Statistics and Computing (1991) **1**, 75–91

Genetic algorithms for numerical optimization

ZBIGNIEW MICHALEWICZ[1] and CEZARY Z. JANIKOW[2]

[1]*Department of Computer Science, University of North Carolina, Charlotte, NC 28223, USA*
[2]*Department of Computer Science, University of North Carolina, Chapel Hill, NC 27599, USA*

Received February 1990 and accepted June 1990

Genetic algorithms (GAs) are stochastic adaptive algorithms whose search method is based on simulation of natural genetic inheritance and Darwinian striving for survival. They can be used to find approximate solutions to numerical optimization problems in cases where finding the exact optimum is prohibitively expensive, or where no algorithm is known. However, such applications can encounter problems that sometimes delay, if not prevent, finding the optimal solutions with desired precision. In this paper we describe applications of GAs to numerical optimization, present three novel ways to handle such problems, and give some experimental results.

Keywords: Genetic algorithm, random algorithm, optimization technique, constraint handling, local tuning, convergence

1. Introduction

In the 1950s, von Neumann created a theory of *self-reproducing automata* (von Neumann, 1966), which laid the foundations for the field of *genetic algorithms.* In the late 1950s Holland (1959) continued this idea. In his more recent research, Holland (1975) discussed the ability of a simple bit-string representation to encode complicated structures and the transformations to improve them. The main result of this work was a demonstration that, with the proper control structure, rapid improvements to bit strings could occur under certain transformations. Even in large and complicated search spaces, given certain conditions on the problem domain, GAs would tend to converge on solutions that were globally optimal or nearly so.

Genetic algorithms (Holland, 1975; DeJong, 1985; Davis, 1987; Goldberg, 1989) implement these ideas; they are a class of probabilistic algorithms that start with a population of randomly generated feasible solutions. These solutions 'evolve' towards better ones by applying genetic operators modeled on the genetic processes occurring in nature. (For a comparison of GAs with other optimization methods the reader is referred to Ackley, 1987.)

GAs have been quite successfully applied to optimization problems like wire routing, scheduling, adaptive control, game playing, cognitive modeling, the transportation problem, the traveling salesman problem, optimal control problems, etc. (see Booker, 1982; DeJong, 1985; Grefenstette, 1985; 1987a; Goldberg, 1989; Schaffer, 1989; Vignaux and Michalewicz, 1989; 1990; Michalewicz and Janikow, 1991; Michalewicz *et al*, 1990).

However, as stated by DeJong (1985):

> because of this historical focus and emphasis on function optimization applications, it is easy to fall into the trap of perceiving GAs themselves as optimization algorithms and then being surprised and/or disappointed when they fail to find an 'obvious' optimum in a particular search space. My suggestion for avoiding this perceptual trap is to think of GAs as a (highly idealized) simulation of a natural process and as such they embody the goals and purposes (if any) of that natural process. I am not sure if anyone is up to the task of defining the goals and purpose of evolutionary systems; however, I think it's fair to say that such systems are not generally perceived as functions optimizers.

The purpose of this paper is twofold. First, we provide a survey of genetic algorithms, discussing what they are, why they work, and how they work. Second, we present various modifications of the classical GAs; these modifica-

0960-3174/91 $03.00 +.12

tions result in a system which can be perceived as a function optimizer.

The paper is organized as follows. In Section 2 we explain the main idea behind GAs, and in Section 3 we describe the major problems that GA implementations encounter. Then, in Sections 4–6 we discuss the major problems and proposed solutions in detail. The discussion is supported by a series of experiments. Section 7 gives conclusions and directions for future work.

2. Genetic algorithms

In this section we introduce genetic algorithms, present their theoretical foundations, and describe their applicability.

2.1 *What they are*

GAs represent a class of adaptive algorithms whose search methods are based on simulation of natural genetics. They belong to the class of probabilistic algorithms; yet, they are very different from random algorithms as they combine elements of directed and stochastic search. Also, for hard optimization problems, they are superior to hill-climbing methods, since at any time GAs provide for both exploitation of the best solutions and exploration of the search space. Because of this, GAs are also more robust than existing directed search methods. Another important property of such genetic-based search methods is their domain-independence.

In general, a GA performs a multi-directional search by maintaining a population of potential solutions and encourages information formation and exchange between these directions. This population undergoes a simulated evolution: at each generation the relatively 'good' solutions reproduce, while the relatively 'bad' solutions die. To distinguish between different solutions we need some evaluation function which plays the role of an environment.

The structure of a simple GA is shown in Fig. 1. During iteration t, the GA maintains a population of potential solutions (called *chromosomes* following the natural terminology), $P(t) = \{x_1^t, \ldots, x_n^t\}$. Each solution x_i^t is evaluated to give some measure of its 'fitness'. Then, a new population (iteration $t+1$) is formed by selecting the more fitted individuals. Some members of this new population undergo reproduction by means of crossover and mutation, to form new solutions.

Crossover combines the features of two parent chromosomes to form two similar offspring by swapping corresponding segments of the parents. For example, if the parents are represented by five-dimensional vectors $(a_1, b_1, c_1, d_1, e_1)$ and $(a_2, b_2, c_2, d_2, e_2)$ (with each element called a *gene*), then crossing the chromosomes after the

```
Procedure genetic algorithm
begin
  t = 0
  initialize P(t)
  evaluate P(t)
  while (not termination-condition) do
  begin
    t = t + 1
    select P(t) from P(t − 1)
    recombine P(t)
    evaluate P(t)
  end
end
```

Fig. 1. *A simple genetic algorithm.*

second gene would produce the offspring $(a_1, b_1, c_2, d_2, e_2)$ and $(a_2, b_2, c_1, d_1, e_1)$. The intuition behind the applicability of the crossover operator is information exchange between different potential solutions.

Mutation arbitrarily alters one or more genes of a selected chromosome, by a random change with a probability equal to the mutation rate. The intuition behind the mutation operator is the introduction of some extra variability into the population.

A genetic algorithm for a particular problem must have the following five components: a genetic representation for potential solutions to the problem; a way to create an initial population of potential solutions; an evaluation function that plays the role of the environment, rating solutions in terms of their 'fitness'; genetic operators that alter the composition of children during reproduction; and values for various parameters that the genetic algorithm uses (population size, probabilities of applying genetic operators, etc.)

2.2 *Why they work*

The theoretical foundations of GAs rely on a binary string representation of solutions, and on the notion of a *schema* (see, e.g. Holland, 1975)—a template allowing exploration of similarities among chromosomes. A schema is built by introducing a new *don't care* symbol ($\star$) into the alphabet of genes—such a schema represents all strings (a hyperplane, or subset of the search space) which match it on all positions other than $\star$. In a population of size n of chromosomes of length m, between 2^m and $2^m n$ different schemata may be represented; at least n^3 of them are processed usefully—Holland has called this property an *implicit parallelism*, as it is obtained without any extra memory or processing requirements.

Two other important notions, associated with the schema, are necessary to derive the theoretical basis. The *schema order*, $o(H)$, is the number of non-don't care positions. It defines the speciality of a schema. The *schema defining length*, $l(H)$, is the distance between the first and the last non-don't care symbols of a chromosome. It defines the compactness of information contained in a schema.

Assuming that the selective probability is proportional to fitness, and independent probabilities, p_c and p_m, for crossover and mutation, respectively, we can derive the following growth equation (see e.g. Goldberg, 1989):

$$m(H, t+1) \geq m(H, t) \times \frac{f(H, t)}{\bar{f}(t)} \times \left[1 - p_c \frac{l(H)}{l-1} - p_m o(H)\right] \qquad (1)$$

where $m(H, t)$ is the number of schema H at time t, $f(H, t)$ is the average fitness of schema H at time t, and $\bar{f}(t)$ is the average fitness of the population. This equation is also based on the assumption that the fitness function f returns only positive values; when applying GAs to optimization problems where the optimization function may return negative values, some additional mapping between optimization and fitness functions is required (see Goldberg, 1989).

The growth equation (1) shows that selection increases sampling rate of the above-average schemata, and that this change is exponential. The sampling itself does not introduce any new schemata (not represented in the initial $t = 0$ sampling). This is exactly why the crossover operator is introduced—to enable structured yet random information exchange. Additionally, the mutation operator introduces greater variability into the population. The combined (disruptive) effect of these operators on a schema is not significant if the schema is short and low-order. This result can be stated as:

Theorem 1 (Schema Theorem) *Short, low-order, above-average schemata receive exponentially increasing trials in subsequent generations of a genetic algorithm.*

An immediate result of this theorem is that GAs explore the search space by short schemata which, subsequently, are used for information exchange during crossover:

Hypothesis 1 (Building Block Hypothesis) *A genetic algorithm seeks near-optimal performance through the juxtaposition of short, low-order, high-performance schemata, called the building blocks.*

As stated by Goldberg (1989):

> Just as a child creates magnificent fortresses through the arrangement of simple blocks of wood, so does a genetic algorithm seek near-optimal performance through the juxtaposition of short, low-order, high performance schemata, or building blocks.

Although some research has been done to prove this hypothesis (e.g. Bethke, 1980), for most non-trivial applications we rely mostly on empirical results. During the last 15 years many GA applications were developed which supported the building block hypothesis in many different problem domains. Nevertheless, this hypothesis suggests that the problem of coding for a GA is critical for its performance, and that such a coding should satisfy the idea of short building blocks.

Since the binary alphabet offers the maximum number of schemata per bit of information of any coding (see Goldberg, 1989), the bit string representation of solutions has dominated genetic algorithm research. Additionally, such coding provides simplicity of analysis and elegance of available operators. But the 'implicit parallelism' result does not depend on the use of bit strings (see Antonisse and Keller, 1987)—hence it may be worthwhile experimenting with richer data structures and other 'genetic' operators. This may be important in particular in the presence of non-trivial constraints on potential solutions to the problem.

2.3 *How they work*

Suppose we wish to maximize a function of k variables, $f(x_1, \ldots, x_k)$: $\mathbb{R}^k \to \mathbb{R}$. Suppose further, that each variable x_i can take values from a domain $D_i = [a_i, b_i] \subseteq \mathbb{R}$. We wish to optimize the function f with some precision: suppose six decimal places for the variables' values is desired.

It is clear that to achieve such precision each domain D_i should be cut into $(b_i - a_i) \times 10^6$ equal size ranges. Let us denote by m_i the smallest integer such that $(b_i - a_i) \times 10^6 \leq 2^{m_i} - 1$. Then, a representation having each variable coded as a binary string of length m_i clearly satisfies the precision requirement. Additionally, the following formula easily interprets each such string:

$$x_i = a_i + \text{decimal}(1001 \ldots 001_2) \times \frac{b_i - a_i}{2^{m_i} - 1}$$

where decimal($string_2$) represents the decimal value of that binary string.

Now, each chromosome (as a potential solution) is represented by a binary string of length $m = \Sigma_{i=1}^{k} m_i$; the first m_1 bits map into a value from the range $[a_1, b_1]$, the next group of m_2 bits map into a value from the range $[a_2, b_2]$, the last group of m_k bits map into a value from the range $[a_k, b_k]$.

To initialize the population, if we do not have any intuition about the distribution of potential optima, we can simply set some number, *pop_size*, of chromosomes randomly in a bitwise fashion. Otherwise, we can provide some initial (potential) solutions.

The rest of the algorithm is straightforward: in each generation we evaluate each chromosome (using the function f on the decoded sequences of variables), select a new population according to the probability distribution based on fitness values, and recombine the chromosomes in the new population by mutation and crossover operators. After a number of generations, when no further improvement is observed, the best chromosome represents a (possibly the global) optimal solution. In practice, we stop the algorithm after a fixed number of iterations, depending on speed and resource criteria.

2.4 *An example*

Let us assume we wish to find the maximum of the following function:

$$f(x_1, x_2, x_3) = 3.5(x_1 - 2.1x_2)^3 - \sqrt{x_1 x_2 + \log_2(x_3 + 1)} \times \sin^2(x_3 + \pi)$$

where $-3.0 \le x_1 \le 12.1$, $4.1 \le x_2 \le 5.8$, and $0.0 \le x_3 \le 50.0$. The required precision is four decimal places for each variable.

The domain of x_1 has length 15.1; the precision requirement implies that the range $[-3.0, 12.1]$ should be divided into at least $15.1 \times 10\,000$ equal size ranges. This means that 18 bits are required for this part of the chromosome:

$$2^{17} < 151\,000 \le 2^{18}$$

The domain of x_2 has length 1.7; the precision requirement implies that the range $[4.1, 5.8]$ should be divided into at least $1.7 \times 10\,000$ equal size ranges. This means that 15 bits are required for this part of the chromosome:

$$2^{14} < 17\,000 \le 2^{15}$$

The domain of x_3 has length 50.0; the precision requirement implies that the range $[0.0, 50.0]$ should be divided into at least $50.0 \times 10\,000$ equal size ranges. This means that 19 bits are required as this part of the chromosome:

$$2^{18} < 500\,000 \le 2^{19}$$

The total length of a chromosome (solution vector) is then $18 + 15 + 19 = 52$ bits; the first 18 bits code x_1, bits 19–33 code x_2, and bits 34–52 code x_3.

Let us consider an example chromosome:

0100010010110100001111100101000101010100001000100101

The first 18 bits

010001001011010000

represent $x_1 = -3.0 + \text{decimal}(010001001011010000_2)$

$$\times \frac{12.1 - (-3.0)}{2^{18} - 1} = -3.0 + 70352 \times \frac{15.1}{262143}$$

$$= -3.0 + 4.052426 = 1.052426.$$

The next 15 bits

111110010100010

represent

$$x_2 = 4.1 + \text{decimal}(111110010100010_2) \times \frac{5.8 - 4.1}{2^{15} - 1}$$

$$= 4.1 + 31906 \times \frac{1.7}{32767} = 4.1 + 1.655330 = 5.755330.$$

The last 19 bits

1010100001000100101

represent

$$x_3 = 0.0 + \text{decimal}(1010100001000100101_2) \times \frac{50.0 - 0.0}{2^{19} - 1}$$

$$= 0.0 + 344613 \times \frac{50.0}{524288} = 32.864857.$$

So the chromosome

0100010010110100001111100101000101010100001000100101

corresponds to

$$(x_1, x_2, x_3) = (1.052426, 5.755330, 32.864857).$$

The fitness value for this chromosome is

$$f(1.052426, 5.755330, 32.864857) = -4704.82.$$

To optimize the function f using a GA, we create a population of such chromosomes (the typical population size for such numerical optimization varies from 50 to 100). All 52 bits for each chromosome are initialized randomly. Each chromosome is evaluated and a new population is formed by selecting the more fitted individuals according to their fitness (we normally assume all evaluations are non-negative; there are ways to enforce that). Some chromosomes from the new population would undergo reproduction by means of crossover and mutation, to form new chromosomes (new solutions). For example, two chromosomes:

01000100101101000011111|00101000101010100001000100101

00001101111110000010101010|10000010101000001000011111100

may be selected as parents for the crossover; the crossover point is (randomly) selected after the 23rd bit (as marked above); the resulting offspring are

01000100101101000011111|10000010101000001000011111100

00001101111110000010101010|00101000101010100001000100101

3. Problems with genetic algorithms

However, applications such as these can encounter problems that sometimes delay, if not prevent, finding the optimal solutions with a desired precision. Such problems originate from many sources,—for example, the coding, which moves the operational search space away from the problem space; insufficient number of iterations, and insufficient population size. These problems then manifest themselves in premature convergence of the entire population to a non-global optimum; inability to perform fine local tuning; or inability to operate in the presence of non-trivial constraints. In this section we describe these three manifestations and in Sections 4–6 we present quite novel ways to handle them.

3.1 *Premature convergence*

Premature convergence is a common problem of GAs and other optimization algorithms. If convergence occurs too rapidly, then the valuable information developed in part of the population is often lost. Implementations of genetic algorithms are prone to converge prematurely before the optimal solution has been found. As stated by Booker (1987):

> While the performance of most implementations is comparable to or better than the performance of many other search techniques, it [the GA] still fails to live up to the high expectations engendered by the theory. The problem is that, while the theory points to sampling rates and search behavior in the limit, any implementation uses a finite population or set of sample points. Estimates based on finite samples inevitably have a sampling error and lead to search trajectories much different from those theoretically predicted. This problem is manifested in practice as a premature loss of diversity in the population with the search converging to a sub-optimal solution.

To improve the performance of GAs, DeJong (1975) investigated five modifications of the basic algorithm. These modifications were called the *elitist* model, the *expected value* model, the *elitist expected value* model, the *crowding factor* model, and the *generalized crossover* model. A few years later, Brindle (1981) examined five further modifications: *deterministic sampling*, *remainder stochastic sampling without replacement*, *stochastic sampling without replacement*, *remainder stochastic sampling with replacement*, and *stochastic tournament*. A detailed discussion of all these modifications is presented by Goldberg (1989). Quite another direction in relaxing this problem borrows ideas from *simulated annealing* (see e.g. Sirag and Weisser, 1987).

In general, most approaches which attempted to improve the convergence of GAs presented some modifications to the selection routine.

3.2 *Fine local tuning*

Genetic algorithms display inherent difficulties in performing local search for the numerical applications. As observed by Grefenstette (1987a):

> Like natural genetic systems, GAs progress by virtue of changing the distribution of high performance substructures in the overall population; individual structures are not the focus of attention. Once the high performance regions of the search space are identified by a GA, it may be useful to invoke a local search routine to optimize the members of the final population.

Local search requires the utilization of schemata of higher order and longer defining length that those suggested by Theorem 1. Holland (1975) suggested that the GA should be used as a preprocessor to perform the initial search, before turning the search process over to a system that can employ domain knowledge to guide the local search. Additionally, there are problems where the domains of parameters are unlimited, the number of parameters is quite large, and high precision is required. These requirements imply that the length of the (binary) solution vector is quite significant (for 100 variables with domains in the range $[-500, 500]$, where the six digits' precision after the decimal point is required, the length of the binary solution vector is 3000). For such problems the performance of GAs is quite poor.

3.3 *Constraints*

The central problem in applications of GAs is that of constraints; until recently there was no promising methodology for handling them.

Traditionally, to solve a constrained optimization problem using the GA approach we use some penalty functions. However, such approaches suffer from the disadvantage of being tailored to the problem and are not sufficiently general to handle a variety of problems.

In this approach we generate potential solutions without considering the constraints, but incorporating in the evaluation function penalties for suppressing illegal candidates. However, though the evaluation function is usually well defined, there is no accepted melthodology for combining it with the penalty (Richardson *et al*, 1989). Davis (1987) discusses a problem in deciding how large a penalty to impose:

> If one incorporates a high penalty into the evaluation routine and the domain is one in which production of an individual violating the constraint is likely, one runs the risk of creating a genetic algorithm that spends most of its time evaluating illegal individuals. Further, it can happen that when a legal individual is found, it drives the others out and the population converges on it without finding better individuals, since the likely paths to other legal individuals require the production of illegal individuals as intermediate structures, and the penalties for violating the constraints make it unlikely that such intermediate structure will reproduce. If one imposes moderate penalties, the system may evolve individuals that violate the constraint

but are rated better than those that do not because the rest of the evaluation function can be satisfied better by accepting the moderate constraint penalty than by avoiding it.

4. Premature convergence

As stated in Section 3, one of the problems encountered by GA applications is premature convergence of the entire population to a non-global optimum. The premature convergence is closely related to the characteristics of the function itself, and to the magnitude and kind of errors introduced by the sampling mechanism.

The problem of premature convergence is primarily related to the existence of local optima, and depends on both function characteristics and sampling of the solution space. For example, assume that $s_v^t \in P(t)$ is close to some local optimum, and $f(s_v^t)$ is much greater than the average evaluation $\bar{f}(t)$. Also, assume that there is no s_w^t close to the global maximum sought; this might be the case for multi-modal functions. In such a case, there is a fast convergence toward that local optimum, with little chance of more global exploration needed to search for other optima. While such a behavior is permissible at the later evolutionary stages, and even desired at the final ones, it is quite disturbing very early. We propose an approach (see Janikow and Michalewicz, 1991; Janikow and Michalewicz, 1990) which diminishes this problem by decreasing the speed of convergence during the early stages of population existence.

The other problem, also reflecting on the convergence, is related to shifts in the average population fitness. Consider two functions: $f_1(s)$, and $f_2(s) = f_1(s) + const$, which have the same relative optima. One would expect that both can be optimized with similar degree of difficulty. However, if $const \gg \bar{f}_1(s)$, then either the function $f_2(s)$ will suffer from much slower convergence than the function $f_1(s)$, or the function $f_1(s)$ will converge possibly to a local optimum. Some of the previous approaches to this problem used rank instead of actual values $f(s_v^k)$ to guide the selective process. Such an approach suffers from some drawbacks. First, it puts the responsibility on the user to decide on the best selection mechanism. Second, it totally ignores information it holds about the absolute evaluations of different chromosomes. Third, it treats all cases uniformly, regardless of the magnitude of the problem.

To deal with such cases we introduce a measure of the problematic characteristics of the function being optimized, which we later incorporate into the sampling mechanism.

4.1 *The scaling mechanism*

We are mostly interested in some average behavior of the function being optimized. However, we know such behavior only through finite sampling from the population. Therefore, we define a measure using statistical random sample variance and mean (we call such a measure a *span*):

$$s_t = \frac{\sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(f(x_i^t) - \bar{f}(t))^2}}{\frac{1}{n}\sum_{i=1}^{n} f(x_i^t)}$$

which can be rewritten as:

$$s_t \simeq \sqrt{\frac{n}{n-1}} \cdot \sqrt{\frac{n \cdot \sum_{i=1}^{n} f^2(x_i^t)}{\left(\sum_{i=1}^{n} f(x_i^t)\right)^2} - 1}$$

This measure is normalized so that it is quite function-independent.

Genetic algorithm applications normally require that all chromosomes evaluate to non-negative numbers; such a requirement relates to the sampling mechanism and can be easily enforced. In such a case $\max(s_t) = \sqrt{n}$ since $\Sigma_i^n f_i^2 \leq (\Sigma_i^n f_i)^2$. However, such a maximal value is obtained only when all but one evaluations are zero—a very unlikely case. Experiments show that most spans are less than one. Moreover, we have determined experimentally, by running different functions with differently scaled s and observing the average optimization behavior, that $s^* = 0.1$ gives the best possible trade-off between space exploration and speed of convergence; therefore, we subsequently treat all cases with respect to the relationship between s_t and s^*. This particular choice of s^* is rather rough and more extensive experiments are planned to approximate it in a more systematic way. Moreover, it would be interesting to see whether it could be approximated theoretically; however, the mechanism we are about to introduce is little affected by some variations of s^*.

To preserve efficiency we do not want to recalculate s_t at each iteration. This is actually not necessary as this measure finds the function's characteristics determinable from a random sample. We have determined experimentally that very often the initial population provides a very good approximation. However, if desired, such sampling may be performed for some time before the algorithm actually starts iterating (we call such a span s_0).

For the construction of our scaling mechanism, which we call a *non-uniform power scaling*, we use two dynamic parameters: the span s and population age t (this parameter is taken to be the iteration number of the algorithm). What we seek is a mechanism which produces more random search for small ages and increasingly selects only the best chromosomes at late stages of the population life. Moreover, the net effect of such a mechansim should depend on the span; a greater span should cause the whole mechanism to put less emphasis on chromosomes' fitness, somehow randomizing the sampling. The scaling itself uses the *power-law scaling*:

$$f_i' = f_i^k$$

where $k \sim 0$ forces a random search, while $k > 1$ forces sampling to be allocated to only the most fitted chromosomes. We construct k so that it changes from small to large values over the population age (with largest changes very early and very late), and the magnitude of k should be larger and the speed of change smaller for problems with lower span. The following equation satisfies these goals:

$$k = \left(\frac{s^*}{s_0}\right)^{p_1} \tan^{p_2\left(\frac{s_0}{s^*}\right)^{\alpha}}\left(\frac{t}{T+1} \times \frac{\pi}{2}\right)$$

where p_1 is a system parameter determining the influence of span s on the magnitude of k, and p_2 and α are system parameters determining the speed of change of k; α determines how much the span affects such changes.

4.2 *The test case*

To experiment with our scaling mechanism we decided to use a single family of functions:

$$f(\mathbf{x}) = a + \sum_{i=1}^{\#elems} \sin(x_i)\sin^{2q}\left(\frac{ix_i^2}{\pi}\right)\Bigg|\, x_k \in [0, \pi], \quad k = 1, \ldots, \#elems$$

which is a sine-modulating ($\#elems$)-dimensional function of components with non-linearly increasing frequency. This function, given appropriate parameters a and q, could simulate functions of totally different characteristics (as mentioned in Section 3.1):

A: $a = 5.0$, $q = 1$. This function, even though non-trivial, exhibits rather nice characteristics: its average span $s_0 \simeq 0.1$ matches the most desired s^*.

B: $a = 0.1$, $q = 250$. This function is very difficult to analyse numerically due to its high non-smoothness. Such highly negative undesired characteristics are captured in its $s_0 \simeq 1.0$. Because of such characteristics, any numerical methods, including GA, will tend toward false optima.

C: $a = 1000$, $q = 1$. This function is a constant transformation of the A function. Such a shift causes the average span to be decreased dramatically to $s_0 \simeq 0.001$.

Note that $f(\mathbf{x}) > 0$ for $a > 0$. In order to visualize better such characteristics, cross-sections of these three functions are given in Fig. 3. For these particular experiments we used $\#elems = 10$. Initial s_0 for these three functions are given in Fig. 2 along with the appropriate behavior of the k exponent.

4.3 *Experiments and results*

To judge the quality of our non-uniform power scaling we tried to maximize the three functions both with the mechanism on and off. We decided on two comparative measures of the GA's performance. First, *accuracy* is defined as the value of the best found chromosome, relative to the value of the global optimum:

$$\frac{f^{\text{best}} - a}{f^{\text{global}} - a}$$

We subtract the a parameter so that we can easily compare the results of different functions.

Second, *imprecision* is defined as the standard deviation of the optimal vector found. This measure evaluates the closeness of individual components selected in the found optimal chromosome as follows:

$$\sqrt{\frac{\sum_{i=1}^{\#elems}(index_i)^2}{\#elems - 1}}$$

where $index_i$ is the index of the ith component of the found optimal chromosome (each dimension has its peaks ordered from 0 to the number of dimensions minus 1 according to the modulated values of its peaks). For example, *imprecision* $= 0$ means the generated chromosome correctly selected the best modulated values along all dimensions; however, it does not have to generate the best function value due to local imperfectness associated with finite iteration time.

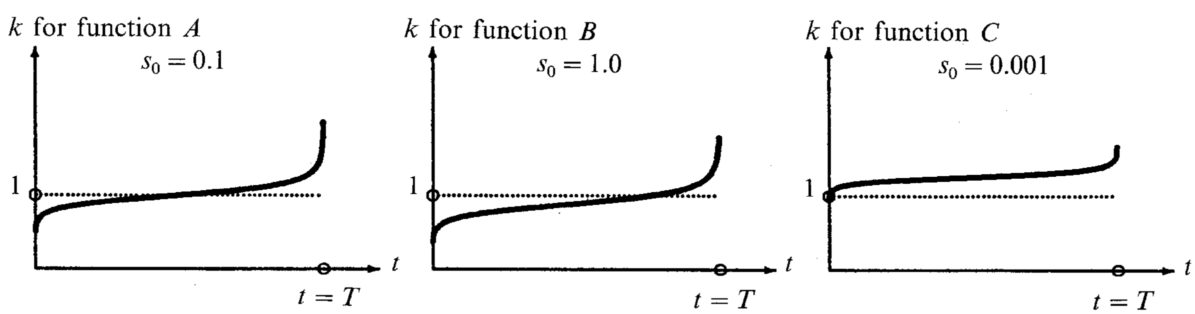

Fig. 2. *k values for functions A, B, and C: $\alpha = 0.1$, $p_1 = 0.05$, $p_2 = 0.1$.*

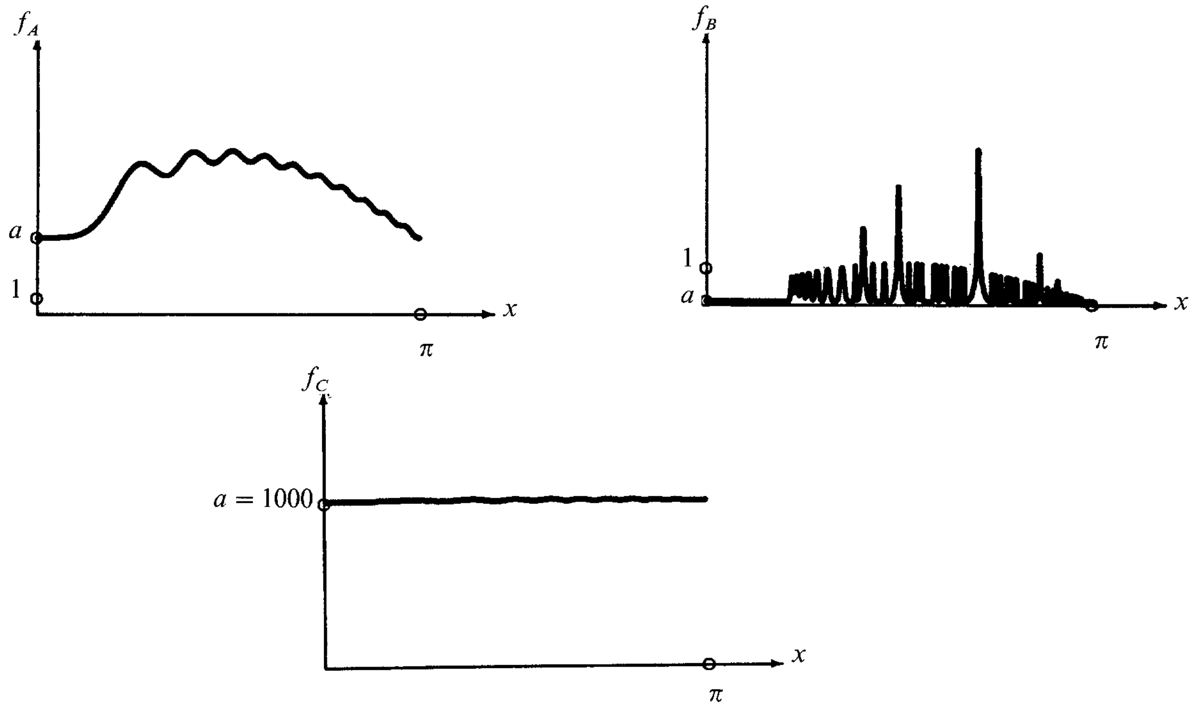

Fig. 3. *Cross-sections of functions A, B, and C along the $x_1 = \ldots = x_{10}$ plane.*

We used a traditional binary representation with 30 bits per variable; therefore, a chromosome was a vector of 260 binary bits for $\#elems = 10$. Moreover, the 30 bits gave us a precision of $\Delta x = \pi/2^{30} \simeq 3 \times 10^{-9}$. This error propagated to function error

$$\Delta f(\mathbf{x}) = \Delta x \sum_{i=1}^{\#\,elems} \frac{\sigma f}{\sigma x_i}$$

which, very pessimistically, can be approximated to

$$\Delta f(\mathbf{x}) = \Delta x \sum_{i=1}^{\#\,elems} (1 + 4qi) = \Delta x(10 + 220q)$$

This, in turn, translates to about 7×10^{-7} for functions A and C, and about 1.6×10^{-4} for function B.

During the runs we used one-bit mutations (0.001 rate on bits), single-point crossover (0.2 rate on chromosomes), and inversion (0.02 rate on chromosomes). The results are summarized in Table 1; they represent an average of 25 independent runs, each with population size of 40 and iterated 5000 times.

As expected, function B turned out to be the most difficult for the GA; its high span $s_0 \simeq 1.0$ caused many faulty local optima to be included in the solutions. This is also the case where our scaling mechanism gave best improvements, both in terms of faulty convergence and of the absolute magnitude of solution vectors. The most average function A also benefited from the mechanism in terms of both measures; this function has a usual span, but is still highly multimodal, difficult for any method. The smallest influence was observed on function C, the one with a very small span; such characteristics prohibited faulty peak selections in a natural way. The increased selective pressure in older populations, generated by higher k-values, slightly improved accuracy, while preserving high precision.

We have yet to conduct more systematic testing in order to optimize the various parameters used in our scaling

Table 1. *Output summary for the three test functions*

Function	The absolute optimum	$f^{\text{global}} - a$	Accuracy: GA without scaling	Accuracy: GA with scaling	Imprecision: GA without scaling	Imprecision: GA with scaling
A	14.70 413	9.70 413	98.94	99.62	0.365	0.298
B	9.75 332	9.65 332	92.16	94.35	1.528	1.194
C	1009.70 413	9.70 413	99.42	99.51	0.258	0.257

mechanism. Nevertheless, these results indicate its usefulness as an automatic problem analyser in cases where the user is not expected to perform such an analysis.

As to the increased computational complexity of calculating the k parameter, this was rather an insignificant change for two reasons. First, the span s_0 capturing the characteristics of the optimized function was calculated only once at iteration $t = 0$. Moreover, to increase the quality of this measure approximated by the final sampling, it was performed without the population size restriction; the number of such samples was set to 200. Second, the formula can be partially rewritten to account for the constant and incremental parts of it.

5. Fine local tuning

To improve the fine local tuning capabilities of a GA, which is a must for high-precision problems, we designed a special mutation operator whose performance is quite different from the traditional one. Recall that a traditional mutation changes one bit of a chromosome at a time; therefore, such a change uses only local knowledge—only the bit undergoing mutation is known. Such a bit, if located in the left portion of a sequence coding a variable, is very significant to the absolute magnitude of the mutation effect on the variable. On the other hand, bits far to the right of such a sequence have quite a smaller influence while mutated. We decided to use such positional global knowledge in the following way: as the population ages, bits located further to the right of each sequence coding one variable have an increased probability of being mutated, while those on the left have a smaller probability. In other words, such a mutation causes global search of the search space at the beginning of the iterative process, but only an increasingly local exploitation later on. We call this a *non-uniform mutation.*

Here we actually selected to use a floating point rather than binary representation for two reasons. First, floating point representation is much more natural for implementing the non-uniform mutation because of equivalency of problem and representation distances. Second, because of the fast propagation of errors in the iterative definitions of states (see the test case in Section 5.2), it would be necessary to use rather more than 30 bits per control state. This, in the presence of 45 coded variables, would create chromosomes over 2000 bits in length, leaving little hope of a resonable performance. A floating point representation requires appropriately different operators; the interested reader should refer to Janikow and Michalewicz (1990).

5.1 *The non-uniform operator*

The non-uniform mutation operator was introduced in two of our earlier papers (see Michalewicz and Janikow, 1990 and Janikow and Michalewicz, 1990). As mentioned, this is the operator responsible for the fine tuning capabilities in a numerical search space. It is defined as follows: if $\mathbf{s}_v^t = (v_1, \ldots, v_m)$ is a chromosome and the element v_k is selected for mutation (the domain of v_k is $[l_k, u_k]$), the result is a vector $\mathbf{s}_v^{t+1} = (v_1, \ldots, v_k', \ldots, v_m)$, with $k \in \{1, \ldots, n\}$,

$$v_k' = \begin{cases} v_k + \Delta(t, u_k - v_k) & \text{if a random digit is 0} \\ v_k - \Delta(t, v_k - l_k) & \text{if a random digit is 1} \end{cases}$$

The function $\Delta(t, y)$ returns a value in the range $[0, y]$ such that the probability of $\Delta(t, y)$ being close to 0 increases as t increases. This property causes this operator to search the space uniformly initially (when t is small), and very locally at later stages. We have used the following function:

$$\Delta(t, y) = y(1 - r^{(1 - t/T)^b}),$$

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.* Figure 4 displays the value of Δ for two selected times; this picture clearly indicates the behavior of the operator.

5.2 *The test case*

To indicate the usefulness of the non-uniform mutation operator we selected a dynamic control problem without constraints (for further results, especially for a successful comparison with a standard numerical optimization package, see Janikow and Michalewicz, 1990). The problem is a one-dimensional linear-quadratic model:

$$\min\left(qx_N^2 + \sum_{k=0}^{N-1} (sx_k^2 + ru_k^2)\right)$$

subject to

$$x_{k+1} = ax_k + bu_k, \quad k = 0, 1, \ldots, N - 1$$

where x_0 is a given initial state, a, b, q, s, r are given constants, $x_k \in \mathbb{R}$ is a state, and $\mathbf{u} \in \mathbb{R}^N$ is the sought control vector. The optimal value can be analytically expressed as

$$J^* = K_0 x_0^2$$

where K_k is the solution of the Riccati equation

$$K_k = s + ra^2 K_{k+1}/(r + b^2 K_{k+1})$$

and $K_N = q$.

*For the binary representation we were using $v_k' = \text{mutate}(v_k, \nabla(t, m_k))$ where mutate(v_k, pos) means mutate value of variable k on bit pos (0 is the least significant), m_k is the binary length of variable k, and

$$\nabla(t, m_k) = \begin{cases} \lfloor \Delta(t, m_k) \rfloor & \text{if a random digit is 0} \\ \lceil \Delta(t, m_k) \rceil & \text{if a random digit is 1} \end{cases}$$

with the b parameter of Δ adjusted appropriately if similar behavior desired.

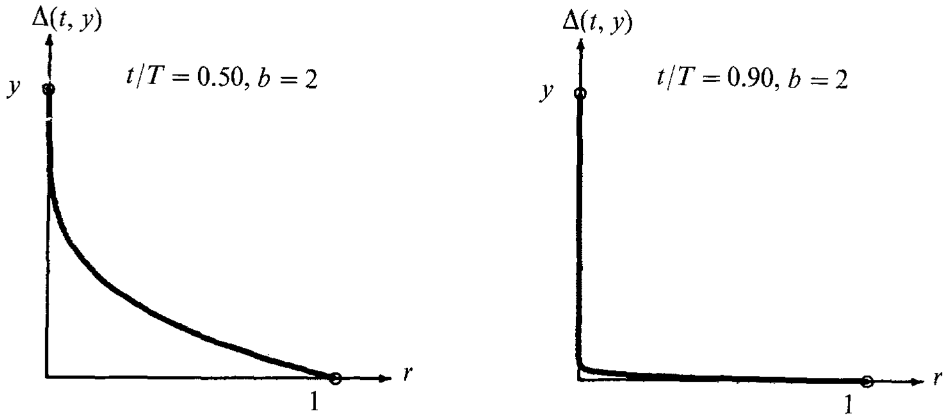

Fig. 4. *$\Delta(t, y)$ for two selected times.*

5.3 *Experiments and results*

The exact representation is as follows: for a problem of m variables, each chromosome, representing a permissible solution, is represented as a vector of m floating point numbers $\mathbf{s}_v^t = (v_1, \ldots, v_n)$ (when the generation number t and the chromosome number i are not important, we simply write s). The precision of such a representation is fixed for a given machine, and based on the precision of the floating point (or double, if needed) type.

We experimented with ten different cases, as defined in Table 2. For each, we repeated three separate runs of 40 000 generations and reported the best (with respect to final value) in Table 3. The same table also gives some intermediate results after a selected number of generations; for example, the values in the 10 000 column indicate the partial results after 10 000 generations, while running 40 000. It is important to note that such values are somehow worse than those obtained while running only 10 000 generations, due to the nature of the non-uniform opera-

Table 2. *The ten test cases for the dynamic control problem*

Case	N	x_0	s	r	q	a	b
1	45	100	1	1	1	1	1
2	45	100	10	1	1	1	1
3	45	100	1000	1	1	1	1
4	45	100	1	10	1	1	1
5	45	100	1	1000	1	1	1
6	45	100	1	1	0	1	1
7	45	100	1	1	1000	1	1
8	45	100	1	1	1	0.01	1
9	45	100	1	1	1	1	0.01
10	45	100	1	1	1	1	100

Table 3. *Results of the modified genetic algorithm on the dynamic control problem*

	Generations							
Case	1	100	1000	10 000	20 000	30 000	40 000	Factor
1	17 807.4	3.27 985	1.74689	1.61 866	1.61 825	1.61 804	1.61 803	10^4
2	13 670.4	5.33 177	1.45968	1.11 349	1.09 205	1.09 165	1.09 163	10^5
3	17 023.8	2.87 485	1.07974	1.00 968	1.00 126	1.00 104	1.00 103	10^7
4	15 077.3	8.64 310	3.75530	3.71 846	3.70 812	3.70 165	3.70 160	10^4
5	59 56.43	12.2559	2.89769	2.87 727	2.87 646	2.87 570	2.87 569	10^5
6	16 657.7	5.07 047	2.05314	1.61 869	1.61 830	1.61 806	1.61 806	10^4
7	288.0666	19.2684	7.02566	1.63 464	1.62 412	1.61 888	1.61 882	10^4
8	116.982	67.1758	1.92764	1.00 009	1.00 005	1.00 005	1.00 005	10^4
9	7.18 263	4.42 849	4.37093	4.31 504	4.31 024	4.31 004	4.31 004	10^5
10	987 0352	138 132	16096.0	1.38 244	1.00 041	1.00 010	1.00 010	10^4

Table 4. *Comparison of solutions for the linear-quadratic dynamic control problem*

Case	Exact solution value	GA w/ non-uniform mutation value	D (%)	GA w/o non-uniform mutation value	D (%)
1	16 180.3399	16 180.3939	0.000	16 234.3233	0.334
2	109 160.7978	109 163.0278	0.000	113 807.2444	4.257
3	10 009 990.0200	10 010 391.3989	0.004	10 128 951.4515	1.188
4	37 015.6212	37 016.0806	0.001	37 035.5652	0.054
5	287 569.3725	287 569.7389	0.000	298 214.4587	3.702
6	16 180.3399	16 180.6166	0.002	16 238.2135	0.358
7	16 180.3399	16 188.2394	0.048	17 278.8502	6.786
8	10 000.5000	10 000.5000	0.000	10 000.5000	0.000
9	431 004.0987	431 004.4092	0.000	431 610.9771	0.141
10	10 000.9999	10 001.0045	0.000	10 439.2695	4.380

Table 5. *Actual improvements on case 1*

GA	Generations 1	10	100	1000	10 000	40 000	CPU time
with non-uniform mutation	17 807 4000	293 564	32 798	17 468	16 186	16 180	719.4 sec*
without non-uniform mutation	17 243 100	415 623	59 872	17 931	17 445	16 234	719.1 sec*

*Times are similar since the GA without the special mutation performed other operations at a higher rate in order to achieve the same rate of breeding. For a comparison, a GA with a binary representation of 30 bits per variable used 12622 sec CPU for the same run (all runs reported from a DEC3100 station).

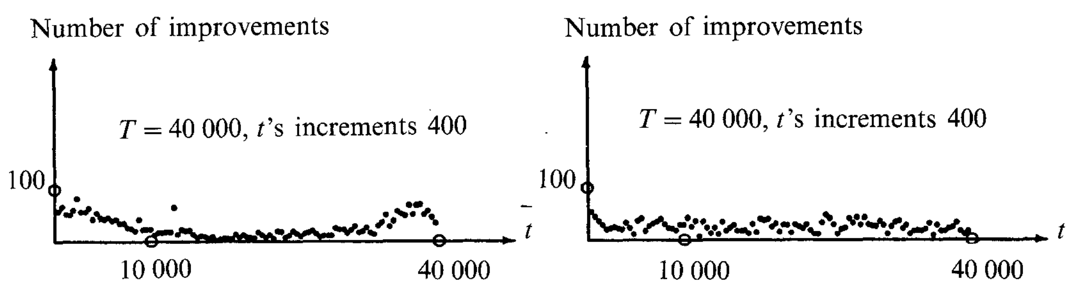

Fig. 5. *Number of improvements on case 1.*

tor. For all cases the population size was fixed at 100. The domain of all variables was set to $[-200, 200]$.

It is interesting to compare these results with the exact solutions as well as those obtained from another GA—exactly the same one but without the non-uniform mutation on. Table 4 summarizes the results; columns labeled D indicate the relative percentage errors. The GA using the non-uniform mutation clearly outperforms the other one with respect to the accuracy of the found optimal solution; while the enhanced GA was rarely in error by more than a few thousandths of 1%, the other one was hardly ever better than 1% out. Moreover, it also converged much faster to that solution (see Table 5 for case 1). In other words it has an additional advantage in time constrained situations.

Figure 5 illustrates the non-uniform mutation's effect on the evolutionary process. The new mutation causes quite an increase in the number of improvements observed in the population at the end of the population's life. Moreover, the smaller number of such improvements prior to that time, together with an actually faster convergence, clearly indicates a better overall search.

In other publications (see Michalewicz *et al*, 1990 and Janikow and Michalewicz, 1990) we present more experiments with other dynamic control problems: these prove the superiority of genetic approaches over commercial optimization packages as well.

6. Constraints handling

The proposed methodology for handling linear constraints by genetic algorithms (see Michalewicz and Janikow, 1991) combines some previous ideas, but in a totally new way and context. The main idea lies in a careful elimination of the equation constraints, and designing *dynamic* operators preserving the inequality constraints. Both types of constraint can be processed very efficiently if they contain only linear equations; so constraints problems include many of the interesting optimization cases.

Linear constraints are of two types: equalities and inequalities (the latter include all variables' domains). We first eliminate all the equalities, reducing the number of variables and modifying the inequalities; reducing the set of variables both decreases the length of the representation and reduces the search space. Left with only linear inequalities, we deal with the convex search space which, in the presence of dynamic and closed operators, can be searched without explicitly considering the constraints. The problem then becomes one of designing such closed operators. We achieve this by defining them as being context-dependent, that is, dynamically adjusting to the current context.

The set of equalities can be eliminated (on a one by one basis) as follows: first transform an equality so that one of its variables is expressed in terms of the others, and then substitute all occurrences of this variable by such an expression, in all remaining equalities and all inequalities. A detailed description of this approach, along with theoretical foundations, can be found in Michalewicz and Janikow (1991). Here, we provide only an example.

Suppose that we wish to minimize a function of six variables:

$$f(x_1, x_2, x_3, x_4, x_5, x_6)$$

subject to the following constraints:

$$x_1 + x_2 + x_3 = 5$$

$$x_4 + x_5 + x_6 = 10$$

$$x_1 + x_4 = 3$$

$$x_2 + x_5 = 4$$

$$x_1 \geq 0, \quad x_2 \geq 0, \quad x_3 \geq 0, \quad x_4 \geq 0, \quad x_5 \geq 0, \quad x_6 \geq 0$$

We can take advantage of the presence of four independent equations and express four variables as functions of the remaining two:

$$x_3 = 5 - x_1 - x_2$$

$$x_4 = 3 - x_1$$

$$x_5 = 4 - x_2$$

$$x_6 = 3 + x_1 + x_2$$

Thus, we have reduced the original problem to the optimization problem of two variables x_1 and x_2:

$$g(x_1, x_2) = f(x_1, x_2, (5 - x_1 - x_2), (3 - x_1), (4 - x_2), (3 + x_1 + x_2))$$

subject to the following constraints (inequalities only):

$$x_1 \geq 0, \quad x_2 \geq 0$$

$$5 - x_1 - x_2 \geq 0$$

$$3 - x_1 \geq 0$$

$$4 - x_2 \geq 0$$

$$3 + x_1 + x_2 \geq 0$$

These inequalities can be further reduced to:

$$0 \leq x_1 \leq 3$$

$$0 \leq x_2 \leq 4$$

$$x_1 + x_2 \leq 5$$

Now, given a chromosome (a point within the constrained solution space), any operator must produce a new feasible solution (this is what we mean by the *closedness* of operators). This can be achieved by working within the current context; e.g. if $\mathbf{x} = (1.8, 2.3)$ is to be mutated on x_1, then the new value of this variable must be taken from the range $[0, 5 - x_2] = [0, 2.7]$. This signifies another fact: it is again much easier to define such operators on a floating point representation, as it is not enough to deal locally with one bit at a time.

6.1 *The dynamic operators*

The genetic operators are dynamic, i.e. the value of a vector component depends on the remaining values of the vector.

The value of the ith component of a feasible solution $\mathbf{s} = (v_1, \ldots, v_m)$ is always in some (dynamic) range $[l, u]$; the bounds l and u depend on the other vector's values $v_1, \ldots, v_{i-1}, v_{i+1}, \ldots, v_m$, and the set of inequalities. We say that the ith component (ith gene) of the vector $\mathbf{s}$ is *movable* if $l < u$.

Before we describe the operators, we present two important characteristics of convex spaces (due to the linearity of the constraints, the solution space is always a convex space $\mathcal{S}$), which play an essential role in the definition of these operators:

(1) For any two points $\mathbf{s}_1$ and $\mathbf{s}_2$ in the solution space $\mathcal{S}$, the linear combination $a\mathbf{s}_1 + (1-a)\mathbf{s}_2$, where $a \in [0, 1]$, is a point in $\mathcal{S}$.

(2) For every point $\mathbf{s}_0 \in \mathcal{S}$ and any line p such that $\mathbf{s}_0 \in p$, p intersects the boundaries of $\mathcal{S}$ at precisely two points, say $\mathbf{l}_p^{s_0}$ and $\mathbf{u}_p^{s_0}$.

Since we are only interested in lines parallel to each axis, to simplify the notation we denote by $l_{(i)}^s$ and $u_{(i)}^s$ the ith components of the vectors $\mathbf{l}_p^s$ and $\mathbf{u}_p^s$, respectively, where the line p is parallel to the axis i. We assume further that $l_{(i)}^s \le u_{(i)}^s$.

Because of intuitional similarities, we cluster the operators into the standard two classes: mutation and crossover. The proposed crossover and mutation operators use the two properties to ensure that the offspring of a point in the convex solution space $\mathcal{S}$ belongs to $\mathcal{S}$. For a detailed discussion on these topics, the reader is referred to Michalewicz and Janikow (1990).

6.1.1 *Mutation group*

Mutations are quite different from the traditional mutation operator with respect to both the actual mutation (a gene, being a floating point number, is mutated in a dynamic range) and the selection of an applicable gene. A traditional mutation is performed on static domains for all genes. In such a case the order of possible mutations on a chromosome does not influence the outcome. This is no longer true with the dynamic domains. To solve the problem we proceed as follows: we randomly select $p_{\text{um}} \times$ *pop_size* chromosomes for uniform mutation, $p_{\text{bm}} \times$ *pop_size* chromosomes for boundary mutation, and $p_{\text{nm}} \times$ *pop_size* chromosomes for non-uniform mutation (all with possible repetitions), where p_{um}, p_{bm}, and p_{nm} are probabilities of the three mutations defined below. Then, we perform these mutations in a random fashion on the selected chromosome:

(1) *Uniform mutations.* For this mutation we select a random gene k (from the set of movable genes of the given chromosome s determined by its current context). If $\mathbf{s}_v^t = (v_1, \ldots, v_m)$ is a chromosome and the kth component is the selected gene, the result is a vector $\mathbf{s}_v^{t+1} = (v_1, \ldots, v_k', \ldots, v_m)$, where v_k' is a random value (uniform probability distribution) from the range $[l_{(k)}^{s_v^t}, u_{(k)}^{s_v^t}]$. The dynamic values $l_{(k)}^{s_v^t}$ and $u_{(k)}^{s_v^t}$ are easily calculated from the set of constraints (inequalities);

(2) *Boundary mutation.* This is a variation of the uniform mutation with v_k' being either $l_{(k)}^{s_v^t}$ or $u_{(k)}^{s_v^t}$, each with equal probability;

(3) *(Dynamic) non-uniform mutation.* This is a version of the operator defined in Section 5.1; the only change relates to the dynamic domains:

$$v_k' = \begin{cases} v_k + \Delta(t, u_{(k)}^{s_v^t} - v_k) & \text{if a random digit is 0} \\ v_k + \Delta(t, v_k - l_{(k)}^{s_v^t}) & \text{if a random digit is 1} \end{cases}$$

6.1.2 *Crossover group*

Chromosomes are randomly selected in pairs for application of the crossover operators according to appropriate probabilities. The operators defined here are versions of the ones outlined in Janikow and Michalewicz (1990) for floating point crossover, but adjusted to our dynamic domains:

(1) *Simple crossover.* This is defined as follows: if $\mathbf{s}_v^t = (v_1, \ldots, v_m)$ and $\mathbf{s}_w^t = (w_1, \ldots, w_m)$ are crossed after the kth position, the resulting offspring are: $\mathbf{s}_v^{t+1} = (v_1, \ldots, v_k, w_{k+1}, \ldots, w_m)$ and $\mathbf{s}_w^{t+1} = (w_1, \ldots, w_k, v_{k+1}, \ldots, v_m)$. Note the the only permissible split points are between individual floating points (using float representation it is impossible to split anywhere else).

However, such an operator may produce offspring outside the convex solution space $\mathcal{S}$. To avoid this problem, we use the property of the convex spaces, which says that there exists $a \in [0, 1]$ such that

$$\mathbf{s}_v^{t+1} = (v_1, \ldots, v_k, w_{k+1}a + v_{k+1}(1-a), \ldots, w_m a + v_m(1-a)) \in \mathcal{S}$$

and

$$\mathbf{s}_w^{t+1} = (w_1, \ldots, w_k, v_{k+1}a + w_{k+1}(1-a), \ldots, v_m a + w_m(1-a)) \in \mathcal{S}$$

The only question still to be answered is how to find the largest a to obtain the greatest possible information exchange: due to the real interval, we cannot perform an extensive search. We implemented a binary search (to some depth only for efficiency). Then, a takes the largest appropriate value found, or 0 if no value satisfied the constraints. The necessity for such actions is small in general and decreases rapidly over the life of the population. Note that the value of a is determined separately for each single arthmetical crossover and each gene.

(2) *Single arithmetical crossover.* This is defined as follows: if $\mathbf{s}_v^t = (v_1, \ldots, v_m)$ and $\mathbf{s}_w^t = (w_1, \ldots, w_m)$ are to be crossed, the resulting offspring are $\mathbf{s}_v^{t+1} = (v_1, \ldots, v_k', \ldots, v_m)$ and $\mathbf{s}_w^{t+1} = (w_1, \ldots, w_k', \ldots, w_m)$, where $k \in [1, m]$, $v_k' = aw_k + (1-a)v_k$, and $w_k' = av_k + (1-a)w_k$. Here, a is a dynamic parameter calculated in the given context (vectors $\mathbf{s}_v$, $\mathbf{s}_w$) so that the operator is closed (points $\mathbf{s}_v^{t+1}$ and $\mathbf{s}_w^{t+1}$ are in the convex constrained space $\mathcal{S}$). Actually, a is a random

choice from the following range:

$$a \in \begin{cases} \left[\max\left(\dfrac{l^{s_w}_{(k)} - w_k}{v_k - w_k}, \dfrac{u^{s_v}_{(k)} - v_k}{w_k - v_k}\right), \min\left(\dfrac{l^{s_v}_{(k)} - v_k}{w_k - v_k}, \dfrac{u^{s_w}_{(k)} - w_k}{v_k - w_k}\right)\right] & \text{if } v_k > w_k \\ [0, 0] & \text{if } v_k = w_k \\ \left[\max\left(\dfrac{l^{s_v}_{(k)} - v_k}{w_k - v_k}, \dfrac{u^{s_w}_{(k)} - w_k}{v_k - w_k}\right), \min\left(\dfrac{l^{s_w}_{(k)} - w_k}{v_k - w_k}, \dfrac{u^{s_v}_{(k)} - v_k}{w_k - v_k}\right)\right] & \text{if } v_k < w_k \end{cases}$$

To increase the applicability of this operator (to ensure that a will be non-zero, which actually always nullifies the results of the operator) it is wise to select the applicable gene as a random choice from the intersection of movable genes of both chromosomes. Note again that the value of a is determined separately for each single arithmetical crossover and each gene.

(3) *Whole arithmetical crossover*. This is defined as a linear combination of two vectors: if $\mathbf{s}_v^t$ and $\mathbf{s}_w^t$ are to be crossed, the resulting offspring are $\mathbf{s}_v^{t+1} = a\mathbf{s}_w^t + (1-a)\mathbf{s}_u^t$ and $\mathbf{s}_w^{t+1} = a\mathbf{s}_v^t + (1-a)\mathbf{s}_w^t$. This operator uses a simpler system parameter $a \in \{0, 1\}$ as it always guarantees closedness (according to characteristic (1) of this section).

6.2 *The test case*

We selected the following problem of 49 variables: minimize

$$f(\mathbf{x}) = \sum_{i=1}^{49} g(x_i)$$

where

$$g(x_i) = \begin{cases} 0, & \text{if } 0 \le x_i \le 2 \\ c_i, & \text{if } 2 < x_i \le 4 \\ 2c_i, & \text{if } 4 < x_i \le 6 \\ 3c_i, & \text{if } 6 < x_i \le 8 \\ 4c_i, & \text{if } 8 < x_i \le 10 \\ 5c_i, & \text{if } 10 < x_i \end{cases}$$

with parameters c_i as given in Table 6 and subject to the following equality constraints:

$$\begin{aligned} x_1 + x_2 + x_3 + x_4 + x_5 + x_6 + x_7 &= 27 \\ x_8 + x_9 + x_{10} + x_{11} + x_{12} + x_{13} + x_{14} &= 28 \\ x_{15} + x_{16} + x_{17} + x_{18} + x_{19} + x_{20} + x_{21} &= 25 \\ x_{22} + x_{23} + x_{24} + x_{25} + x_{26} + x_{27} + x_{28} &= 20 \\ x_{29} + x_{30} + x_{31} + x_{32} + x_{33} + x_{34} + x_{35} &= 20 \\ x_{36} + x_{37} + x_{38} + x_{39} + x_{40} + x_{41} + x_{42} &= 20 \\ x_{43} + x_{44} + x_{45} + x_{46} + x_{47} + x_{48} + x_{49} &= 20 \end{aligned}$$

$$\begin{aligned} x_1 + x_8 + x_{15} + x_{22} + x_{29} + x_{36} + x_{43} &= 20 \\ x_2 + x_9 + x_{16} + x_{23} + x_{30} + x_{37} + x_{44} &= 20 \\ x_3 + x_{10} + x_{17} + x_{24} + x_{31} + x_{38} + x_{45} &= 20 \\ x_4 + x_{11} + x_{18} + x_{25} + x_{32} + x_{39} + x_{46} &= 23 \\ x_5 + x_{12} + x_{19} + x_{26} + x_{33} + x_{40} + x_{47} &= 26 \\ x_6 + x_{13} + x_{20} + x_{27} + x_{34} + x_{41} + x_{48} &= 25 \\ x_7 + x_{14} + x_{21} + x_{28} + x_{35} + x_{42} + x_{49} &= 26 \end{aligned}$$

Given that, $\forall i,\ x_i \ge 0$, we also have additional implicit domain inequalities, given in Table 7.

The above is actually an example of a transportation problem with a reasonable stepwise cost function; for more on that the reader is referred to Michalewicz, Vignaux and Hobbs (1991) and Vignaux and Michalewicz (1990).

Table 6. *Parameters c_i*

$c_1 = 0$	$c_2 = 21$	$c_3 = 50$	$c_4 = 62$	$c_5 = 93$	$c_6 = 77$	$c_7 = 1000$
$c_8 = 21$	$c_9 = 0$	$c_{10} = 17$	$c_{11} = 54$	$c_{12} = 67$	$c_{13} = 1000$	$c_{14} = 48$
$c_{15} = 50$	$c_{16} = 17$	$c_{17} = 0$	$c_{18} = 60$	$c_{19} = 98$	$c_{20} = 67$	$c_{21} = 25$
$c_{22} = 62$	$c_{23} = 54$	$c_{24} = 60$	$c_{25} = 0$	$c_{26} = 27$	$c_{27} = 1000$	$c_{28} = 38$
$c_{29} = 93$	$c_{30} = 67$	$c_{31} = 98$	$c_{32} = 27$	$c_{33} = 0$	$c_{34} = 47$	$c_{35} = 42$
$c_{36} = 77$	$c_{37} = 1000$	$c_{38} = 67$	$c_{39} = 1000$	$c_{40} = 47$	$c_{41} = 0$	$c_{42} = 35$
$c_{43} = 1000$	$c_{44} = 48$	$c_{45} = 25$	$c_{46} = 38$	$c_{47} = 42$	$c_{48} = 35$	$c_{49} = 0$

Table 7. *Upper bounds for variables x_i*

$x_1 \le 20$	$x_2 \le 20$	$x_3 \le 20$	$x_4 \le 23$	$x_5 \le 26$	$x_6 \le 25$	$x_7 \le 26$
$x_8 \le 20$	$x_9 \le 20$	$x_{10} \le 20$	$x_{11} \le 23$	$x_{12} \le 26$	$x_{13} \le 25$	$x_{14} \le 26$
$x_{15} \le 20$	$x_{16} \le 20$	$x_{17} \le 20$	$x_{18} \le 23$	$x_{19} \le 25$	$x_{20} \le 25$	$x_{21} \le 25$
$x_{22} \le 20$	$x_{23} \le 20$	$x_{24} \le 20$	$x_{25} \le 20$	$x_{26} \le 20$	$x_{27} \le 20$	$x_{28} \le 20$
$x_{29} \le 20$	$x_{30} \le 20$	$x_{31} \le 20$	$x_{32} \le 20$	$x_{33} \le 20$	$x_{34} \le 20$	$x_{35} \le 20$
$x_{36} \le 20$	$x_{37} \le 20$	$x_{38} \le 20$	$x_{39} \le 20$	$x_{40} \le 20$	$x_{41} \le 20$	$x_{42} \le 20$
$x_{43} \le 20$	$x_{44} \le 20$	$x_{45} \le 20$	$x_{46} \le 20$	$x_{47} \le 20$	$x_{48} \le 20$	$x_{49} \le 20$

There are 13 independent and one dependent equations here; therefore, we eliminate thirteen variables: $x_1, \ldots, x_8, x_{15}, x_{22}, x_{36}, x_{44}$. All remaining variables are renamed $y_1, \ldots, y_{36}$ preserving order, i.e. $y_1 = x_9$, $y_2 = x_{10}, \ldots, y_6 = x_{14}, y_7 = x_{16}, \ldots, y_{36} = x_{49}$. Each of these variables has to satisfy four two-sided inequalities, which result from the initial domains and our transformations. Now, each chromosome is a float vector $(y_1, \ldots, y_{36})$.

6.3 *Experiments and results*

For comparative experiments we planned on using a standard GA approach to constraints by penalties (see e.g. Richardson *et al*, 1989) and a version (the student version) of GAMS, a package for the construction and solution of large and compex mathematical programming models (Brooke *et al*, 1988); we used the MINOS version of the optimizer. However, we did not succeed in the former: the major issue in using the penalty functions approach is assigning weights to the constraints—these weights play the role of penalties if a potential solution does not satisfy them. In experiments with the above problem the evaluation function Eval was composed of the optimization function f and the penalty P:

$$\text{Eval}(\mathbf{x}) = f(\mathbf{x}) + P$$

For our experiments we used a suggestion (see Richardson *et al*, 1989) to start with relaxed penalties and to tighten them as the run progresses. We used

$$P = k \times \left(\frac{t}{T}\right)^p \times \bar{f} \times \Sigma_{i=1}^{14} d_i$$

where $\bar{f}$ is the average fitness of the population at the given generation, t, k and p are parameters, T is the maximum number of generations, and d_i returns the 'degree of constraint violation'. For example, for a constraint:

$$\Sigma_{i \in W} x_i = val, \quad W \subseteq \{1, \ldots, 49\},$$

and a chromosome $(v_1, \ldots, v_{49})$, the penalty (degree of constraint violation) d_i was

$$d_i = \left|\Sigma_{i \in W} v_i - val\right|.$$

We experimented with various values of p (close to 1), k (close to $\frac{1}{14}$, where 14 is total number of equality constraints—the static domain constraints were naturally satisfied by a proper representation), and $T = 8000$. However, this method did not lead to feasible solutions: in over 1200 runs (with different random number generator seeds and various values for parameters k and p) the best chromosomes (after 8000 generations) violated significantly at least three constraints. For example, very often the algorithm converged to a solution where the numbers on one diagonal were equal to 20 and all others were zeros:

$$x_1 = x_9 = x_{17} = x_{25} = x_{33} = x_{41} = x_{49} = 20$$

$$x_i = 0 \text{ for other values of } i$$

As to GAMS, which only works with continuous functions, we reimplemented the problem using arctangent functions to approximate each of the five steps. A parameter P_A was used to control the 'tightness' of the fit:

$$g(x_i) = c_i \begin{pmatrix} \arctan(P_A(x_i - 2))/\pi + \frac{1}{2} + \\ \arctan(P_A(x_i - 4))/\pi + \frac{1}{2} + \\ \arctan(P_A(x_i - 6))/\pi + \frac{1}{2} + \\ \arctan(P_A(x_i - 8))/\pi + \frac{1}{2} + \\ \arctan(P_A(x_i - 10))/\pi + \frac{1}{2} \end{pmatrix}$$

This method allowed the system to find a feasible solution, but one much worse than those of our GA (Tables 8 and 9); the GA found an optimal value of 24.15 while the other system found one of 96.00.

The results indicate that the usefulness of our method in the presence of many constraints and its superiority over some standard systems on non-trivial problems. Our modified genetic algorithm was run for 8000 iterations, with a population size of 40. The parameters used in all runs are displayed in Table 10. A single run of 8000 iterations took 2:28 CPU sec on a Cray Y-MP. GAMS was run on an Olivetti 386 with a math coprocessor and a single run took 1:20 sec.

7. Conclusions

In this paper we discussed the use of genetic algorithms for numerical optimization problems. In particular, we concentrated on various modifications of classical genetic

Table 8. *Solution found by our GA: f(x) = 24.15*

$x_1 = 20.00$	$x_2 = 0.00$	$x_3 = 0.00$	$x_4 = 1.93$	$x_5 = 1.63$	$x_6 = 1.47$	$x_7 = 1.97$
$x_8 = 0.00$	$x_9 = 20.00$	$x_{10} = 2.88$	$x_{11} = 1.76$	$x_{12} = 1.47$	$x_{13} = 1.89$	$x_{14} = 0.00$
$x_{15} = 0.00$	$x_{16} = 0.00$	$x_{17} = 17.12$	$x_{18} = 1.90$	$x_{19} = 1.99$	$x_{20} = 1.10$	$x_{21} = 2.89$
$x_{22} = 0.00$	$x_{23} = 0.00$	$x_{24} = 0.00$	$x_{25} = 16.26$	$x_{26} = 0.85$	$x_{27} = 1.38$	$x_{28} = 1.51$
$x_{29} = 0.00$	$x_{30} = 0.00$	$x_{31} = 0.00$	$x_{32} = 0.00$	$x_{33} = 19.65$	$x_{34} = 0.00$	$x_{35} = 0.35$
$x_{36} = 0.00$	$x_{37} = 0.00$	$x_{38} = 0.00$	$x_{39} = 0.43$	$x_{40} = 0.41$	$x_{41} = 19.16$	$x_{42} = 0.00$
$x_{43} = 0.00$	$x_{44} = 0.00$	$x_{45} = 0.00$	$x_{46} = 0.72$	$x_{47} = 0.00$	$x_{48} = 0.00$	$x_{49} = 19.28$

Table 9. *Solution found by GAMS: $f(\mathbf{x}) = 96.00$*

$x_1 = 20.00$	$x_2 = 1.29$	$x_3 = 0.95$	$x_4 = 1.58$	$x_5 = 1.52$	$x_6 = 1.58$	$x_7 = 0.08$
$x_8 = 0.00$	$x_9 = 18.71$	$x_{10} = 0.39$	$x_{11} = 1.59$	$x_{12} = 1.58$	$x_{13} = 0.12$	$x_{14} = 5.61$
$x_{15} = 0.00$	$x_{16} = 0.00$	$x_{17} = 18.66$	$x_{18} = 1.56$	$x_{19} = 1.47$	$x_{20} = 1.59$	$x_{21} = 1.72$
$x_{22} = 0.00$	$x_{23} = 0.00$	$x_{24} = 0.00$	$x_{25} = 18.27$	$x_{26} = 1.25$	$x_{27} = 0.00$	$x_{28} = 0.48$
$x_{29} = 0.00$	$x_{30} = 0.00$	$x_{31} = 0.00$	$x_{32} = 0.00$	$x_{33} = 19.47$	$x_{34} = 0.53$	$x_{35} = 0.00$
$x_{36} = 0.00$	$x_{37} = 0.00$	$x_{38} = 0.00$	$x_{39} = 0.00$	$x_{40} = 0.00$	$x_{41} = 20.00$	$x_{42} = 0.00$
$x_{43} = 0.00$	$x_{44} = 0.00$	$x_{45} = 0.00$	$x_{46} = 0.00$	$x_{47} = 0.71$	$x_{48} = 1.18$	$x_{49} = 18.11$

Table 10. *Parameters used for the 7 × 7 transportation problem*

Parameter	Value
pop_size	40
$prop\_mut_{um}$	0.08
$prob\_mut_{bm}$	0.03
$prob\_mut_{nm}$	0.07
$prob\_cross_{sc}$	0.10
$prob\_cross_{sa}$	0.10
$prob\_cross_{wa}$	0.10
a	0.25
b	2.0

Key: population size (*pop_size*), probability of uniform mutation ($prob\_mut_{um}$), probability of boundary mutation ($prob\_mut_{bm}$), probability of non-uniform mutation ($prob\_mut_{nm}$), probability of simple crossover ($prob\_cross_{sc}$), probability of single arithmetical crossover ($prob\_cross_{sa}$), probability of whole arithmetical crossover ($prob\_cross_{wa}$), coefficient a for the whole arithmetical crossover, coefficient b for Δ of the non-uniform mutation.

algorithms to overcome three major problems: handling of constraints, premature convergence, and local fine tuning. The preliminary results of several experiments are more than encouraging and suggest that the methods are very useful. They may lead to the solution of some difficult operations research problems. We are currently in the process of implementing a single system to bring all the above ideas together. When completed, the system will be compared with many software optimization packages on different functions with non-trivial constraints. The system should be able to deal with very complex problems (thousands of variables, hundreds of constraints) since we need to represent only a relatively small population of potential solution vectors and apply new genetic operators to them.

Also, further extensions are planned to handle discrete variables (integer, boolean, nominal) and to handle some classes of non-linear constraints.

References

Ackley, D. H. (1987) An empiricial study of bit vector function optimization, in *Genetic Algorithms and Simulated Annealing*, Davis, L. (ed), Pitman, London, pp. 170–204.

Antonisse, J. (1989) A new interpretation of schema notation that overturns the binary encoding constraint, in *Proceedings of the Third International Conference on Genetic Algorithms*, J. Schaffer (ed.), Morgan Kaufmann, San Mateo, California, pp. 86–91.

Antonisse, H. J. and Keller, K. S. (1987) Genetic operators for high level knowledge representation, in *Proceedings of the Second International Conference on Genetic Algorithms*, MIT, Cambridge, Mass., J. J. Grenfenstette (ed.), Lawrence Erlbaum, Hillsdale, New Jersey.

Bellman, R. (1957) *Dynamic Programming*, Princeton University Press, Princeton, NJ.

Bethke, A. D. (1980) Genetic algorithms as function optimizers. PhD dissertation, University of Michigan.

Bertsekas, D. P. (1987) *Dynamic Programming. Deterministic and Stochastic Models*, Prentice Hall, Englewood Cliffs, NJ.

Booker, L. B. (1982) Intelligent behavior as an adaptation to the task environment, PhD dissertation, University of Michigan.

Booker, L. B. (1987) Improving search in genetic algorithms, in *Genetic Algorithms and Simulated Annealing*, L. Davis (ed.), Pitman, London, pp. 61–73.

Bosworth, J., Foo, N. and Zeigler, B. P. (1972) *Comparison of Genetic Algorithms with Conjugate Gradient Methods* (CR-2093), Washington, DC: National Aeronautics and Space Administration.

Brindle, A. (1981) Genetic algorithms for function optimization. PhD dissertation, University of Alberta, Edmonton.

Brooke, A., Kendrick, D., Meeraus, A. (1988) *GAMS: A User's Guide*, The Scientific Press, Redwood City, California.

Davis, L. (ed.) (1987) *Genetic Algorithms and Simulated Annealing*, Pitman, London.

DeJong, K. A. (1975) An analysis of the behaviour of a class of genetic adaptive systems. PhD dissertation, University of Michigan.

DeJong, K. A. (1985) Genetic algorithms: a 10 year perspective, in *Proceedings of the First International Conference on Genetic*

Algorithms, Pittsburgh, 24–26 July, J. J. Grefenstette (ed.), Lawrence Erlbaum, Hillsdale, New Jersey.

Dhar, V. and Ranganathan, N. (1990) Integer programming vs. expert systems: an experimental comparison. *Communications of ACM*, **33**, 323–336.

Goldberg, D. E. (1989) *Genetic Algorithms in Search, Optimization and Machine Learning*, Addison-Wesley, Reading, Massachusetts.

Grefenstette, J. J. (1985) *Proceedings of the First International Conference on Genetic Algorithms*, Pittsburgh, 24–26 July, Lawrence Erlbaum, Hillsdale, New Jersey.

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. (1987a) Incorporating problem specific knowledge into genetic algorithms, in *Genetic Algorithms and Simulated Annealing*, L. Davis (ed.), Pitman, London, pp. 42–60.

Grefenstette, J. J. (1987b) *Proceedings of the Second International Conference on Genetic Algorithms*, MIT, Cambridge, Mass., 28–31 July, Lawrence Erlbaum, Hillsdale, New Jersey.

Groves, L., Michalewicz, Z., Elia, P. and Janikow, C. (1990) Genetic algorithms for drawing directed graphs, in *Proceedings of the Fifth International Symposium on Methodologies of Intelligent Systems*, Knoxville, 25–27 October, pp. 268–276.

Holland, J. (1959) A universal computer capable of executing an arbitrary number of sub-programs simultaneously, in *Proceedings of the 1959 EJCC*, pp. 108–113.

Holland, J. (1975) *Adaptation in Natural and Artificial Systems*, University of Michigan Press, Ann Arbor.

Janikow, C. and Michalewicz, Z. (1990) Specialized genetic algorithms for numerical optimization problems, in *Proceedings of the International Conference on Tools for AI*, Washington, 6–9 November, pp. 798–804.

Janikow, C. and Michalewicz, Z. (1991) Convergence problem in genetic algorithms, submitted for publication.

Kirkpatrick, S., Gellatt, C. and Vecchi, M. (1983) Optimization by simulated annealing, *Science*, **220**(4598), 671.

Laarhoven, P. J. M. van and Aarts, E. H. L. (1987) *Simulated Annealing: Theory and Applications*, D. Reidel, Dordrecht.

Michalewicz, Z., Vignaux, G. A. and Groves, L. (1989) Genetic algorithms for optimization problems, in *Proceedings of the 11th NZ Computer Conference*, Wellington, New Zealand, 16–18 August, pp. 211–223.

Michalewicz, Z., Vignaux, G. A. and Hobbs, M. (1991) A genetic algorithm for the nonlinear transportation problem, *ORSA Journal on Computing*, **3**(4).

Michalewicz, Z. and Janikow, C. (1991) GENOCOP: a genetic algorithm for numerical optimization problems with constraints, to appear in *Communications of ACM*.

Michalewicz, Z., Jankowski, A., Vignaux, G. A. (1990) The constraints problem in genetic algorithms, in *Methodologies for Intelligent Systems: Selected Papers*, M. L. Emrich, M. S. Phifer, B. Huber, M. Zemankova and Z. W. Ras (eds), Proceedings of the Fifth International Symposium on Methodologies of Intelligent Systems, Knoxville, 25–27 October, pp. 142–157.

Michalewicz, Z., Kazemi, M., Krawczyk, J. and Janikow, C. (1990) On dynamic control problem, *Proceedings of the 29th IEEE Conference on Decision and Control*, Honolulu, 5–7 December.

Neumann, J. von (1966) *Theory of Self-Reproducing Automata*, Burks (ed.), University of Illinois Press.

Richardson, J. T., Palmer, M. R., Liepins, G. and Hilliard, M. (1989) Some guidelines for genetic algorithms with penalty functions, in *Proceedings of the Third International Conference on Genetic Algorithms*, 4–7 June, J. Schaffer (ed.), Morgan Kaufmann, London.

Sasieni, M., Yaspan, A. and Friedman, L. (1959) *Operations Research Methods and Problems*, John Wiley and Sons.

Schaffer, J. (1989) *Proceedings of the Third International Conference on Genetic Algorithms*, 4–7 June, Morgan Kaufmann, Redwood City, California.

Schaffer, J., Caruana, R., Eshelman, L. and Das, R. (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*, 4–7 June, J. Schaffer (ed.), Morgan Kaufmann, pp. 51–60.

Sirag, D. J. and Weisser, P. T. (1987) Toward a unified thermodynamic genetic operator, in *Proceedings of the Second International Conference on Genetic Algorithms*, 28–31 July, Lawrence Erlbaum, Hillsdale, New Jersey.

Taha, H. A. (1982) *Operations Research: An Introduction*, Collier Macmillan, London.

Ulam, S. (1949) On the Monte Carlo Method, *Proceedings of the 2nd Symposium on Large Scale Digital Calculating Machinery*, Harvard University Press, pp. 207–212.

Vignaux, G. A. and Michalewicz, Z. (1989) Genetic algorithms for the transporation problem, *Proceedings of the 4th International Symposium on Methodologies for Intelligent Systems*, Charlotte, NC, 12–14 October, pp. 252–259.

Vignaux, G. A. and Michalewicz, Z. (1990) A genetic algorithm for the linear transportation problem. *IEEE Transactions on Systems, Man, and Cybernetics*, **21**(2).