determine the direction and/or magnitude of the change to the strategy parameter.

Self-adaptive, also called implicitly adaptive, parameter control stands for situation in which parameters to be adapted are encoded into the chromosomes and undergo mutation and recombination. The better values of encoded parameters lead to better individuals, which, in turn, are more likely to survive and produce offspring and hence propagate these better parameter values. In other terminology, parameter control is divided to 1) uncoupled/absolute, which encompasses deterministic and adaptive control, and 2) tightly coupled/empirical, which coincides with self-adaptation (Angeline 1995; Eiben et al. 1999).

Population structure based parameter control uses the population structure and spatial position of individuals to determine their parameters (Ursem 2003).

4.3 Parameter Control Techniques

The following part gives a summary of the experimental works to control the parameters of evolutionary algorithms. These are divided according to what has been controlled: 1) individual representation, 2) evaluation function, 3) mutation operators and their probabilities, 4) crossover operators and their probabilities, 5) parent selection and 6) population.

4.3.1 Representation

Premature convergence and Hamming cliff problem, which occurred when GAs were first applied to numeric optimization problems, were reasons why most works have been done to adapt GA representation.

The ARGOT strategy uses one function variable per gene for a flexible mapping of the function variables to the genes (Shaefer 1987). This allows adaptation of resolution, i.e. the number of bits used to represent the gene, and the range and center point of the range of the values the genes are mapped into. The degree of convergence of the genes, the variance of the gene values, and the distance between the gene values and current range boundaries form the basis of adaptation of representation and mapping (Eiben et al. 1999).

Delta coding uses a GA with multiple restarts, where the first run is used to find an interim solution. Subsequent runs decode the genes as distances or delta values from the last solution. As a result, each restart forms a new hypercube with the interim solution as its origin. Restarts are triggered when the Hamming distance (the number of bits which differ between two bit strings of equal size) between the best and the worst strings of the continuing population are not greater than one. To expand or contract the search space the resolution of the delta values at the restarts can be altered (Whitley, Mathias & Fitzhorn 1991; Eiben et al. 1999).

The *dynamic parameter encoding* (DPE) is based on altering the mapping of the gene to its phenotypic value. “The phenotype of an individual organism is either its total physical appearance and constitution or a specific manifestation of a trait, such as size or eye color, that varies between individuals" (Wikipedia 2005). The DPE does not modify the actual number of bits used to represent a function parameter. Each generation the algorithm constructs a histogram of the population over the quarters of the current search interval formed by the two most significant bits of the gene. By summing over neighboring quarters, the population counts for three overlapping target intervals are obtained. If the largest count exceeds a given trigger threshold, the population is considered to have converged on the corresponding target interval. Then the binary gene of every individual in the population is manipulated such that those lying inside the target interval are left in place. Those lying outside are folded onto the target interval in the process. Subsequently, this technique can only increase the resolution of genes. Resolution denotes the number of alleles in a gene. (Schraudolph & Belew 1992.)

Messy GA (Goldberg, Deb & Korb 1990; Goldberg, Deb & Korb 1991) is targeted to fixed-length representations for solving deceptive binary string problems. A problem is deceptive if for two incompatible schemata A and B the average fitness of schema A is greater than B even though B includes a string that has a greater fitness than any member of A (Eshelman 2000). Low-order partitions in deceptive binary strings contain misleading information about higher-order partitions. Therefore, it will be hard for a GA to find the optimum of deceptive problem since building blocks cannot be formed.

Messy GA adapts its representation to a particular instance of the problem and allows over and under specification of representation of one. Chromosomes are of variable lengths and their genes contain their bit values and positions. Chromosomes may contain too few or too many bits for the representation. If more than one gene specifies a bit position (over specification), the first one encountered is used. If some bit positions are not specified by the chromosome (under specification), they are filled in from the so called competitive templates. Messy GAs do not use mutation and use cut and splice operators in place of crossover. A messy GA run happens in two phases: 1) a primordial phase which enriches the proportion of good building blocks and reduces the population size using only selection, 2) a juxtapositional phase which uses all the reproduction operators.

Self-adaptive control is also used for the dominance mechanism of diploid chromosomes, where there are two copies of each chromosome in each individual. The extra chromosome encodes an alternate solution and dominance decides which of the solutions will be expressed. An evolvable dominance value can be added to each gene and the gene with the highest dominance value is dominant (Bagley 1967; Eiben et al. 1999). The dominance effect can also be determined with particular enzymes being expressed (Rosenberg 1967; Eiben et al. 1999).

4.3.2 Evaluation Function

Penalty functions have also been put under adaptation. Various mechanisms have been used for varying penalties according to predefined deterministic schedule. One mechanism is to define the evaluation function *eval* with an additional parameter τ in the following way:

$$eval(x,\tau) = f(x) + \frac{1}{2\tau}\sum_J f_j^2(x)\,. \tag{16}$$

Here *eval* is decreased every time the evolutionary algorithm converges, typically by replacing τ with $\tau/10$. The best solutions found are taken as the initial population of the next iteration with the new decreased value of τ. Therefore, there are several cycles within a single run of the system. The evaluation function is fixed for each cycle: $J \subseteq \{1,...,m\}$ is a set of active constraints at the end of a cycle and the penalty pressure increases only when switching from one cycle to another (Michalewicz & Attia 1994; Eiben et al. 1999; Michalewicz et al. 2004).

The evaluation function can be changed based on the performance of an EA run so that the weights (penalties) of those constraints which are violated by the best individual after termination are raised and the new weights are used in the next run (Eiben & Ruttkay 1996; Eiben et al. 1999).

The penalties of constraints in a constrained optimization problem can also be adapted during a run so that the penalty measure depends on the number of violated constraints, the best feasible objective function found, and the best objective function value found (Smith & Tate 1993; Eiben et al. 1999).

The weights of constraints can be adaptively changed during the evolution by re-evaluating them when the algorithm gets stuck in a local optimum (Morris 1993; Eiben et al. 1999). The evaluation function can also be adaptively changed by periodically checking the best individual in the population and raising the weights (penalties) of those constraints this individual violates. In this so-called stepwise adaptation of weights (SAW) the run continues with the new evaluation function (Eiben & van der Hauw 1997 a, Eiben & van der Hauw 1997 b; Eiben et al. 1999).

A decoder-based approach for solving constrained numerical optimization problems is described in (Koziel & Michalewicz 1999; Eiben et al. 1999). The method defines a homomorphous mapping between n-dimensional cube and a feasible search space. The performance of the system was enhanced by introducing adaptive location of the reference point of the mapping, where the best individual in the current population served as the next reference point. Consequently, a change in a reference point resulted in changes in evaluation function for all individuals (Koziel & Michalewicz 1998; Eiben et al. 1999).

Co-evolution has been applied among others to constraint satisfaction and data mining problems as an evaluation function adaptive method. The adaptive mechanism is implemented as the interaction of the two subpopulations, where each subpopulation mutually influences the fitness of the members of the other subpopulation (Paredis 1994; Paredis 1995 a; Paredis 1995 b; Pagie & Hogeweg 1997; Eiben et al. 1999).

4.3.3 Mutation Operators

Mutation is a bit reversal event that occurs with small probabilities p_m per bit. Efforts to tune the mutation probability have resulted to different values and hence leaving practitioners in ambiguity. As results of tuning "optimal" mutation rate, the best rate found to be $p_m \approx 0.001$ (De Jong 1975), $p_m \approx 0.01$ (Grefenstette 1986),

$p_m \in [0.005, 0.01]$ (Schaffer et al. 1989) and $p_m = 1/L$ (Mühlenbein 1992), where L is the length of the bit string (Michalewicz et al. 2004).

Different techniques for updating the mutation rate over time have also been presented by different researchers. A time dependent schedule for controlling the mutation is presented in (Laumanns, Rudolph & Schwefel 1998; Laumanns, Rudolph & Schwefel 2001). Here the mutation step sizes are discounted by a constant factor each time an offspring is produced. Fogarty presented three ideas for controlling the mutation rate in bit-flip mutation (Fogarty 1989; Ursem 2003). The first idea was to control the mutation rate $p_m(t)$ by an exponential decreasing function of the generation number t according to the following formula:

$$p_m(t) = \frac{1}{240} + \frac{0.11375}{2^t}. \tag{17}$$

The second idea was to have different mutation rates for different bits in the chromosome. More precisely to have high mutation rates on the least significant bits and low mutation rates for the most significant bits according to the following formula:

$$p_m(i) = \frac{0.3528}{2^{i-1}}, \tag{18}$$

where i is the bit number and $i = 1, 2, \ldots 10$, with $i = 1$ being the least significant bit.

The third idea was to combine two previous formulas and control the mutation rates according to the following formula:

$$p_m(t,i) = \frac{28}{1905.2^{i-1}} + \frac{0.4026}{2^{t+i-1}}. \tag{19}$$

Fogarty compared these three schemes with a scheme using constant mutation rate of $p_m = 0.01$ and noticed that the combination of varying across both generations and bit number was the best of all four techniques (Ursem 2003).

A theoretically optimal schedule for deterministically changing p_m for the counting-ones function is presented in (Hesser & Männer 1991; Eiben et al. 1999; Michalewicz et al. 2004). Here, the value for p_m is defined in the following way over time t:

$$p_m(t) = \sqrt{\frac{\alpha}{\beta}} \times \frac{\exp(\frac{-\gamma t}{2})}{\lambda \sqrt{L}}, \tag{20}$$

where α, β and γ are constants, λ is the population size, L is the string length and t is the generation number.

The function to control the decrease of p_m is defined as a linear decreasing function from 0.5 to $1/L$ in generation T in (Bäck & Schütz 1996; Eiben et al. 1999 and Ursem 2003). The value of $p_m(t)$ is constrained so that $p_m(0) = 0.5$, $p_m(T) = \frac{1}{L}$ and otherwise:

$$p_m(t) = \left(2 + \frac{L-2}{T} t\right)^{-1}, \quad \text{if } 0 \le t \le T. \tag{21}$$

An optimal schedule for decreasing the mutation rate as a function of the distance to the optimum is defined in (Bäck 1992 a; Eiben et al. 1999; Michalewicz et al. 2004) in the following way:

$$p_m(f(x)) \approx \frac{1}{2(f(x)+1) - L}. \tag{22}$$

The idea of varying the mutation rate over both bit index number and generation was revisited and implemented in slightly different form from the traditional bit-flip mutation operation in (Janikow et al. 1991; Ursem 2003). The new mutation operator was called *non-uniform mutation* for binary encoding and was derived from the version used for real encoding. The operator mutates the k'th $(k = 1,2,3,...q)$ binary encoded parameter x_k of a candidate solution $\mathrm{x} = (x_1,...,x_k,...,x_q)$ according to:

$$x_k^{t+1} = \begin{cases} x_k^t + \Delta(t, righ(k) - x_k), & \text{if a random binary digit is 0} \\ x_k^t - \Delta(t, x_k - left(k)), & \text{if a random binary digit is 1} \end{cases}. \tag{23}$$

The functions $left(k)$ and $right(k)$ determine the valid range $\left[left(k), right(k)\right]$ for each point x_k in the search space, where other variables x_i ($i = 1,...,k-1,k+1,...,q$) remain fixed. The function $\Delta(t, y)$ returns a value in the range $\left[0, y\right]$ so that the probability of $\Delta(t, y)$ being close to 0 increases as the generation number, t increases. This property causes this operator to search the space uniformly initially, when t is small, and very locally at later stages. Function $\Delta(t, y)$ is defined as:

$$\Delta(t, y) = yr(1 - \frac{t}{T})^b, \tag{24}$$

where r is a random number from [0, 1], T is the maximal generation number, and b is a system parameter determining the degree of non-uniformity.

Controlling the variance in Gaussian mutation is very critical in successful application of real-encoded EAs. The standard approach for doing this is to set the variance of the mutation according to a monotonic decreasing function depending on the generation number. These heuristic functions are usually developed by scrutinizing the experimental data and forming a hypothesis from the relationship between the performance of the algorithm and how the parameters were changed (Ursem 2003).

A classical adaptive method for changing the mutation step size is Rechenberg's 1/5 rule for Gaussian mutation in (1+1)-EAs presented in (Rechenberg 1973; Eiben et al. 1999). This rule states that the ratio of successful mutations to all mutations should be 1/5 measured over a number of generations. The standard deviation σ should increase if the ratio is above 1/5, decrease if it is below, and remain unchanged if it is 1/5 (Ursem 2003). It is assumed that maximum progress can be achieved through mutation step sizes leading to a success probability of approximately 20%.

The ratio between crossovers and mutations is regulated based on their performance in (Julstrom 1995; Eiben et al. 1999). Both operators are used separately to create an offspring, and the algorithm keeps a tree of their recent contributions to the new offspring and rewards them accordingly.

Mutation rates in a parallel GA are adapted with a farming model in (Lis 1996; Eiben et al. 1999). Probability of mutation, crossover and the population size in an algorithm of parallel farming model has been adapted in (Lis & Lis 1996; Eiben et al. 1999). They use parallel populations and each of them has one value, out of a possible three different values for p_m, p_c and the population size. The populations are compared after a certain period of time, and then the values for p_m, p_c and the population size are shifted one level towards the values of the most successful population (Eiben et al. 1999).

Gaussian mutation of a real-encoded variable x_i is usually performed according to (Ursem 2003):

$$x_i' = x_i + N(0, \sigma_i(t)). \tag{25}$$

The traditional approach to set the mutation variance is using either a linear or an exponentially decreasing function such as:

$$\sigma_i(t) = 1 \big/ \sqrt{1+t}\ . \tag{26}$$

Self-adaptive control of mutation step sizes has been reported in (Schwefel 1995; Bäck 1996; Bäck et al. 1997; Eiben et al. 1999). Mutating a floating-point object variable x_i happens in the following way:

$$x_i' = x_i + \sigma_i N(0,1), \tag{27}$$

where the mean step sizes are modified lognormally:

$$\sigma_i' = \sigma_i \exp(\tau' N(0,1) + \tau N_i(0,1)), \tag{28}$$

where τ and τ' are the so-called learning rates. The value of σ can also be modified normally:

$$\sigma_i' = \sigma_i + \zeta \sigma_i N(0,1), \tag{29}$$

where ζ is a scaling constant (Fogel 1995; Eiben et al. 1999). Empirical evidence suggests that lognormal perturbation of mutation rates is preferable to Gaussian perturbations on fixed-length real-valued representations (Saravanan & Fogel 1994; Saravanan, Fogel & Nelson 1995; Eiben et al. 1999). However, a slight advantage of

Gaussian perturbations over lognormal updates for self-adaptively evolving finite state machines is reported in (Angeline, Fogel & Fogel 1996; Eiben et al. 1999).

Significant improvement can be achieved by controlling the mutation variance by other techniques than a strictly decreasing function (Ursem 2003). The so-called sand pile model was used to generate power-law distributed numbers for controlling the variance in Gaussian mutation (Krink, Thomsen & Rickers 2000; Ursem 2003). The sandpile model is a simple approach to study many complex phenomena found in nature and is an example of how self-organized criticality (SOC) can be generated by very simple means (Bak 1996; Ursem 2003).

Self-adaptation of the mutation step size for optimizing numeric functions in a real valued GA is applied in (Hinterding 1995; Eiben et al. 1999). In another experimentation individuals are replaced by their offspring. Probabilities of crossover and mutation for each chromosome are added to its bit string and adapted in proportion to the population maximum and mean fitness (Srinivas & Patnaik 1994; Eiben et al. 1999). In (Kursawe 1991; Laumanns et al. 2001) the selection criterion changes randomly over time. To make individuals cope with the changing environment they are supplied with a set of step sizes for each objective function through polyploidy. Polyploidy is a situation when the number of chromosomes in a cell becomes doubled. This can happen by a mutation that simply makes two copies. It can also happen when the chromosomes from two different species are mixed.

The mutation rate of GAs can also be self-adapted by adding the rate of mutating p_m, coded in bits, to every individual. Then the new p_m is used to mutate the individual's object variables. This is based on the idea that better p_m rates will produce better offspring and then hitchhike on their improved children to new generations, while bad rates will die out (Bäck 1992 a, Bäck 1992 b; Eiben et al. 1999). The same idea with an implementation of 1/5 success rule for mutation has been applied on a steady-state GA in (Smith & Fogarty 1996; Eiben et al. 1999).

Laumanns et al. (2001) discuss the problem of controlling mutation strength in multi-objective evolutionary algorithms and its implications for the convergence to the Pareto set. A Pareto set is defined to be the set of all Pareto optimal decision vectors and a Pareto optimal vector is defined to be a vector, which makes the optimization function converge the most. Convergence here refers to the iterative approach of populations to the Pareto set of the underlying optimization problem.

An algorithm should ensure convergence to Pareto set and provide a "good" distribution of solutions in order to find or to approximate the set of efficient or Pareto-optimal solutions. Fitness assignment methods based on the notion of dominance seem to produce better solution distributions than plain aggregating methods. Density based selection methods maintain diversity and the use of elitism speeds up the search in the direction of the Pareto set and ensure convergence properties. (Laumanns, Zitzler & Thiele 2001; Rudolph & Agapie 2000; Laumanns et al. 2001). However, virtually all implementations focusing on the role the variation operators in evolutionary multi-objective optimization use standard non-adaptive operators from the single objective case (Laumanns et al. 2001).

Approximating the Pareto set of a multi-objective optimization problem by evolutionary algorithms faces two main problems:

1. The velocity and reliability of convergence to the Pareto set, same as in single objective optimization.
2. Distribution of solutions, caused by the existence of multiple Preto-optimal solutions in the multi-objective case.

Laumanns et al. (2001) present the so-called Predator-Prey model. In this model predator individuals move across the spatial structure so that they delete the worst prey individual of their neighborhood according to their associated objective function. The authors present two adaptation rules capable of increasing the step sizes on need for the model: 1) the standard mutative self-adaptation and 2) self-adaptation through recombination frequency. These rules have worked well. The model converges to the Pareto set due to the superior success probability of single criteria selection in the vicinity of the Pareto set, where it gets increasingly difficult to make cooperative steps from one mutation alone.

The second method, self-adaptation through recombination frequency, combines the implicit evaluation of good "inner models" through fitness evaluation and selection with a fixed but flexible schedule depending on the recombination frequency. In this way all individuals use bigger step sizes for the early offspring and smaller for later ones. The mutation of the step sizes is done deterministically according to the rule:

$$\sigma^{(t+1)} = \gamma^{d-d_0}\sigma^{(t)}, \quad \gamma \in \left]0,1\right[. \tag{30}$$

Here d denotes the number of descendants an individual has produced so far. The delay parameter, d_0 determines the number of descendants that must be created before smaller step sizes are passed on to the offspring. The adaptation rate can be controlled by γ. In this schedule for step sizes just one order of magnitude greater than the optimal ones the success rate rapidly goes to zero. Therefore it is a matter of time until appropriate step sizes will be found.

4.3.4 Crossover Operators

In contrast to the mutation operator that acts on bits, the crossover rate p_c acts on a pair of chromosomes and states the probability that the selected pair undergoes crossover. Crossover rate is a number between zero and one that indicates how frequently crossover is applied to a given population (Haupt et al. 2004). Some pre-set crossover rates tuned for a binary GA are reported in Eiben et al. 1999 to be $p_c \approx 0.6$ (De Jong 1975), $p_c = 0.95$ (Grefenstette 1986) and $p_C \in [0.75, 0.95]$ (Schaffer et al. 1989; Bäck 1996; Michalewicz et al. 2004). It is generally believed that the crossover rate should not be too low and not below 0.6 (Eiben et al. 1999).

The rate of operators can be adapted by rewarding those individuals, which have been successful in creating better offspring. This idea is implemented in Davis's adaptive operator fitness. This reward is diminishingly propagated back to operators of a few generations back, which helped setting it all up; the reward is a shift up in probability at the cost of other operators (Davis 1991; Eiben et al. 1999).

The same idea seems to have been used in classifier systems as credit assignment principle (Goldberg 1989; Eiben et al. 1999). Cost based operator rate adaptation (COBRA) on adaptive crossover rates has been studied in (Tuson & Ross 1996; Eiben et al. 1999). As it was mentioned earlier values of p_m, p_c and the population size in an algorithm of a parallel farming model have been adapted in (Lis et al. 1996).

Probability adaptation for allele exchange of uniform crossover is investigated in (White & Oppacher 1994; Eiben et al. 1999). A discrete $p_c \in \left[0, \frac{1}{N}, \frac{2}{N}, ..., 1\right]$ is assigned to each bit in each chromosome and bits are exchanged at position i by crossover if $\sqrt{p(parent_1)_c^i . p(parent_2)_c^i} \geq rnd(0,1)$. The offspring created inherits bit x_i and p_c^i from its parents. The finite state automata used here amounts to updating these probabilities p_c^i in the offspring in the following way:

```
if $f(child) > f(parent)$ then
    raise $p_c^i$ in child for $i's$ from parent
if $f(child) < f(parent)$ then
    lower $p_c^i$ in child for $i's$ from parent
else
    modify randomly $parent \in \{parent_1, parent_2\}$
```

The choice between the two different crossovers, 2-point crossover and uniform crossover has been self-adapted in (Spears 1995; Eiben et al. 1999) by adding one extra bit to each individual. This extra bit determines which type of crossover, e.g. uniform crossover, 2-point crossover, is used for that individual and this choice of crossover is inherited from the parents to their offspring. If the parents' crossover bits are different, then one operator is chosen randomly. The crossover operator can also be controlled by using the so-called gene linkage bits. Here, two binary linkage genes are added to each gene and the values of these bits determine whether the crossover is allowed or not; if both link bits are 1, crossover is not allowed (Krink 2005).

Individuals are replaced by their offspring in (Srinivas & Patnaik 1994; Eiben et al. 1999). Each chromosome has its own probabilities p_m and p_c added to their bitstrings. These probabilities are adapted in proportion to the population maximum and mean fitness.

The number and locations of crossover points are adapted in (Schaffer et al. 1987; Eiben et al. 1999) by introducing special marks into offspring representation, which keep track of the sites in the string where crossover occurred.

In multiparent operators the number of parents taking part in recombination is also a parameter. To adjust the arity of recombination an adaptive mechanism based on competing subpopulations (Schlierkamp-Voosen & Mühlenbein 1996) is used in (Eiben Sprinkhuizen-Kuyper & Thijssen 1998). The idea is that the population is divided into disjoint subpopulations each using a different crossover, i.e. arity. Subpopulations develop independently for a certain period of time and exchange information by allowing migration after each period. The process is controlled so that more successful populations grow in size and less successful populations become smaller. However, subpopulations and their crossover operators are saved from complete extinction. By using a high quality six-parent crossover variant, this method performed comparably with a traditional GA using one population, one crossover. But, it failed to clearly identify the better operators by making the corresponding subpopulation larger (Eiben et al. 1999).

4.3.5 Parent Selection

The selection pressure can be varied along the course of the evaluation according to a pre-defined cooling scheduling using Boltzmann selection mechanism, where a minimal energy level is sought by state transitions (Mahfoud 1997; Eiben et al. 1999). The chance of moving from state i to state j is determined by the Metropolis criterion:

$$P\left[accept\ j\right] = \exp\left(\frac{E_i - E_j}{K_b.T}\right), \tag{31}$$

where E_i, E_j are the energy levels, K_b is a parameter called the Boltzmann constant, and T is the temperature. The Metropolis criterion is applied for defining the evaluation of a chromosome in (De la Maza & Tidor 1993; Eiben et al. 2000; Michalewicz et al. 2004). The selection pressure is changed by changing the acceptation threshold over time in a multi-population GA framework presented in (Rudolph & Sprave 1995; Eiben et al. 1999).

The parent selection component of an EA has not been commonly used in an adaptive manner, but there are selection methods whose parameters can be easily adapted (Eiben et al. 1999). For instance, in linear ranking a selection probability proportional to the rank of individual i is assigned to each individual in the following way:

$$p(i) = \frac{2 - b + 2i(b-1)/(pop\_size - 1)}{pop\_size}. \tag{32}$$

Here the parameter b represents the expected number of offspring to be allocated to the best individual. The selective pressure of the algorithm can be adapted by changing the value of b within the range of $[1,\ 2]$. Here the rank of the worst individual is 0 and the rank of the best individual is $pop\_size - 1$ (Eiben et al. 2000; Michalewicz et al. 2004).

4.3.6 Replacement Operator

Like parent selection mechanisms, replacement operator (survivor selection) is not commonly used in an adaptive manner. In simulated annealing (SA), solutions are searched and tested and it can be envisioned as an evolutionary process with population size of 1, problem dependent representation and mutation mechanism, and a specific survivor selection mechanism. During the run the selective pressure increases in the Boltzmann-style. The main cycle of SA for minimization problems is:

```
procedure SA
begin
    generate $j \in S_i$
    if $f(j) < f(i)$ then $i := j$
    else
        if $\exp(\frac{f(i) - f(j)}{c_k}) > random[0,1]$ then $i := j$
end
```

Figure 2. The simulated annealing algorithm for minimization problems.

Here the probability of accepting inferior solutions decreases as temperature c_k is decreasing.

4.3.7 Population

A theoretical analysis of the optimal population size have been provided in (Goldberg, Deb & Clark 1992 a; Goldberg, Deb & Clark 1992 b; Smith 1997); Eiben et al. 1999). In addition, there have been efforts to find the optimal population size empirically. The optimal population size has recommended being [50, 100] (De Jong 1975; Eiben et al. 1999), [30, 80] (Grefenstette 1986; Eiben et al. 1999) and [20, 30] (Schaffer et al. 1989; Eiben et al. 1999).

An algorithm, which adjusts the population size with respect to the probability of selection error, is presented in (Smith 1993; Eiben et al. 1999). An adaptive GA, which works on varying population size, is presented in (Hinterding, Michalewicz & Peachey 1996; Eiben et al. 1999). This algorithm consists of three subpopulations and the sizes of these populations are adjusted at regular intervals based on the current state of the search. Another competition scheme is presented in (Schlierkamp-Voosen & Mühlenbein 1994; Eiben et al. 1999). This scheme changes the sizes of subpopulations, while keeping the total number of individuals fixed. In the genetic algorithm with varying population, size (GAVaPS) the age of the chromosome replaces the concept of selection. The age of a chromosome is the number of generations the chromosome stays alive. Since the age of chromosome depends on the fitness of the individual, it influences the size of population at every stage of the process (Arabas, Michalewicz & Mulawka 1994; Eiben et al. 1999).

A recent idea in parameter control is to use the population structure to set the parameters of the algorithm (Ursem 2003). There are two main approaches:

1. To divide the entire population into a number of disjoint subpopulations and use the success of each subpopulation to control the parameters of other subpopulations.
2. To use a spatial population structure and map the position of each individual to a corresponding set of parameter values.

A subpopulation-based approach to control the probability of mutation p_m in a binary encoded GA is suggested in (Lis 1996; Ursem 2003). In this approach designed for parallel computers, each subpopulation has its own level of mutation probability. The subpopulations are assigned to 12 consecutive levels form a predefined scheme of mutation probabilities: {0.0001, 0.0002, 0.0005, 0.001, 0.002, 0.005, 0.01, 0.02, 0.05, 0.1, 0.2, 0.5}. A main processor coordinates and distributes subpopulations to other processors. The main processor first creates a random initial population, which is then copied and distributed to each processor along with the number of generations and sends the best individual to the main processor. The algorithm constantly adapts the window of mutation probability towards the most successful probability. (Ursem 2003)

In spatial control, a large part of the parameter space is represented by a spatial population structure and location of the individual is interpreted as a set of parameters. The Terrain-Based Genetic Algorithm (TBGA) uses a spatial population structure, where each cell holds an individual with a binary encoded genome (Gordon, Pirie, Wachter & Sharp 1999; Ursem 2003). The two-dimensional grid position of an individual is interpreted as its offspring's mutation rate and number of crossover points. Hence, the individuals of a TBGA population apply the entire variety of different parameter combinations at each step. The advantage of this approach is that optimal parameters for the type of optimization task and the current state of the optimization process can be exploited. However, since the individuals in the cellular EA are fixed at their grid position, only a few of them are able to take advantage of this set-up simultaneously.

4.3.8 Combining Forms of Control

Most studies on control of parameters in EAs consider control of one parameter only or a few parameters related to a single component of EA. The reason for this might be 1) because the exploration of capabilities of adaptation was done experimentally, or 2) it is easier to report positive results in such similar cases. The combination forms of control are much more difficult. This is because the interactions of even static parameter settings for different components of EAs depend on the objective functions and representations used, and hard to understand (Eiben et al. 1999).

A fuzzy system was used to control the change in population size, crossover rate and mutation rate in (Lee & Takagi 1993). The system used the two ratios $(\mathit{average\ fitness})/(\mathit{best\ fitness})$ and $(\mathit{average\ fitness})/(\mathit{worst\ fitness})$ together with the fitness velocity to control changes. The system was first tested with simple artificial benchmark problems and then used to set the parameters of an EA evolving a controller for a pole-balancing problem. The fuzzy parameter control system showed better online performance but similar offline performance compared to a simple EA. Experiments indicated that the learned fuzzy rules may be generally applicable, but they require a significant investment in programming since a fuzzy interface engine needs to be implemented before being used in practice (Ursem 2003).

A combination of one generational measure and two instantaneous measures (called state vector) were used to set three parameters (called control vector) of the algorithm in (Kee, Airey & Cyre 2001). The state vector consists of the generational fitness velocity and the instantaneous fitness variance and population diversity measured by Hamming distance. The control vector sets the probability of mutation p_m, the probability of crossover p_c and the so-called power fitness scaling factor α. A simple rule system is used to map the current state vector to a control vector. The state values are divided into low, medium and high ranges, which give a system with $3^3 = 27$ rules. The control values are set to a low, a medium or a high value. Hence, each rule maps a state three-tuple to a corresponding control three-tuple, like (low, high, high) to (medium, low, low). The rule learning is carried out while the EA is optimizing the problem (Ursem 2003).

The most common form of combined parameter control is related to mutation. A number of parameters can control the operation of Gaussian mutation. The mutation can be controlled by 1) controlling its direction or 2) by setting the standard deviation of the mutations (mutation step size) at a global level for each individual or for genes within an individual (Michalewicz et al. 2004).

Controlling parameters taken from different components of the EA are much rare (Eiben et al. 1999). Self-adaptation of the mutation step size was combined with the feedback-based adaptation of the population size in (Hinterding et al. 1996; Eiben et al. 1999). Here feedback from a cluster of three EAs, which use self-adaptive Gaussian mutation and different population sizes were used to adjust the population size of one or more of the EAs at 1000 evaluation epochs. The EA adapted different strategies for different type of test functions: for unimodal functions, it adapted to small population sizes for all the EAs, and for multimodal functions, it adapted one of the EAs to a large but oscillating population size to help it escape from local optima.

The mutation step size and preferred crossover points in an EA have been self-adapted in (Smith & Fogarty 1996 a; Smith & Fogarty 1996 b; Eiben et al. 1999). Here each gene in the chromosome includes: a) the problem encoding component, b) a mutation rate for gene, c) two linkage flags; one at each end of the gene, which are used to link genes into larger blocks, so that two adjacent genes have their adjacent linkage flags set. Crossover is a multiparent crossover and occurs at block boundaries, whereas the mutation can affect all the components of a block and the rate is the average of the mutation rates in a block.

The adaptation of mutation probability, crossover rate and population size have been combined in (Lis et al. 1996; Eiben et al. 1999). Here a parallel GA was used over a number of epochs so that in each epoch the parameter settings for the individual GAs was determined by using the Latin Square (a matrix consisting of n sets of the numbers 1 to n arranged in such a way that no orthogonal row or column contains the same number twice) experiment design. This was done so that the best combination of three values for each of the three parameters could be determined using the fewest number of experiments. At the end of each of epoch, the middle level parameters for the next epoch were set to be the best values from the last epoch.

Minimizing the number of separate populations is the reason why feedback-based rather than self-adaptation was used to control the population size in (Hinterding et al. 1996; Eiben et al. 1999). The complexity of interactions between adaptive parameters seems to be the reason, why combined parameter control is commonly self-adaptive. Using self-adaptation in combined parameter control is the most promising way since we leave the process to evolution itself to determine the beneficial interactions among parameters while finding a near-optimal solution to the problem (Eiben et al. 1999).

Combined parameter control may trigger additional problems related to transitory behavior of EAs and reduce the chances of utilizing good operators at later stages of the process. For instance, a population may be arranged in a number of disjoint subpopulations, which use different crossover rates. If the size of subpopulation depends on the merit of its crossover, the operator which performs poorly at some stage of the process would have difficulties to recover as its subpopulation shrinked in the meantime and smaller populations usually perform worse than larger ones.

5. The Distributional Decimal Binary Genetic Algorithm

In the following, the new genetic algorithm developed in this research is described in detail. The algorithm is called distributional decimal binary genetic algorithm (DDBGA). The algorithm uses the information from the population distribution for breeding. It uses the binary encoding of candidate solutions, but utilizes operators originated form real value encoding in novel ways. All components of the algorithm are designed and developed here for experimentation with function optimization. Some of these components are well known, but they are implemented in a new way and combined with some other new operators. For instance, the classic one point crossover is a well-known operator also used in this algorithm, but selection for breeding prior to the crossover is a new method developed here. Components of the algorithm described here are: the problem encoding (representation), initialization, breeding, operators and replacement. For each component, the pseudo code is also provided.

5.1 Representation

The binary representation was selected for encoding individuals for both binary and real value problems. The binary representation of variables is the target of all operators. However, in order to make the binary operators more efficient, conversion between the real value and the binary representation is frequently used. Since the random multiple point mutation of a long binary string, particularly in a non-uniform manner requires lots of trials, making use of decimal mutation techniques can speed up the process significantly. The effect can be particularly observed during the fine-tuning of candidate solutions.

The mapping between binary strings into floating-point numbers and vice versa is implemented according to the following well-known steps:

1. The distance between the upper and the lower bounds of variables is divided according to the required precisions, $precision$ (e.g. the precision for 6 digits after the decimal point is $1000000_{(10)}$) in the following way:

$$(upper - lower) \times precision\,. \tag{33}$$

2. Then an integer number l is found so that:

$$(upper - lower) \times precision \leq 2^l\,. \tag{34}$$

   Thus, l determines the length of binary representation, which implies that each chromosome in the population is l bits long.

Therefore, if we have a binary string x' of length l, in order to convert it to a real value x, we first convert the binary string to its corresponding integer value in base 10, $x'_{(10)}$ and then calculate the corresponding floating-point value x according to the following formula:

$$x = lower + x'_{(10)} \times \frac{upper - lower}{2^l - 1}\,. \tag{35}$$

This formula can also be used in order to represent binary chromosomes in integer form. Since the corresponding decimal value of the binary string in base 10 is always a positive number smaller than or equal to 2^l, instead of binary strings the population can consist of randomly generated integer numbers between 0 and 2^l. This can be an interesting technique to avoid the main drawback of binary representation for resulting in too long binary strings and search spaces. This means that, e.g. for 100 variables with domains in the range $[-500,\ 500]$ and precision of six digits, instead of maintaining 3000 bits long binary strings, we can have an array of integer numbers of just 100 locations.

Whenever a binary operation should be carried out, the integer number can easily be converted to its binary representation and after the operation is complete, the new binary string can be converted to its corresponding integer number in base 10.

5.2 Population Size

The population size is the only fixed parameter. It is set to the smallest feasible value, i.e. 12. Size 12 seems to be just big enough for maintaining some variation in the population; there will be a group of three individuals in the lower quartile of the population (0.25*12), six individuals in the interquartile range of the population (0.5*12) and three individuals in the higher quartile of the population (12-0.75*12). Populations of smaller size would not have enough individuals to form a group of reasonable size in the higher quartile. On the other hand, populations of bigger size would cause unnecessary function evaluations and waste resources.

The search space is explored more only when it was necessary. This can be described as: “just-in-time search space exploration” or “search space exploration on demand”; the search is started with a population of relatively small size and is refreshed when it is necessary. If the fitness values of the best individual and the worst individual are the same, then the population is considered to be overtaken by a single individual. In that case, the algorithm explores the search space more by keeping one individual and replacing the rest of the individuals with new randomly generated ones. This approach gives a dynamic characteristic to the search space exploration.

5.3 Initialization of the Population

Binary initialization of individuals is performed to assure the real random initialization of the population. This is simply done by randomly initializing each locus of gene in each chromosome with zero and one. This approach proved to be a more efficient way for uniformly initializing the population randomly. Real value initializations could lead to certain distributions, where the whole search space would not be necessarily covered.

During initialization, for each individual a mate of the opposite sex is created. This is done simply by inverting each gene in the individual to the opposite value, like:

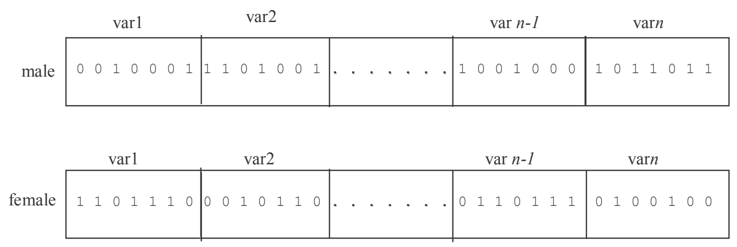

Figure 3. The male and female chromosome mapping example.

The motivation for this operation is the observation that flipping all bits in a string could lead to rapid fitness improvement. Moreover, crossover with two bit strings of opposite values increases the chance of producing better offspring. For each individual, both male and female chromosomes are evaluated. Of these chromosomes, the fitter one is set to be the male chromosome and the other one is se to be the female chromosome. This process is repeated until all members of the population have been created. Therefore, each individual is actually presented by two chromosomes: a *male* chromosome and a *female* chromosome. However, operations are aimed at the male chromosomes by default. Figure 4 provides the pseudo code for the initialization process of the population:

```
procedure PI
begin
    for each individual in the population do
      for each gene in the individual do
        select the binary gene value $gv \in \{0,1\}$ for the male chromosome
        calculate the gene value for the female chromosome $fgv = 1 \text{XOR}\, gv$
      done
      evaluate male and female chromosomes
      set the better chromosome as male chromosome
      set the worse chromosome as female chromosome
    done
end
```

Figure 4. The pseudo code for the population initialization (PI) algorithm.

After each evaluation of the population, individuals are sorted in ascending order according to their fitness values. This helps dividing the population to three separate parts: 25th percentile (lower quartile), 75th percentile (higher quartile) and interquartile range (above the lower quartile and below the higher quartile). This division is necessary in order to recognize the most critical areas in the distribution of the population and focus genetic operators on most promising individuals. This will also help implementing genetic operators more precisely and avoiding precious processing time. Figure 5 shows how the population is divided to different areas according to the individuals' fitness distribution.

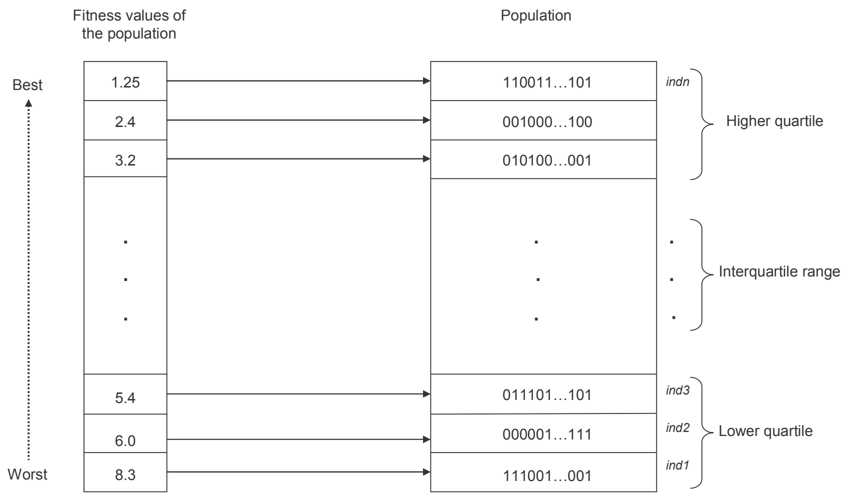

Figure 5. Division of the population distribution according to the fitness values of individuals for a minimization problem.

As a result, genetic operators will be more efficient and improve the population more rapidly. This division will also help maintaining diversity in the population while for instance individuals in the higher quartile will go through continuous evolution process and improve more. Furthermore, the division of the population helps focusing evolution operators intentionally on certain individuals instead of hoping that a random process would take care of the process and select fitter individuals for different operators.

5.4 Survivor Selection

After the initialization, the population goes through various evolutionary operators to get developed. After each operator application, new offspring are evaluated and compared to the population individuals starting from the best individual. Once a worse individual is found, it is checked whether the worse individual is the worst one in the population or not. If the individual is not the worst one, it first replaces the worst individual before itself being replaced by the new offspring. This means that offspring do not automatically replace their parents. On the other hand, offspring do not need to be better than their own parents to be moved to the population. Practically this is equal to replacing the worst individual with a fitter one. In this manner no chance for improving the population average fitness value is lost. Also no effort is lost and the advantageous results of the crossover operations are used efficiently. Figure 6 provides the pseudo code for the survivor selection procedure.

```
procedure SSP(offspring)
begin
    evaluate offspring
    if $\exists ind \in population$ so that offspring is fitter than ind then
        if ind is fitter than $population\_worst\_individual$ then
                replace population_worst_individual with ind
        endif
        replace ind with offspring
    endif
end
```

Figure 6. The pseudo code for the survivor selection procedure (SSP).

This survivor selection mechanism is implemented after each operator later in the text this mechanism will be referred to as the survivor selection procedure (SSP).

5.5 Genetic Operators

Several genetic operators are designed and implemented. Some of these operators are new and some others are previously known classical operators, which are implemented here in a new way. The common factor in the design and implementation of the operators is the fact that they are intentionally restricted to a certain area of the fitness value distribution of the population and therefore they manipulate individuals according to their position in the distribution.

The mutation operator is not used explicitly at all. It may have occurred according to a random process in the random building block operator explained later, but it has not been planned to happen in the evolution process.

Several variations of the crossover operator have been used. These have been designed to deterministically manipulate individuals in each quartile of the fitness population. For each quartile of the population, the crossover operator is repeated at least for the expected number of individuals in that quartile. This means that during the crossover operation for e.g. the higher quartile, the operation is repeated at least 3 times (0.25*12=3), and so on. After repeating the crossover operator for the number of individuals in each population area, the application of the operator is repeated until it fails to produce any better offspring.

5.5.1.1 The Higher Quartile Crossover Operator

The *higher quartile crossover* (HQC) operator implements the idea of the well-known one-point crossover. However, parents are selected for breeding in a new way. Individuals in the higher quartile of the population are randomly selected to go through this operation. For this operator, two different parents, p_1 and p_2 and a crossover point cp are randomly selected. Then, the male chromosome of parent p_1 is crossed over with the male and female chromosomes of parent p_2. As a result, four new offspring are created. These new offspring then go through the survivor selection procedure explained earlier for possible replacement. Figure 7 describes how the crossover between two individuals is implemented.

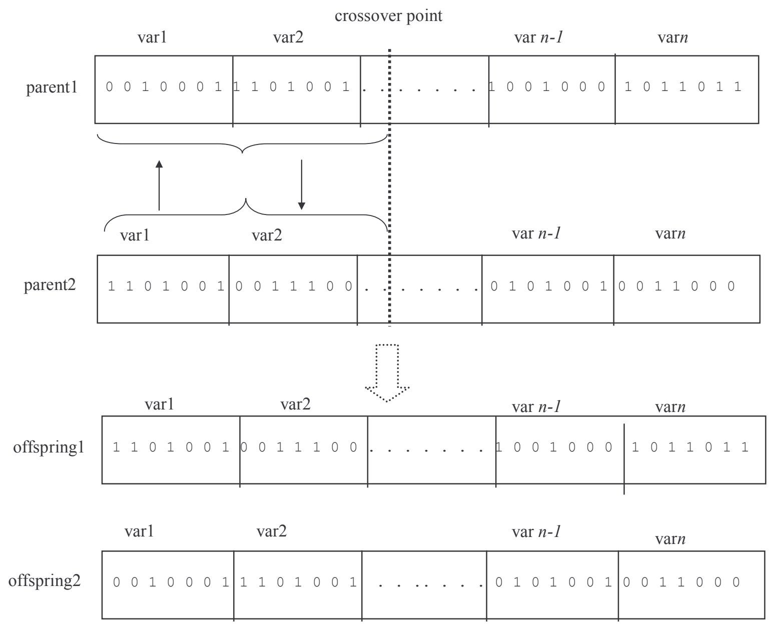

Figure 7. The crossover operator. Two individuals are randomly crossed over at a single point to produce new offspring.

The new idea here is that the crossover operator is repeated with the same parent indexes and on the same crossover point as long as a better offspring is created. It is important to notice that the same parent indexes do not necessarily mean the same parents since a better offspring might have replaced the parent during the previous survivor selection. Crossover points are selected so that they are at least two loci far from the end points of the binary representations of the chromosomes. This is to make sure that the operator is really crossover and not mutation. Figure 8 describes the pseudo code for the higher quartile crossover operator.

```
procedure HQC
begin
    for the number of individuals in the population higher quartile do
        select randomly parent1 from the population higher quartile
        select randomly parent2 from the population higher quartile
        select randomly crossover point $cp \in [2, \text{length}(parent1) - 2]$
        do
            crossover parent1 with the male chromosome of parent2 on cp
            crossover parent1 with the female chromosome of parent2 on cp
            failed=true
            for each resulting offspring do
              call the survivor selection procedure (SSP) for offspring
              if offspring is better than a population member then
                failed=false
              endif
            done
        while (failed==false)
    done
end
```

Figure 8. The pseudo code for higher quartile crossover operator (HQC).

5.5.2 The Random Building Block Operator

The *random building block* (RBB) operator is also a new operator proposed here. During the classical crossover operation, building blocks of two or more individuals of the population are exchanged in the hope that a better building block from one individual will replace a worse building block in the other individual and improve the individual's fitness value. However, the random building block operator involves only one individual. The idea with the random building block operator is that it randomly produces new building blocks of random length and replaces a randomly selected building block in an individual.

This operator can help breaking the possible deadlock when the classic crossover operator fails to improve individuals. It can also refresh the population by injecting better building blocks into individuals, which are not currently found from the population. Figure 9 describes the random building block operator.

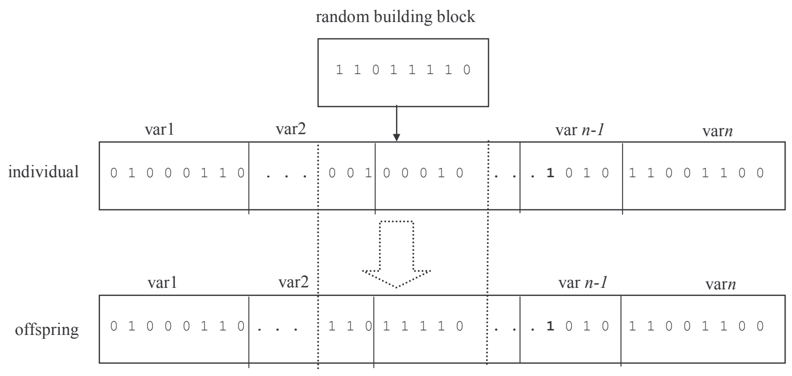

Figure 9. The random building block operator. A random building block is generated and copied to an individual to produce a new offspring.

All population members go through this operation. This operation is implemented in the following order: 1) for each individual *ind* two crossover points *cp1* and *cp2* are randomly selected, 2) a random bit string *bstr* of length $l = |cp2 - cp1|$ is generated, and 3) bits between the crossover points on the individual *ind* are replaced by the bit string *bstr* . Figure 10 provides the pseudo code for the random building block operator.

```
**procedure RBB**
**begin**
    **for** each *individual* in the population **do**
        make a *clone* of *individual*
        select crossover point $cp1 \in [0, \mathrm{length}(individual))$
        select crossover point $cp2 \in [0, \mathrm{length}(individual))$ so that $cp2 \neq cp1$
        **if** $cp2 < cp1$ **then**
            exchange values of $cp1$ and $cp2$
        **endif**
        generate random bit string *rbs* of length $l = |cp2 - cp1|$
        replace bits between $cp1$ and $cp2$ on *clone* with *rbs*
        call the survivor selection procedure (SSP) for *clone*
    **done**
**end**
```

Figure 10. The pseudo code for the random building block (RBB) operator. Bit indexes start from 0.

5.5.3 The Variable Crossover Operator

The *variable crossover* (VC) operator is a modification of the standard uniform crossover operator designed and implemented here. This operator is meaningful only for multiple variable problems. The main idea is the same as with the standard crossover. However, the crossover points are always the borders between variables in the individuals. This means that during this operation the whole binary representations of different variables in one individual are replaced by the binary representations of some variables from some other individuals.

The motivation for this operator is simply the fact that even in an individual with low fitness value, the value of one or more variables might be very close to their optimal values, while the values of other variables are far from their optimal values. Since the value of each variable plays an important role in determining the individual's fitness value, removing "unhealthy" parts from the individual and replacing them with "healthier" parts would definitely increase the individual's fitness value. Therefore, the variable crossover operator can be compared to a surgical operation in which healthier organs from better individuals are transferred to “sick” individuals in the hope of rehabilitating them.

The variable crossover operator is implemented separately on the higher quartile and the elitist. Both implementations are described below. Figure 11 describes the main idea with the variable crossover operator.

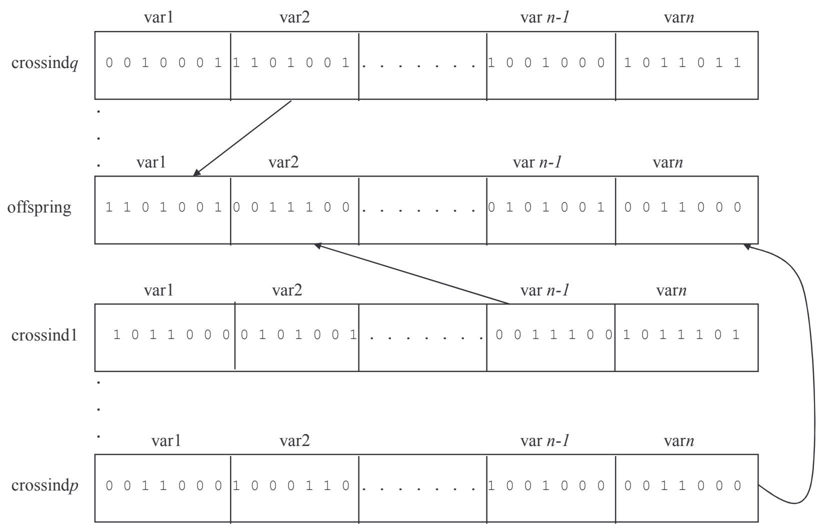

Figure 11. The variable crossover (VC) operator. The binary representation of one or more (but not all) variables in an individual is replaced by the binary representation of other variables from different individuals.

The *higher quartile variable crossover* (HQVC) operator crosses over each individual in the population higher quartile with one or more randomly selected individuals from the population spectrum. The number of crossover points cps is randomly determined so that it is at least 1 and at most the number of variables minus one: $cps \in [1, number\_of\_variables)$. This is necessary to avoid 1) useless attempts with no crossing point, 2) replacing an individual completely with other ones.

This operator is a multiple parent crossover operator. After determining the number of crossover points, the crossover points and also parents involved in the operation are randomly selected. For each crossover point a parent is also randomly selected. Finally, the selected individual from the higher quartile is crossed over with randomly selected individuals on randomly selected crossover points. This means that each individual contributes one variable to the offspring. Figure 12 provides the pseudo code for the higher quartile variable crossover operator.

```
procedure HQVC
begin
   for each individual in the higher quartile of the population do
        make a clone of individual
        select the number of crossover points $cps \in [1, number\_of\_variables)$
        for the number of crossover points cps do
           select a crossover variable $cv \in [1, number\_of\_variables]$
           select a respective individual cvind from the population
           replace bit string of cv in clone with bit string of cv in cvind
        done
        call survivor selection procedure (SSP) for clone
   done
end
```

Figure 12. The pseudo code for higher quartile variable crossover (HQVC) operator.

The variable crossover operator is also implemented on the elitist individual. See the description of the elitist variable crossover operator.

5.5.4 The Variable Replacement Operator

The *variable replacement* (VR) operator is a new operator introduced here. The motivation for this operator is that the value of a variable (a substring of the binary representation of the chromosome) in the individual may be in such an optimal position in the search space that will help also other variables to improve the fitness value of the individual. This can normally happen for a multiple variable problem, whose optimal value is achieved when all variables of the problem have the same certain value. The main idea with the variable replacement operator is that the value of each variable in turn replaces the values of all other variables in the same individual and then the individual is evaluated again. This process is repeated for all variables in the same individual. If any of these variable replacement trials generates a better offspring than the original one, the offspring is saved as a candidate to replace the original individual. At the end of the variable replacement process, the original individual is either replaced by the fitter individual produced by the variable replacement operator, or remains unchanged.

Two versions of this operator are implemented. One of them receives an individual as an argument and modifies it. The other version implements the variable replacement operator on all individuals in the higher quartile of the population. Figure 13 describes the idea of the variable replacement operator.

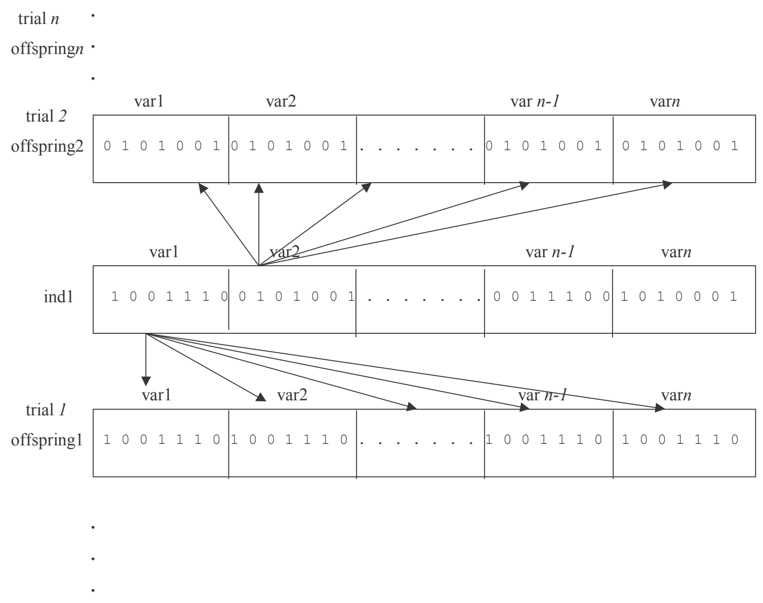

Figure 13. The variable replacement (VR) operator. The binary representation of each variable in the individual replaces the binary representations of all other variables in turn to produce a new offspring.

5.5.4.1 The Single Variable Replacement Operator

The *single variable replacement* (SVR) operator carries out the variable replacement process only for a selected individual. This operator receives an individual as the input and implements the variable replacement process for each variable in the individual as described earlier. If this produces a better offspring, this offspring will replace the individual received as the input. In the DDBGA this operator is called in conjunction with some other operators, like the integer mutation operator described later. Figure 14 provides the pseudo code for this operator.

```
procedure SVR(individual)
begin
    for each variable var in individual do
        make a clone of individual
        replace the values of other variables in clone with the value of var
    done
    find the best clone bclone
    if bclone is better than individual then
        replace individual with bclone
    endif
end
```

Figure 14. The pseudo code for the single variable replacement (SVR) operator.

5.5.4.2 The Higher Quartile Variable Replacement Operator

The *higher quartile variable replacement* (HQVR) operator carries out extensive replacement operator on all individuals in the higher quartile of the population. The reason for this is simply the fact that individuals in the higher quartile are the best ones and there is a high probability that one or more variables in these individuals are close to the optimal values of variables. Figure 15 provides the pseudo code for the higher quartile variable replacement operator.

```
procedure HQVR
begin
  for each individual in the higher quartile of the population do
    for each variable var in individual do
      make a clone of individual
      replace the values of other variables in clone with the value of var
    done
    find the best clone bclone
    if bclone is better than individual then
      replace individual with bclone
    endif
   call the survivor selection procedure (SSP) for individual
  done
end
```

Figure 15. The pseudo code for the higher quartile variable replacement (HQVR) operator.

5.5.5 Decimal Operators

One major problem with the classical implementation of binary mutation, the multiple point mutation or the crossover operator is that it is difficult to control their effect or to restrict changes caused by them within certain limits. Using the traditional techniques of doing random bit changes it is hard to modify the binary representation of a variable so that the value increases or decreases by a certain value. For example, using previously introduced technique for mapping binary strings to real values between –100 and 100 with 6 digits precision, the following bit string x :

1 0 0 0 0 1 0 1 0 0 0 1 1 1 1 0 1 0 1 1 1 0 0 0 0 0 0 0

represents 4, since $-100 + x_{(10)} \times \frac{200}{2^{28}-1} = 4$ and the following bit string y :

1 0 0 0 0 0 1 1 1 1 0 1 0 1 1 1 0 0 0 0 1 0 1 0 0 0 0 0

represents 3, since $-100 + y_{(10)} \times \frac{200}{2^{28}-1} = 3$. Comparing these bit strings shows how improbable it would be to modify the first bit string properly with traditional binary operators in order to get the second bit string. Of 28 bits, 10 bits on scattered positions need to be mutated in order to get the latter bit string from the earlier one. This would require 10 consecutive successful mutation operations provided that the same individual was selected each time. Thus, if the mutation probability for each bit is 0.01, then the theoretical probability for such a mutation would be $0.01^{10} \times 1/28 \times 1/27 \times 1/26 \ldots \times 1/19 = 10^{-20} \times 2.1 \times 10^{-14} = 2.1 \times 10^{-34}$. The probability for modifying the first bit string to the later one through a uniform crossover of size 10 would be $0.5^{10} \times \binom{28}{10}^{-1} = \frac{0.5^{10} \times 10!18!}{28!} = 5 \times 10^{-10} \times 4.8 \times 10^{-19} = 2.4 \times 10^{-28}$

Therefore, several techniques are developed to implement the genetic operators intelligently so that the resulting modifications on the binary string will cause changes in the real values within the desired limits. This idea is implemented so that the real value of the variable is randomly changed within the desired limits and the modified value then is converted to the binary representation and stored as the value of the variable. In this way we can cause more intelligent mutations in the bit strings and make sure that changes in real values are within the desired bounds.

Apparently, changes of different magnitudes are required at different stages of the evolutionary process. Thus, two types of decimal mutation operators have been implemented:

1. For modifying variables with integer values. The bounds for the absolute values of such changes are at least 1 and at most the integer part of the real value representation of the variable. This means that the upper bound of the range may vary even for each variable of the same individual. The randomly selected mutation value may be either positive or negative. Thus, if the integer part of the variable is $|\text{int}(\mathit{variable})|$, the integer mutation range is $\pm[1, |\text{int}(\mathit{variable})|]$.

2. For modifying variables with values from the range $(0,1)$. The lower bound for the absolute value of such changes is determined by the required precision of the real value presentation of the variable, like 10^{-6}. The upper bound for the absolute value of such changes is determined by decimal part of the variable. Here also the mutation value can be either positive or negative. Thus, if the number of digits after the decimal point for a variable is $\text{precision}(\mathit{variable})$, and the decimal part of the variable is $\text{decimal}(\mathit{variable})$, the range for the real mutation values is $\pm\left[10^{-\text{precision}(\mathit{variable})}, \left|\text{decimal}(\mathit{variable})\right|\right]$.

Two variations of these operators are implemented; one for the elitist individual and the other for the rest of the individuals of the population. These implementations differ: 1) in the range of mutation values and 2) the individuals, which are subject of these mutations.

5.5.5.1 The Integer Mutation Operator

The *integer mutation* (IM) operator mutates the individuals of the population in relatively great magnitudes. During this operation an integer mutation value Γ is selected randomly from the following range:

$$\Gamma \in \pm\left[1, \left|\text{int}(\mathit{variable})\right|\right] \tag{36}$$

and added to the variable under mutation. Here, $\left|\text{int}(\mathit{variable})\right|$ stands for the absolute value of the integer part of the variable. Clearly, this integer part does not necessarily cover the whole range of the variable. To avoid wasting resources special care is taken to make sure that the generated random number is not 0. Thus, the upper bound for the integer mutation value is different for each variable and is defined by the absolute value of the integer part of the variable. This will make the process more flexible and intelligent.

If the upper bound of the mutation value is set to a fixed value, the operator becomes inefficient or the probability for its failure will rise. For instance, if the optimal value of a variable is 0.05 and its present value 80.64, we will need 80 successful integer mutations of magnitude 1 in order to get close to the optimal value of the variable. However, if the magnitude of the integer mutation value can be dynamically determined by the magnitude of the variable, the operator will have a much greater chance to improve the value of the variable dramatically in a short time.

During this operation for each variable in the individual, first a Boolean value is randomly generated that determines whether the mutation operation for the variable at hand should take place or not. If the Boolean value is true, then a randomly generated integer number within the specified bounds is added to the decimal value of the variable. This process is repeated for each variable of the individual separately and the binary representation of the resulting offspring is updated. The offspring is then evaluated and put through the survivor selection procedure. Once this process is ready, the new offspring goes through the variable replacement operator for possible improvement. Next, the offspring, which has possibly been modified by the variable replacement operator, goes through the survivor selection procedure for possible substitution. There is no fixed rate for this operator. All population members go through this operator at least once. This operation is combined with the variable replacement operator described previously. After going through the variable replacement operator, the offspring goes again through the survivor selection procedure (SSP) for evaluation and possible selection. Figure 16 provides the pseudo code for the operator.

```
procedure IM
begin
    for each individual in the population do
        make a clone of individual
        for each variable $var$ in the clone do
            if a random Boolean value is true then
                select randomly the integer mutation value $imv \in \pm\left[1, \left|\text{int}(var)\right|\right]$
                add $imv$ to $var$ in clone
            endif
        done
        call the survivor selection procedure (SSP) for clone
        call the single variable replacement (SVR) operator for clone
        call the survivor selection procedure (SSP) for clone
    done
end
```

Figure 16. The pseudo code for the integer mutation (IM) operator.

5.5.5.2 The Decimal Mutation Operator

The *decimal mutation* (DM) operator is used in order to implement changes of smaller magnitudes on individuals. During this operation non-zero decimal numbers in the specific range are randomly generated and added to the randomly selected variables in the individual. The upper bound for the decimal mutation values is determined by the absolute value of the decimal part of variables. If the decimal part of a *variable* is denoted by $\text{decimal}(variable)$, the maximum distance of the mutation values from 0 is $\left|\text{decimal}(variable)\right|$. The lower bound of the mutation range is determined by the number of digits after the decimal point. Thus, if $\text{precision}(variable)$ shows the number of required decimal places of a *variable*, the decimal mutation value is determined in the following way:

$$\varphi \in \pm\left[10^{-\text{precision}(variable)}, \left|\text{decimal}(variable)\right|\right] \tag{37}$$

The difference between this operator and the integer mutation operator is the way the mutation value is determined. Otherwise, these operators are similar. Figure 17 provides the pseudo code for the operator.

```
procedure DM
begin
  for each individual in the population do
    make a clone of individual
    for each variable in the clone do
      if a random Boolean value is true then
        select randomly the decimal mutation value:
        $dmv \in \pm\left[10^{-precision}, \left|\text{decimal}(variable)\right|\right]$
        add dmv to variable in clone
      endif
    done
    call the survivor selection procedure (SSP) for clone
    call the single variable replacement (SVR) operator for clone
    call the survivor selection procedure (SSP) for clone
  done
end
```

Figure 17. The pseudo code for the decimal mutation (DM) operator.

5.6 Elitism

A lot of attention is paid to the best individual, i.e. elitist in the population and several operators have been designed in order to improve the quality of the elitist individual. However, measures are also taken in order to avoid dominance of the elitist in the population. In operations aimed at improving the elitist, the new offspring is copied to the population only if the offspring is better than the best individual of the previous generation or there is at least one individual, whose fitness value is worse than the new offspring.

Operators aimed at improving the elitist individual are the *elitist crossover*, the so-called *elitist variable crossover, elitist integer mutation* and *elitist decimal mutation*. Of these operators, the elitist crossover is the only standard one, which can be found from the literature. Other operators are developed in this research. At least they have not been seen in any other sources. Each operator will be explained separately in the following.

5.6.1 The Elitist Crossover Operator

The *elitist crossover* (EC) operator carries out an extensive crossover operator of the best individual with the rest of individuals in the population. Each individual is selected in turn for crossing over with the best one. The crossover point cp is a random index selected in the following way:

$$cp \in \left[2, \text{length}(individual) - 2\right], \tag{38}$$

where $\text{length}(individual)$ is the length of the binary representation of *individual*. The length of the crossover section is set to be at least 2 to assure that the operator is really crossover and not mutation.

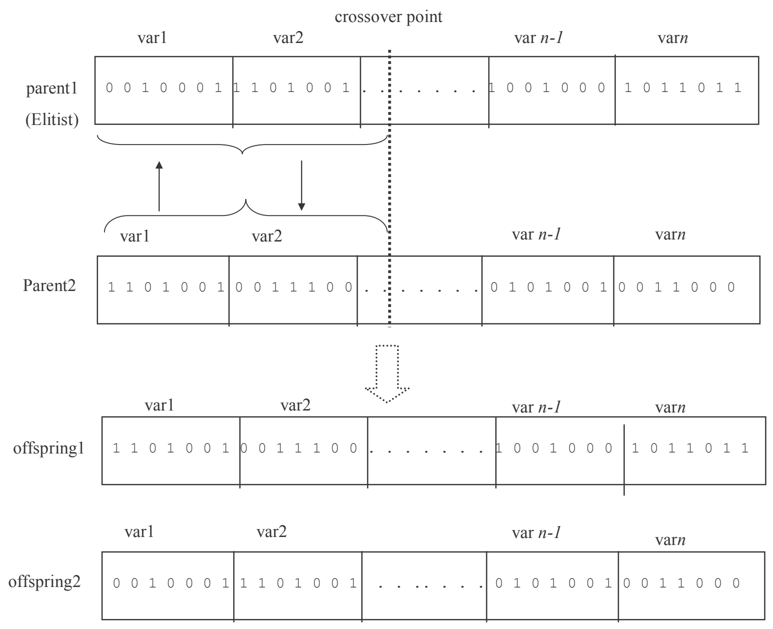

Figure 18. The elitist crossover operator. The best individual is crossed over with another individual at a single point to produce new offspring.

Thus, two offspring are generated by combining one part form the elitist individual and the other part from another individual in the population. The crossover actually happens with both the male and female chromosomes of the individual; the male chromosome of the elitist individual is crossed over once with the male chromosome and then with the female chromosome of the other parent.

Another modification to the traditional implementation of the crossover operator is that the elitist crossover process on the same individual indexes and the same crossing point is repeated as long as it produced better offspring. Clearly, repeating the process will be possible if it produces in its first round better offspring and replaces at least one of its parents. If the crossover process fails to produce any better offspring or fails to replace any of its parents, it will not be repeated after the first time. Figure 19 provides the pseudo code for the elitist crossover operator.

```
procedure EC
begin
    for each individual other than the elitist one in the population do
        select the elitist individual as parent1
        select individual as parent2
        select randomly crossover point $cp \in [2, \text{length}(parent1) - 2]$
        do
            crossover parent1 with the male chromosome of parent2 on cp
            crossover parent1 with the female chromosome of parent2 on cp
            failed=true
            for each resulting offspring do
                call the survivor selection procedure (SSP) for offspring
                if offspring is better than a population member then
                    failed=false
                endif
            done
        while (failed==false)
    done
end
```

Figure 19. The pseudo code for the elitist crossover (EC) operator.

5.6.2 The Elitist Variable Crossover Operator

The difference between the implementations on the elitist and the higher quartile is the fact that in the *elitist variable crossover* (EVC) the elitist individual is involved in each trial. Figure 20 describes the elitist variable crossover operator.

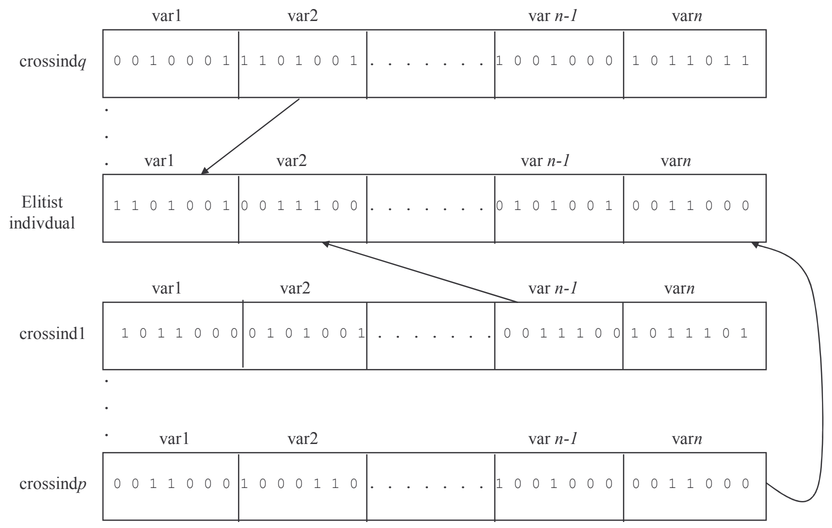

Figure 20. The elitist variable crossover (EVC) operator. Different individuals contribute a variable to the operator by replacing the binary representation of a variable in the best individual to produce a new offspring.

In each crossover cycle, first the number of crossover points is randomly generated. Then the crossover points and respective parents are randomly generated. At the end, the elitist and other parents take part into the variable crossover operator. The target individual for changes is only the elitist individual; other individuals are not modified. This means that all other selected individuals contribute one variable to the crossover operation by replacing the value of one variable in the elitist individual. Then the resulting offspring is put through the survivor selection procedure. This process is repeated for the number of individuals in the population. Figure 21 provides the pseudo code for the elitist variable crossover operator.

```
procedure EVC
begin
  for the number of individuals in the population do
     make a clone of the best individual
     select randomly the number of crossover points:
     $cps \in [1,\ number\_of\_variables)$
     for the number of crossover points cps do
          select a crossover point $cp \in [1,\ number\_of\_variables]$
          select randomly respective parent $p \in [1, population\_size]$
     done
     crossover the clone on each crossover point with its respective parent
     call the survivor selection procedure (SSP) for clone
  done
end
```

Figure 21. The pseudo code for elitist variable crossover (EVC) operator.

5.6.3 The Elitist Integer Mutation Operator

The *elitist integer mutation* (EIM) operator is used in order to fine-tune the best candidate solution in the population. Changes of the smallest possible integer magnitude are imposed in order to mutate the best candidate solution in a cautious way with more chances of success. Thus, the integer mutation value is determined in the following way:

$$\Gamma \in \{-1,1\} \tag{39}$$

This operation is performed for each variable in the best individual of the population. For each variable, first a Boolean value with $\mathrm{P}\{true\} = 0.5$ is randomly generated that determines whether the mutation for the variable at hand should take place or not. If the Boolean value is true, then a randomly generated integer number is added to the decimal value of the variable. Otherwise, the variable remains intact. Once this process is completed for all variables in the elitist individual, the binary representation of the resulting offspring is updated and put through the survivor selection procedure. Then the new offspring is forced to go through the variable replacement operator and the survivor selection procedure again.

There is no fixed rate for this operation. The operation on the elitist individual is repeated as long as it generates individuals better than at least one individual in the population. Figure 22 describes the pseudo code for the operator.

```
procedure EIM
begin
  do
      make a clone of the elitist individual
      for each variable in clone do
        if a random Boolean value is true then
          select randomly the integer mutation value imv $\in \{-1,1\}$
          add imv to the variable in clone
        endif
      done
        failed=true
      call the survivor selection procedure for clone
      if clone is better than a population member then
        failed=false
      endif
      call the single variable replacement (SVR) operator on clone
      call the survivor selection procedure (SSP) for clone
  while (failed==false)
end
```

Figure 22. The pseudo code for the elitist integer mutation (EIM) operator.

5.6.4 The Elitist Decimal Mutation Operator

The *elitist decimal mutation* (EDM) operator is used after the elitist integer mutation operator in order to implement changes of smaller magnitudes on the elitist individual. The minimum decimal mutation value is determined by the required number of digits after the decimal point, giving the minimum mutation value $\pm 10^{-\mathrm{precision}(variable)}$. The maximum decimal mutation value is restricted to 0.1 in order to increase the chance for successfulness of the operation. Thus, the decimal mutation value is randomly determined in the following way:

$$\varphi \in \pm\left[10^{-\mathrm{precision}(variable)}, 0.1\right] \tag{40}$$

This operation is repeated for each variable in the elitist individual. Except for the mutation value, the rest of the process is similar to the elitist integer mutation operator. Figure 23 describes the pseudo code for the elitist decimal mutation operator.

```
procedure EDM
begin
   do
        make a clone of the elitist individual
        for each variable in clone do
           if a random Boolean value is true then
              select randomly the decimal mutation value $dmv \in \pm\left[10^{-precision}, 0.1\right]$
              add $dmv$ to variable in clone
           endif
        done
           failed=true
        call the survivor selection procedure for clone
        if clone was better than a population member then
           failed=false
        endif
        call the single variable replacement operator (SVR) on clone
        call the survivor selection procedure (SSP) for clone
   while (failed==false)
end
```

Figure 23. The pseudo code for the elitist decimal mutation (EDM) operator.

There is no fixed rate for this operation. The operation on the best individual is repeated as long as it generated individuals better than at least one individual in the population.

5.6.5 The Population Refreshing Procedure

The variation of the fitness values may occasionally decrease due to the dominance of an overwhelmingly fitter individual. This is known to slow down the search space exploration procedure. In order to overcome this problem, the population refreshing procedure (PRP) is implemented. This procedure is called if the population is totally overtaken by a single individual and the difference between the fitness value of the best and the worst individuals is 0. The population refreshing procedure keeps the best individual and replaces the rest of individuals with new randomly generated individuals.

Filling the population makes it possible to leave out some operators especially when the number of variables is small. Therefore, for some problems with a small number of variables this procedure was called during the run. Figure 24 describes the pseudo code for the procedure.

```
procedure PRP
begin
  for each individual except for the elitist in the population do
    for each gene in the individual do
      select randomly the binary gene value $gv \in \{0,1\}$ for the male chromosome
      calculate the gene value for the female chromosome $fgv = 1 \text{ XOR } gv$
    done
    evaluate male and female chromosomes
    set the fitter chromosome as male and the worse chromosome as female
    replace the individual's male chromosome
    replace the individual's female chromosome
  done
end
```

Figure 24. The pseudo code for the population refreshing procedure (PRP).

5.7 The DDBGA Algorithm

The distributional decimal binary genetic algorithm (DDBGA) starts with the initialization of the population and calls the previously described operators in the pre-specified order. Survivor selection happens during each operation. Once the operator has been implemented for one or more individuals, the fitness value of the best individual is compared to the desired optimal value and if they are equal, the algorithm is terminated. This action is naturally done in order to save processing time when testing the ability of the algorithm to find optimal solutions of hard problems. When the optimum is not known a priori, the number of generations or the best achieved fitness value within certain pre-set bounds value can work as the termination condition.

The population is sorted again in ascending order after application of each operator, and the population diversity is checked. This is done by comparing the fitness values of the best and the worst individual. If they are equal, the population refreshing procedure

(PRP) is called for solving some problems. After each operator call the termination condition is checked for possible process termination. The idea behind the order of operator calls is to carry out operations of greater effect first, and then focus on operations of smaller effect and fine-tuning. Figure 25 describes the DDBGA algorithm.

```
procedure DDBGA
begin
  initialize the population
  sort individuals in ascending order based on their fitness values
  do
     call the higher quartile crossover operator
     sort the population in ascending order
     if ( termination_condition is true) then
         break
     endif

     if the problem is of the multiple variable type then
       call the higher quartile variable crossover operator
       sort the population in ascending order
        if ( termination_condition is true) then
          break
       endif
     endif

     if the problem is of the multiple variable type then
       call the higher quartile variable replacement operator
       sort the population in ascending order
       if ( termination_condition is true) then
          break
       endif
     endif

     call the random building block operator
     sort the population in ascending order
      if ( termination_condition is true) then
         break
     endif

     call the integer mutation operator
     sort the population in ascending order
      if ( termination_condition is true) then
         break
     endif

     call the decimal  mutation operator
     sort the population in ascending order
```

```
        if ( termination_condition is true) then
            break
        endif

        call the elitist crossover operator
        sort the population in ascending order
        if ( termination_condition is true) then
            break
        endif

        if the problem is of the multiple variable type then
            call the elitist variable crossover operator
            sort the population in ascending order
            if ( termination_condition is true) then
                break
            endif
        endif

        call the elitist integer mutation operator
        sort the population in ascending order
        if ( termination_condition is true) then
            break
        endif

        call the elitist decimal mutation operator
        sort the population in ascending order
        if ( termination_condition is true) then
            break
        endif

        if ( $\text{fitness}(best\_individual) - \text{fitness}(worst\_individual) == 0$ ) then
            call the population refreshing procedure
        endif

    while (termination_condition is false)
end
```

Figure 25. The pseudo code for the DDBGA algorithm.

6. Experimentation

The DDBGA was implemented in Java2. It was tested on various widely used optimization test functions. Test runs were performed on Mobile Intel Pentium, 4-M CPU 1.80 GHz, 1.18 GHz, 512 MB of RAM running Microsoft Windows XP professional version 2002 with service pack 2.

Each test run consisted of 10 trials of the run sequence. One run sequence, in turn consisted of five distinct runs. This means that for five consecutive times the algorithm was initiated from scratch and let run until the termination condition came true. This proved to be a fruitful method for solving problems. The main justification for such a trick is the fact that sometimes the random initialization of the population is not as desirable as it should be. Consequently, operators might not be able to produce better offspring as quickly and efficiently as one would wish. Therefore, new chances were given to the population to be born anew.

The reborn of population was mainly used in order to test the ability of the algorithm to find all global optimum values for problems which have several global optimum values. The number of repetition of runs was selected to be five here, but it could be easily changed if there is any suspicion, that the optimization problem might have more global optimum values.

6.1 Test Functions

The DDBGA was tested on the following widely used minimization test functions.

1. The Ackley function:

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

where $-32.768 \leq x_i \leq 32.768$.

2. The Colville function:

$$100(\mathrm{x}_2 - x_1^2)^2 + (1 - x_1)^2 + 90(x_4 - x_3^2)^2 + (1 - x_3)^2 + 10.1((x_2 - 1)^2 + (x_4 - 1)^2) + 19.8(x_2 - 1)(x_4 - 1), \text{ where } -10 \le x_i \le 10 \quad \text{for i} = 1,\ldots 4.$$

3. The De Jong function F1:

$$\sum_{i=1}^{3} x_i^2 \text{, where } -5.12 \le x_i \le 5.12 \,.$$

4. The De Jong function F2:

$$100(x_1^2 - x_2)^2 + (1 - x_1)^2 \text{, where } -2.048 \le x_i \le 2.048 \,.$$

5. The De Jong function F3:

$$\sum_{i=1}^{5} \text{integer}(x_i) \text{, where } -5.12 \le x_i \le 5.12 \,.$$

6. The De Jong function F4:

$$\sum_{i=1}^{30} i x_i^4 + N(0,1) \text{, where } -1.28 \le x_i \le 1.28 \,.$$

7. The De Jong function F5:

$$\frac{1}{1/k + \sum_{j=1}^{25} 1/f_j(x_1, x_2)},$$

where $-65.536 \le x_i \le 65.536, K = 500$, $f_j(x_1, x_2) = j + \sum_{i=1}^{2} (x_i - a_{ij})^6$ and

$$a_{ij} = \begin{bmatrix} -32 & -16 & 0 & 16 & 32 & -32 & -16 & \ldots & 0 & 16 & 32 \\ -32 & -32 & -32 & -32 & -32 & -16 & -16 & \ldots & 32 & 32 & 32 \end{bmatrix}.$$

8. The Griewank function F1:

$$\sum_{i=1}^{n} (x_i - 100)^2 - \prod_{i=1}^{n} \cos(\frac{x_i - 100}{\sqrt{i}}) + 1 \text{, where } -600 \le \mathrm{x_i} \le 600 \,.$$

9. The Rastrigin function:

$$\sum_{i=1}^{n} (x_i^2 - 10\cos(2\pi x_i) + 10) \text{, where } -5.12 \le \mathrm{x_i} \le 5.12 \,.$$

10. The Rosenbrock function:

$$\sum_{i=1}^{n-1} (100(x_{i+1} - x_i^{\,2})^2 + (x_i - 1)^2) \text{, where } -100 \le x_i \le 100 \,.$$

11. The Schaffer function F6:

$$0.5+\frac{\sin^2\sqrt{x_1^2+x_2^2}-0.5}{\left[1.0+0.001(x_1^2+x_2^2)\right]^2}, \text{ where } -100 \le x_i \le 100.$$

12. The Schaffer function F7:

$$(x_1^2+x_2^2)^{0.25}\left[\sin^2(50(x_1^2+x_2^2)^{0.1})+1.0\right], \text{ where } -100 \le x_i \le 100.$$

For multidimensional problems with optional number of dimensions (n), the algorithm was tested for $n = 1$, 2, 3, 4, 5, 10, 20, 50 and 100.

Of the test functions, the De Jong functions reflect different degrees of complexity. Test functions F1-F4 are unimodal with only one optimum, whereas F5 is a multimodal function with many local optima, but only one global optimum. The Rosenbrock function is considered difficult, because it has a very narrow ridge. The tip of the ridge is very sharp and it runs around a parabola. Algorithms that are not able to discover good directions to find the solutions underperform this problem (Digalakis & Konstantinos 2002).

The Rastrigin and Griewank functions are typical examples of non-linear multimodal optimization problems. The Rastrigin function is fairly difficult for GAs due to the large search space and the large number of local minima. This function has a complexity of $O(n\ln(n))$, where n is the number of the function parameters. This means that the Rastrigin function does not grow at a faster rate than $O(n\ln(n))$. It contains millions of local optima in the interval of consideration (Digalakis et al. 2002).

The Griewank function has a complexity of $O(n\ln(n))$, where n is the number of the function's parameters. The terms of the summation produce a parabola, while the local optima are above the parabola level. The dimensions of the search range increase on the basis of the product, which results in the decrease of the local minimums. The more the search range increases, the flatter the function becomes. Generally speaking this is a good function for testing GA performance since the product creates sub-populations strongly codependent to parallel GAs models. Most algorithms have difficulties to converge close to the minimum of such functions, because the probability of making progress decreases rapidly as the minimum is approached (Digalakis et al. 2002).

6.2 Performance Issues

Krink (2005) lists some performance criteria for optimization algorithms: 1) the accuracy of results (precision), 2) whether good results can be obtained repeatedly (robustness), 3) computation time (efficiency), 4) the number of different problems the algorithm can solve (generality), 5) how much effort the implementation requires (simplicity). Thus, in reporting the test results the above criteria have been taken into account.

The main performance criterion used here is the number of function evaluations performed by the algorithm to solve the test problem. The number of function evaluations was calculated by incrementing a counter variable by one each time the fitness value of a candidate solution was calculated. The required precision was set to 6 decimal digits. This means that if the global optimal value was 0, it was considered to be found when the difference between the best achieved fitness value and the optimal value was less than 0.000001. For problems with a global optimum different from zero, the optimum value was considered to be found, if the difference between the best fitness value and the global optimum was zero.

The ability of the algorithm to achieve an approximate optimal value was also tracked. The approximate optimal precision value was set to 0.001. This means that when the difference between the achieved best fitness value and the pre-set optimal value was less than 0.001, then the global approximate optimal value was considered to be found. The descriptive statistics of the fitness population were calculated after each operator call in order to track the progress of the population minimum and maximum fitness values. The descriptive statistics of the initial and the final population in the best and the worst run of each test case (50 test runs) are reported to give an insight of how the population developed. These statistics were the minimum, maximum, mean, median, mode, standard deviation, skewness and kurtosis.

Median is used to point out the middle of a distribution: half the scores are above the median and half are below the median. The median is less sensitive to extreme scores than the mean and this makes it a better measure than the mean for highly skewed distributions. The mode is the most common (frequent) value. A population can have more than one mode. A mode is a relative maximum. Mode is especially useful to point out the most typical occurrence of a value. (Neter, Wasserman & Whitmore 1993)

The standard deviation and indicators of the fitness population distribution, i.e. skewness and kurtosis are used to describe how fitness values were distributed in the beginning and at the end of each run. The standard deviation is the positive square root of the variance. The variance s^2 is the average of the squared deviations about the mean, $(x_i - \overline{x})^2$. The variance sometimes is called the second moment about the mean and denoted by m_2. The standard deviation σ of a set of observations $x_1, x_2, ...x_n$ with mean $\overline{x}$ is defined as (Neter et al. 1993):

$$\sigma = \sqrt{s^2} = \sqrt{\frac{\sum_{i=1}^{n}(x_i - \overline{x})^2}{n-1}}\,. \tag{41}$$

Skewness is a measure of the lack of symmetry. A distribution, or data set, is symmetric if it looks the same to the left and right of the center point. Negative values for the skewness indicate data that are skewed left and positive values for the skewness indicate data that are skewed right. Skewed left means that the left tail is heavier than the right tail. Similarly, skewed right means that the right tail is heavier than the left tail. Some measurements have a lower bound and are skewed right. Skewness m_3 is the third moment about the mean of a set of observations $x_1, x_2, ...x_n$ with mean $\overline{x}$ and is defined as (Neter et al. 1993):

$$m_3 = \frac{\sum_{i=1}^{n}(x_i - \overline{x})^3}{n-1}\,. \tag{42}$$

The standardized skewness measure m_3' for a data set is defined as: (Neter et al. 1993)

$$m_3' = \frac{m_3}{\sigma^3}. \qquad (43)$$

A data set is skewed positively or negatively according to whether m_3' is positive or negative. If the values in the data set are symmetric about the mean, m_3'=0.

Kurtosis is a measure of whether the data are peaked or flat relative to a normal distribution. That is, data sets with high (positive) kurtosis tend to have a distinct peak near the mean, decline rather rapidly, and have heavy tails. Data sets with low (negative) kurtosis tend to have a flat top near the mean rather than a sharp peak. A uniform distribution would be the extreme case. Kurtosis m_4 is the fourth moment about the mean of a set of observations $x_1, x_2, ...x_n$ with mean $\overline{x}$ and is defined as (Neter et al. 1993):

$$m_4 = \frac{\sum_{i=1}^{n}(x_i - \overline{x})^4}{n-1}. \qquad (44)$$

The standardized kurtosis measure m_4' of a data set is defined as (Neter et al. 1993):

$$m_4' = \frac{m_4}{\sigma^4}. \qquad (45)$$

The kurtosis for a standard normal distribution is 3. If in calculations m_4' is replaced by $m_4' - 3$, the kurtosis of the normal distribution becomes zero and kurtosis of other distributions can be compared with zero.

6.3 The Ackley Function F1

The Ackley function F1 is a multimodal minimization problem and its mathematical representation is the following:

$$20 + e - 20\exp\left(-0.2\sqrt{\frac{1}{n}\sum_{i=1}^{n}x_i^2}\right) - \exp\left(\frac{1}{n}\sum_{i=1}^{n}\cos(2\pi.x_i)\right), \qquad (46)$$

where $-32.768 \le x_i \le 32.768$ for $i = 1,...,n$.

This function has a global minimum value of 0 at $x_i = 0$ for $i = 0,1,2,...n$. The approximate optimal value was set to 0.001 for this function. All operators were used for solving this problem. A summary of test results for $n = 2$ and $n = 100$ is reported in Table 1. A summary of test results for other 7 cases is presented in Appendix A.

In Table 1 and other similar ones, n is the number of variables in the function. *Run Summary* gives a summary of statistics of the worst run, the best run and all 50 runs. *Generations* indicates the number of elapsed generations. *Evaluations to Approx. Optimal* indicates the number of performed function evaluations to achieve the approximate global optimum. *Time to Approx. Optimal(s)* indicates the elapsed time to achieve the approximate optimal value. *Total Evaluations* indicates the number of performed function evaluations to achieve the global optimal value and *Total Elapsed Time(s)* indicates the elapsed time to achieve the global optimal value.

Table 1. Test results for the Ackley function F1 with different number of variables (n).

n	Run Summary	Generations	Evaluations to Approx. Optimal	Time to Approx. Optimal(s)	Total Evaluations	Total Elapsed Time(s)
2	**Best Run**	1	207	0.00	207	0.00
	Worst Run	13	1543	1.00	2384	1.00
	Mean	5.34	730.12	0.16	941.76	0.32
	Median	5.00	709.50	0.00	802.00	0.00
	Mode	4.00	#N/A	0.00	1004.00	0.00
	Std. Dev.	2.20	291.61	0.37	405.45	0.51
100	**Best Run**	1	2344	5.00	2344	5.00
	Worst Run	2	2787	6.00	5344	13.00
	Mean	1.18	2835.28	6.88	3202.22	7.92
	Median	1.00	2771.00	7.00	2771.00	7.00
	Mode	1.00	2787.00	6.00	2964.00	6.00
	Std. Dev.	0.39	487.82	1.56	940.98	2.66

Statistics drawn from 450 runs (50 runs for each value of n) reveal that the algorithm found the global optimal value of the Ackley function on average within 4 generations, 1453 evaluations and 1.42 seconds. When the effect of extreme cases is eliminated, the median value suggests that the algorithm solved the problem within 4 generations, 1011 evaluations in less than 1 second. In the most typical case, the algorithm solved the problem in 2 generations in less than 1 second.

The most important descriptive statistics of populations were tracked in order to point out the changes through which the fitness population has gone before achieving its optimal value. The following table summarizes these statistics for the worst and the best populations of the best and the worst runs for the number of variables being 2 and 100. The descriptive statistics for other cases of the function are presented in Appendix A.

In Table 2 and other similar ones n is the number of variables in the function. *Runs* indicates the type of the run; the best run is the run in which the algorithm solved the problem with the smallest number of function evaluations, and the worst run is the run in which the algorithm solved the problem with the greatest number of function evaluations. For the best and the worst runs, descriptive statistics of the worst and the best populations are reported. The worst population is the initial population with poor fitness values, and the best population is the latest population with the best achieved fitness values. *Min* is the minimum fitness value, *Max* is the maximum fitness value and *Mean* is the mean of fitness values in the population. *Std. Dev.* is the standard deviation, *Skew*. is the skewness and *Kurt.* is the kurtosis of the fitness population.

Table 2. Descriptive statistics of the population fitness values for the test runs on the Ackley function F1 with different numbers of variables (n).

n	Runs	Pops	Min	Max	Mean	Std. Dev.	Skew.	Kurt.
2	Best Run	Worst Pop.	-17.27	-10.31	-14.09	2.00	0.00	0.00
		Best Pop.	0.00	0.00	0.00	0.00	0.00	-3.00
	Worst Run	Worst Pop.	-19.09	-10.18	-13.74	3.00	0.00	-1.00
		Best Pop.	0.00	0.00	0.00	0.00	0.00	-3.00
100	Best Run	Worst Pop.	-21.15	-21.04	-21.08	0.00	0.00	-3.00
		Best Pop.	0.00	0.00	0.00	0.00	0.00	-3.00
	Worst Run	Worst Pop.	-21.17	-20.96	-21.05	0.00	0.00	-3.00
		Best Pop.	0.00	0.00	0.00	0.00	0.00	-3.00

Since the DDBGA was originally designed for maximization problems, the function has been multiplied by -1 to adapt it to the algorithm. Statistics drawn from 450 runs suggest that the lowest fitness value in the initial population was –21.18. The population average fitness value was in the worst case -21.08 and in the best case –3E-6. The lowest standard deviation was 0 and the highest one was 3. The population skewness was at least 0 and at most 16. The population kurtosis was at least –3 and at most 96.

The following diagrams depict the role of each operator in improving the population minimum and maximum fitness value in the worst run and the best run in cases where the number of variables was 2 and 100. The horizontal axis shows operators with their generation index. Generation indexes start with 0; e.g. *RBB0* stands for the random building block operator called in generation 1. The vertical axis shows the relative improvement made by each operator. The relative improvement made by each operator was calculated in the following way: if the fitness value of the worst (or the best) individual in the population before and after calling an operator is f_{old} and f_{new} respectively, the relative improvement Δ of the population minimum (or maximum) fitness value is calculated as: $\Delta = \frac{f_{new} - f_{old}}{f_{old}} \times 100$.

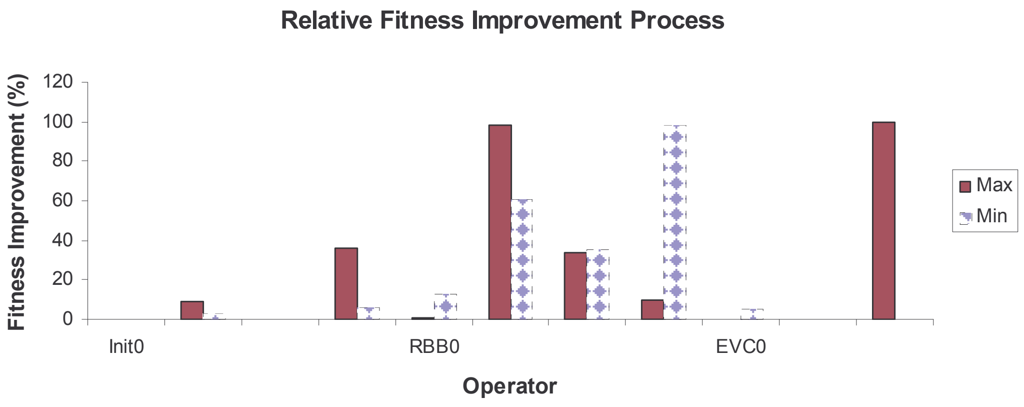

Figure 26. The relative fitness improvement made by each operator in the best run for the Ackley function with 2 variables.

The smallest improvement to the population minimum fitness value was made by the higher quartile crossover operator and the greatest improvement (98%) was made by the elitist crossover operator. The smallest improvement to the population maximum fitness value was made by the random building block operator, and the greatest improvement (100%) was made by the elitist decimal mutation operator in the only generation 1. Figure 27 illustrates the development of population fitness values in the worst run when the number of variables was 2.

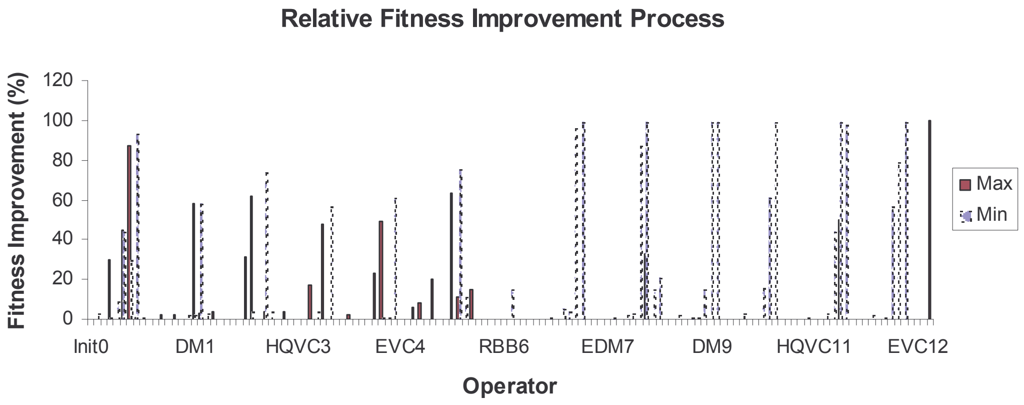

Figure 27. The relative fitness improvement made by each operator for the worst run for the Ackley function with 2 variables.

The smallest improvement to the population minimum fitness value was made by the higher quartile variable replacement operator and the greatest improvement (93%) was made by the elitist crossover operator in generation 1. The last improvement was made by the decimal mutation operator in generation 13. The smallest improvement to the population maximum fitness value was made by the elitist crossover operator and the greatest improvement (87%) was made by the decimal mutation operator in generation 1. The last improvement was made by the elitist decimal mutation operator in generation 13. Figure 28 illustrates the role of each operator in improving the population fitness value in the best run in case the number of variables is 100.

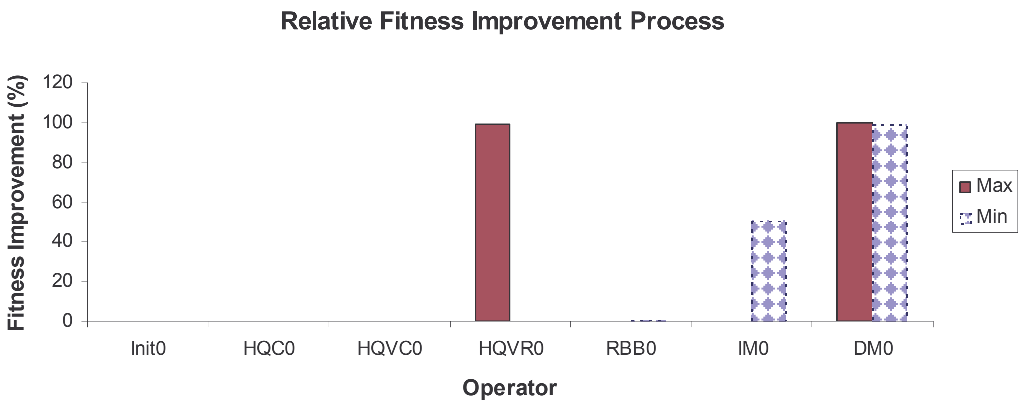

Figure 28. The relative fitness improvement made by each operator in the best run for the Ackley function with 100 variables.

The greatest improvement (99%) to the population minimum fitness value was made by the decimal mutation operator in the only generation 1. The population maximum fitness value was almost equally improved by the higher quartile variable replacement operator and the decimal mutation operator. Figure 29 depicts development of the population fitness values in the worst run when the number of variables was 100.

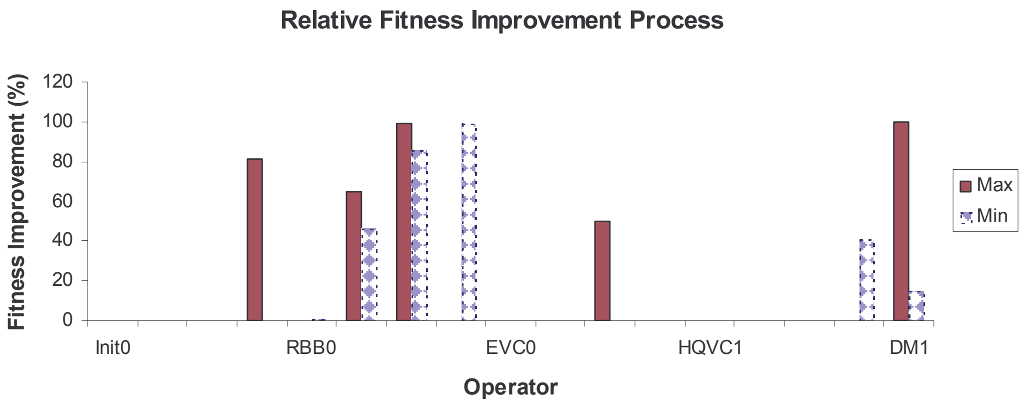

Figure 29. The relative fitness improvement made by each operator for the worst run for the Ackley function with 100 variables.

The smallest improvement to the population minimum fitness value was made by the random building block operator and the greatest improvement (99%) was made by the elitist crossover operator in generation 1. The smallest improvement to the population maximum fitness value was made by the elitist decimal mutation operator and the greatest improvement (99%) was made by the decimal mutation operator in generation 1.

In addition, the survival of male and female chromosomes in the selection and crossover operations was tracked. Table 3 summarizes the survival statistics.

Table 3. Comparison of survival of male and female chromosomes.

Variables	Chromosome Sex	Survival Rate in Selection (%)	Survival Rate in Crossover (%)	
			Worst Run	Best Run
2	Male	50	60	72
	Female	50	39	27
100	Male	50	52	100
	Female	50	47	0

The statistics point out that male and female chromosome survived equally in the selection. However, in the crossover operation male chromosomes produced better offspring. When the number of variables was 2 in the worst run male chromosomes produced better offspring in 60% of cases while female offspring produced better offspring in 39% of cases. In the best run, the survival rate of male and female chromosomes in the crossover operator was 72% and 27% respectively.

When the number of variables was 100, in the worst run male and female chromosomes produced good offspring almost equally. However, in the best run male chromosomes produced better offspring than female chromosomes in all cases.

6.4 The Colville Function

The Colville function is a minimization function and its mathematical presentation is the following:

$$100(\mathrm{x}_2 - x_1^2)^2 + (1-x_1)^2 + 90(x_4 - x_3^2)^2 + (1-x_3)^2 + 10.1((x_2-1)^2 + (x_4-1)^2) + 19.8(x_2-1)(x_4-1), \text{ where } -10 \le x_i \le 10 \quad \text{for i} = 1,\ldots 4. \tag{47}$$

This function has a global minimum value of 0 at $(x_1, x_2, x_3, x_4) = (1,1,1,1)$. All operators were used for solving this problem. The population refreshing procedure was not used. Table 4 summarizes the statistics retrieved from these test runs.

Table 4. Summary of test runs for the Colville function.

Summary	**Generations**	**Evaluations to approx. optimal**	**Time to approx. optimal (s)**	**Total Evaluations**	**Total Elapsed Time (s)**
Best Run	1	54	0	54	0
Worst Run	1	181	0	181	0
Mean	1	107.84	0	107.84	0
Median	1	129	0	129	0
Mode	1	129	0	129	0
Std. Dev.	0	37.98945	0	37.98945	0

The algorithm solved the Colville function in the best case in 1 generation, 54 evaluations and less than 1 second. In the worst case, solving the function required 1 generation and 181 evaluations. On average, the algorithm solved the problem in 1 generation, 108 evaluations and less than 1 second. Most typically, the algorithm solved the problem in 1 generation, 129 evaluations and less than 1 second. Table 5 summarizes the most descriptive statistics of the population in the beginning and at the end of the best and the worst runs.

Table 5. Summary of the most descriptive statistics of the fitness population for the best and the worst runs for the Colville function.

Runs	Pops	Min	Max	Mean	Std. Dev.	Skew.	Kurt.
Best Run	**Worst Pop.**	-850914.40	-5577.100	-340357.025	337778.000	0.0	-2.0
	Best Pop.	-720976.00	0.000	-205315.550	275863.000	0.0	-2.0
Worst Run	**Worst Pop.**	-1177926.10	-2215.900	-335294.608	363236.000	-1.0	0.0
	Best Pop.	-42.00	0.000	-24.500	21.000	0.0	-2.0

The function has been multiplied by -1 to adapt it to DDBGA. The population maximum and mean fitness values in the initial population of the best run were worse than the respective values in the initial population of the worst run. Also the algorithm seems to have improved the final population in the worst run much more than in the best run. Figure 30 illustrates the role of each operator in improving the population fitness value during the best run.

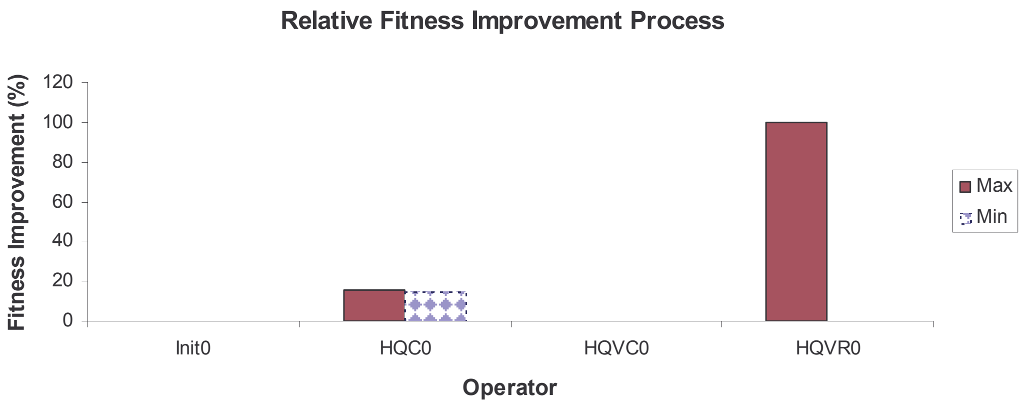

Figure 30. The relative fitness improvement made by each operator in the best run for the Colville function.

The population minimum fitness value was improved 15% by the higher quartile crossover operator. The greatest improvement (100%) to the population maximum fitness value was made by the higher quartile variable replacement operator in the only generation 1. Figure 31 illustrates the role of each operator in improving the population fitness value during the worst run.

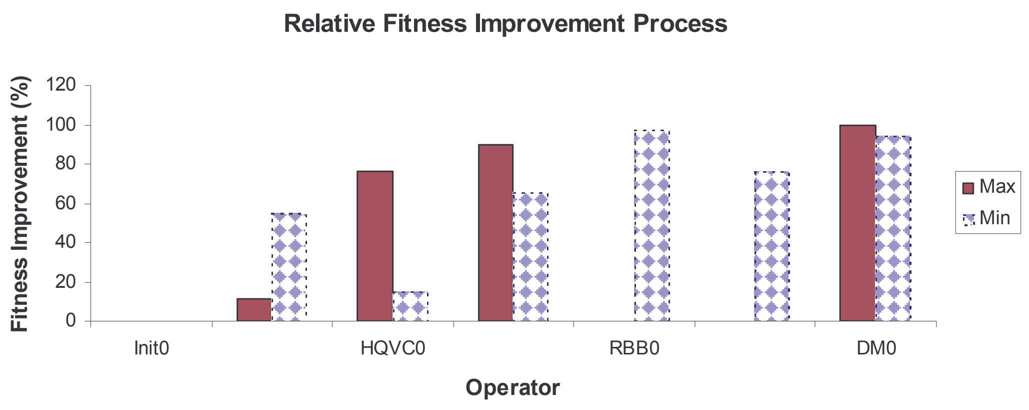

Figure 31. The relative fitness improvement made by each operator in the worst run for the Colville function.

The smallest improvement to the population minimum fitness value was made by the higher quartile variable crossover operator and the greatest improvement (97%) was made by the random building block operator. The smallest improvement to the population maximum fitness value was made by the higher quartile crossover operator and the greatest improvement (100%) was made by the decimal mutation operator in generation 1. Table 6 summarizes the survival statistics of male and female chromosomes in the initialization phase and the crossover operations.

Table 6. Comparison of survival of male and female chromosomes.

Chromosome Sex	**Survival Rate in Selection (%)**	**Survival Rate in Crossover (%)**	
		Worst Run	**Best Run**
Male	50	75	33
Female	49	25	66

The male and female chromosomes seem to have survived equally in the selection phase. However, their successfulness in producing offspring seems to have varied.

During the worst run, male chromosomes produced three times better offspring than the female ones. On the other hand, female chromosomes produced two times better offspring than the male ones during the worst run.

6.5 The De Jong Function F1

The De Jong function F1 is a minimization problem and its mathematical representation is the following:

$$\sum_{i=1}^{3} x_i^2 \text{ , where } -5.12 \le x_i \le 5.12 \text{ .} \tag{48}$$

This function has a global minimum value of 0 at $(x_1, x_2, x_3) = (0,0,0)$. This function is considered very simple for every serious minimization method (Storn & Price 1995). The following operators were used to solve this problem: the decimal mutation operator, the elitist integer mutation operator and the elitist decimal mutation operator. Experimentation showed that using other operators would increase the number of function evaluations unnecessarily. The population refreshing procedure was not used. Table 7 summarizes the statistics for these runs.

Table 7. Summary of test runs for the De Jong function F1.

Summary	Generations	Evaluations to approx. optimal	Time to approx. optimal (s)	Total Evaluations	Total Elapsed Time (s)
Best Run	2.00	140	0	147	0
Worst Run	14.00	92	0	818	1
Mean	5.78	185.16	0	352.68	0.04
Median	5.00	196	0	315	0
Mode	6.00	196	0	361	0
Std. Dev.	2.44	65.79	0	136.71	0.20

The algorithm solved the problem in the best run in 2 generations, 147 evaluations and in less than 1 second. In the worst run, it took the algorithm 14 generations, 818 evaluations and 1 second to solve the problem. However, this seems to be a single extreme case. On average, the algorithm solved the problem in 6 generations and 353 evaluations in less than 1 second. The median value indicates that when the effect of extreme cases is eliminated, the algorithm solved the problem in 5 generations, 315 evaluations in less than 1 second. Most typically, the algorithm solved the problem in 6 generations and 361 evaluations in less than 1 second. Table 8 summarizes the most descriptive statistics of the initial and the final population in the best and the worst run.

Table 8. Descriptive statistics of the population fitness values for the test runs on De Jong function F1.

Runs	Pops	Min	Max	Mean	Std. Dev.	Skew.	Kurt.
Best Run	**Worst Pop.**	-46.40	-5.01	-22.50	11.00	0.00	0.00
	Best Pop.	-0.09	0.00	-0.04	0.00	0.00	-3.00
Worst Run	**Worst Pop.**	-39.18	-1.30	-21.32	14.00	0.00	-2.00
	Best Pop.	0.00	0.00	0.00	0.00	0.00	-3.00

The function has been multiplied by -1 to adapt it to DDBGA. The most important descriptive statistics of the initial population in the best run were worse than the respective values in the worst run. Nevertheless, deviation in the fitness population was greater in the initial population of the worst run. The situation of the final fitness populations in both runs seem to have been virtually equal. Figure 32 shows the role of each operator in improving the population minimum and maximum fitness value in the best run.

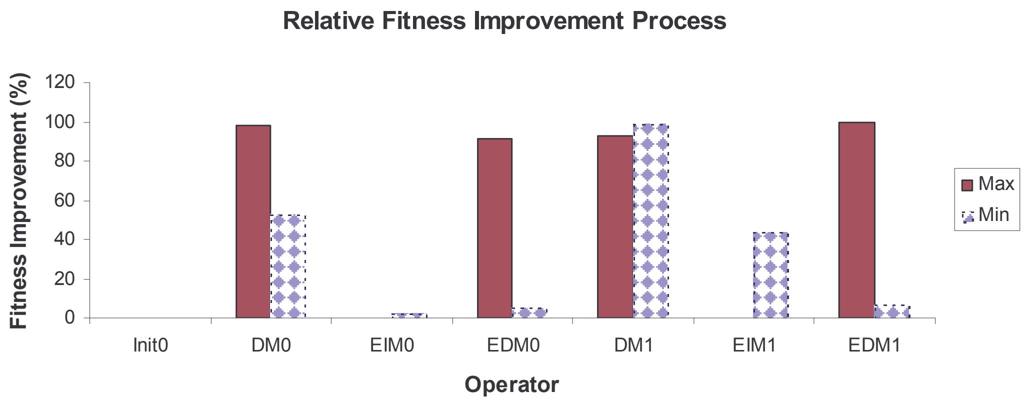

Figure 32. The relative fitness improvement made by each operator in the best run for the De Jong function F1.

The smallest improvement to the population minimum fitness value was made by the elitist integer mutation operator and the greatest improvement (53%) was made by the decimal mutation operator in generation 1. The smallest improvement to the population maximum fitness value was made by the elitist decimal mutation operator and the greatest improvement (98%) was made by the decimal mutation operator in generation 1. Figure 33 shows the population fitness improvement process in the worst run.

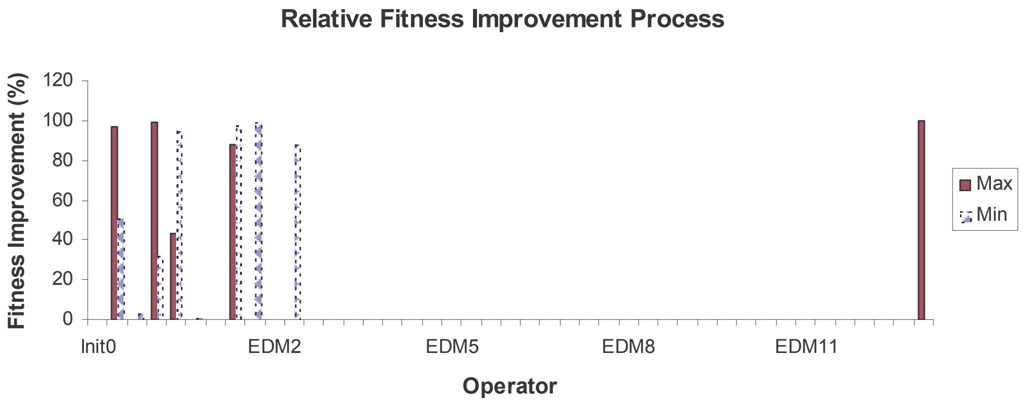

Figure 33. The relative fitness improvement made by each operator for the worst run for the De Jong function F1.

The smallest improvement to the population minimum fitness value was made by the elitist integer mutation operator and the greatest improvement (51%) was made by the decimal mutation operator in generation 1. The smallest improvement to the population maximum fitness value was made by the decimal mutation operator and the greatest improvement (99%) was made by the elitist decimal mutation operator in generation 1. Table 9 summarizes the survival rate of male and female chromosomes in the initialization and the crossover operations.

Table 9. Comparison of survival of male and female chromosomes.

Chromosome Sex	Survival Rate in Selection (%)	Survival Rate in Crossover (%)	
		Worst Run	Best Run
Male	50	-	-
Female	50	-	-

Male and female chromosomes seem to have survived equally during the initialization phase. Since the classical crossover operator was not used, there was no information on the superiority of male and female chromosomes during the crossover operation.

6.6 The De Jong Function F2

The De Jong function F2 is a minimization problem and its mathematical representation is the following:

$$100(x_1^2 - x_2)^2 + (1 - x_1)^2 \text{, where } -2.048 \le x_i \le 2.048 \text{, for } i = 1,2 . \qquad (49)$$

This function has a global minimum value of 0 at $(x_1, x_2) = (1,1)$. This function has the reputation of being a difficult minimization problem (Storn et al. 1995). Operators used for solving this problem were: the decimal mutation operator and the elitist decimal mutation operator. In addition, the population refreshing procedure at the end of each generation was called. Table 10 gives a summary of these test runs.

Table 10. Summary of test runs for the De Jong function F2.

Summary	Generations	Evaluations to approx. optimal	Time to approx. optimal (s)	Total Evaluations	Total Elapsed Time (s)
Best Run	3	111	0.00	148	0.00
Worst Run	33	150	0.00	1417	0.00
Mean	9.28	249.70	0.00	410.56	0.04
Median	8.50	190.50	0.00	382.00	0.00
Mode	3.00	189.00	0.00	149.00	0.00
Std. Dev.	5.80	148.30	0.00	242.01	0.20

The algorithm solved the problem on average in 9 generations and 411 evaluations in less than 1 second. If the effect of extreme cases is eliminated, the algorithm solved the problem on average in 9 generations and 382 evaluations in less than 1 second. In the best run, the algorithm solved the problem in 3 generations and 148 evaluations in less than 1 second. In the worst run, solving the problem took 33 generations, 1417 evaluations and less than 1 second. Most typically, the algorithm solved the problem in 3 generations and 149 evaluations in less than 1 second. Table 11 summarizes the most important descriptive statistics of the fitness populations of the best run and the worst run.

Table 11. Descriptive statistics of the population fitness values for the test runs on the De Jong function F2.

Runs	Pops	Min	Max	Mean	Std. Dev.	Skew.	Kurt.
Best Run	**Worst Pop.**	-2679.18	-3.33	-500.06	742.00	-2.00	3.00
	Best Pop.	-0.01	0.00	0.00	0.00	0.00	-3.00
Worst Run	**Worst Pop.**	-1276.00	-23.42	-407.49	419.00	0.00	-1.00
	Best Pop.	-2.34	0.00	-0.43	0.00	-1.00	-1.00

The function has been multiplied by -1 to adapt it to DDBGA. The minimum and the mean fitness values in the initial population of the worst run were better than the respective values in the best run, but the maximum fitness value in the initial population of the worst run was almost eight times worse than the respective value in the best run. The fitness value deviation in the initial population of the best run was also twice greater than the respective value in the worst run. The statistics of the final populations indicate that the algorithm has improved the population members more in the best run. Figure 34 illustrates the role of each operator in improving the population fitness value in the best run.

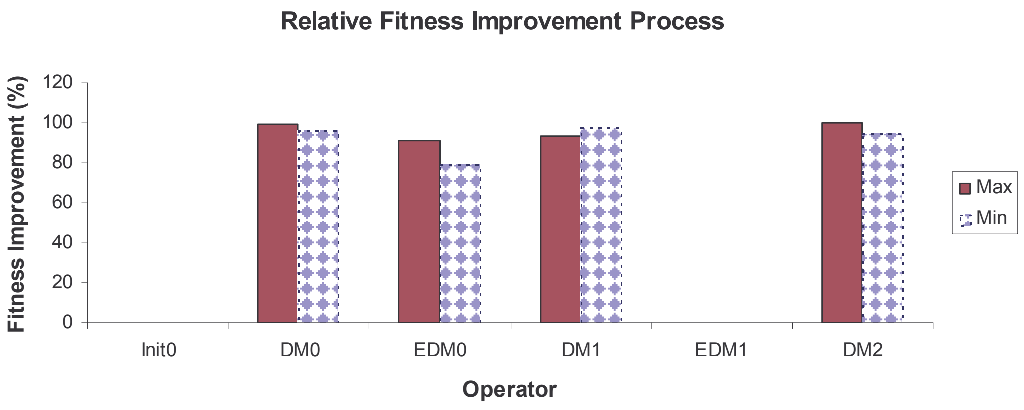

Figure 34. The relative fitness improvement made by each operator in the best run for the De Jong function F2.

The smallest improvement to the population minimum fitness value was made by the elitist decimal mutation operator and the greatest improvement (96%) was made by the decimal mutation operator in generation 1. The smallest improvement to the population maximum fitness value was made by the elitist decimal mutation operator and the greatest improvement (99%) was made by the decimal mutation operator in generation 1. Figure 35 depicts the improvement of the population fitness value during the worst run.

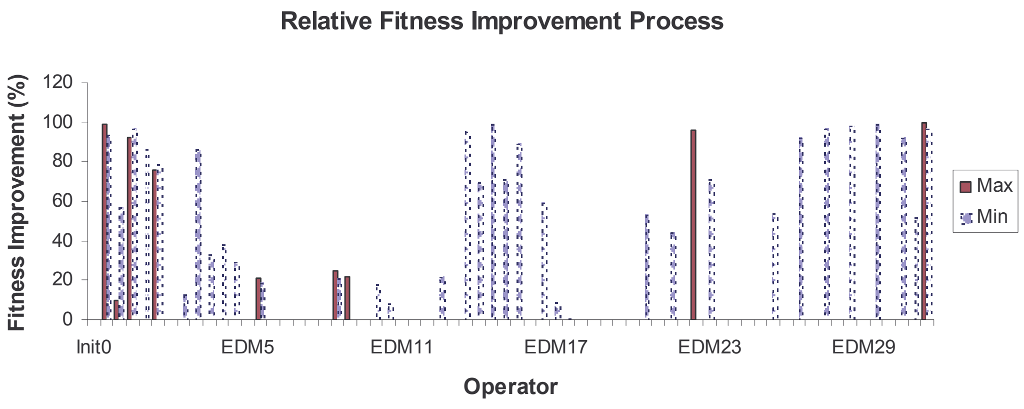

Figure 35. The relative fitness improvement made by each operator for the worst run for the De Jong function F2.

The smallest improvement to the population minimum fitness value was made by the elitist decimal mutation operator and the greatest improvement (94%) was made by the decimal mutation operator in generation 1. The smallest improvement in the population maximum fitness value was made by the elitist decimal mutation operator and the greatest improvement (99%) was made by the decimal mutation operator in generation 1. Table 12 summarizes the survival rates of male and female chromosomes.

Table 12. Comparison of survival of male and female chromosomes.

Chromosome Sex	Survival Rate in Selection (%)	Survival Rate in Crossover (%)	
		Worst Run	Best Run
Male	52	-	-
Female	47	-	-

The female chromosomes were practically as good as male chromosomes in the initialization phase. Since no classic crossover operator was used to solve this problem, there is no information about the productivity of male and female chromosomes during the crossover operations.

6.7 The De Jong Function F3

The De Jong function F3 is a minimization problem and its mathematical representation is the following:

$$\sum_{i=1}^{5} \text{integer}(x_i)\text{, where } -5.12 \le x_i \le 5.12\text{, for } i = 1,...,5\,. \qquad (50)$$

Here $\text{integer}(x_i)$ is the integer part of x_i, like $\text{integer}(-5.12) = -6$. This function has a global minimum value of -30 for all $-5.12 \le x_i \le -5.0$. This function exhibits many plateaus, which pose a considerable problem for many minimization algorithms (Storn et al. 1995). The following operators were used to solve this problem: the higher quartile variable crossover operator, the higher quartile variable replacement operator, the random building block operator, the integer mutation operator, the decimal mutation operator and the elitist decimal mutation operator. Also the population refreshing procedure at the end of each generation was called. Table 13 presents a summary of test runs.

Table 13. Summary of test runs for the De Jong function F3.

Summary	Generations	Evaluations to approx. optimal	Time to approx. optimal (s)	Total Evaluations	Total Elapsed Time (s)
Best Run	1.00	42.00	0.00	42.00	0.00
Worst Run	3.00	559.00	0.00	559.00	0.00
Mean	1.28	202.00	0.00	202.00	0.00
Median	1.00	196.00	0.00	196.00	0.00
Mode	1.00	43.00	0.00	43.00	0.00
Std. Dev.	0.54	134.24	0.00	134.24	0.00

The algorithm solved the problem on average in 1 generation, 202 evaluations and less than 1 second. In the best case, the algorithm needed 1 generation, 42 evaluations and less than 1 second to solve the problem. In the worst case, the algorithm needed 3 generations, 559 evaluations and less than 1 second to solve the problem. When the effect of outliers is ignored, the algorithm solved the problem on average in 1 generation, 196 evaluations and less than 1 second. Most typically, the algorithm solved the problem in 1 generation and 43 evaluations in less than 1 second. Table 14 summarizes the most important descriptive statistics of the initial and final populations in the best run and the worst run.

Table 14. Descriptive statistics of the population fitness values for the test runs on the De Jong function F3.

Runs	Pops	Min	Max	Mean	Std. Dev.	Skew.	Kurt.
Best Run	**Worst Pop.**	-42.00	-15.00	-31.75	8.00	0.00	-1.00
	Best Pop.	-38.00	0.00	-26.08	11.00	1.00	0.00
Worst Run	**Worst Pop.**	-47.00	-17.00	-32.33	10.00	0.00	-2.00
	Best Pop.	-6.00	0.00	-4.67	1.00	8.00	40.00

The function has been multiplied by -1 to adapt it to DDBGA. The descriptive statistics point out that the initial population in the best run had a better quality. However, the final population in the worst run seems to have been improved more and therefore become more homogeneous than the respective one in the worst run. Figure 36 illustrates the role of each operator in improving the population fitness value in the best run.

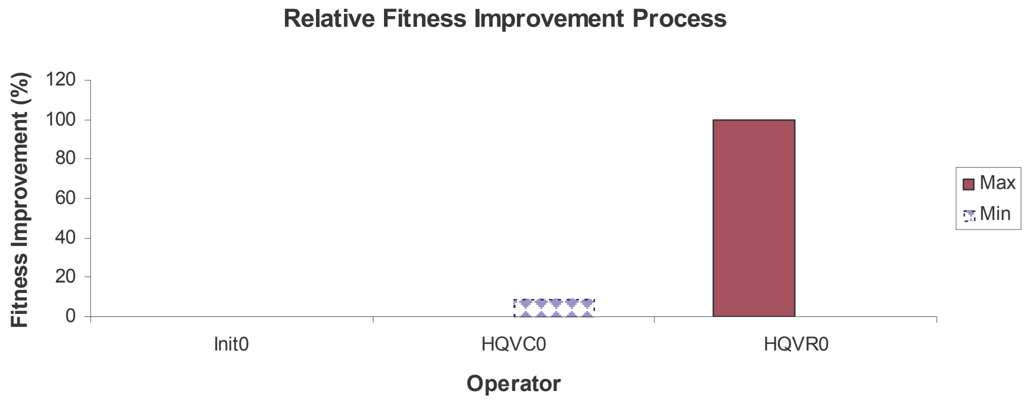

Figure 36. The relative fitness improvement made by each operator in the best run for the De Jong function F3.

The population minimum fitness value was improved 9% by the higher quartile variable crossover operator. The maximum fitness value was improved 100% by the higher quartile variable replacement operator in the only generation 1. Figure 37 illustrates the role of each operator in improving the population fitness value in the worst run.

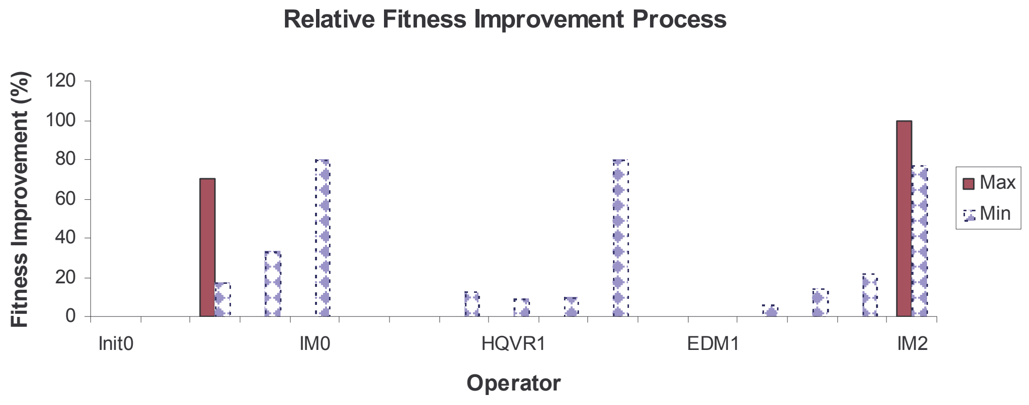

Figure 37. The relative fitness improvement made by each operator for the worst run for the De Jong function F3.

The smallest improvement to the population minimum fitness value was made by the higher quartile variable replacement operator and the greatest improvement (80%) was made by the integer mutation operator in generation 1. The population maximum fitness value was finally improved 100% by the integer mutation operator in generation 3. Table 15 summarizes the survival rates of male and female chromosomes during the initialization and crossover operations.

Table 15. Comparison of survival of male and female chromosomes.

Chromosome Sex	**Survival Rate in Selection (%)**	**Survival Rate in Crossover (%)**	
		Worst Run	**Best Run**
Male	50	-	-
Female	49	-	-

Male and female chromosomes survived equally during the initialization phase. However, since the classic crossover operator was not used to solve this problem, there is no information about the survival rates of chromosomes of different sex during the crossover operations.

6.8 The De Jong Function F4

The De Jong function F4 is a minimization problem and its mathematical representation is the following:

$$\sum_{i=1}^{30} ix_i^4 + N(0,1)\text{, where } -1.28 \le x_i \le 1.28\text{, for } i = 1,\ldots,30. \qquad (51)$$

$N(0,1)$ is a random variable produced by Gaussian noise having mean value 0 and standard deviation 1. This function without Gaussian noise has a global minimum value of 0 at $(x_1, x_2,\ldots, x_{30}) = (0,0,\ldots,0)$. This function with noise is designed to test the behavior of a minimization algorithm in the presence of noise. This function appears to be flawed as no definite global minimum exists (Ingber & Rosen 1992; Storn et al. 1995). The algorithm performance was tested on this function with and without noise. All other operators except for the integer mutation operator were used to solve this problem without noise. There would be no need for the integer mutation operator due to the short valid range of variables. The population refreshing procedure was not used too. Table 16 summarizes the statistics of test runs for the function without noise.

Table 16. Summary of test runs for the De Jong function F4 without noise.

Summary	Generations	Evaluations to approx. optimal	Time to approx. optimal (s)	Total Evaluations	Total Elapsed Time (s)
Best Run	1	55	0.00	55	0.00
Worst Run	1	518	0.00	518	0.00
Mean	1.00	149.92	0.00	352.02	0.00
Median	1.00	141.00	0.00	414.00	0.00
Mode	1.00	141.00	0.00	61.00	0.00
Std. Dev.	0.00	94.67	0.00	159.56	0.00

The algorithm solved the problem without noise on average in 1 generation and 352 evaluations in less than 1 second. The optimal value was found in the best case in 1 generation only after 55 evaluations in less than 1 second. In the worst case, the global optimum was found in 1 generation and 518 evaluations in less than 1 second. When the effect of extreme cases is eliminated, the optimal value was found on average in 1 generation, 414 evaluations and less than 1 second. Most typically, the algorithm solved the problem in 1 generation and 61 evaluations in less than 1 second.

The algorithm performance was also tested on the noisy version of the function. The fitness value for the noisy function was calculated by first calculating the sum of the first part and then adding a randomly generated Gaussian number to the sum. The desired fitness value precision was set to ± 0.001 and the maximum number of function evaluations was set to 1735. The average of achieved optimal values over 50 runs was 0.00006 ± 0.001596 . Table 17 summarizes test statistics.

Table 17. Summary of test runs for the De Jong function F4 with noise.

Summary	Generations	Evaluations to approx. optimal	Time to approx. optimal (s)	Total Evaluations	Total Elapsed Time (s)
Best Run	1	512	0.00	512	0.00
Worst Run	3	525	0.00	1735	1.00
Mean	2.78	499.92	0.02	1581.68	0.90
Median	3.00	525.00	0.00	1731.00	1.00
Mode	3.00	525.00	0.00	1731.00	1.00
Std. Dev.	0.46	160.99	0.14	304.41	0.51

The algorithm solved the problem on average in 3 generations and 1582 evaluations in 1 second. The optimal value was found in the best case in 1 generation after 512 evaluations in less than 1 second. When the effect of extreme cases is eliminated, the optimal value was found on average in 3 generations and 1731 evaluations in 1 second. Most typically, the algorithm solved the problem in 3 generations and 1731 evaluations in 1 second. Table 18 summarizes the most important descriptive statistics of the test runs.

Table 18. Descriptive statistics of the population fitness values for the test runs on the De Jong function F4 without noise.

Runs	Pops	Min	Max	Mean	Std. Dev.	Skew.	Kurt.
Best Run	**Worst Pop.**	-333.96	-130.95	-243.90	59.00	0.00	-1.00
	Best Pop.	-104.35	0.00	-17.99	37.00	-1.00	1.00
Worst Run	**Worst Pop.**	-464.86	-146.01	-233.31	93.00	-1.00	1.00
	Best Pop.	-0.02	0.00	0.00	0.00	0.00	-3.00

The function has been multiplied by -1 to adapt it to DDBGA. The mean fitness value in the initial population of the best run was worse than the respective value in the worst run. However, in the worst run the algorithm seems to have improved all individuals so that the fitness population deviation in the final population is much smaller than the respective value in the best run and is virtually 0. The role of each operator in improving the population fitness values during the best run is depicted in Figure 38.

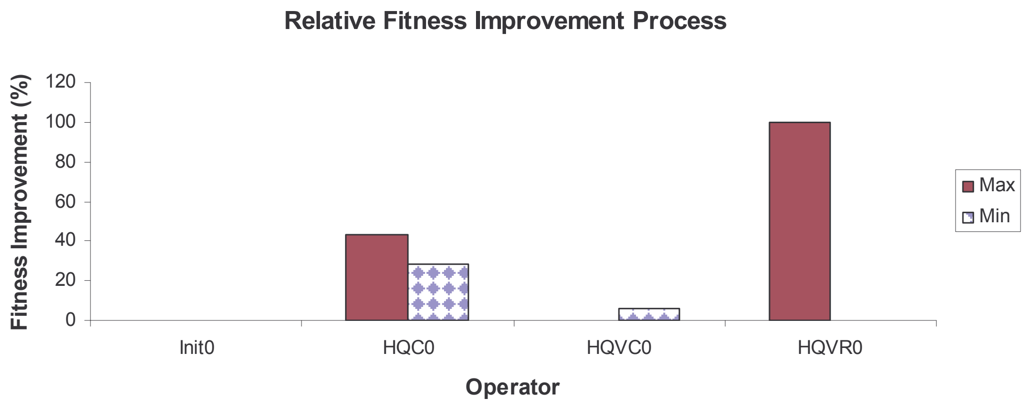

Figure 38. The relative fitness improvement made by each operator in the best run for the De Jong function F4 without noise.

The greatest improvement (28%) to the population minimum fitness value was made by the higher quartile crossover operator in the only generation 1. The greatest improvement (100%) to the population maximum fitness value was made by the higher quartile variable replacement operator. Figure 39 illustrates the role of each operator in improving the population fitness value in the worst run.

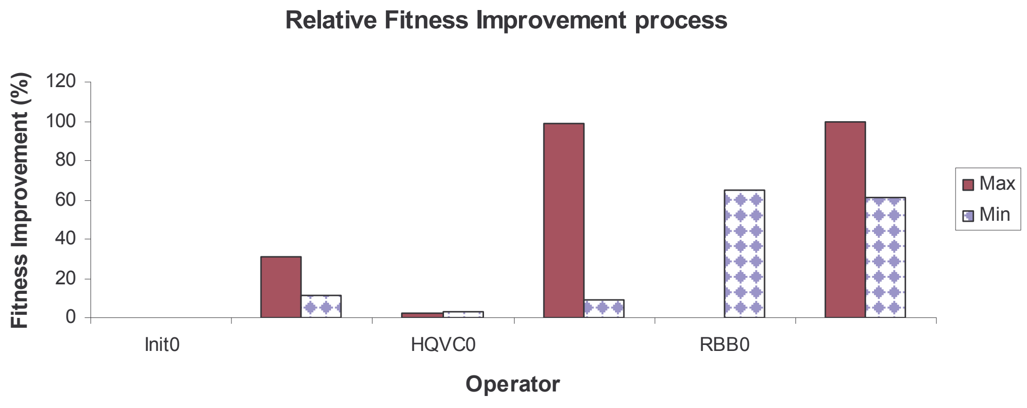

Figure 39. The relative fitness improvement made by each operator for the worst run for the De Jong function F4 without noise.

The smallest improvement to the population minimum fitness value was made by the higher quartile variable crossover operator and the greatest improvement (65%) was made by the random building block operator in the only generation 1. The smallest improvement to the population maximum fitness value was made by the higher quartile variable crossover operator and the greatest improvement (100%) was made by the decimal mutation operator. Table 19 summarizes the survival rate statistics of the male and female chromosomes.

Table 19. Comparison of survival of male and female chromosomes.

Chromosome Sex	**Survival Rate in Selection (%)**	**Survival Rate in Crossover (%)**	
		Worst Run	**Best Run**
Male	68	33	46
Female	31	66	53

Male chromosomes seem to have survived in more than twice as many cases as female chromosomes during the initialization phase. However, female chromosomes have produced better offspring. The superiority of female chromosomes in producing better offspring is more obvious during the worst run.

6.9 The De Jong Function F5

The De Jong function F5 is a minimization function and its mathematical representation is the following:

$$\frac{1}{1/k + \sum_{j=1}^{25} 1/f_j(x_1,x_2)}, \tag{52}$$

where $-65.536 \le x_i \le 65.536$, for $i = 1,2$, $k = 500$, $f_j(x_1,x_2) = j + \sum_{i=1}^{2}(x_i - a_{ij})^6$ and

$$a_{ij} = \begin{bmatrix} -32 & -16 & 0 & 16 & 32 & -32 & -16 & \ldots & 0 & 16 & 32 \\ -32 & -32 & -32 & -32 & -32 & -16 & -16 & \ldots & 32 & 32 & 32 \end{bmatrix}$$

This function has a global minimum value of 0.998004 at $(x_1, x_2) = (-32, -32)$. All operators and the population refreshing procedure were used to solve this problem. Table 20 summarizes the test runs statistics.

Table 20. Summary of test runs for the De Jong function F5.

Summary	Generations	Evaluations to approx. optimal	Time to approx. optimal (s)	Total Evaluations	Total Elapsed Time (s)
Best Run	1	97	0.00	97	0.00
Worst Run	9	1624	1.00	1812	1.00
Mean	2.58	372.22	0.14	469.86	0.14
Median	2.00	280.50	0.00	341.00	0.00
Mode	2.00	99.00	0.00	323.00	0.00
Std. Dev.	1.69	342.99	0.40	353.93	0.40

The algorithm solved the problem on average in 3 generations and 470 evaluations in 0.14 seconds. In the best case, it took 1 generation, 97 evaluations and less than 1 second to solve the problem. In the worst case, the algorithm solved the problem in 9 generations, 1812 evaluations and 1 second. When the effect of outliers is eliminated, the algorithm solved the problem on average in 2 generations, 341 evaluations and in less than 1 second. Most typically, the algorithm solved the problem in 2 generations, 323 evaluations and in less than 1 second. Table 21 gives a summary of the most important descriptive statistics of the initial and the final populations in the best and the worst run.

Table 21. Descriptive statistics of the population fitness values for the test runs on the De Jong function F5.

Runs	Pops	Min	Max	Mean	Std. Dev.	Skew.	Kurt.
Best Run	**Worst Pop.**	-500.00	-181.39	-457.83	95.00	2.00	3.00
	Best Pop.	-167.44	-0.998	-50.23	60.00	-1.00	-1.00
Worst Run	**Worst Pop.**	-500.00	-407.90	-490.34	26.00	2.00	6.00
	Best Pop.	-0.998	-0.998	-0.998	0.00	0.00	-3.00

The function has been multiplied by -1 to adapt it to DDBGA. The minimum fitness values in the initial populations of both the best run and the worst run were equal.

However, the maximum fitness value in the initial population of the best run was significantly better than the respective value in the worst population. The mean fitness value in the initial population of the best run was also better than the respective value in the worst run. Moreover, fitness values were much more deviated in the initial population of the best run. Figure 40 illustrates the role of each operator in improving the population fitness values.

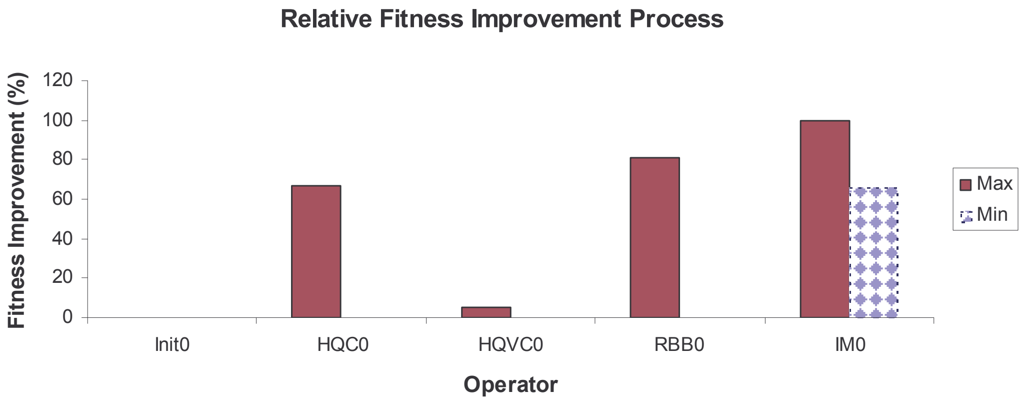

Figure 40. The relative fitness improvement made by each operator in the best run for the De Jong function F5.

The population minimum fitness value was improved 66% by the integer mutation operator in the only generation 1. The smallest improvement to the maximum fitness value was made by the higher quartile variable crossover operator and the greatest improvement (100%) was made by the integer mutation operator. Figure 41 depicts development of the population fitness values in the worst run.

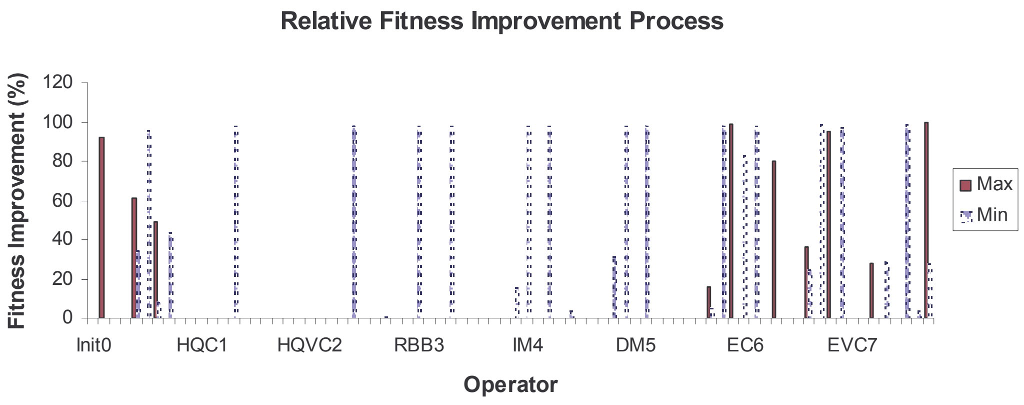

Figure 41. The relative fitness improvement made by each operator in the worst run for the De Jong function F5.

The smallest improvement to the population minimum fitness value was made by the elitist crossover operator and the greatest improvement (96%) was made by the decimal mutation operator in generation 1. The smallest improvement to the population maximum fitness value was made by the elitist crossover operator and the greatest improvement (92%) was made by the higher quartile crossover operator in generation 1. Table 22 summarizes the survival of male and female chromosomes in the selection and crossover operations.

Table 22. Comparison of survival of male and female chromosomes.

Chromosome Sex	**Survival Rate in Selection (%)**	**Survival Rate in Crossover (%)**	
		Worst Run	**Best Run**
Male	49	78	33
Female	50	21	66

The female chromosomes survived almost equally during the initialization. However, female chromosomes produced better offspring two times more than male chromosomes during the best run. But, male chromosomes produced better offspring almost four times more than female chromosomes during the worst run.

6.10 The Griewank Function F1

The Griewank function F1 is a minimization function and its mathematical presentation is the following:

$$\frac{1}{4000}\sum_{i=1}^{n}(x_i-100)^2-\prod_{i=1}^{n}\cos\left(\frac{x_i-100}{\sqrt{i}}\right)+1\text{, where } -600 \le x_i \le 600\text{.} \tag{53}$$

This function has a global minimum value of 0 at $x_i = 100$ for $i = 1,2,\ldots n$. This function has also many local minima; therefore it is very difficult to find the true minimum. When the number of variables was one, all operators except for the following ones were used: the higher quartile variable crossover operator, the higher quartile variable replacement and elitist variable crossover operator. However, for other cases all operators and the population refreshing procedure at the end of each generation were used. The summary of test runs results for the number of variables being 2 and 100 is reported in Table 23. For the summary of test runs results for other cases refer to Appendix A.

Table 23. Summary of test runs for the Griewank function with different number of variables (n).

n	Run Summary	Generations	Evaluations to approx. optimal	Time to approx. optimal(s)	Total Evaluations	Total elapsed Time(s)
2	Best Run	2	336	0	336	0
	Worst Run	35	5759	4	6587	5
	Mean	10.84	1714.74	1.08	2009.3	1.36
	Median	10	1435.5	1	1866	1
	Mode	3	362	0	557	0
	Std. Dev.	7.52	1342.07	1.24	1441.07	1.5
100	Best Run	1	191	0	191	0
	Worst Run	1	2787	6	2954	6
	Mean	1	2399.12	4.82	2435.54	4.88
	Median	1	2519	5	2519	5
	Mode	1	2599	5	2599	5
	Std. Dev.	0	444.3	1.04	414.4	0.98

Statistics drawn from 450 runs point out that the algorithm solved the problem on average in 5 generations, 1460 evaluations and 1 second. When the effect of outliers is eliminated, the statistics reveal that the algorithm solved the problem in 3 generations and 1065 generations in less than 1 second. Most typically, the algorithm solved the problem in 3 generations and less than 1 second. Table 24 summarizes the descriptive statistics for test runs in case the number of variables is 2 and 100. For the descriptive statistics for other cases, refer to Appendix A.

Table 24. Descriptive statistics of the population fitness values for the test runs on the Griewank function with different numbers of variables (n).

n	Runs	Pops	Min	Max	Mean	Std. Dev.	Skew.	Kurt.
2	Best Run	Worst Pop.	-143.25	-12.83	-81.44	44.00	0.00	-2.00
		Best Pop.	-0.96	0.00	-0.62	0.00	0.00	-3.00
	Worst Run	Worst Pop.	-189.57	-29.97	-80.34	53.00	-1.00	-1.00
		Best Pop.	0.00	0.00	0.00	0.00	0.00	-3.00
100	Best Run	Worst Pop.	-4098.27	-2516.00	-3223.00	405.00	0.00	0.00
		Best Pop.	-0.86	0.00	-0.20	0.00	0.00	-3.00
	Worst Run	Worst Pop.	-3995.68	-2909.00	-3434.00	370.00	0.00	-2.00
		Best Pop.	-3140.14	0.00	-1279.00	1579.00	0.00	-2.00

The function has been multiplied by -1 to adapt it to DDBGA. When the number of variables was 2, the minimum and maximum population fitness values were better in the worst population of the best run than the respective values in the worst population of the worst run. However, the mean fitness value was slightly worse in the worst population of the best run than the same value in the worst population of the worst run. Fitness variation in the worst population of the best run seems to have been less than that of the worst population of the worst run. During both runs the algorithm seems to have improved all individuals in the best populations.

When the number of variables was 100, the mean fitness value in the worst population of the best run was worse than the same value in the worst population of the worst run. Nevertheless, the fitness variation in the worst population of the best run was greater than the same value in the worst population of the worst run. It also seems that the algorithm has improved almost all individuals during the best run, but there is still a lot of fitness variation in the best population of the worst run.

The ability of operators to improve the quality of the fitness population was tracked. Below, the role of each operator in improving the minimum and maximum fitness value of the population for the best run and the worst run for the number of variables being 2 and 100 are presented graphically. Figure 42 illustrates development of the population fitness values in the best run when the number of variables was 2.

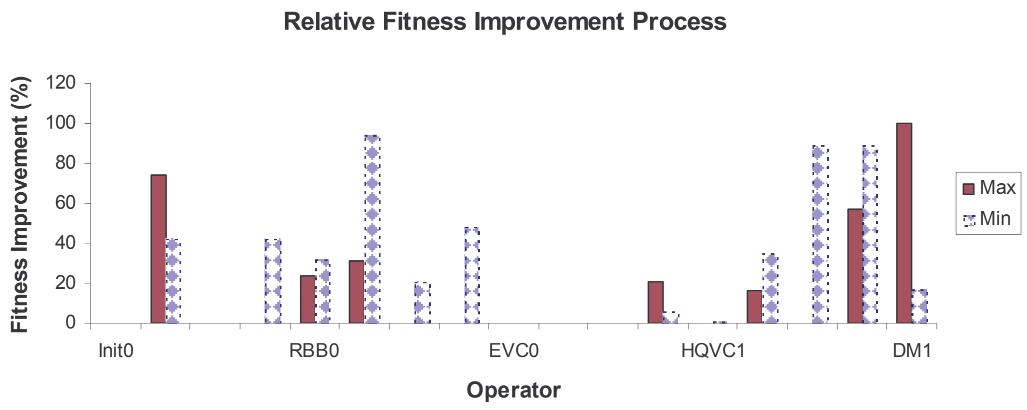

Figure 42. The relative fitness improvement made by each operator in the best run for the Griewank function with 2 variables.

The smallest improvement to the population minimum fitness value was made by the decimal mutation operator and the greatest improvement (94%) was made by the integer mutation operator in generation 1. The smallest improvement to the population maximum fitness value was made by the random building block operator and the greatest improvement (74%) was made by the higher quartile crossover operator in generation 1. Figure 43 illustrates the development of population fitness values in the worst run when the number of variables was 2.

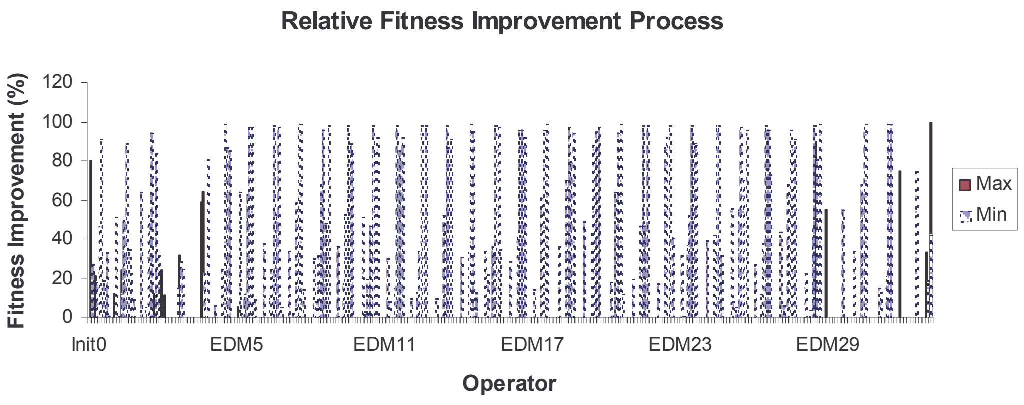

Figure 43. The relative fitness improvement made by each operator for the worst run for the Griewank function with 2 variables.

The smallest improvement to the population minimum fitness value was made by the elitist variable crossover operator and the greatest improvement (91%) was made by the integer mutation operator in generation 1. The smallest improvement to the population maximum fitness value was made by the elitist crossover operator and the greatest improvement (90%) was made by the higher quartile crossover operator in generation 1. Figure 44 illustrates the development of the population fitness values in the best run when the number of variables was 100.

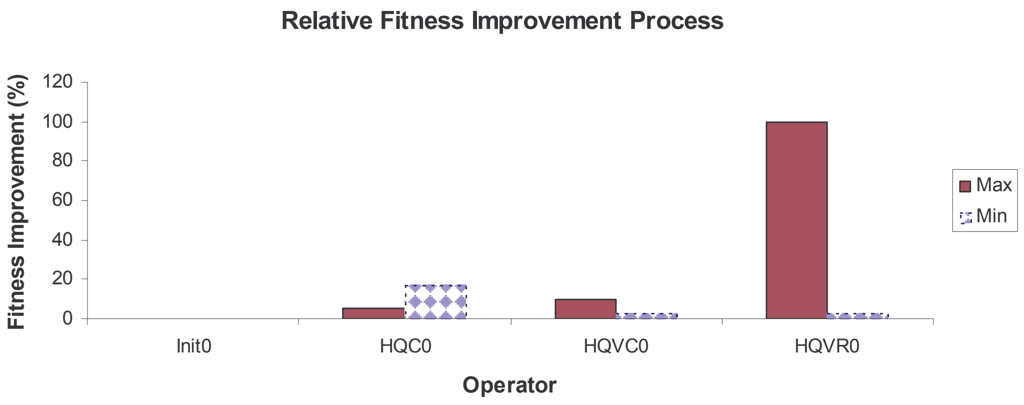

Figure 44. The relative fitness improvement made by each operator in the best run for the Griewank function with 100 variables.

The smallest improvement to the population minimum fitness value was made by the higher quartile variable crossover operator and the greatest improvement (17%) was made by the higher quartile crossover operator. The smallest improvement to the population maximum fitness value was made by the higher quartile crossover operator and the greatest improvement (100%) was made by the higher quartile variable replacement operator. Figure 45 illustrates the role of each operator in improving the population fitness value in the worst run when the number of variables was 100.

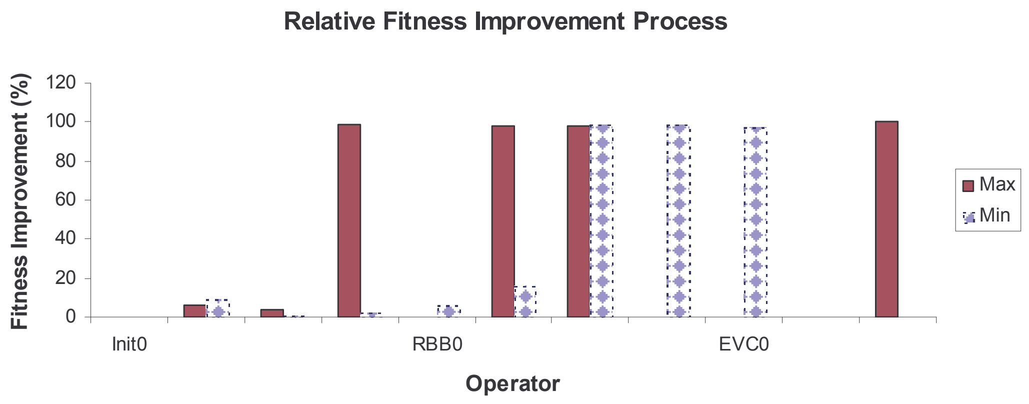

Figure 45. The relative fitness improvement made by each operator for the worst run for the Griewank function with 100 variables.

The smallest improvement to the population minimum fitness value was made by the higher quartile variable crossover operator and the greatest improvement (99%) was made by the decimal mutation operator in the only generation 1. The smallest improvement to the population maximum fitness value was made by the higher quartile variable crossover operator and the greatest improvement (100%) was made by the elitist decimal mutation operator in generation 1. Table 25 summarizes the survival rates of the male and female chromosomes.

Table 25. Comparison of survival of male and female chromosomes.

Variables	Chromosome Sex	Survival Rate in Selection (%)	Survival Rate in Crossover (%)	
			Worst Run	Best Run
2	Male	50	92	100
	Female	49	7	0
100	Male	0	81	0
	Female	100	18	100

When the number of variables was 2, the male and female chromosomes survived equally in the initialization phase. However, the male chromosomes produced better offspring in the crossover operations in almost all cases both during the worst and the best run.

When the number of variables was 100, the female chromosomes survived overwhelmingly better than the male chromosomes in the initialization phase. Also in the best run female chromosomes produced better offspring in all cases in the crossover operations. However, the situation was almost opposite in the worst run when the ratio of producing better offspring was 81% over 18% in favor of male chromosomes.

6.11 The Rosenbrock Function F1

The Rosenbrock function F1 is a multimodal minimization function and its mathematical presentation is the following:

$$\sum_{i=1}^{n-1}(100(x_{i+1} - x_i^{\,2})^2 + (x_i - 1)^2)\text{, where } -100 \le x_i \le 100 . \tag{54}$$

This function has a global minimum value of 0 at $x_i = 1.0$ for $i = 1,2,...,n$. All operators were used to solve this problem. The population refreshing procedure was not used. The summary of test results for the number of variables being 2 and 100 is reported in Table 26. See Appendix A for the summary of test results for other cases.

Table 26. Summary of test runs for the Rosenbrock function with different number of variables (n).

n	Summary	Generations	Evaluation to approx. optimal	Time to approx. optimal (s)	Total Evaluations	Total Elapsed Time (s)
2	**Best Run**	2.00	181.00	0.00	250.00	0.00
	Worst Run	14.00	2672.00	1.00	2672.00	1.00
	Mean	6.20	957.58	0.40	1130.94	0.46
	Median	6.00	734.50	0.00	1079.50	0.00
	Mode	4.00	1480.00	0.00	1480.00	0.00
	Std. Dev.	3.33	626.32	0.70	647.30	0.71
100	**Best Run**	1.00	75.00	0.00	75.00	0.00
	Worst Run	1.00	2963.00	7.00	2963.00	7.00
	Mean	1.00	2325.68	4.24	2325.68	4.24
	Mean	1.00	2325.68	4.24	2325.68	4.24
	Median	1.00	2420.00	4.00	2420.00	4.00
	Mode	1.00	2486.00	4.00	2486.00	4.00
	Std. Dev.	0.00	414.44	1.02	414.44	1.02

Statistics drawn from 400 test runs reveal that on average the algorithm solved the problem in 2 generations, 881 evaluations and 0.7 second. When the effect of extreme cases is eliminated, the median of run statistics point out that the algorithm solved the problem in 2 generations, 619 evaluations in less than 1 second. No typical run could be distinguished from the statistics. Table 27 presents the summary of descriptive statistics for test runs in which the number of variables is 2 and 100. See Appendix A for descriptive statistics of other test cases.

Table 27. Descriptive statistics of the population fitness values for the test runs on Rosenbrock function with different numbers of variables (n).

n	Runs	Pops	Min	Max	Mean	Std. Dev.	Skew.
2	Best Run	Worst Pop.	-2.90E+07	-6.07E+01	-3.37E+06	8.58E+06	-2
		Best Pop.	-4.80E-05	0.00E+00	-3.90E-05	0.00E+00	0
	Worst Run	Worst Pop.	-1.20E+07	-6.90E+01	-1.19E+06	3.53E+06	-2
		Best Pop.	-3.00E-06	0.00E+00	-2.00E-06	0.00E+00	0
100	Best Run	Worst Pop.	-1.60E+11	-1.30E+11	-1.47E+11	1.01E+10	0
		Best Pop.	-8.28E+01	0.00E+00	-4.17E+01	4.20E+01	0
	Worst Run	Worst Pop.	-1.80E+11	-1.00E+11	-1.49E+11	2.45E+10	0
		Best Pop.	-1.24E-02	0.00E+00	-6.03E-03	0.00E+00	0

The function has been multiplied by -1 to adapt it to DDBGA. The algorithm seems to have improved the population members more during the worst runs so that the minimum and the mean fitness values in the final populations of the worst runs are better than the respective values in the final populations of the best runs. Figure 46 depicts the role of each operator in improving the population fitness value for the best run in case the number of variables was 2.

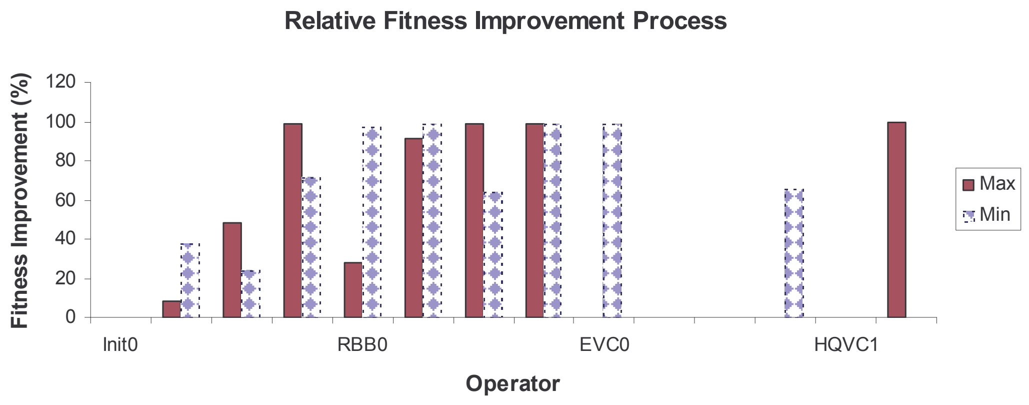

Figure 46. The relative fitness improvement made by each operator in the best run for the Rosenbrock function with 2 variables.

The smallest improvement to the population minimum fitness value was made by the higher quartile variable crossover operator and the greatest improvement (99%) was made by the elitist variable crossover operator in generation 1. The smallest improvement to the population maximum fitness value was made by the higher quartile crossover operator and the greatest improvement (99%) was made by the elitist crossover operator in generation 1. Figure 47 depicts the role of each operator in improving the population fitness value for the worst run in case the number of variables was 2.

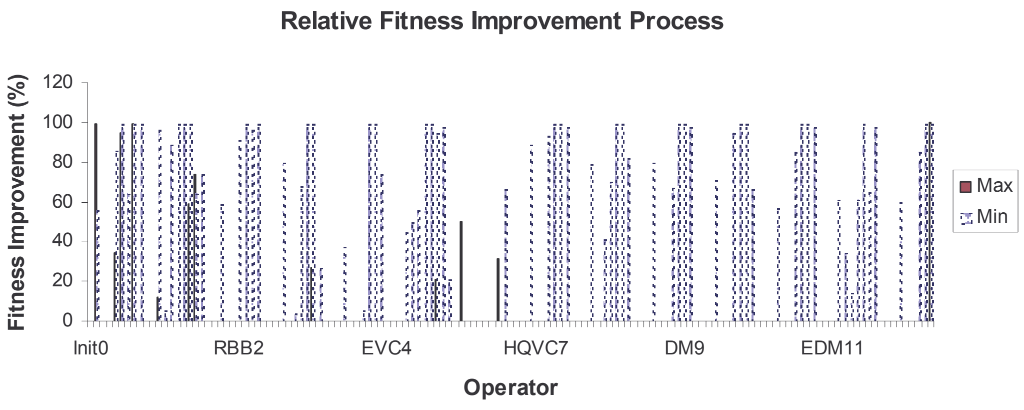

Figure 47. The relative fitness improvement made by each operator for the worst run for the Rosenbrock function with 2 variables.

The smallest improvement to the population minimum fitness value was made by the higher quartile crossover operator and the greatest improvement (99%) was made by the elitist variable crossover operator in generation 1. The smallest improvement to the population maximum fitness value was made by the random building block operator and the greatest improvement (99%) was made by the elitist crossover operator in generation 1. Figure 48 illustrates the role of each operator in improving the population fitness value for the best run in case the number of variables was 100.

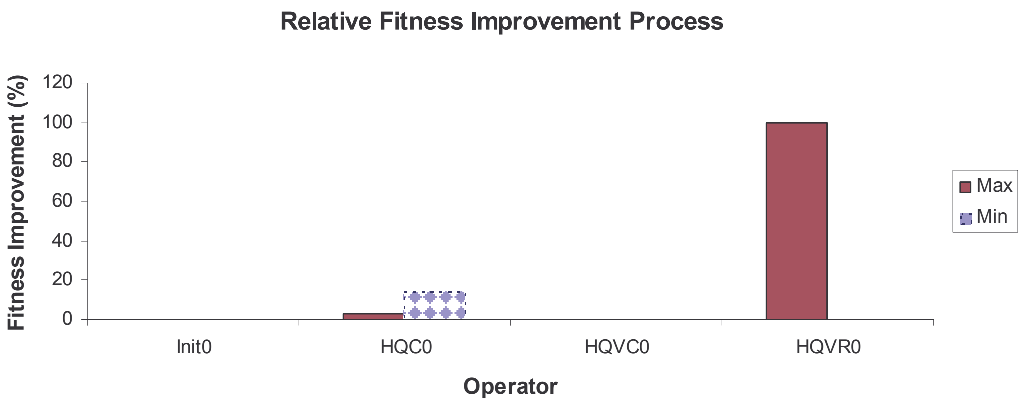

Figure 48. The relative fitness improvement made by each operator in the best run for the Rosenbrock function with 100 variables.

The population minimum fitness value was improved 14% by the higher quartile crossover operator in the only generation 1. The greatest improvement (100%) to the population maximum fitness value was made by the higher quartile variable replacement operator. Figure 49 illustrates the role of each operator in improving the population fitness value for the worst run in case the number of variables was 100.

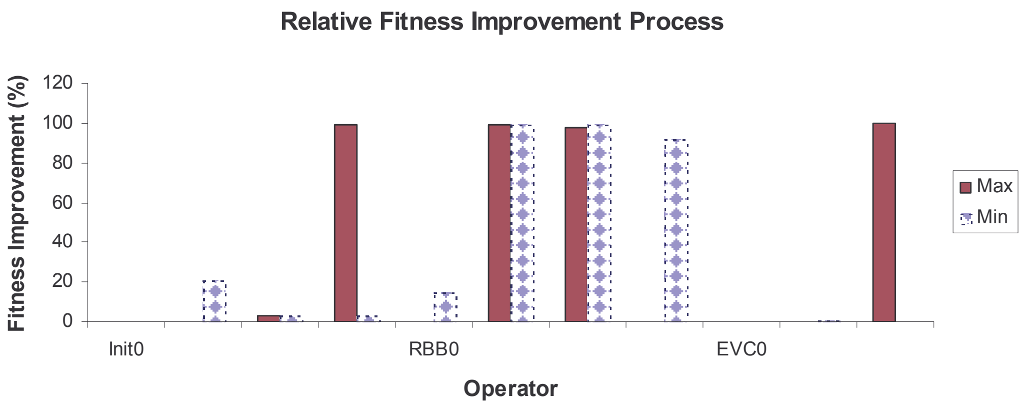

Figure 49. The relative fitness improvement made by each operator for the worst run for the Rosenbrock function with 100 variables.

The smallest improvement to the population minimum fitness value was made by the elitist integer mutation operator and the greatest improvement (99%) was made by the decimal mutation operator in generation 1. The smallest improvement to the population maximum fitness value was made by the higher quartile variable crossover operator and the greatest improvement (100%) was made by the elitist decimal mutation operator in generation 1. Table 28 summarizes the survival rates of male and female chromosomes.

Table 28. Comparison of survival of male and female chromosomes.

Variables	Chromosome Sex	Survival Rate in Selection (%)	Survival Rate in Crossover (%)	
			Worst Run	Best Run
2	Male	50	100	100
	Female	49	0	0
100	Male	50	100	100
	Female	50	0	0

The statistics drawn from 50 runs reveals that when the number of variables was 2, on average the male chromosomes were better in 50% of cases and female chromosomes were better in 49% of cases. In both the worst run and the best run, male chromosomes produced better offspring than female chromosomes in all cases during the crossover operation.

When the number of variables was 100, the male and female chromosomes survived equally during the selection. Also here in both the worst run and the best run, male chromosomes produced better offspring than female chromosomes in all cases during the crossover operation.

6.12 The Rastrigin Function F1

The Rastrigin function F1 is a multimodal minimization function and its mathematical presentation is the following:

$$\sum_{i=1}^{n}(x_i^2 - 10\cos(2\pi x_i) + 10) \quad \text{where} - 5.12 \leq \mathrm{x_i} \leq 5.12\,. \tag{55}$$

This function and has a global minimum value of 0 at $x_i = 0$, for $i = 1,2,...n$. All operators were used for solving this problem. However, the population refreshing procedure was not used. The following tables provide the summary of test runs for the number of variables being 2 and 100. For the summary of test runs for other cases, refer to Appendix A.

Table 29. Summary of test runs for the Rastrigin function F1 with different number of variables (n).

n	Summary	Generations	Evaluations to approx. optimal	Time to approx. optimal (s)	Total Evaluations	Total Elapsed Time (s)
2	**Best Run**	1	54	0	54	0
	Worst Run	8	1457	1	1457	1
	Mean	3.14	409.12	0.02	551.64	0.04
	Median	3	378.5	0	456	0
	Mode	2	374	0	303	0
	Std. Dev.	1.48	228.79	0.14	272.21	0.2
100	**Best Run**	1	1717	3	1717	3
	Worst Run	1	2922	6	2922	6
	Mean	1	2399.38	4.06	2399.38	4.06
	Median	1	2447	4	2447	4
	Mode	1	#N/A	4	#N/A	4
	Std. Dev.	0	218.5	0.62	218.5	0.62

Statistics drawn form 450 runs point out that the algorithm solved the problem on average in 2 generations and 893 evaluations in 0.62 second. When the effect of extreme cases is eliminated, the algorithm solved the problem in 2 generations and 499 evaluations in less than 1 second. Most typically, the algorithm solved the problem in 2 generations in less than 1 second. Table 30 gives a summary of the most relevant descriptive statistics of test runs in case the number of variables was 2 and 100. For the summary of descriptive statistics for other cases, refer to Appendix A.

Table 30. Descriptive statistics of the population fitness values for the test runs on Rastrigin function with different numbers of variables (n).

n	Runs	Pops	Min	Max	Mean	Std. Dev.	Skew.	Kurt.
2	Best Run	Worst Pop.	-67.30	-2.60	-14.58	17.0	-2.0	5.0
		Best Pop.	0.00	0.00	0.00	0.0	0.0	-3.0
	Worst Run	Worst Pop.	-50.69	-6.78	-19.87	13.0	-1.0	0.0
		Best Pop.	0.00	0.00	0.00	0.0	0.0	-3.0
100	Best Run	Worst Pop.	-1760.95	-1586.74	-1653.10	58.0	0.0	-2.0
		Best Pop.	-1.97	0.00	-1.15	1.0	0.0	-2.0
	Worst Run	Worst Pop.	-1758.26	-1614.97	-1676.31	52.0	0.0	-2.0
		Best Pop.	-0.01	0.00	0.00	0.0	0.0	-3.0

The function has been multiplied by -1 to adapt it to DDBGA. The interesting point observable from Table 30 is that as the number of variables increases, the distance between the fitness of the best individual in the population and the optimal fitness value increases. The fitness of the best individual in initial population in the best run in case the number of variables was 2, was –2.6, but the same value in case the number of variables was 100, was almost -1587. However, the algorithm needed only 1569 more generations and 3 more seconds to reach the optimal fitness value. Figure 50 illustrates the role of each operator in improving the population fitness value of the best run in case the number of variables was 2.

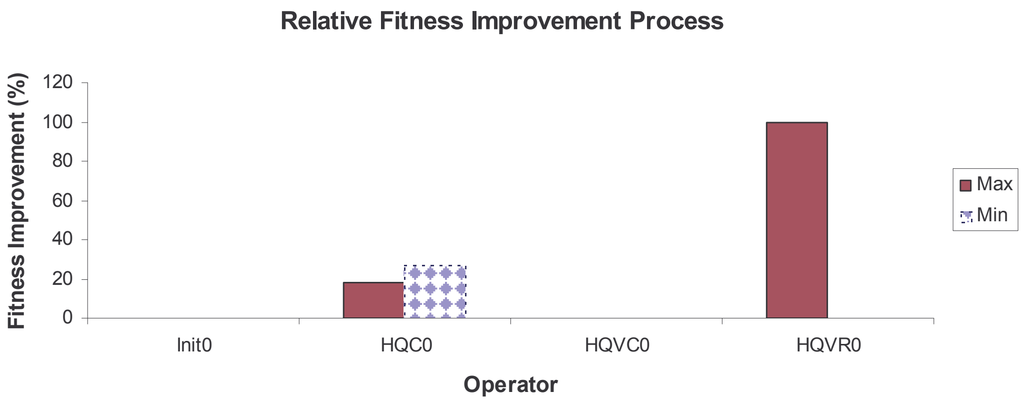

Figure 50. The relative fitness improvement made by each operator in the best run for the Rastrigin function with 2 variables.

The population minimum fitness value was improved 27% by the higher quartile crossover operator. The greatest improvement (100%) by the population maximum fitness value was made by the higher quartile variable replacement operator in the only generation 1. Figure 51 illustrates the role of each operator in improving the population fitness value of the worst run in case the number of variables was 2.

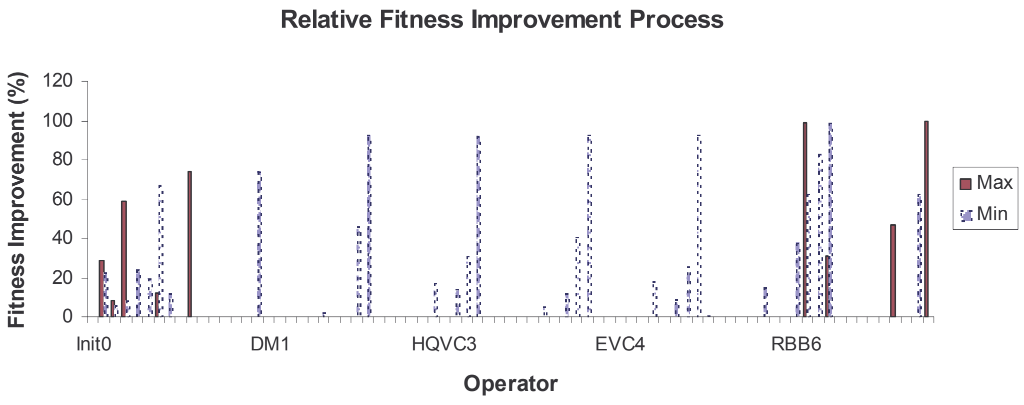

Figure 51. The relative fitness improvement made by each operator in the worst run for the Rastrigin function with 2 variables.

The smallest improvement to the population minimum fitness value was made by the higher quartile variable crossover operator and the greatest improvement (67%) was made by the decimal mutation operator in generation 1. The smallest improvement to the population maximum fitness value was made by the higher quartile variable crossover operator and the greatest improvement (74%) was made by the elitist integer mutation operator in generation 1. Figure 52 depicts the population fitness improvement process for the worst run when the number of variables was 100.

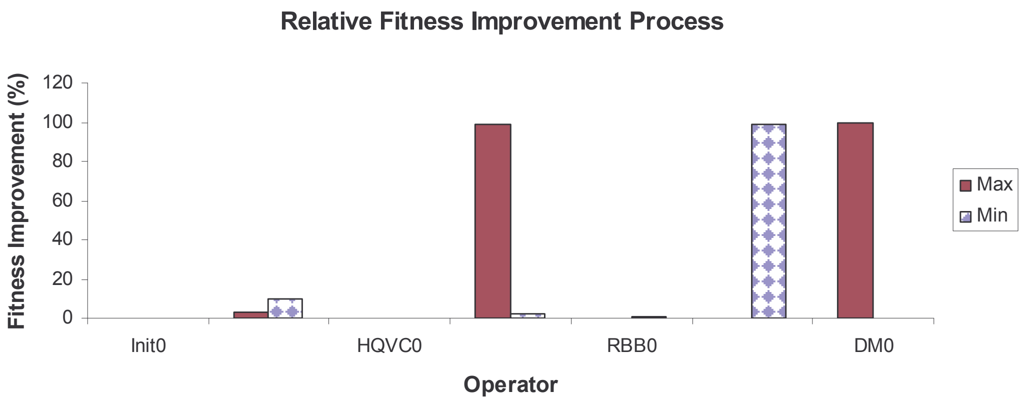

Figure 52. The relative fitness improvement made by each operator in the best run for the Rastrigin function with 100 variables.

The smallest improvement to the population minimum fitness value was made by the random building block operator and the greatest improvement (99%) was made by the integer mutation operator in the only generation 1. The smallest improvement to the population maximum fitness value was made by the higher quartile crossover operator and the greatest improvement (100%) was made by the decimal mutation operator. Figure 53 illustrates the population fitness improvement process for the worst run.

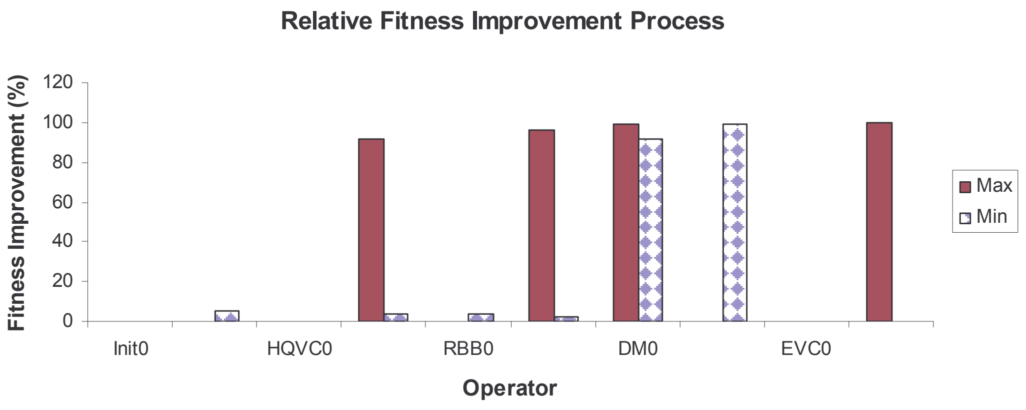

Figure 53. The relative fitness improvement made by each operator for the worst run for the Rastrigin function with 100 variables.

The smallest improvement to the population minimum fitness value was made by the integer mutation operator and the greatest improvement (99%) was made by the elitist crossover operator. The smallest improvement to the population maximum fitness value was made by the higher quartile variable replacement operator and the greatest improvement (100%) was made by the elitist integer mutation operator in the only generation 1. Table 31 summarizes the survival rates of male and female chromosomes during the population initialization and the crossover operations.

Table 31. Comparison of survival of male and female chromosomes.

Variables	Chromosome Sex	Survival Rate in Selection (%)	Survival Rate in Crossover (%)	
			Worst Run	Best Run
2	Male	77	80	92
	Female	22	20	8
100	Male	50	57	66
	Female	50	42	33

The statistics drawn from 50 runs revealed that on average the male chromosomes were better in slightly more than 77% of cases and female chromosomes were better in almost 22% of cases when the number of variables was 2.

When the number of variables was 2, in the best run during the crossover operator the male chromosomes produced better offspring in 92% of cases and female chromosomes produced better offspring in 8% of cases. In the worst run, during the crossover operator the male chromosomes produced better offspring in 80% of cases and female chromosomes produced better offspring in 20% of cases.

When the number of variables was 100, male and female chromosomes were equally good; in 50% of cases, male chromosomes were selected and in 50% of cases, female chromosomes were selected. In the worst run, male chromosomes produced better offspring in 57% of cases and female chromosomes produced better offspring in 42% of cases. In the best run, male chromosomes produced better offspring in 66% of cases and female chromosomes produced better offspring in 33% of cases.

6.13 The Schaffer Function F6

The Schaffer function F6 is a minimization function and its mathematical representation is the following:

$$0.5+\frac{\sin^2\sqrt{x_1^2+x_2^2}-0.5}{\left[1.0+0.001(x_1^2+x_2^2)\right]^2}\text{, where } -100 \le x_i \le 100\text{, for } i=1,2\,. \tag{56}$$

This function has a global minimum of 0 at $(x_1, x_2) = (0,0)$. The following operators were not used for solving the problem: the decimal mutation operator and the elitist decimal mutation operator. The population refreshing procedure was not used too. Table 32 summarizes the statistics of these test runs.

Table 32. Summary of test runs for the Schaffer function F6.

Summary	Generations	Evaluations to approx. optimal	Time to approx. optimal (s)	Total Evaluations	Total Elapsed Time (s)
Best Run	1	54	0	54	0
Worst Run	9	533	0	1033	0
Mean	3.02	231.7	0	345.9	0.06
Median	3	220	0	334.5	0
Mode	2	220	0	335	0
Std. Dev.	1.857033	116.0136	0	212.3081	0.239898

The algorithm solved the problem on average in 3 generations, 346 evaluations and 0.06 seconds. In the best case, solving the problem took 1 generation, 54 evaluations and less than 1 second. In the worst case, it took 9 generations, 1033 evaluations and less than 1 second to achieve the optimal value. Most typically, the algorithm solved the problem in 2 generations, 335 evaluations and less than 1 second. Table 33 gives a summary of the most important descriptive statistics of the initial and the final populations in the best and the worst run.

Table 33. Descriptive statistics of the population fitness values for the test runs on the Schaffer function F6.

Runs	Pops	Min	Max	Mean	Std. Dev.	Skew.	Kurt.
Best Run	Worst Pop.	-0.51	-0.49	-0.50	0.00	0.00	-3.00
	Best Pop.	-0.50	0.00	-0.45	0.00	0.00	-3.00
Worst Run	Worst Pop.	-0.58	-0.44	-0.50	0.00	0.00	-3.00
	Best Pop.	-0.01	0.00	-0.01	0.00	0.00	-3.00

The function has been multiplied by -1 to adapt it to DDBGA. The descriptive statistics of the population fitness values seem to have been quite similar in both the best run and the worst run. However, the algorithm seems to have had time to improve the individuals in the population more during the worst run. Although the population minimum fitness value in the initial populations of both the best run and the worst run was almost equal, in the worst run the algorithm seem to have improved the minimum fitness value quite close to the optimal value. Figure 54 shows the role of each operator in improving the population fitness value in the best run.

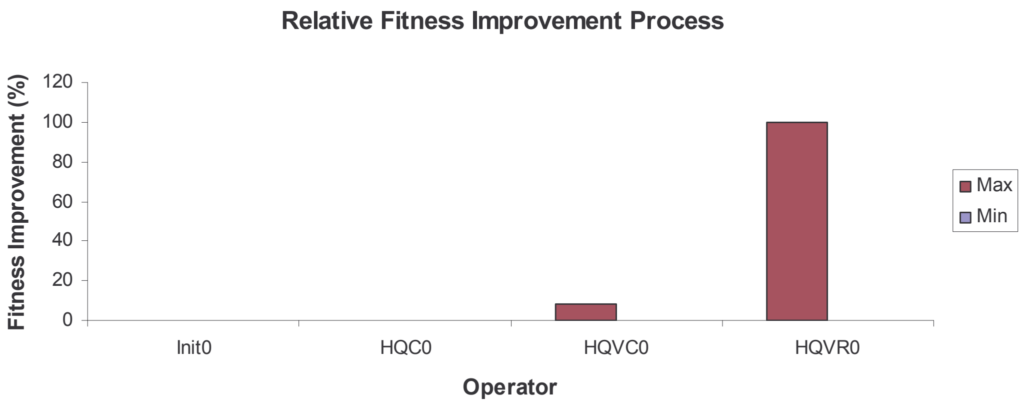

Figure 54. The relative fitness improvement made by each operator in the best run for the Schaffer function F6.

The population minimum fitness value was not improved at all. The greatest improvement (100%) to the population maximum fitness value was made by the higher quartile variable replacement operator in the only generation 1. Figure 55 depicts the relative fitness improvement in the worst run.

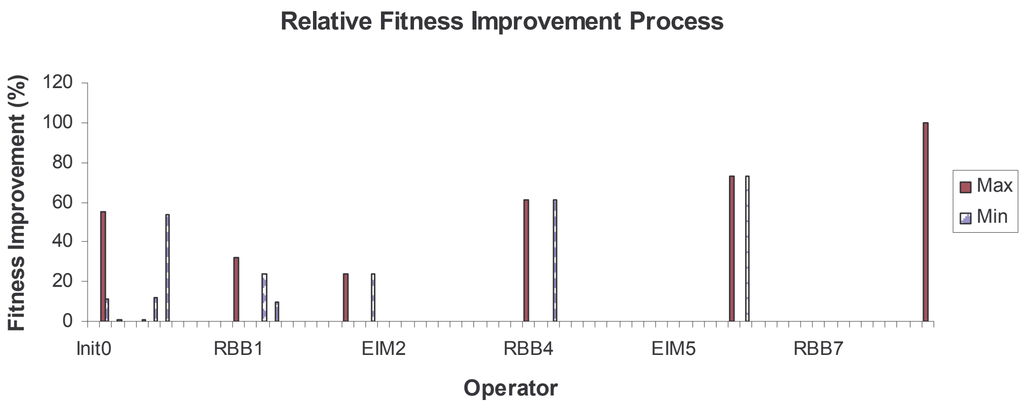

Figure 55. The relative fitness improvement made by each operator for the worst run for the Schaffer function F6.

The smallest improvement to the population minimum fitness value was made by the higher quartile variable crossover operator, and the greatest improvement (54%) was made by the elitist crossover operator in generation 1. Further improvement was made throughout the run. However, no more improvement was made to the population minimum fitness value during the last generation. The smallest improvement to the population maximum fitness value was made by the higher quartile crossover operator in generation 1. The final improvement (100%) was made by the integer mutation operator in the last generation. Table 34 summarizes the survival rates of male and female chromosomes during the initialization and crossover operations.

Table 34. Comparison of survival of male and female chromosomes.

Chromosome Sex	**Survival Rate in Selection (%)**	**Survival Rate in Crossover (%)**	
		Worst Run	**Best Run**
Male	50	77	66
Female	50	22	33

The male and female chromosomes seem to have survived equally in the selection phase. During the best run, male chromosomes produced better offspring in the crossover operation two times more. During the worst run, male chromosomes produced better offspring in 3.5 times more cases.

6.14 The Schaffer Function F7

The Schaffer function F7 is a minimization problem and its mathematical representation is the following:

$$(x_1^2 + x_2^2)^{0.25}\left[\sin^2(50(x_1^2 + x_2^2)^{0.1}) + 1.0\right] \text{, where } -100 \le x_i \le 100 \text{, for } i = 1,2 . \qquad (57)$$

This function has a global minimum value of 0 at $(x_1, x_2) = (0,0)$. The following operators were not used for solving the problem: the decimal mutation operator and the elitist decimal mutation operator. The population refreshing procedure was not used too. Table 35 gives a summary of the statistics of test runs.

Table 35. Summary of test runs for the Schaffer function F7.

Summary	Generations	Evaluations to approx. optimal	Time to approx. optimal (s)	Total Evaluations	Total Elapsed Time (s)
Best Run	1	54	0	54	0
Worst Run	4	421	0	421	0
Mean	1.84	206.8	0.02	206.8	0.02
Median	2	213.5	0	213.5	0
Mode	1	104	0	104	0
Std. Dev.	0.841767	96.90538	0.141421	96.90538	0.141421

The algorithm found the global optimal value in all cases. In the best case, it took 1 generation, 54 evaluations and less than 1 second to solve the problem. In the worst case, it took 4 generations, 421 evaluation and less than 1 second to solve the problem. On average, it took 2 seconds, 207 evaluations and less than 1 second to solve the problem. Most typically, the algorithm solved the problem in 1 generation, 104 evaluations and less than 1 second. Table 36 summarizes the most important descriptive statistics of the best run and the worst run.

Table 36. Descriptive statistics of the population fitness values for the test runs on the Schaffer function F7.

Runs	Pops	Min	Max	Mean	Std. Dev.	Skew.	Kurt.
Best Run	**Worst Pop.**	-16.475	-2.263	-11.323	3.0	2.0	5.0
	Best Pop.	-13.642	0.000	-8.571	4.0	1.0	0.0
Worst Run	**Worst Pop.**	-21.940	-7.432	-13.488	4.0	0.0	0.0
	Best Pop.	-1.228	0.000	-1.099	0.0	0.0	-3.0

The function has been multiplied by -1 to adapt it to DDBGA. The most important descriptive statistics of the fitness values of the initial population were better in the best run than the respective values in the worst run. However, the final population in the worst run seems to be much more homogenous than the respective one in the best run; the population members have been improved so that the mean fitness value in the last population is much better than the respective value in the best run. Figure 56 shows the role of each operator in improving the population fitness value in the best run.

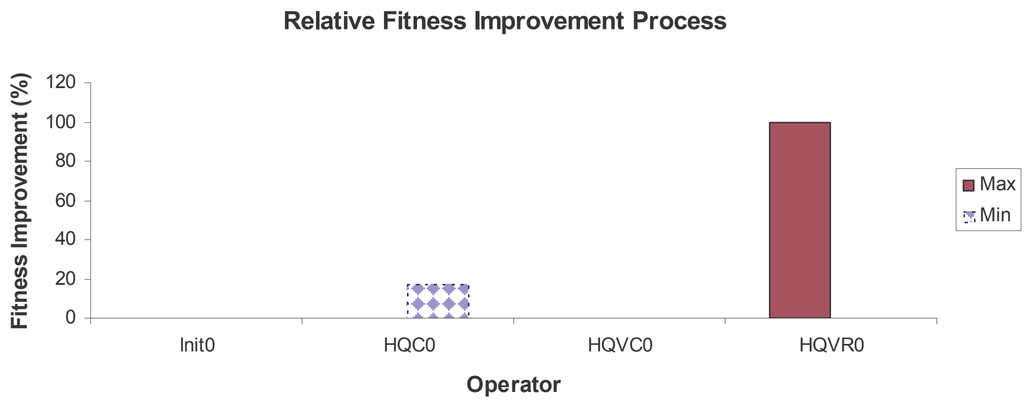

Figure 56. The relative fitness improvement made by each operator in the best run for the Schaffer function F7.

The population minimum fitness value was improved only by the higher quartile crossover operator in the only generation 1. The population maximum fitness value was improved only by the higher quartile variable replacement operator after which the optimal value was achieved. Figure 57 depicts the relative fitness improvement process for the worst run.

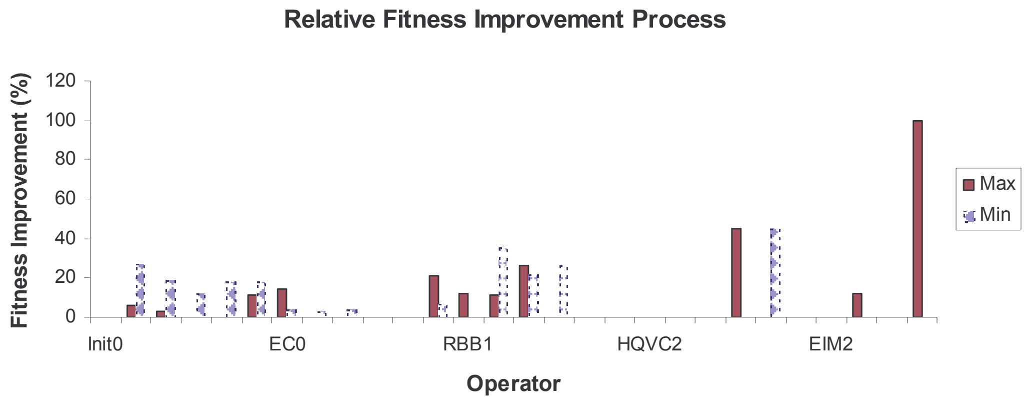

Figure 57. The relative fitness improvement made by each operator in the worst run for the Schaffer function F7.

The smallest improvement to the population minimum fitness value was made by the elitist variable crossover operator, and the greatest improvement (27%) was made by the higher quartile crossover operator. More improvement was made throughout the run. However, no more improvement was made to the population minimum fitness value during the last generation. The smallest improvement to the population maximum fitness value was made by the higher quartile variable crossover operator and the greatest improvement (14%) was made by the elitist crossover operator in generation 1. Table 37 summarizes the survival rates of male and female chromosomes.

Table 37. Comparison of survival of male and female chromosomes.

Chromosome Sex	**Survival Rate in Selection (%)**	**Survival Rate in Crossover (%)**	
		Worst Run	**Best Run**
Male	50	44	33
Female	50	55	66

Male and female chromosomes survived equally in the selection phase. However, female chromosomes seem to have produced better offspring during the crossover operation in the worst run as well as in the best run. Superiority of female chromosomes in producing better offspring seems to be clearer during the best run than the worst run.

7. Summary of Test Runs

The DDBGA algorithm was tested on 44 test problems in 2200 runs. Table 38 summarizes statistics drawn from the results of all runs.

Table 38. Summary statistics for 2200 runs.

Run Summary	Generations	Evaluations to approx. optimal	Time to approx. optimal(s)	Total Evaluations	Total elapsed Time(s)
Mean	3.36	886.99	0.69	1050.31	0.81
Std. Dev.	2.69	667.41	1.48	702.00	1.59
Median	2	706.75	0	809.50	0
Mode	1	#N/A	0	#N/A	0

The overall summary statistics for the runs indicate that the algorithm solved the problems on average in 3 generations and 1050 function evaluations in less than 1 second. When the effect of extreme cases is eliminated, the algorithm solved the problem on average in 2 generation, 810 function evaluations in less than 1 second. Most typically the algorithm solved the problem in 1 generation and less than 1 second.

However, finding the approximate optimal values took even less time and resources. The summary statistics point out that the algorithm solved the problems to their approximate optimal values on average in 887 evaluations in less than 1 second. When extreme cases are ignored, on average the algorithm solved the problems to their approximate optimal values in 707 function evaluations in less than 1 second.

The efficiency of the operators in finding better individuals and improving the population fitness values was also studied. The efficiency of operators in improving the minimum, maximum and mean fitness values of the population was analyzed. This analysis pointed out that operators can be put in slightly different orders for improving different fitness value statistics of the population. Table 39 summarizes the statistics. These statistics are the average values of the relative improvement achieved after calling each operator and are calculated based on the statistics reported for each test function in Chapter 6.

Table 39. Comparison of the efficiency of operators in improving the population fitness values. Operators are: integer mutation (IM), decimal mutation (DM), elitist crossover (EC), random building block (RBB), elitist variable crossover (EVC), higher quartile crossover (HQC), elitist decimal mutation (EDM), higher quartile variable replacement (HQVR), elitist integer mutation (EIM), higher quartile variable crossover (HQVC).

Operator	Average Improvement of Population Min. Fitness	Average Improvement of Population Max. Fitness	Average Improvement of Population Mean Fitness
IM	59.60	26.86	2183214396.09
DM	55.53	43.15	14848738.01
EC	47.08	10.51	138298.35
RBB	24.32	4.68	2081.63
EVC	11.23	0.02	792.67
HQC	14.85	7.63	45.17
EDM	8.67	12.75	43.23
HQVR	7.23	28.06	36.94
EIM	4.40	6.55	24.10
HQVC	3.75	2.44	9.43

The operators can be put into two distinct categories based on their efficiency in improving the population mean fitness value. The more efficient category includes: 1) integer mutation operator, 2) decimal mutation operator, 3) elitist crossover operator, 4) random building block operator, and 5) elitist variable crossover operator. The less efficient category includes the rest of operators.

The efficiency order of the operators in improving the population minimum fitness value is different: 1) integer mutation operator, 2) decimal mutation operator, 3) elitist crossover operator, 4) random building block operator, 5) higher quartile crossover operator, 6) elitist variable crossover operator, 7) elitist decimal mutation operator, 8) higher quartile variable replacement operator, 9) elitist integer mutation operator and 10) higher quartile variable crossover operator. The operators can be put into two distinct categories based on their efficiency in improving the population minimum fitness value: 1) the more efficient category includes operators 1-4, 2) the less efficient category includes operators 5-10.

The efficiency order of the operators in improving the population maximum fitness value is: 1) decimal mutation operator, 2) higher quartile variable replacement operator, 3) integer mutation operator, 4) elitist decimal mutation operator, 5) elitist crossover operator, 6) higher quartile crossover operator, 7) elitist integer mutation operator, 8) random building block operator, 9) higher quartile variable crossover operator, and 10) elitist variable crossover operator. Operators 1-3 form the category of more efficient operators while the rest of operators form the less efficient category.

The most efficient operators are the integer mutation and the decimal mutation operators, which have been able to improve the population fitness values the most. The least efficient operator seems to be the higher quartile variable crossover operator.

The usefulness of repeating the higher quartile crossover operator and elitist crossover operator on the same individual indexes and crossover points was also studied. The study revealed that this resulted into more successful crossover operations in around less than 1% of cases depending on how often the crossover operators were called. In a test run with 10-dimensional Griewank function, in 40 cases the operation was repeated with the same parent indexes and crossover points two times, in 18 cases the operation was repeated three times, in seven cases the operation was repeated four times and in one case the operation was repeated up to five times.

The survival of male and female chromosomes throughout test runs was also studied. The analysis revealed that male and female chromosomes survived equally during initialization. However, male chromosomes produced better offspring in 68% of cases while female chromosomes produced better offspring in around 32% of cases.

7.1 Comparison with Other Known Algorithms

The performance of the DDBGA was compared to the performance of some other well known algorithms, like classical GA, Differential Evolution (DE) (Storn et al. 1995), Adaptive Simulated Annealing (ASA) (Ingber 1993), Annealed Nedler & Mead approach (ANM) (Press, Teukolsky, Vetterling & Flannery 1992) and Particle Swarm Optimization (PSO) (Kennedy & Eberhart 1995). According to (Krink 2005) DE is the best default choice for numerical optimization of single-objective problems. The comparison test run results for other algorithms are based on the data presented in (Storn et al. 1995) and (Krink, Filipič & Fogel 2004; Krink 2005). In addition, the GA Playground Toolkit (Dolan 2005) was used and tested for some problems. This toolkit presents a classic GA. The problems tested on the GA Playground were the 10-dimensional versions of the Ackley, the Griewank, the Rastrigin and the Rosenborck functions. The most important parameters in the GA Playground were: *population size=30*, *crossover rate=100%* and *mutation rate=3% (5%* for the Rosenbrock function). Table 40 summarizes the test results.

Table 40. Comparison of the average fitness values achieved by the GA Playground and the DDBGA. The GA Playground was run 30 times up to more than 2000 function evaluations for each test function.

Function	**Average Fitness**	
	GA Playground	**DDBGA**
Ackley (10D)	5.56±0.93	0.0±0.0
Griewank (10D)	2.51±0.66	0.0±0.0
Rastrigin (10D)	9.40± 2.66	0.0±0.0
Rosenbrock (10D)	13.81± 11.58	0.0±0.0

The performance and efficiency of the GA Playground was not comparable with the DDBGA. The DDBGA achieved the global optimal value for the 10-dimensional versions of these functions in 1056 function evaluations for the Ackley function, in 1357 function evaluations for the Griewank function, in 561 function evaluations for the Rastrigin function and in 609 function evaluations for the Rosenbrock function.

More runs up to more than 1000000 function evaluations were made with each test problem in order to find out how close the GA Playground can get to the optimal values. The best optimal value achieved was 0.0130449 while the real global optimal value is 0.

In the following, the statistics presented in (Storn et al. 1995) are used to compare the DDBGA with some other well-known algorithms. The Griewank function is presented in a slightly different form in (Storn et al. 1995):

$$\frac{1}{4000}\sum_{i=1}^{n}(x_i)^2 - \prod\nolimits_{i=1}^{n}\cos(\frac{x_i}{\sqrt{i}})+1, \quad \text{where } -400 \le \mathrm{x_i} \le 400 \text{, for } i = 1,...,n\,. \qquad (58)$$

Also the De Jong function F4 is presented by including a random variable η with uniform distribution and bounded by $[0,1)$ inside the summation instead of just adding it to the summation result:

$$\sum_{i=0}^{29}(x_i^4(i+1)+\eta), \text{ where } x_i \in [-1.28, 1.28] \text{, for } i = 0,...,29\,. \qquad (59)$$

This modified version of the De Jong F4 function has a minimum value of $f(0) \le 30\mathrm{E}[\eta] = 15$, where $\mathrm{E}[\eta]$ is the expectation of η (Storn et al. 1995).

Therefore, the performance of DDBGA was also tested for these functions with new run sequences consisting of 50 test runs for each function. Comparing the number of function evaluations presented in Table 41 points out that the DDBGA has competed very well with other well-known algorithms and has performed much better than both versions of differential evolution.

Table 41. Comparison of Annealed Nedler & Mead (ANM), Adaptive Simulated Annealing (ASA), two versions of Differential Evolution (DE) and the DDBGA for the number of function evaluations to achieve the optimal value. A hyphen means misconvergence (Storn et al. 1995).

Function	ANM	ASA	DE1	DE2	DDBGA

De Jong F1	95	397	490	392	353
De Jong F2	106	11275	746	615	411
De Jong F3	90258	354	915	1300	202
De Jong F4	-	4812	2378	2873	713
De Jong F5	-	1379	735	828	470
Griewank (D10)	-	-	22167	12804	686

All operators were used for the Griewank function and the De Jong F5 function. For other functions, the used operators are mentioned earlier. The DDBGA has been clearly superior to the both versions of the differential evolution algorithm (DE) in all test cases. The most significant difference is observable in solving the 10 dimensional Griewank function, where DDBGA finds the problem's global optimum value in only a fraction of function evaluations compared to both versions of DE. According to (Storn et al. 1995) the Griewank function has many local minima and it is very difficult to find the true minimum. However, while the DDBGA solved the problem on average in only 686 trials, the DE needed on average at least 12804 and at most 22167 evaluations to solve the problem. The DDBGA has also been superior to ASA in all cases and far superior to ANM in solving the De Jong function F3.

In (Krink et al. 2004; Krink 2005) the efficiency of DE, PSO and an EA algorithm in solving the Schaffer F6 (2D), the sphere (5D), the Griewank (50D), the Rastrigin F1 (50D) and the Rosenbrock (50D) functions with and without noise have been compared and some statistics have been presented. The Sphere (5D) function is the same as the De Jong function F3 with this difference that for the Sphere function the valid variable range is $\left[-100,100\right]$. Candidate solutions were represented as arrays of floating-point numbers in algorithms. Parameters for the algorithms are reported in Table 42, where $popSize$ is the population size, CF is the crossover factor, f is the scaling factor, ω is the inertia weight, whose value is linearly decreasing from 1.0 to 0.7 during the run, φ_{min}, φ_{max} are the lower and upper bounds of the random velocity rule weights, p_c is the crossover rate, p_m is the mutation rate, σ_m is the mutation variance, n is the elitist size.

Table 42. Algorithm parameters for differential evolution (DE), particle swarm optimization (PSO) and evolutionary algorithm (EA). (Krink et al. 2004).

DE		**PSO**		**EA**	
$popSize$	50	$popSize$	490	$popSize$	100
CF	0.8	ω	$1.0 \rightarrow 0.7$	p_c	1.0
f	0.5	φ_{min}	915	p_m	0.3
		φ_{max}	2378	σ_m	0.01
				n	10

Re-sampling is the most common way to tackle noise in the noisy functions. The idea with re-sampling is to evaluate the same candidate solution n times and to estimate the real fitness value by the mean of the samples (Krink et al. 2004). This means that for each re-sampling rate s, the fitness of each candidate solution is replaced by the average of s noisy fitness (re-)evaluations. In other words, for a noisy function f the averaged function f' with re-sampling rate s is calculated in the following way:

$$f' = \frac{\sum_{i=1}^{s} f}{s} . \tag{60}$$

This means that during the algorithm run, the fitness value of any candidate solution was calculated and summed together for the number of re-sampling rate and then averaged. In (Krink et al. 2004) the noise re-sampling rate has been selected to be 1, 5, 20, 50 and 100. Therefore, the DDBGA was also tested on these problems to give more comparative statistics for these algorithms. The DDBGA was run 50 times for each test case.

In the following tables the average values of the best fitness function values found by algorithms are reported. The global optimal value for all functions is 0. For the noisy versions of the functions the achieved optimal value should be as close to 0 as possible. Also the standard deviation values related to the calculation of average values are reported. For DDBGA these statistics are drawn from 50 test runs. For other algorithms, they are estimated from 30 values (Krink 2005).

Table 43. Comparison of the average of the best fitness values achieved with different algorithms for the Schaffer function F6 after 100000 fitness function evaluations. *s* stands for the re-sampling rate.

Noise	DE	PSO	EA	DDBGA
s=0	0±0	0.00453±0.00090	3.33067E-17±0	0.0±0.0
s=1	0.48988±0.00582	0.44486±0.01667	0.25829±0.03045	2.20E-07±1.92876E-05
s=5	0.40360±0.03030	0.37603±0.02978	0.12859±0.01678	1.61E-05±8.48851E-05
s=20	0.16597±0.02753	0.19964±0.03328	0.06730±0.01066	-9.68E-05±0.000464474
s=50	0.12729±0.01829	0.09242±0.01634	0.04769±0.00757	0.0020315±0.009497522
s=100	0.09795±0.01209	0.06972±0.00639	0.06277±0.00743	-2.77E-02±0.062652344

These average values found by the algorithm for the function variables were: (0.975±59.586, 1.739±58.693) for *s*=1, (-3.304±55.971,-9.356±53.738) for *s*=5, (-1.609±19.467, 2.056±18.554) for *s*=20, (8.208±37.646, 9.753±39.548), for *s*=50, (-3.218±19.013, -1.143±23.217) for *s*=100. The global optimum value for the non-noisy version of the Schaffer function F6 is at (0,0).

Table 44. Comparison of the average of the best fitness values achieved with different algorithms for the Sphere (5D) function after 100000 fitness function evaluations. *s* stands for the re-sampling rate.

Noise	DE	PSO	EA	DDBGA
s=0	4.12744E-152±0	2.51130E-8±0	6.71654E-20±0	0±0
s=1	0.36484±0.05182	0.25249±0.02603	0.04078±0.00543	-5.4E-6±3.6236E-5
s=5	0.13315±0.01266	0.16702±0.03072	0.02690±0.00363	-1.6765E-5±8.998E-5
s=20	0.07364±0.00811	0.11501±0.01649	0.02205±0.00290	-4.7E-6±2.3E-4
s=50	0.07004±0.00686	0.06478±0.00739	0.01765±0.00233	-1.59E-5±3.19E-4
s=100	0.08165±0.00800	0.07135±0.00938	0.03929±0.00396	1.39E-4±1.17E-3

These average values found by the algorithm for the function variables were: -0.0551±0.2666 for *s*=1, 0.02998±0.18368 for *s*=5, -0.00154±0.16540 for *s*=20, -0.00248±0.11527 for *s*=50 and -0.00298±0.10253 for *s*=100. The global optimum for the non-noisy version of the Sphere (5D) function is at 0 for all variables.

Table 45. Comparison of the average of the best fitness values achieved with different algorithms for the Griewank (50D) function after 500000 fitness function evaluations. *s* stands for the sampling rate.

Noise	DE	PSO	EA	DDBGA
s=0	0±0	1.54900±0.06695	0.00624±0.00138	0±0
s=1	3.31514±0.07388	11.2462±0.50951	1.14598±0.00307	-2.18E-05±4.32E-04
s=5	2.42183±0.03616	16.6429±0.70800	1.10223±0.00342	-4.60E-07±1.11E-05
s=20	2.67093±0.03895	85.4865±2.13148	1.44349±0.01381	-2.80E-07±3.49215E-05
s=50	46.8197±0.96449	143.021±2.33228	3.69626±0.13127	1.61E-05±5.91E-05
s=100	233.802±6.25840	194.188±4.90959	18.0858±0.99646	-7.08E-06±7.39E-05

These average values found by the algorithm for the function variables were: 99.5224±3.2656 for *s*=1, 99.9924±0.3576 for *s*=5, 100.0638±0.2109 for *s*=20, 99.969±0.1545 for *s*=50 and 100.0042±0.1312 for s=100. The global optimum for the non-noisy version of Griewank (50D) function is at 100 for all variables.

Table 46. Comparison of the average of the best fitness values achieved with different algorithms for the Rastrigin (50D) function after 500000 fitness function evaluations. *s* stands for re-sampling rate.

Noise	DE	PSO	EA	DDBGA
s=0	0±0	13.1162±1.44815	32.6679±1.94017	0.0±0.0
s=1	2.35249±0.06062	55.9704±2.19902	30.7511±1.32780	0.0±0.0
s=5	14.0355±0.47935	160.500±2.67500	31.4725±2.02356	0.0±0.0
s=20	167.628±2.12569	313.184±3.93659	39.1777±2.11529	0.0±0.0
s=50	314.762±2.88650	380.178±4.88706	74.8577±2.69437	0.0±0.0
s=100	438.036±3.67504	418.265±5.35434	147.800±2.93208	0.0±0.0

These average values found by the algorithm for the function variables were: -0.00095± 0.00611 for *s*=1, 0.00061± 0.00496 for *s*=5, -0.00085±0.0036 for *s*=20, -0.0006±0.0027 for *s*=50 and -0.00067±0.00260 for *s*=100. The global optimum for the non-noisy version of the Rastrigin (50D) function is at 0 for all variables.

Table 47. Comparison of the average of the best fitness values achieved with different algorithms for the Rosenbrock (50D) function after 500000 fitness function evaluations. *s* stands for re-sampling rate.

Noise	DE	PSO	EA	DDBGA
s=0	35.3176±0.27444	5142.45±2929.47	79.8180±10.4477	0±0
s=1	47.6188±0.15811	4884.68±886.599	118.940±13.2322	-1.58E-06±3.28E-05
s=5	47.0404±0.13932	368512±39755.5	341.788±49.6738	3.82E-06±1.10E-04
s=20	7917.46±352.851	1.61E+7±1.18E+6	1859.06±261.844	3.36E-05±2.35E-04
s=50	1.65E+7±903677	5.57E+7±2.38E+6	35477.7±4656.17	-3.56E-03±2.53E-02
s=100	2.98E+8±1.04E+7	1.17E+8±7.38E+6	257488±19371.2	-0.11610826±0.382981454

These average values found by the algorithm for the function variables were: 1.0008±0.0078 for *s*=1, 1.0006±0.0062 for *s*=5, 0.9994±0.0024 for *s*=20, 0.9998±0.0014 for s=50 and 0.9998±0.006 for *s*=100. The global optimum for the non-noisy version of the Rosenbrock (50D) function is at 1 for all variables.

Comparing the DDBGA with six other well-known algorithms on 36 different test problems clearly proves the superiority of the DDBGA over other algorithms. The DDBGA converged to the global optimum of the test functions in all cases. The DDBGA seems to be far superior to the other algorithms when the test problems include many variables and/or are noisy.

8. Conclusions

In order to introduce new ideas for implementing the theory of genetic algorithms and methods for controlling its parameters, a literature survey was done and a novel method to implement the genetic algorithm was presented. Implementing the genetic algorithm required re-thinking of all phases in the evolutionary process of the genetic algorithm.

As a result, all phases of the genetic algorithm were implemented in more or less new ways. The algorithm is called distributional decimal binary genetic algorithm (DDBGA). The population size was set to 12. Otherwise, the algorithm did not use any pre-set parameter. No stochastic method was used to adjust the selective pressure during the run. Instead, a set of operators was developed to control the search in an implicit self-adaptive way. The algorithm makes heavy use of the population distribution and decimal operators together with binary operators.

Combining the binary and real value operators as it is presented here, removes the dilemma of whether to choose the binary or real value representation. The DDBGA combines advantages of binary and real value representations and exploits the search space in a polymorphic way.

Each run was implemented as a sequence of short runs with relatively small number of generations. A sequence of short runs seems to be a good alternative to a single long run. In each short run the population is generated from scratch. This would offer new chances to the operators to work with new solution candidates and avoid getting stuck in local optimum values. This approach is actually a solution for multimodal problems with many global optimal values. Each distinct run with new population lets the algorithm start from different starting points in the search space and find all possible global optimum values.

Male and female chromosomes were introduced and utilized during the algorithm run. A female chromosome was created by flipping each bit in the original chromosome to its opposite value. Experimentations showed that this technique could frequently convert individuals very quickly to much better ones and hence improve the population quality. The summary of statistics related to the survival of male and female chromosomes reported in Chapter 6 revealed that female chromosomes have managed equally with male chromosomes in the initialization. However, female chromosomes produced better offspring than male chromosomes in only around 32% of cases.

The survivor selection procedure used in the algorithm lets new offspring replace any individual with lower fitness and also gives a chance to improve the worst individual in the population by replacing it with a better individual. The very interesting point here is that individuals can be eliminated and replaced by a new offspring without even participating to any competition. However, offspring are not automatically copied to the population. On the other hand, offspring do not need to be better than their own parents in order to be copied to the population. This survivor selection procedure makes use of any successful genetic operator and does not miss any chance to improve the population average fitness. This also reduces the number of necessary repetitions of genetic operators for improving the population since some individuals can be replaced by better ones as the sub-effect of operators on other individuals.

New approaches for implementing classical genetic operators and several new operators were presented. However, to avoid unnecessary function evaluations expertise knowledge could be utilized to select the most relevant operators for each problem. For instance, if the valid range of a variable is between 0 and less then 2, using the operator, which tries to find a better solution of the greater integer magnitude will lead to unsuccessful trials and increase the number of function evaluations.

Parents for genetic operators were selected mainly from the higher quartile of the fitness population. Intensive efforts were made to improve individuals in the higher quartile and also the elitist individual.

Repeating the higher quartile crossover operator and elitist crossover operator on the same individual indexes and crossover points produced better offspring occasionally in around less than 1% of cases.

For multidimensional problems, the variable crossover operator makes sure that the exchanged bit strings between individuals in the crossover operator represent whole variables and not only a portion of them.

In addition, the variable replacement operator for multiple variable problems tests in turn whether substituting the rest of variables with one variable in the same individual could improve the fitness value of the individual.

The so-called random building block operator selects randomly two crossover points on an individual, then randomly generates a binary string of length determined by the index difference between the crossover points and then copies the generated random building block to the individual on crossover points.

The so-called decimal operators also take care of implementing changes of different magnitudes. These are tricky and very efficient operators, which modify the binary representations of individuals in an intelligent manner so that the result will be within certain decimal limits. These operators proved to be a very efficient solution to the Hamming cliff phenomenon, since they do not let the binary representation fox the algorithm.

The integer mutation operator, which can be compared to the multiple point mutation operator, selects randomly an integer mutation number between 0 and the magnitude of the integer part of the variable and randomly decides whether to add or subtract the mutation number from the variable.

The decimal mutation operator selects randomly a decimal mutation value greater than 0 and less than 1. Then it randomly decides whether to add or subtract the mutation decimal value from the variable.
These decimal operators were implemented separately for the elitist individual in smaller magnitudes. The integer mutation operator for the elitist individual selects randomly either 1 or –1 and adds it to the variable in the elitist individual. The elitist mutation operator for the elitist individual selects randomly a decimal mutation number

greater than zero and less than or equal to 0.1. Then it randomly decides whether to add or subtract this mutation number from the variable.

Analyzing efficiency of these operators revealed that the most efficient operator was the integer mutation operator and the least efficient operator was the higher quartile variable crossover operator. For many dimensional and noisy problems all operators were normally used. However, for some problems with few variables some operators were left out to avoid unnecessary function evaluations. The criteria for selecting operators were 1) the valid range of the variables and 2) the observation about the efficiency of operators through experimentation. The set of most appropriate operators that would lead to the minimum function evaluation was not the same for all test functions.

Also a so-called just-in-time or on-demand search space exploration technique was presented. In this approach a population of fairly small size is initialized and maintained in the beginning. After each operation, the diversity in the population is measured and if it is zero, the elitist individual is reserved and the rest of individuals are replaced by new randomly generated individuals. This approach saves resources by significantly reducing the number of unnecessary evaluations and at the same time gives the algorithm access to new potential solutions in the search space. Two versions of on-demand search space exploration were implemented for different problems: 1) after each operator, 2) after each generation. The algorithm proved to be able to overcome the lack of diversity without refreshing the population for most test problems. However, in some cases refreshing the population seemed to improve the efficiency of the algorithm by reducing the number of function evaluations. Refreshing the population after each operator ensures diversity in the population. However, in some cases it increases the number of function evaluations unnecessarily. Therefore, in some cases the population was refreshed only at the end of each generation if the population consisted of only one distinct individual. This was particularly observable in solving the Griewank function.

The algorithm proved to be very efficient, robust and capable of solving all benchmark test functions. Therefore, based on the achieved results it can be recommended that instead of trying to develop techniques for controlling parameters, focus should be shifted to developing new operators and also new methods for implementing traditional operators. So far, major attention has been paid to the quantitative aspects of the operators, like rates. However, it seems that the quality of operators plays a significant role in the successfulness of operators. Techniques presented here, suggest a straightforward approach for how to select individuals for breeding and how to implement changes in a more deterministic way without forgetting the role of randomness in the evolutionary process.

Furthermore, presented statistics and the way the efficiency of each operator is tracked, show a new way to justify the use of new operators. Each time a new operator is suggested, its efficiency in improving the population fitness compared to previous population can be presented as it is done here.

References

Aarts, E. H. L. & P. J. M. van Laarhoven (1987). *Simulated Annealing: Theory and Applications*. Kluwer Academic Publishers.

Aarts, E. H. L. & J. Korst (1989). *Simulated Annealing and Boltzman Machines: A Stochastic Approach to Combinatorial Optimization and Neural Computing*. Chichester: Wiley .

Altenberg, L. (1995). The schema theorem and Price's theorem. In: *Foundations of Genetic Algorithms 3,* 23-49. Eds Darrell Whitley & Michael Vose. San Francisco: Morgan Kaufmann.

Angeline, P.J. (1995). Adaptive and self-adaptive evolutionary computation. In: *Computational Intelligence: A Dynamic System Perspective*, 152-161. Eds M. Palaniswami, Y. Attikiouzel, R.J. Marks R., D. Fogel & T. Fukuda. IEEE Press.

Angeline, P.J., D.B. Fogel & L.J. Fogel (1996). A comparison of self-adaptation Methods for finite state machines in dynamic environments. In: *Proceedings of the 5th Annual Conference on Evolutionary Programming*. Eds L.J. Fogel, P.J. Angeline & T. Bäck, MIT Press.

Arabas, J., Z. Michalewicz & J. Mulawka (1994). GAVaPS- A genetic algorithm with varying population size. In: *Proceedings of the 1st IEEE Conference on Evolutionary Computation*, 73-78. IEEE Press.

Arts, E. & J.K. Lenstra, eds (1997). *Local Search in Combinatorial Optimization*. John Wiley & Sons.

Ausiello, G., P. Crescenzi, G. Gambosi, V. Kann, A. Marchetti-Spaccamela & M. Protasi (2003). *Complexity and Approximation, Combinatorial Optimization Problems and Their Approximability Properties*. Germany: Springer-Verlag. ISBN 3-540-65431-3.

Bak, P. (1996). *How Nature Works*. Copernicus, Springer-Verlag, 1st edition.

Baker, J. E. (1985). Adaptive selection methods for genetic algorithms. In: *Proceedings of the First International Conference on Genetic Algorithms and Their Applications*, 101-111. Ed. J. J. Grefenstette. Erlbaum.

Baker, J. E. (1987). Reducing bias and inefficiency in the selection algorithm. In: *Genetic Algorithms and Their Applications: Proceedings of the Second International Conference on Genetic Algorithms*, 14-21. Ed. J. J. Grefenstette. Erlbaum.

Bagley, J.D. (1967). *The Behavior of Adaptive Systems Which Employ Genetic and Correlation Algorithms*. PhD Thesis, University of Michigan. Dissertation Abstracts International, 28(12), 5106B. University Microfilms, 68-7556.

Banzhaf, Wolfgang, Peter Nordin, Robert E. Keller & Frank D. Francone (1998). *Genetic Programming—An Introduction: On the Automatic Evolution of Computer Programs and Its Applications*. USA: Morgan Kaufmann Publishers, Inc. and dpunkt—Verlag für digitale Technologie GmbH. ISBN 1-55860-510-X.

Black, Paul E., ed. (2005). Algorithms and theory of computation handbook. Ed. Mikhail J. Atallah. CRC-Press, 1999. In: *Dictionary of Algorithms and Data Structures*. Available at: http://www.nist.gov/dads/. Checked in February 2006.

Blickle, Tobias (2000). Tournament selection. In: *Evolutionary Computation 1, Basic Algorithms and Operators*. Eds T. Bäck, D.B. Fogel & Z. Michalewicz. United Kingdom: Institute of Physics Publishing Ltd, Bristol and Philadelphia. ISBN 0750306645.

Booker, L. B. (1985). Improving the performance of genetic algorithms in classifier systems. In*: Proceedings of the First International Conference on Genetic Algorithms and Their Applications*. Ed. J. J. Grefenstette. Erlbaum.

Bremermann, H.J., M. Rogson, & S. Salaff (1996). Global properties of evolution processes. In: *Natural Automata and Useful Simultations*, 3-41. Eds Pattee et al. Spartan Books.

Bäck, T. & F. Hoffmeister (1991). Extended selection mechanisms in genetic algorithms. In: *Proceedings of the Fourth International Conference on Genetic Algorithms*, 92-99. Ed. R. Belew & L. Booker. San Mateo, CA.: Morgan Kaufmann Publishers.

Bäck, T., D. Fogel & Z. Michalewicz, editors (1997). *Handbook of Evolutionary Computation*. Institute of Physics Publishing Ltd. New York: Bristol and Oxford University Press.

Bäck, T., F. Hoffmeister & H.-P. Schwefel (1991). A survey of evolution strategies. In: *Proceedings of the Fourth International Conference on Genetic Algorithms*, 2-9. Eds Belew, R. & Booker. San Mateo, CA: L. Morgan Kaufmann Publishers.

Bäck, T. (1992 a). The interaction of mutation rate, selection, and self-adaptation within a genetic algorithm. In: *Proceedings of the 2nd Conference on Parallel Problem Solving from Nature*, 85-94. Eds R. Männer & B. Manderick, B. North-Holland.

Bäck, T. (1992 b). Self-adaptation in genetic algorithms. In: *Toward a Practice of Autonomous Systems: Proceedings of the 1st European Conference on Artificial Life*, 263-271. Eds F. J. Varela & P. Bourgine. MIT Press.

Bäck, Thomas (1993*)*. Optimal mutation rates in genetic search. In: *Proceedings of the Fifth International Conference on Genetic Algorithms*, 2-8. Ed. Stephanie Forrest. San Mateo, CA.: Morgan Kaufmann Publishers.

Bäck, T. (1996). *Evolutionary Algorithms in Theory and Practice*. New York: Oxford University Press.

Bäck, T. & M. Schütz (1996*)*. Intelligent mutation rate control in canonical genetic algorithms. In: Foundations of intelligent systems, 1079. In: *Lecture Notes in Artificial Intelligence*, 158-167. Eds Z. Ras & M. Michalewicz. Springer-Verlag.

Bäck, Thomas, Ulrich Hammel & Hans-Paul Schwefel (1997). Evolutionary computation: comments on the history and current state. In: *Handbook of Evolutionary Computation*. Eds Thomas Bäck, D. B. Fogel & Z. Michalewicz. New York: Oxford University Press.

Bäck, Thomas, David B. Fogel, Darrell Whitely & Peter J. Angeline (2000). Mutation operators. In: *Evolutionary Computation 1, Basic Algorithms and Operators*. Eds T. Bäck, D.B. Fogel & Z. Michalewicz. United Kingdom: Institute of Physics Publishing Ltd, Bristol and Philadelphia. ISBN 0750306645.

Cormen, Thomas H., Charles E. Leiserson, Ronald L. Rivest & Clifford Stein (2003). *Introduction to Algorithms*. Second edition. USA: The McGraw-Hill Book Company. ISBN 0-07-013151-1.

Davis, L. D. (1989). Adapting operator probabilities in genetic algorithms. In: *Proceedings of the 1st International Conference on Genetic Algorithms and Their Applications*, 61-69. Ed. J. J. Grefenstette. Lawrence Erlbaum Associates.

Davis, L. D., ed. (1991). *Handbook of Genetic Algorithms*. Van Nostrand Reinhold.

Deb, K. & D.E. Goldberg (1989). An investigation of niche and species formation in genetic function optimization. In: *Proceedings of the Third International Conference on Genetic Algorithms*. Ed. J. D. Schaffer. Morgan Kaufmann.

De Jong, K. A. (1975). *An Analysis of the Behavior of a Class of Genetic Adaptive Systems*. Ph.D. thesis, University of Michigan. Michigan: Ann Arbor.

De la Maza, M. & B. Tidor (1993). An analysis of selection procedures with particular attention paid to boltzmann selection. In: *Proceedings of the 5th International Conference on Genetic Algorithms* 124-131. Ed. S. Forrest. Morgan Kaufmann.

Digalakis, Jason G. & Konstantinos G. Margaritis (2002). An experimental study of benchmarking functions for genetic algorithms. *Intern. J. Computer Math*., 79(4), 403-416. Taylor and Francis Ltd.

Dolan, Ariel (2005). *GA Playground Toolkit*. Available at: http://www.aridolan.com/ga/gaa/gaa.html. Checked in september 2005.

Eiben, A.E. & Zs. Ruttkay (1996). Self-adaptivity for constraint satisfaction: learning penalty functions. In: *Proceedings of the 3rd IEEE Conference on Evolutionary Computation*, 258-261. IEEE Press.

Eiben, Ágoston E., Robert Hinterding & Zbigniew Michalewicz (1999). Parameter control in evolutionary algorithms. *IEEE Transactions on Evolutionary Computation* 3(2), 124-141. This paper has won the 2001 IEEE Transactions on Evolutionary Computation Outstanding Paper.

Eiben, A.E. & J.K. van der Hauw (1997 a). Solving 3-sat with adaptive genetic algorithms, 81-86. In: *Proceedings of the 4th IEEE Conference on Evolutionary Computation*. IEEE Press.

Eiben, A.E. & J.K. van der Hauw (1997 b). Adaptive penalties for evolutionary graph-coloring. In: artificial evolution'97 1363, 95-106. In: *LNCS*. Eds J.K. Hao, E. Lutton, E. Ronald, M. Schoenauer & D. Snyers. Berlin : Springer.

Eiben, A.E., I.G. Sprinkhuizen-Kuyper & B.A. Thijssen (1998). Competing crossovers in adaptive ga framework. In: *Proceedings of the 5th IEEE Conference on Evolutionary Computation*, 787-792. IEEE Press.

Eshelman, L. J. (1991). The CHC adaptive search algorithm: how to have safe search when engaging in nontraditional genetic recombination. In: *Foundations of Genetic Algorithms*. Ed. G. Rawlins. Morgan Kaufmann.

Eshelman, Larry J. (2000). Genetic algorithms. In: *Evolutionary Computation 1, Basic Algorithms and Operators*. Eds T. Bäck, D.B. Fogel & Z. Michalewicz. United Kingdom: Institute of Physics Publishing Ltd, Bristol and Philadelphia. ISBN 0750306645.

Eshelman, L. J. & J.D. Schaffer (1991). Preventing premature convergence in genetic algorithms by preventing incest. In: *Proceedings of the Fourth International Conference on Genetic Algorithms*. Eds R. K. Belew & L. B. Booker. San Mateo, CA : Morgan Kaufmann Publishers.

Eshelman, L. J., R. A. Caruana & J. D. Schaffer (1989). Biases in the crossover landscape. In: *Proceedings of the Third International Conference on Genetic Algorithms*. Ed. J. D. Schaffer. Morgan Kaufmann.

Fogarty, T. (1989). Varying the probability of mutation in the genetic algorithm. In: *Proceedings of the 3rd International Conference on Genetic Algorithms*, 104-109. Ed. J.D. Schaffer. Morgan Kaufmann.

Fogel, D.B. (1995). *Evolutionary Computation*. IEEE Press.

Fogel, L. J., A.J. Owens & M.J. Walsh (1966). *Artificial Intelligence through Simulated Evolution*. Chichester, UK: John Wiley.

Forrest, S. (1985). Scaling fitnesses in the genetic algorithm. In: *Documentation for Prisoners Dilemma and Norms Programs That Use the Genetic Algorithm*. Unpublished manuscript.

Gen, Mitsuo & RunWei Cheng (2000). *Genetic Algorithms and Engineering Optimization*. A Wiley-Interscience Publication. John Wiley & Sons, Inc. ISBN 0-471-31531-1.

Glover, F. (1977). Heuristics for integer programming using surrogate constraints. *Decision Sciences* 8(1), 156-166.

Glover, F. (1989). Tabu search–part I. *ORSA Journal on Computing* 1(3), 190-206.

Glover, F. (1990). Tabu search–part II. *ORSA Journal on Computing* 2(1), 4-32.

Goldberg, D. E. (1989). *Genetic Algorithms in Search, Optimization and Machine Learning*. Reading, MA: Addison Wesley.

Goldberg, D.E. (1990). Real-coded genetic algorithms, virtual alphabets, and blocking. In: *Technical Report 90001*. University of Illinois at Urbana-Chapaign.

Goldberg, David E. (2000). Genetic algorithms: an indiosyncratic tutorial. In: *2000 Genetic and Evolutionary Computation Conference, Tutorial Program*. Las Vegas, Nevada, July 9.

Goldberg, D. E., K. Deb & B. Korb (1990). Messy genetic algorithms revisited: studies in mixed size and scale. *Complex Systems* 4, 415-444.

Goldberg, D. E., K. Deb & B. Korb (1991). Do not worry, be messy. In: *Proceedings of the Fourth International Conference on Genetic Algorithms*, 24-30. Eds R. Belew & L. Booker. San Mateo, CA: Morgan Kaufmann Publishers.

Goldberg, D.E., K. Deb & J.H. Clark (1992 a). Accounting for noise in the sizing of populations. In: *Foundations of Genetic Algorithms 2*, 127-140. Ed. L.D. Whitely. Morgan Kaufmann.

Goldberg, D.E., K. Deb & J.H. Clark (1992 b). Genetic algorithms, noise and the sizing of populations. *Complex Systems* 6, 333-362.

Goldberg, D. E. & J. Richardson (1987). Genetic algorithms with sharing for multimodal function optimization. In: *Genetic Algorithms and Their Applications: Proceedings of the Second International Conference on Genetic Algorithms*. Ed. J. J. Grefenstette. Erlbaum.

Goldberg, D.E. & R.E. Smith (1987). Nonstationary function optimization using genetic algorithms with dominance and diploidy. In: *Proceedings of The 2nd International Conference on Genetic Algorithms*, 59-68. Ed. J.J. Grefenstette. Lawrence Erlbaum Associates.

Gordon, V. S., R. Pirie, A. Wachter & S. Sharp (1999). Terrain-based genetic algorithm (tbga): modeling parameter space as terrain. In: *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-1999)* 1, 229-235. Eds Banzhaf et al..

Grefenstette, J. J. (1986). Optimization of control parameters for genetic algorithms. *IEEE Transactions on Systems, Man, and Cybernetics 16*, 1, 122-128.

Grefenstette, J. J. (1987). Incorporating problem specific knowledge into genetic algorithms. In: *Genetic Algorithms and Simulated Annealing*, 42-60. Ed. L. Davis. San Mateo, CA: Morgan Kaufmann Publishers.

Grefenstette, John (2000). Rank-based selection. In: *Evolutionary Computation 1, Basic Algorithms and Operators*. Eds T. Bäck, D.B. Fogel & Z. Michalewicz. United Kingdom: Institute of Physics Publishing Ltd, Bristol and Philadelphia. ISBN 0750306645.

Haupt, Randy L. & Sue Ellen Haupt (2004). *Practical Genetic Algorithms.* Second edition. Hoboken, New Jersey: A John Wiley & Sons, Inc., Publication. ISBN 0-471-45565-2.

Herdy, M. (1991). Application of the evolution strategy to discrete optimization problems. In: Proceedings of the first international conference on parallel problem solving from nature (ppsn). Eds H.-P. Schwefel & R. Männer. In: *Lecture Notes in Computer Science* 496, 188-192. Springer-Verlag.

Hesser, J. & R. Männer (1991). Towards an optimal mutation probability for genetic algorithms. In: Proceedings of the 1st conference on parallel problem solving from nature. Eds H.-P. Schwefel & R. Männer. In: *Lecture Notes in Computer Science* 496, 23-32. Springer-Verlag.

Hillis, W. D. (1992). Co-evolving parasites improve simulated evolution as an optimization procedure. In: *Artificial Life II*. Eds C. G. Langton, C. Taylor, J. D. Farmer & S. Rasmussen. Addison-Wesley.

Hinterding, R. (1995). Gaussian mutation and self-adaptation in numeric genetic algorithms. In: *Proceedings of the 2nd IEEE Conference on Evolutionary Computation*, 384-389. IEEE Press.

Hinterding, R., Z. Michalewicz & T.C. Peachey (1996). Self-adaptive genetic algorithm for numeric functions. In: Proceedings of the 4th conference on parallel problem solving from nature. Eds H.-M. Voigt, W. Ebeling, I. Rechenberg & H.-P. Schwefel. In: *Lecture Notes in Computer Science* 1141, 420-429. Berlin:Springer.

Hinterding, R., Z. Michalewicz & A.E. Eiben (1997). Adaptation in evolutionary computation: a survey. In: *Proceedings of the 4th IEEE Conference on Evolutionary Computation*, 65-69. IEEE Press.

Holland, J. H. (1975). *Adaptation in Natural and Artificial Systems*. Ann Arbor: MI: University of Michigan Press.

Holland, J. H. (1986). Escaping Brittleness: the possibilities of general-purpose learning algorithms applied to parallel rule-based systems. In: *Machine Learning II*. R. S. Michalski, J. G. Carbonell & T. M. Mitchell. Morgan Kaufmann.

Ingber, L. (1993). *Simulated Annealing: Practice Versus Theory*. Mathl. Comput. Modelling, 18(11), 29-57.

Ingber, L. & B. Rosen (1992). Genetic algorithms and very fast simulated reannealing: a comparison. J. Mathl. *Comput. Modelling* 16(11), 87-100.

Janikow, C. Z. & Z. Michalewicz (1991). An experimental comparison of binary and floating-point representations in genetic algorithms. In: *Proceedings of the Fourth International Conference on Genetic Algorithms*. Eds R. K. Belew & L. B. Booker. Morgan Kaufmann.

Julstrom, B.A. (1995). What have you done lately for me? Adapting operator probabilities in a steady-state genetic algorithm. In: *Proceedings of the 6th International Conference on Genetic Algorithms*, 81-87. Ed. L. Eshelman. Morgan Kaufmann.

Kalyanmoy, Deb (2000). Introduction to representations. In: *Evolutionary Computation 1, Basic Algorithms and Operators*. Eds T. Bäck, D.B. Fogel & Z. Michalewicz. United Kingdom: Institute of Physics Publishing Ltd, Bristol and Philadelphia. ISBN 0750306645.

Kee, E., S. Airey & W. Cyre (2001). An adaptive genetic algorithm. In: *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001)* 391-397. Eds Spector et al.

Kennedy, J. & R. C. Eberhart (1995). Particle swarm optimization. In: *Proceedings of the 1995 IEEE International Conference on Neural Networks IV*, 1942-1948. Piscataway, NJ: IEEE Service Center.

Koza, J. R. (1992). *Genetic Programming*. Cambridge, MA: MIT Press.

Koziel, S. & Z. Michalewicz (1998). A decoder-based evolutionary algorithm for constrained parameter optimization problems. In: Proceedings of the 5th international on parallel problem solving from nature. Eds A.E. Eiben, Th. Bäck, M. Schoenauer & H.-P. Schwefel. In: *Lecture Notes in Computer Science* 1498, 231-240. Berlin: Springer.

Koziel, S. & Z. Michalewicz (1999). Evolutionary algorithms, homomorphous mappings, and constrained parameter optimization. *Evolutionary Computation* 7(1), 19-44.

Krink, Thiemo, Bogdan Filipič & Gary B. Fogel (2004). *Noisy Optimization Problems-A Particular Challenge for Differential Evolution?* Available at: http://www.natural-selection.com/Library/2004/CEC04_Noisy.pdf. Checked in December 2005.

Krink, T., R. Thomsen & P. Rickers (2000). Applying self-organized criticality to evolutionary algorithms. In: *Parallel Problem Solving from Nature VI (PPSN-2000)*, 1, 375-384. Eds Schoenauer et al.

Krink, Thiemo (2005). *Foundations of Evolutionary Computation*, *Lecture Notes*. Available at: http://www.daimi.au.dk/~krink/fec05/index.html. Checked in June 2005.

Kursawe, Frank (1991). A variant of evolution strategies for vector optimization. In: *Parallel Problem Solving from Nature*, 193-197. Eds H.-P. Schwefel & R. Männer. Berlin: Springer.

Lee, M. & H. Takagi (1993). Dynamic control of genetic algorithms using fuzzy logic techniques. In: *Proceedings of the Fifth International Conference on Genetic Algorithms*, 76-83. Ed. Forrest.

Lis, J. & M. Lis (1996). Self-adapting parallel genetic algorithm with the dynamic mutation probability, crossover rate and population size. In: *Proceedings of the 1st Polish National Conference on Evolutionary Computation*, 324-329. Ed. J. Arabas. Oficina Wydawnica Politechniki Warszawskiej.

Lis, J. (1996). Parallel Genetic algorithm with dynamic parameter control. In: *Proceedings of the 3rd IEEE Conference on Evolutionary Computation,* 324-329. IEEE Press.

Laumanns, Marco, Günter Rudolph & Hans-Paul Schwefel (1998). A spatial predator-prey approach to multi-objective optimization: a preliminary study. In: *Parallel Problem Solving From Nature (PPSN-V)*, 241-249. Eds Agoston E. Eiben et al. Berlin: Springer.

Laumanns, Marco, Günter Rudolph & Hans-Paul Schwefel (2001). Mutation control and convergence in evolutionary multi-objective optimization. In: *7th International Conference on Soft Computing MENDEL 2001*, Brno, Czech Republic, June 6-8.

Laumanns, Marco, Eckart Zitzler & Lothar Thiele (2001). On the effects of archiving, elitism and density based selectio in evolutionary multi-objective optimization. In: Evolutionary multi-criterion optimization (emo 2001). Ed. E. Zitzler. In: *Lecture Notes in Computer Science* 1993, 181-196. Berlin: Springer-Verlag.

Magyar, Gábor (2000). *On Solution Approaches for Some Industrially Motivated Combinatorial Optimization Problems*. TUCS dissertation 23. Turku Center for Computer Science (TUCS) and University of Turku, Department of Mathematical Sciences. Turku, Finland.

Mahfoud, S.W. (1997). Boltzmann selection. In: *Handbook of Evolutionary Computation*, C2.5, 1-4. Eds T. Bäck, D. Fogel & Z. Michalewicz. New York: Institute of Physics Publishing Ltd, Bristol and Oxford University Press.

Mahfoud, Samir W. (2000). Boltzmann selection. In: *Evolutionary Computation 1, Basic Algorithms and Operators*. Eds T. Bäck, D.B. Fogel & Z. Michalewicz. United Kingdom: Institute of Physics Publishing Ltd, Bristol and Philadelphia. ISBN 0750306645.

Michalewicz, Zbigniew (1996). *Genetic Algorithms + Data Structures = Evolution Programs*. Third, Revised and Extended Edition. USA: Springer. ISBN 3-540-60676-9.

Michalewicz, Zbigniew (2000). Introduction to search operators. In: *Evolutionary Computation 1, Basic Algorithms and Operators*. Eds T. Bäck, D.B. Fogel & Z. Michalewicz. United Kingdom: Institute of Physics Publishing Ltd, Bristol and Philadelphia. ISBN 0750306645.

Michalewicz, Z. & N. Attia (1994). Evolutionary optimization of constrained problems. In : *Proceedings of the 3rd Annual Conference on Evolutionary Programming*, 98-108. Eds A.V. Sebald & L.J. Fogel. World Scientific.

Michalewicz, Zbigniew & David B. Fogel (2004). *How to Solve It: Modern Heuristics.* Second, Revised and Extended Edition. Germany: Springer-Verlag Berlin Heidelberg. ISBN 3-540-22494-7.

Miller, Brad L. & M. J. Shaw (1996). Genetic algorithms with dynamic niche sharing for multimodal function optimization. *Technical Report 95010*, IlliGAL, University of Illinois.

Mitchell, Melanie (1998). *An Introducton to Genetic Algorithms*. United States of America: A Bradford Book. First MIT Press Paperback Edition.

Morris, P. (1993). The breakout method for escaping from local minima. In: *Proceedings of the 11th National Conference on Artificial Intelligenc*, 40-45. AAAI-93, AAAI Press/The MIT Press.

Mühlenbein, H. (1992). How genetic algorithms really work: 1. mutation and hill-climbing. In: *Parallel Problem Solving from Nature 2*. Eds R. Männer & B. Manderick. North-Holland.

Mühlenbein, H. & D. Schlierkamp-Vosen (1993). Predictive models for the breeder genetic algorithm. *Evolutionary Computation* 1(1), 25-49.

Neter, John, William Wasserman & G.A. Whitmore (1993). *Applied Statistics*. Fourth edition. USA: Simon & Schuster, Inc. ISBN 0-205-13478-5.

O'Reilly, U.-M. & F. Oppacher (1995). The troubling aspects of a building block hypothesis for genetic programming. In: *Foundations of Genetic Algorithms 3*. Eds L. D. Whitley & M. D. Vose. Morgan Kaufmann.

Pagie, L. & P. Hogeweg (1997). Evolutionary consequences of coevolving targets. *Evolutionary Computation* 5(4), 401-418.

Papadimitriou, C. H. & K. Steiglitz (1982). *Combinatorial Optimization: Algorithms and Complexity*. New Jersey: Prentice-Hall.

Paredis, J. (1994). Co-evolutionary constraint satisfaction. In: Proceedings of the 3rd conference on parallel problem solving from nature. Eds Y. Davidor, H.-P. Schwefel & R. Männer. In: *Lecture Notes in Computer Science* 866, 46-56. Springer-Verlag.

Paredis, J. (1995 a). Co-evolutionary computation. *Artificial Life* 2(4), 355-375.

Paredis, J. (1995 b). The symbolic evolution of solutions and their representations. In: *Proceedings of the 6th International Conference on Genetic Algorithms*, 359-365. Ed. L. Eshelman. Morgan Kaufmann.

Press, W.H., S.A. Teukolsky, W.T. Vetterling & B.P. Flannery (1992). *Numerical Recipes in C*. Cambridge University Press.

Rechenberg, I. (1973). *Evolutionsstrategie*: *Optimierung technischer Systeme nach Prinzipien der biologischen Evolution*. Stuttgart: Frommann-Holzboog Verlag.

Reeves, Colin R., ed. (1993). *Modern Heuristic Techniques for Combinatorial Problems*. Oxford Blackwell Scientific Publications. Printed in Great Britain. ISBN 0-470-22079-1.

Reeves, C. R., ed. (1995). *Modern Heuristic Techniques for Combinatorial Problems*. McGraw-Hill.

Reinelt, G. (1994). The traveling salesman. In: *Lecture Notes in Computer Science,* 840, Springer-Verlag.

Ronald, E. (1995). When selection meets seduction. In: *Proceedings of the Sixth International Conference on Genetic Algorithms*, 167-173. Ed. L. J. Eshelman. San Mateo, CA: Morgan Kaufmann.

Rosenberg, R.S. (1967). *Simulation of Genetic Populations with Biochemical Properties*. PhD Thesis, University of Michigan. Dissertation Abstracts International 28(7), 2732B. University Microfilms 67-17, 836.

Rudolph, Günter (2000). Evolution strategies. In: *Evolutionary Computation 1, Basic Algorithms and Operators*. Eds T. Bäck, D.B. Fogel & Z. Michalewicz. United Kingdom: Institute of Physics Publishing Ltd, Bristol and Philadelphia. ISBN 0750306645.

Rudolph, G. & A. Agapie (2000). Convergence properties of some multi-objective evolutionary algorithms. In: *Congress on Evolutionary Computation (CEC 2000)* 2, 1010-1016. Piscataway, NJ: IEEE Press.

Rudolph, G. & J. Sprave (1995). A cellular genetic algorithm with self-adjusting acceptance threshold. In: *Proceedings of the 1st IEE/IEEE International Conference on Genetic Algorithms in Evolutionary Systems: Innovations and Applications*, 365-372. London: IEE.

Saravanan, F. & D.B. Fogel (1994). Learning strategy parameters in evolutionary programming: an empirical study. In: *Proceedings of the 3rd Annual Conference on Evolutionary Programming*. Eds A.V. Sebald & L.J. Fogel. World Scientific.

Saravanan, N., D.B. Fogel & K.M. Nelson (1995). A comparison of methods for self-adaptation in evolutionary algorithms. *BioSystems* 36, 157-166.

Sarma, Jayshree & Kenneth De Jong (2000). Generation gap methods. In: *Evolutionary Computation 1, Basic Algorithms and Operators*. Eds T. Bäck, D.B. Fogel & Z. Michalewicz. United Kingdom: Institute of Physics Publishing Ltd, Bristol and Philadelphia. ISBN 0750306645.

Shaefer, C.G. (1987). The ARGOT strategy: adaptive representation genetic optimizer technique. In: *Proceedings of the 2nd International Conference on Genetic Algorithms and Their Applications*, 50-55. Ed. J.J. Grefenstette. Lawrence Erlbaum Associates.

Schaffer, J. D., R.A. Caruana, L.J. Eshelman & R. Das (1989). A study of control parameters affecting online performance of genetic algorithms for function optimization. In: *Proceedings of the Third International Conference on Genetic Algorithms*. Ed. J. D. Scahffer. Morgan Kaufmann.

Schraudolph, N. & R. Belew (1992). Dynamic parameter encoding for genetic algorithms. In: *Machine Learning* 9(1), 9-21.

Schwefel, H.-P. (1981). *Numerical Optimization for Computer Models*. UK: John Wiley, Chichester.

Schwefel, H.-P. (1995). *Evolution and Optimum Seeking*. New York: Wiley.

Schwefel, Hans-Paul (2000). Advantages (and disadvantages) of evolutionary computation over other approaches. In: *Evolutionary Computation 1, Basic Algorithms and Operators*. Eds T. Bäck, D.B. Fogel & Z. Michalewicz. United Kingdom: Institute of Physics Publishing Ltd, Bristol and Philadelphia. ISBN 0750306645.

Smith, R. (1993). Adaptively resizing populations: an algorithm and analysis. In: *Proceedings of the 5th International Conference on Genetic Algorithms*. Ed. S. Forest. Morgan Kaufmann.

Smith, R. (1997). Population size. In: *Handbook of Evolutionary Computation*. Eds T. Bäck, D. Fogel & Z. Michalewicz. Institute of Physics Publishing Ltd. New York: Bristol and Oxford University Press.

Smith, J. & T. Fogarty (1996 a). Adaptively parameterized evolutionary systems: self adaptive recombination and mutation in genetic algorithm. In: Proceedings of the 4th conference on parallel problem solving from nature, 441-450. Eds H.-M. Voigt, W. Ebeling, I. Rechenberg & H.-P. Schwefel. In: *Lecture Notes in Computer Science* 1141. Berlin: Springer.

Smith, J. & T. Fogarty (1996 b). Self-adaptation of mutation rates in a steady-state genetic algorithm. In: *Proceedings of the 3rd IEEE Conference on Evolutionary Computation*, 318-323. IEEE Press.

Smith, R. E., S. Forrest & A.S. Perelson (1993). Population diversity in an immune system model: implications for genetic search. In: *Foundations of Genetic Algorithms 2*. Ed. L.D. Whitely. Morgan Kaufmann.

Smith, A.E. & D.M. Tate (1993). Genetic optimization using a penalty function. In: *Proceedings of the 5th International Conference on Genetic Algorithms*, 499-503. Ed. S. Forrest. Morgan Kaufmann.

Soule, T. & J.A. Foster (1997). Code size and depth flows in genetic programming. In: *Proceedings of the 2nd Annual Conference on Genetic Programming*, 313-320. Eds J.R. Koza, K. Deb, M. Dorigo, D.B. Fogel, M. Garzon, H. Iba & R.L. Riolo. MIT Press.

Spears, W. M. & K.A. De Jong (1991). On the virtues of parametrized uniform crossover. In: *Proceedings of the Fourth International Conference on Genetic Algorithms*. Eds R. K. Belew & L. B. Booker. Morgan Kaufmann.

Spears, W. M. (1993). Crossover or mutation? In: *Foundations of Genetic Algorithms 2*. Ed. L. D. Whitely. Morgan Kaufmann.

Spears, W. M. (1995). Adapting crossover in evolutionary algorithms. In: *Proceedings of the 4th Annual Conference on Evolutionary Programming*, 367-384. Eds J.R. McDonnell, R.G. Reynolds & D.B. Fogel. MIT Press.

Srinivas, M. & L.M. Patnaik (1994). Adaptive probabilities of crossover and mutation in genetic algorithms. *IEEE Transactions on Systems, Man and Cybernetics* 24(4), 17-26.

Schlierkamp-Voosen, D. & H. Mühlenbein (1994). Strategy adaptation by competing subpopulations. In: Proceedings of the 3rd conference on parallel problem solving from nature, 866. Eds Y. Davidor, H.-P. Schwefel & R. Männer. In: *Lecture Notes in Computer Science* 199-208. Springer-Verlag.

Schlierkamp-Voosen, D. & H. Mühlenbein (1996). Adaptation of population sizes by competing subpopulations. In: *Proceedings of the 3rd IEEE Conference on Evolutionary Computation*, 330-335. IEEE Press.

Storn, Rainer & Kenneth Price (1995). Differential evolution-a simple and efficient adaptive scheme for global optimization over continuous spaces. *Technical Report* 95-012.

Syswerda, G. (1991). Scheduled optimization using genetic algorithms. In: *Handbook of Genetic Algorithms* 332-349. L. Davis. Van Nostrand Reinhold.

Tackett, W. A. (1994). *Recombination, Selection, and the Genetic Construction of Computer Programs*, Ph.D. thesis, Department of Computer Engineering, University of Southern California.

Tuson, A. & P. Ross (1996). Cost based operator rate adaptation: an investigation. In: *Proceedings of the 4th Conference on Parallel Problem Solving from Nature*. Eds H.-M. Voigt, W. Ebeling, I. Rechenberg & H.-P. Schwefel. In: *Lecture Notes in Computer Science* 1141, 461-469. Berlin: Springer.

Ursem, Rasmus K. (2003). *Models for Evolutionary Algorithms and Their Applications in System Identification and Control Optimization (PhD Dissertation)*. A Dissertation Presented to the Faculty of Science of the University of Aarhus in Partial Fulfillment of the Requirements for the PhD Degree. Department of Computer Science, University of Aarhus, Denmark.

White, T. & F. Oppacher (1994). Adaptive crossover using automata. In: *Proceedings of the 3rd Conference on Parallel Problem Solving from Nature*. Eds Y. Davidor, H.-P. Schwefel & R. Männer. In: *Lecture Notes in Computer Science* 866, 229-238. Springer-Verlag.

Whitley, D. (1989). The GENITOR algorithm and selection pressure: why rank-based allocation of reproductive trials is best. In: *Proceedings of the Third International Conference on Genetic Algorithms*, 116-121. Ed. J. Schaffer. San Mateo, CA: Morgan Kaufmann Publishers.

Whitley, D., V. Gordan & K. Mathias (1994). Lamarckian evolution, the Baldwin effect and function optimization. In: *Parallel Problem Solving from Nature*, 6-15. Eds Y. Davidor, H. Schwefel & R. Männer. PPSN III. Berlin: Springler-Verlag.

Whitley, L.D., K. Mathias & P. Fitzhorn (1991). Delta coding: an iterative strategy for genetic algorithms. In: *Proceedings of the 4th International Conference on Genetic Algorithms* 77-84. Eds R.K. Belew & L.B. Booker. Morgan Kaufmann.

Whitley, Darrell (2000). Permutations. In: *Evolutionary Computation 1, Basic Algorithms and Operators*. Eds T. Bäck, D.B. Fogel & Z. Michalewicz. United Kingdom: Institute of Physics Publishing Ltd, Bristol and Philadelphia. ISBN 0750306645.

Wilf, Herbert S. (1994). *Algorithms and Complexity*. University of Pennsylvania. Philadelphia, PA 19104-6395. Internet edition. Available at: http://www.math.upenn.edu/%7Ewilf/AlgoComp.pdf. Checked in February 2006.

Wikipedia, The Free Encyclopedia. Wikimedia Foundation, Inc. Available at: *http://en.wikipedia.org/wiki/Main_Page*. Checked in December 2005.

Wolpert, D.H. & W.G. Macready (1995). No free lunch theorems for search. *Technical Report*, SFI-TR-02-010, Santa Fe Institute.

Wright, A. H. (1991). Genetic algorithms for real parameter optimization. In: *Foundations of Genetic Algorithms*. Ed. G. Rawlings. Morgan Kaufmann.

Appendix A

Table A1. Summary of test runs for the Ackley function with different number of variables (n).

n	Run Summary	Generations	Evaluations to approx. optimal	Time to approx. optimal(s)	Total Evaluations	Total elapsed Time(s)
1	**Best Run**	2.00	232.00	0.00	232.00	0.00
	Worst Run	25.00	445.00	0.00	3682.00	2.00
	Mean	8.22	528.22	0.02	1143.46	0.40
	Median	6.00	529.00	0.00	843.00	0.00
	Mode	6.00	340.00	0.00	#N/A	0.00
	Std. Dev.	5.47	214.43	0.14	820.90	0.67
3	**Best Run**	2.00	375.00	0.00	375.00	0.00
	Worst Run	8.00	1685.00	1.00	1685.00	1.00
	Mean	4.88	789.98	0.10	990.58	0.20
	Median	5.00	804.50	0.00	967.50	0.00
	Mode	5.00	#N/A	0.00	#N/A	0.00
	Std. Dev.	1.57	278.23	0.30	329.72	0.45
4	**Best Run**	1.00	259.00	0.00	259.00	0.00
	Worst Run	9.00	734.00	1.00	2227.00	3.00
	Mean	4.14	729.20	0.04	964.28	0.20
	Median	4.00	706.00	0.00	903.00	0.00
	Mode	3.00	706.00	0.00	729.00	0.00
	Std. Dev.	1.69	241.14	0.20	414.35	0.53
5	**Best Run**	2.00	473.00	0.00	473.00	0.00
	Worst Run	10.00	1179.00	0.00	2582.00	1.00
	Mean	4.12	836.30	0.06	1061.26	0.16
	Median	4.00	784.50	0.00	1037.50	0.00
	Mode	4.00	496.00	0.00	1066.00	0.00
	Std. Dev.	1.57	291.95	0.31	406.09	0.42
10	**Best Run**	1.00	357.00	0.00	439.00	0.00
	Worst Run	5.00	1623.00	1.00	2043.00	1.00
	Mean	2.64	889.00	0.14	1056.20	0.22
	Median	2.50	772.00	0.00	1011.00	0.00
	Mode	2.00	760.00	0.00	761.00	0.00
	Std. Dev.	0.88	319.65	0.40	340.12	0.51
20	**Best Run**	1.00	612.00	0.00	612.00	0.00
	Worst Run	3.00	1365.00	1.00	2150.00	1.00
	Mean	2.18	1150.22	0.28	1386.12	0.56
	Median	2.00	1301.50	0.00	1311.00	0.00
	Mode	2.00	627.00	0.00	1311.00	0.00
	Std. Dev.	0.52	369.17	0.50	390.71	0.64
50	**Best Run**	1.00	1340.00	1.00	1340.00	1.00
	Worst Run	2.00	3028.00	4.00	3138.00	4.00
	Mean	1.64	1885.40	2.18	2334.88	2.80
	Median	2.00	1557.50	2.00	2721.00	3.00
	Mode	2.00	1437.00	2.00	1563.00	3.00
	Std. Dev.	0.48	621.18	1.12	651.79	1.01

Table A2. Descriptive statistics of the population fitness values for the test runs on the Ackley function with different numbers of variables (n).

n	Runs	Pops	Min	Max	Mean	Std. Dev.	Skew.	Kurt.
1	**Best Run**	**Worst Pop.**	-22.08	-0.64	-17.37	5.00	3.00	8.00
		Best Pop.	0.00	0.00	0.00	0.00	0.00	-3.00
	Worst Run	**Worst Pop.**	-22.28	-3.64	-19.31	5.00	2.00	5.00
		Best Pop.	0.00	0.00	0.00	0.00	0.00	-3.00
3	**Best Run**	**Worst Pop.**	-20.09	-15.94	-17.58	1.00	1.00	6.00
		Best Pop.	0.00	0.00	0.00	0.00	0.00	-3.00
	Worst Run	**Worst Pop.**	-20.15	-14.33	-19.03	1.00	9.00	41.00
		Best Pop.	0.00	0.00	0.00	0.00	0.00	-3.00
4	**Best Run**	**Worst Pop.**	-20.84	-14.36	-18.65	1.00	6.00	35.00
		Best Pop.	-1.66	0.00	-0.20	0.00	0.00	0 -3.0
	Worst Run	**Worst Pop.**	-20.65	-13.23	-18.96	1.00	16.00	96.00
		Best Pop.	0.00	0.00	0.00	0.00	0.00	-3.00
5	**Best Run**	**Worst Pop.**	-20.86	-17.19	-19.36	1.00	0.00	1.00
		Best Pop.	-0.02	0.00	0.00	0.00	0.00	-3.00
	Worst Run	**Worst Pop.**	-20.42	-16.24	-19.05	1.00	1.00	3.00
		Best Pop.	0.00	0.00	0.00	0.00	0.00	-3.00
10	**Best Run**	**Worst Pop.**	-20.89	-19.23	-20.14	0.00	0.00	-3.00
		Best Pop.	-1.01	0.00	-0.61	0.00	0.00	-3.00
	Worst Run	**Worst Pop.**	-20.69	-19.75	-20.21	0.00	0.00	-3.00
		Best Pop.	-0.02	0.00	0.00	0.00	0.00	-3.00
20	**Best Run**	**Worst Pop.**	-21.01	-20.32	-20.70	0.00	0.00	-3.00
		Best Pop.	-0.01	0.00	-0.01	0.00	0.00	-3.00
	Worst Run	**Worst Pop.**	-20.98	-20.43	-20.74	0.00	0.00	-3.00
		Best Pop.	-0.12	0.00	-0.01	0.00	0.00	-3.00
50	**Best Run**	**Worst Pop.**	-21.18	-20.87	-21.03	0.00	0.00	-3.00
		Best Pop.	0.00	0.00	0.00	0.00	0.00	-3.00
	Worst Run	**Worst Pop.**	-21.16	-20.32	-20.95	0.00	0.00	-3.00
		Best Pop.	-0.01	0.00	-0.01	0.00	0.00	-3.00

Table A3. Summary of test runs for the Griewank function with different number of variables (n).

n	Run Summary	Generations	Evaluations to approx. optimal	Time to approx. optimal(s)	Total Evaluations	Total elapsed Time(s)
1	**Best Run**	1.00	148.00	0.00	152.00	0.00
	Worst Run	46.00	6638.00	5.00	6712.00	5.00
	Mean	9.86	956.52	0.48	1357.12	0.72
	Median	6.00	340.00	0.00	837.00	0.00
	Mode	6.00	148.00	0.00	268.00	0.00
	Std. Dev.	10.53	1574.26	1.16	1541.77	1.20
3	**Best Run**	1.00	166.00	0.00	166.00	0.00
	Worst Run	30.00	6326.00	4.00	6456.00	4.00
	Mean	9.46	1767.92	0.76	1991.60	0.88
	Median	8.00	1323.00	1.00	1639.50	1.00
	Mode	6.00	168.00	0.00	#N/A	0.00
	Std. Dev.	6.62	1435.85	0.94	1438.69	0.96
4	**Best Run**	1.00	191.00	0.00	191.00	0.00
	Worst Run	14.00	3181.00	1.00	3399.00	2.00
	Mean	5.40	1016.30	0.30	1236.30	0.40
	Median	5.00	857.00	0.00	1064.50	0.00
	Mode	3.00	416.00	0.00	475.00	0.00
	Std. Dev.	3.32	732.44	0.68	797.69	0.76
5	**Best Run**	1.00	146.00	0.00	146.00	0.00
	Worst Run	13.00	3449.00	4.00	3449.00	4.00
	Mean	3.70	781.26	0.10	921.42	0.14
	Median	3.00	736.00	0.00	834.00	0.00
	Mode	3.00	472.00	0.00	715.00	0.00
	Std. Dev.	1.84	507.54	0.58	487.59	0.61
10	**Best Run**	1.00	348.00	0.00	348.00	0.00
	Worst Run	4.00	1575.00	1.00	1643.00	1.00
	Mean	2.20	682.72	0.08	851.78	0.10
	Median	2.00	752.00	0.00	817.00	0.00
	Mode	2.00	752.00	0.00	426.00	0.00
	Std. Dev.	0.81	287.35	0.27	309.01	0.30
20	**Best Run**	1.00	551.00	0.00	551.00	0.00
	Worst Run	2.00	1312.00	0.00	1398.00	0.00
	Mean	1.40	782.92	0.10	893.72	0.18
	Median	1.00	627.00	0.00	708.50	0.00
	Mode	1.00	627.00	0.00	714.00	0.00
	Std. Dev.	0.49	285.73	0.30	314.04	0.39
50	**Best Run**	1.00	825.00	1.00	1273.00	2.00
	Worst Run	2.00	825.00	0.00	2726.00	3.00
	Mean	1.04	1360.40	1.14	1444.30	1.26
	Median	1.00	1374.50	1.00	1379.00	1.00
	Mode	1.00	1437.00	1.00	1275.00	1.00
	Std. Dev.	0.20	123.93	0.40	261.59	0.53

Table A4. Descriptive statistics of the population fitness values for the test runs on the Griewank function with different numbers of variables (n).

n	Runs	Pops	Min	Max	Mean	Std. Dev.	Skew.	Kurt.
1	Best Run	Worst Pop.	-72.1	0	-18.09	22	-1	0
		Best Pop.	0	0	0	0	0	-3
	Worst Run	Worst Pop.	-92.1	-0.61	-38.57	32	0	-2
		Best Pop.	-0.46	0	-0.19	0	0	-3
3	Best Run	Worst Pop.	-204	-29.51	-92.48	56	0	-1
		Best Pop.	-95	0	-42.41	38	0	-2
	Worst Run	Worst Pop.	-226	-20.36	-109.88	58	0	-1
		Best Pop.	0	0	0	0	0	-3
4	Best Run	Worst Pop.	-198	-9.4	-102.86	55	0	-1
		Best Pop.	-0.04	0	-0.01	0	0	-3
	Worst Run	Worst Pop.	-289	-32.18	-160.52	80	0	-2
		Best Pop.	0	0	0	0	0	-3
5	Best Run	Worst Pop.	-349	-92.87	-209.77	79	0	-1
		Best Pop.	-0.05	0	-0.01	0	0	-3
	Worst Run	Worst Pop.	-295	-20.02	-170.39	72	0	-1
		Best Pop.	0	0	0	0	0	-3
10	Best Run	Worst Pop.	-511	-204	-376.49	107	0	-2
		Best Pop.	0	0	0	0	0	-3
	Worst Run	Worst Pop.	-476	-212	-342.38	84	0	-2
		Best Pop.	0	0	0	0	0	-3
20	Best Run	Worst Pop.	-902	-379	-603.1	155	0	-1
		Best Pop.	-2.6	0	-1.12	1	0	-2
	Worst Run	Worst Pop.	-834	-330	-590.2	150	0	-1
		Best Pop.	-672	0	-56	193	-2	6
50	Best Run	Worst Pop.	-1918	-1439	-1649	170	0	-2
		Best Pop.	-1.22	0	-0.38	0	0	-3
	Worst Run	Worst Pop.	-2279	-1407	-1665	256	-1	0
		Best Pop.	-1623	0	-520.4	769	0	-2

Table A5. Summary of test runs for the Rastrigin function F1.

n	Summary	Generations	Evaluations to approx. optimal	Time to approx. optimal (s)	Total Evaluations	Total Elapsed Time (s)
1	Best Run	1	148	0	148	0
	Worst Run	22	264	0	3219	2
	Mean	5.48	350.78	0.04	751.3	0.18
	Median	4	285.5	0	524.5	0
	Mode	3	152	0	446	0
	Std. Dev.	4.13	226.35	0.2	602.01	0.48
3	Best Run	1	162	0	162	0
	Worst Run	5	793	0	1080	0
	Mean	2.64	453.1	0.06	533.66	0.06
	Median	3	413	0	486	0
	Mode	2	854	0	372	0
	Std. Dev.	0.96	201.48	0.24	216.84	0.24
4	Best Run	1	191	0	191	0
	Worst Run	8	1548	0	1994	1
	Mean	2.5	489.6	0.04	563.04	0.06
	Median	2	444	0	487	0
	Mode	2	428	0	428	0
	Std. Dev.	1.15	223.58	0.2	289.07	0.24
5	Best Run	1	144	0	144	0
	Worst Run	6	994	0	1555	1
	Mean	2.22	457.52	0.04	562.94	0.1
	Median	2	467.5	0	499	0
	Mode	2	222	0	548	0
	Std. Dev.	1.04	207.13	0.2	278.95	0.3
10	Best Run	1	329	0	329	0
	Worst Run	3	954	0	954	0
	Mean	1.52	561.44	0.02	561.44	0.02
	Median	1	446.5	0	446.5	0
	Mode	1	447	0	447	0
	Std. Dev.	0.68	207.3	0.14	207.3	0.14
20	Best Run	1	342	0	342	0
	Worst Run	2	1447	1	1447	1
	Mean	1.26	795.78	0.1	795.78	0.1
	Median	1	707.5	0	707.5	0
	Mode	1	713	0	713	0
	Std. Dev.	0.44	287.72	0.3	287.72	0.3
50	Best Run	1.00	79.00	0.00	79.00	0.00
	Worst Run	1.00	1595.00	1.00	1595.00	1.00
	Mean	1.00	1317.88	0.96	1317.88	0.96
	Median	1.00	1359.50	1.00	1359.50	1.00
	Mode	1.00	1383.00	1.00	1383.00	1.00
	Std. Dev.	0.00	259.32	0.35	259.32	0.35

Table A6. Descriptive statistics of the population fitness values for the test runs on Rastrigin function with different numbers of variables (n).

n	Runs	Pops	Min	Max	Mean	Std. Dev.	Skew.	Kurt.
1	**Best Run**	**Worst Pop.**	-28.8	-0.06	-14.6	8	0	-1
		Best Pop.	0	0	0	0	0	-3
	Worst Run	**Worst Pop.**	-31.8	-1.5	-12.7	10	0	-2
		Best Pop.	0	0	0	0	0	-3
3	**Best Run**	**Worst Pop.**	-83.81	-18.54	-40.06	21	0	-1
		Best Pop.	0	0	0	0	0	-3
	Worst Run	**Worst Pop.**	-63.36	-14.01	-35.92	17	0	-2
		Best Pop.	0	0	0	0	0	-3
4	**Best Run**	**Worst Pop.**	-64.28	-31.18	-42.69	11	0	-1
		Best Pop.	0	0	0	0	0	-3
	Worst Run	**Worst Pop.**	-52.07	-27.33	-42.03	7	1	1
		Best Pop.	0	0	0	0	0	-3
5	**Best Run**	**Worst Pop.**	-69.92	-23.12	-51.47	15	0	-1
		Best Pop.	-0.5	0	-0.17	0	0	-3
	Worst Run	**Worst Pop.**	-64.15	-25.37	-44.84	14	0	-2
		Best Pop.	0	0	0	0	0	-3
10	**Best Run**	**Worst Pop.**	-155.96	-114.52	-136	15	0	-2
		Best Pop.	0	0	0	0	0	-3
	Worst Run	**Worst Pop.**	-155.29	-114.47	-133.08	13	0	-2
		Best Pop.	0	0	0	0	0	-3
20	**Best Run**	**Worst Pop.**	-322.45	-258.69	-288.6	19	0	-1
		Best Pop.	-3.47	0	-1.69	1	0	5
	Worst Run	**Worst Pop.**	-323.34	-280.92	-305.43	15	0	-2
		Best Pop.	-0.07	0	-0.03	0	0	-3
50	**Best Run**	**Worst Pop.**	-863.62	-723.98	-804.08	46	0	-1
		Best Pop.	-0.01	0	-0.01	0	0	-3
	Worst Run	**Worst Pop.**	-850.15	-752.68	-804.04	34	0	-2
		Best Pop.	-0.06	0	-0.05	0	0	-3

Table A7. Summary of test runs for the Rosenbrock function with different number of variables (n).

n	Summary	Generations	Evaluation to approx. optimal	Time to approx. optimal (s)	Total Evaluations	Total Elapsed Time (s)
3	**Best Run**	1.00	57.00	0.00	57.00	0.00
	Worst Run	8.00	1711.00	1.00	1711.00	1.00
	Mean	3.10	615.74	0.14	615.74	0.14
	Median	3.00	522.50	0.00	522.50	0.00
	Mode	2.00	363.00	0.00	363.00	0.00
	Std. Dev.	1.53	334.24	0.35	334.24	0.35
4	**Best Run**	1.00	192.00	0.00	192.00	0.00
	Worst Run	6.00	1329.00	0.00	1329.00	0.00
	Mean	2.68	602.86	0.06	602.86	0.06
	Median	2.00	512.00	0.00	512.00	0.00
	Mode	2.00	672.00	0.00	672.00	0.00
	Std. Dev.	1.15	264.89	0.24	264.89	0.24
5	**Best Run**	1.00	146.00	0.00	146.00	0.00
	Worst Run	5.00	1383.00	1.00	1383.00	1.00
	Mean	2.34	592.86	0.08	592.86	0.08
	Median	2.00	533.50	0.00	533.50	0.00
	Mode	2.00	218.00	0.00	218.00	0.00
	Std. Dev.	0.92	258.16	0.27	258.16	0.27
10	**Best Run**	1.00	341.00	0.00	341.00	0.00
	Worst Run	3.00	1252.00	0.00	1252.00	0.00
	Mean	1.60	609.20	0.04	609.20	0.04
	Median	2.00	722.50	0.00	722.50	0.00
	Mode	2.00	423.00	0.00	423.00	0.00
	Std. Dev.	0.53	199.46	0.20	199.46	0.20
20	**Best Run**	1.00	539.00	0.00	539.00	0.00
	Worst Run	2.00	1409.00	1.00	1409.00	1.00
	Mean	1.18	745.78	0.10	745.78	0.10
	Median	1.00	619.00	0.00	619.00	0.00
	Mode	1.00	596.00	0.00	596.00	0.00
	Std. Dev.	0.39	252.65	0.30	252.65	0.30
50	**Best Run**	1.00	61.00	0.00	61.00	0.00
	Worst Run	1.00	1562.00	1.00	1562.00	1.00
	Mean	1.00	1302.48	1.08	1302.48	1.08
	Median	1.00	1342.50	1.00	1342.50	1.00
	Mode	1.00	1341.00	1.00	1341.00	1.00
	Std. Dev.	0.00	212.17	0.34	212.17	0.34

Table A8. Descriptive statistics of the population fitness values for the test runs on Rosenbrock function with different numbers of variables (n).

n	Runs	Pops	Min	Max	Mean	Std. Dev.	Skew.	Kurt.
3	Best Run	Worst Pop.	-3.10E+08	-227568	-6.20E+07	3.12E+08	-166	-2
		Best Pop.	-35445	0	-8285	14557	-1	1
	Worst Run	Worst Pop.	-1.00E+09	-333713	-2.20E+08	2.91E+08	2	-2
		Best Pop.	-1.00E-06	0	-1.00E-06	0	0	-3
4	Best Run	Worst Pop.	-1.40E+09	-180369	-2.90E+08	4.09E+08	-1	-2
		Best Pop.	-3.12E-04	0	-2.84E-04	0	0	0
	Worst Run	Worst Pop.	-4.80E+08	-1.20E+07	-1.20E+08	1.28E+08	-1	-2
		Best Pop.	-1.40E-05	0	-9.00E-06	0	0	1
5	Best Run	Worst Pop.	-1.90E+09	-1.30E+08	-6.60E+08	4.92E+08	-1	-2
		Best Pop.	-0.06406	0	-0.03287	0	0	0
	Worst Run	Worst Pop.	-2.30E+09	-1.20E+08	-7.20E+08	7.55E+08	-1	-2
		Best Pop.	-2.00E-06	0	-1.00E-06	0	0	-3
10	Best Run	Worst Pop.	-9.50E+09	-1.90E+09	-4.40E+09	2337467715	0	-1
		Best Pop.	-5.66E-04	0	-4.26E-04	2413738	2	-3
	Worst Run	Worst Pop.	-9.90E+09	-6.70E+08	-5.61E+09	3153121821	0	-2
		Best Pop.	-1.00E-06	0	-1.00E-06	0	0	-3
20	Best Run	Worst Pop.	-2.40E+10	-1.30E+10	-1.93E+10	3813356653	0	-2
		Best Pop.	-19.7376	0	-5.56449	8	-1	-1
	Worst Run	Worst Pop.	-2.60E+10	-1.70E+10	-2.16E+10	3768494242	0	-2
		Best Pop.	-6.38073	0	-1.86768	2	-1	1
50	Best Run	Worst Pop.	-8.60E+10	-5.50E+10	-6.68E+10	10815749638	0	-1
		Best Pop.	-212.635	0	-101.159	102	0	-2
	Worst Run	Worst Pop.	-7.20E+10	-5.20E+10	-6.46E+10	6854101733	0	-1
		Best Pop.	-1.41223	0	-0.95826	0	0	-3