Heuristic Methods for Evolutionary Computation Techniques

Zbigniew Michalewicz

Department of Computer Science
University of North Carolina
Charlotte, NC 28223, USA
and
Institute of Computer Science
Polish Academy of Sciences
ul. Ordona 21
01-237 Warsaw, Poland

ph. (704) 547-4873
fax (704) 547-3516
e-mail: zbyszek@uncc.edu

Abstract

Evolutionary computation techniques, which are based on a powerful principle of evolution: survival of the fittest, constitute an interesting category of heuristic search. In other words, evolutionary techniques are stochastic algorithms whose search methods model some natural phenomena: genetic inheritance and Darwinian strife for survival.

Any evolutionary algorithm applied to a particular problem must address the issue of genetic representation of solutions to the problem and genetic operators that would alter the genetic composition of offspring during the reproduction process. However, additional heuristics should be incorporated in the algorithm as well; some of these heuristic rules provide guidelines for evaluating (feasible and infeasible) individuals in the population. This paper surveys such heuristics and discusses their merits and drawbacks.

Key words: Constrained optimization, evolutionary computation, genetic algorithms, infeasible individuals.

1 Introduction

During the last two decades there has been a growing interest in algorithms which are based on the principle of evolution (survival of the fittest). A common term, accepted recently, refers to such techniques as *evolutionary computation* (EC) methods. The best known algorithms in this class include genetic algorithms, evolutionary programming, evolution strategies, and genetic programming. There are also many hybrid systems which incorporate various features of the above paradigms, and consequently are hard to classify; anyway, we refer to them just as EC methods.

It is generally accepted that any evolutionary algorithm to solve a problem must have five basic components:

- a genetic representation of solutions to the problem,
- a way to create an initial population of solutions,
- an evaluation function (i.e., the environment), rating solutions in terms of their 'fitness',
- 'genetic' operators that alter the genetic composition of children during reproduction, and
- values for the parameters (population size, probabilities of applying genetic operators, etc.).

It is interesting to note that for a successful implementation of an evolutionary technique for a particular real-world problem, the basic components listed above require some additional heuristics. These heuristic rules apply to genetic representation of solutions, to 'genetic' operators that alter their composition, to values of various parameters, to methods for creating an initial population. It seems that one item only from the above list of five basic components of the evolutionary algorithm—the evaluation function—usually is taken "for granted" and does not require any heuristic modifications. Indeed, in many cases the process of selection of an evaluation function is straightforward (e.g., classical numerical and combinatorial optimization problems). Consequently, during the last two decades, many difficult functions have been examined; often they served as test-beds for different selection methods, various operators, different representations, and so forth. However, the process of selection of an evaluation function might be quite complex by itself, especially, when we deal with feasible and infeasible solutions to the problem; several heuristics usually are incorporated in this process. In this paper we examine some of these heuristics and discuss their merits and drawbacks.

The paper is organized as follows. The next section provides a short introductory description of evolutionary algorithms. Section 3 discusses some heuristics incorporated in evolutionary algorithms except those which are used in their "evaluation of population" parts. Section 4 states the problem by defining feasible and infeasible individuals and

Section 5 provides a detailed discussion on evaluation methods for evolutionary techniques. Section 6 concludes the paper.

2 Evolutionary computation

In general, any abstract task to be accomplished can be thought of as solving a problem, which, in turn, can be perceived as a search through a space of potential solutions. Since usually we are after "the best" solution, we can view this task as an optimization process. For small spaces, classical exhaustive methods usually suffice; for larger spaces special artificial intelligence techniques must be employed. The methods of evolutionary computation are among such techniques; they are stochastic algorithms whose search methods model some natural phenomena: genetic inheritance and Darwinian strife for survival. As stated in Davis (1987):

> "... the metaphor underlying genetic algorithms[1] is that of natural evolution. In evolution, the problem each species faces is one of searching for beneficial adaptations to a complicated and changing environment. The 'knowledge' that each species has gained is embodied in the makeup of the chromosomes of its members."

The idea behind evolutionary algorithms is to do what nature does. Let us take rabbits as an example:[2] at any given time there is a population of rabbits. Some of them are faster and smarter than other rabbits. These faster, smarter rabbits are less likely to be eaten by foxes, and therefore more of them survive to do what rabbits do best: make more rabbits. Of course, some of the slower, dumber rabbits will survive just because they are lucky. This surviving population of rabbits starts breeding. The breeding results in a good mixture of rabbit genetic material: some slow rabbits breed with fast rabbits, some fast with fast, some smart rabbits with dumb rabbits, and so on. And on the top of that, nature throws in a 'wild hare' every once in a while by mutating some of the rabbit genetic material. The resulting baby rabbits will (on average) be faster and smarter than these in the original population because more faster, smarter parents survived the foxes. (It is a good thing that the foxes are undergoing similar process—otherwise the rabbits might become too fast and smart for the foxes to catch any of them).

Evolutionary computation methods follow a step-by-step procedure that closely matches the story of the rabbits; they mimic the process of natural evolution, following the principles of natural selection and "survival of the fittest". In these algorithms a population of individuals (potential solutions) undergoes a sequence of unary (mutation type) and higher order (crossover type) transformations. These individuals strive for survival: a selection scheme, biased towards fitter individuals, selects the next generation. This

[1]The best known evolutionary computation techniques are genetic algorithms; very often the terms *evolutionary computation* methods and *GA-based* methods are used interchangeably.

[2]We 'borrowed' this example from Michalewicz 1994.

new generation contains a higher proportion of the characteristics possessed by the 'good' members of the previous generation; in this way good characteristics (e.g., being fast) are spread over the population (e.g., of rabbits) and mixed with other good characteristics (e.g., being smart). After some number of generations, the program converges and the best individual represents a near-optimum solution.

Another analogy was presented recently on the *Internet* (comp.ai.neural-nets, Sarle, 1993): it provides also a nice comparison between hill-climbing, simulated annealing, and genetic algorithm techniques:

> "Notice that in all [hill-climbing] methods discussed so far, the kangaroo can hope at best to find the top of a mountain close to where he starts. There's no guarantee that this mountain will be Everest, or even a very high mountain. Various methods are used to try to find the actual global optimum.
>
> In simulated annealing, the kangaroo is drunk and hops around randomly for a long time. However, he gradually sobers up and tends to hop up hill.
>
> In genetic algorithms, there are lots of kangaroos that are parachuted into the Himalayas (if the pilot didn't get lost) at random places. These kangaroos do not know that they are supposed to be looking for the top of Mt. Everest. However, every few years, you shoot the kangaroos at low altitudes and hope the ones that are left will be fruitful and multiply".

As already mentioned earlier in the paper, the best known techniques in the class of evolutionary computation methods are genetic algorithms, evolution strategies, evolutionary programming, and genetic programming. There are also many hybrid systems which incorporate various features of the above paradigms; however, the structure of any evolutionary computation algorithm is very much the same; a sample structure is shown in Figure 1.

```
procedure evolutionary algorithm
begin
  $t \leftarrow 0$
  initialize $P(t)$
  evaluate $P(t)$
  while (not termination-condition) do
  begin
    $t \leftarrow t + 1$
    select $P(t)$ from $P(t-1)$
    alter $P(t)$
    evaluate $P(t)$
  end
end
```

Figure 1: The structure of an evolutionary algorithm

The evolutionary algorithm maintains a population of individuals, $P(t) = \{x_1^t, \ldots, x_n^t\}$ for iteration t. Each individual represents a potential solution to the problem at hand, and is implemented as some data structure S. 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 fit individuals (select step). Some members of the new population undergo transformations (alter step) by means of "genetic" operators to form new solutions. There are unary transformations m_i (mutation type), which create new individuals by a small change in a single individual ($m_i : S \rightarrow S$), and higher order transformations c_j (crossover type), which create new individuals by combining parts from several (two or more) individuals ($c_j : S \times \ldots \times S \rightarrow S$). After some number of generations the algorithm converges—it is hoped that the best individual represents a near-optimum (reasonable) solution.

Despite powerful similarities between various evolutionary computation techniques there are also many differences between them (often hidden on a lower level of abstraction). They use different data structures S for their chromosomal representations, consequently, the 'genetic' operators are different as well. They may or may not incorporate some other information (to control the search process) in their genes. There are also other differences; for example, the two lines of the Figure 1:

```
select $P(t)$ from $P(t-1)$
alter $P(t)$
```

can appear in the reverse order: in evolution strategies first the population is altered and later a new population is formed by a selection process. Moreover, even within a particular technique, say, within genetic algorithms, there are many flavors and twists. For example, there are many methods for selecting individuals for survival and reproduction. These methods include (1) proportional selection, where the probability of selection is proportional to the individual's fitness, (2) ranking methods, where all individuals in a population are sorted from the best to the worst and probabilities of their selection are fixed for the whole evolution process,[3] and (3) tournament selection, where some number of individuals (usually two) compete for selection to the next generation: this competition (tournament) step is repeated population-size number of times. Within each of these categories there are further important details. Proportional selection may require the use of scaling windows or truncation methods, there are different ways for allocating probabilities in ranking methods (linear, nonlinear distributions), the size of a tournament plays a significant role in tournament selection methods. It is also important to decide on a generational policy. For example, it is possible to replace the whole population by a population of offspring, or it is possible to select the best individuals from two populations (population of parents and population of offspring)—this selection can be

[3]For example, the probability of selection of the best individual is always 0.15 regardless its precise evaluation; the probability of selection of the second best individual is always 0.14, etc. The only requirements are that better individuals have larger probabilities and the total of these probabilities equals to one.

done in a deterministic or nondeterministic way. It is also possible to produce few (in particular, a single) offspring, which replace some (the worst?) individuals (systems based on such generational policy are called 'steady state'). Also, one can use an 'elitist' model which keeps the best individual from one generation to the next[4]; such model is very helpful for solving many kinds of optimization problems.

For a particular chromosomal representation there is a variety of different genetic operators. We can consider various types of mutation, where probability of mutation depends on generation number and/or location of a bit. Also, apart from 1-point crossover, we may experiment with 2-point, 3-point, etc. crossovers, which exchange appropriate number of segments between parent chromosomes, as well as 'uniform crossover', which exchanges single genes from both parents. When a chromosome is a permutation of integer numbers $1, \ldots, n$, there are also many ways to mutate such chromosome and crossover two chromosomes (e.g., PMX, OX, CX, ER, EER crossovers).[5]

The variety of structures, operators, selection methods, etc. indicate clearly that some versions of evolutionary algorithms perform better than other versions on particular problems; many comparisons of different sort have been reported in literature (e.g., evolutionary strategies versus genetic algorithms, 1-point crossover versus 2-point crossover versus uniform crossover, etc.) As a result, in building a successful evolutionary algorithm for a particular problem (or class of problems) the user uses a 'common knowledge': a set of heuristic rules which emerged during the last two decades as a summary of countless experiments with various systems and various problems. The next section describes briefly some heuristics for selecting appropriate components of evolutionary algorithm, whereas Section 5 provides a detailed discussion on heuristics used for evaluating an individual in a population.

3 EC techniques and heuristics

The data structure used for a particular problem and a set of 'genetic' operators constitute the most essential components of any evolutionary algorithm. For example, the original genetic algorithms (GAs), devised to model *adaptation processes*, mainly operated on binary strings and used a recombination operator with mutation as a background operator (Holland 1975). Mutation flips a bit in a chromosome and crossover exchanges genetic material between two parents: if the parents are represented by five-bits strings, say $(0,0,0,0,0)$ and $(1,1,1,1,1)$, crossing the vectors after the second component would produce the offspring $(0,0,1,1,1)$ and $(1,1,0,0,0)$.[6]

[4]It means, that if the best individual from a current generation is lost due to selection or genetic operators, the system force it into next generation anyway.

[5]In most cases, crossover involves just two parents, however, it need not be the case. In a recent study (Eiben et al. 1994) the authors investigate the merits of 'orgies', where more than two parents are involved in the reproduction process. Also, scatter search techniques (Glover 1977) propose the use of multiple parents.

[6]This is an example of so-called 1-point crossover.

Evolution strategies (ESs) were developed as a method to solve parameter optimization problems (Back 1991, Schwefel 1981); consequently, a chromosome represents an individual as a pair of float-valued vectors,[7] i.e., $\vec{v} = (\vec{x}, \vec{\sigma})$. Here, the first vector $\vec{x}$ represents a point in the search space; the second vector $\vec{\sigma}$ is a vector of standard deviations: mutations are realized by replacing $\vec{v}$ by $(\vec{x}', \vec{\sigma}')$, where

$$\vec{\sigma}' = \vec{\sigma} \cdot e^{N(0,\Delta\vec{\sigma})} \text{ and}$$
$$\vec{x}' = \vec{x} + N(0, \vec{\sigma}'),$$

where $N(0, \vec{\sigma})$ is a vector of independent random Gaussian numbers with a mean of zero and standard deviations $\vec{\sigma}$ and $\Delta\vec{\sigma}$ is a parameter of the method.

The original evolutionary programming (EP) techniques (Fogel et al. 1966) aimed at evolution of artificial intelligence and finite state machines (FSM) were selected as a chromosomal representation of individuals. Offspring (new FSMs) are created by random mutations of parent population (see Figure 2). There are five possible mutation operators: change of an output symbol, change of a state transition, addition of a state, deletion of a state, and change of the initial state (there are some additional constraints on the minimum and maximum number of states).

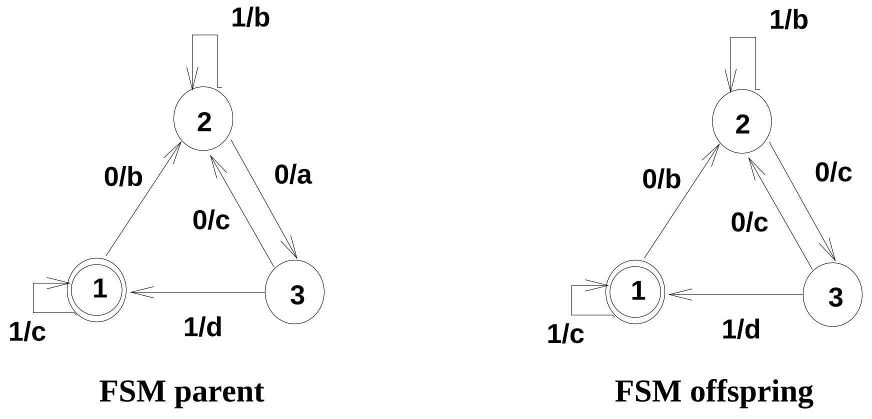

Figure 2: A FSM and its offspring. Machines start in state 1

Genetic programming (GP) techniques (Koza 1992) provide a way to run a search of the space of possible computer programs[8] for the best one (the most fit). For example, two structures e_1 and e_2 (Figure 3) represent expressions $2x + 2.11$ and $x \cdot \sin(3.28)$, respectively. A possible offspring e_3 (after crossover of e_1 and e_2) represents $x \cdot \sin(2x)$.

Many researchers modified further evolutionary algorithms by 'adding' the problem specific knowledge to the algorithm. Several papers (Antonisse and Keller 1987, Forrest 1985, Fox and McMahon 1990, Grefenstette 1987, Starkweather et al. 1991) have discussed initialization techniques, different representations, decoding techniques (mapping

[7]However, they started with integer variables as an experimental optimum-seeking method.
[8]Actually, Koza has chosen LISP's S-expressions for all his experiments.

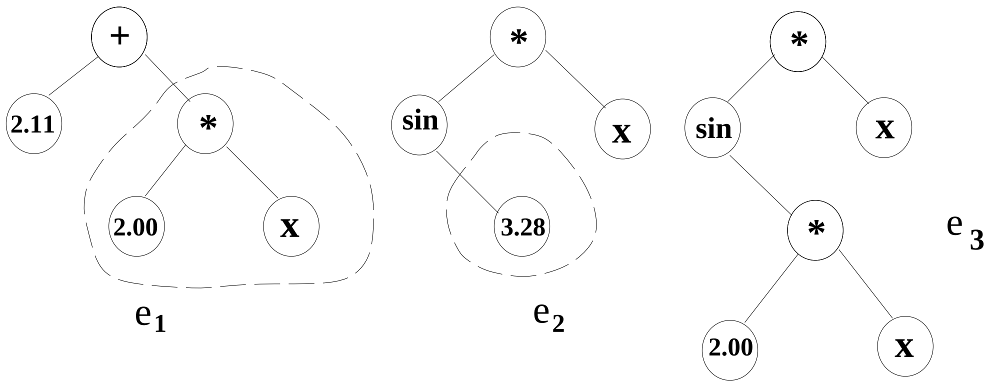

Figure 3: Expression e_3: an offspring of e_1 and e_2. Broken line includes areas being exchanged during the crossover operation

from genetic representations to 'phenotypic' representations), and the use of heuristics for genetic operators. Davis (1989) wrote (in the context of classical, binary GAs):

> "It has seemed true to me for some time that we cannot handle most real-world problems with binary representations and an operator set consisting only of binary crossover and binary mutation. One reason for this is that nearly every real-world domain has associated domain knowledge that is of use when one is considering a transformation of a solution in the domain [...] I believe that genetic algorithms are the appropriate algorithms to use in a great many real-world applications. I also believe that one should incorporate real-world knowledge in one's algorithm by adding it to one's decoder or by expanding one's operator set."

Such hybrid/nonstandard systems enjoy a significant popularity in evolutionary computation community. Very often these systems, extended by the problem-specific knowledge, outperform other classical evolutionary methods as well as other standard techniques (Michalewicz 1993, Michalewicz 1994).

There are few heuristics to guide a user in selection of appropriate data structures and operators for a particular problem. It is a common knowledge that for numerical optimization problem one should use an evolutionary strategy[9] or genetic algorithm with floating point representation, whereas some versions of genetic algorithm would be the best to handle combinatorial optimization problems. Genetic programs are great in discovery of rules given as a computer program, and evolutionary programming techniques can be used successfully to model a behavior of the system (e.g., prisoner dilemma problem, see Fogel 1993). An additional popular heuristic in applying evolutionary algorithms to real-world problems is based on modifying the algorithm by the problem-specific knowledge; this problem-specific knowledge is incorporated in chromosomal data structures

[9]Evolutionary programming techniques were generalized also to handle numerical optimization problems, see, e.g., Fogel 1992.

and specialized genetic operators (Michalewicz 1994). For example, a system Genetic-2N (Michalewicz et al. 1991) constructed for the nonlinear transportation problem used a matrix representation for its chromosomes, a problem-specific mutation (main operator, used with probability 0.4) and arithmetical crossover (background operator, used with probability 0.05). It is hard to classify this system: it is not really a genetic algorithm, since it can run with mutation operator only without any significant decrease of quality of results. Moreover, all matrix entries are floating point numbers. It is not an evolution strategy, since it did not encode any control parameters in its chromosomal structures. Clearly, it has nothing to do with genetic programming or evolutionary programming approaches. It is just an evolutionary technique aimed at particular problem.

Another possibility is based on hybridization; in (Davis 1991) the author wrote:

> "When I talk to the user, I explain that my plan is to hybridize the genetic algorithm technique and the current algorithm by employing the following three principles:
>
> - *Use the Current Encoding.* Use the current algorithm's encoding technique in the hybrid algorithm;
> - *Hybridize Where Possible.* Incorporate the positive features of the current algorithm in the hybrid algorithm;
> - *Adapt the Genetic Operators.* Create crossover and mutation operators for the new type of encoding by analogy with bit string crossover and mutation operators. Incorporate domain-based heuristics as operators as well."

Some of these ideas were embodied earlier in the evolutionary procedure called *scatter search* (Glover 1977). The process generates initial populations by screening good solutions produced by heuristics. The points used as parents are then joined by linear combinations with context-dependent weights, where such combinations may apply to multiple parents simultaneously. The linear combination operators are further modified by adaptive rounding processes to handle components required to take discrete values. (The vectors operated on may contain both real and integer components, as opposed to strictly binary components.) Finally, preferred outcomes are selected and again subjected to heuristics, whereupon the process repeats. The approach has been found useful for mixed integer and combinatorial optimization. (For background and recent developments see, e.g., Glover 1994.)

There are a few heuristics available for creating an initial population: one can start from a randomly created population, or use an output from some deterministic algorithm to initialize it (with many other possibilities in between these extremes). There are also some general heuristic rules for determining values for the various parameters; for many genetic algorithms applications, population size stays between 50 and 100, probability of crossover—between 0.65 and 1.00, and probability of mutation—between 0.001 and 0.01.

Additional heuristic rules are often used to vary the population size or probabilities of operators during the evolution process.

It seems that neither of the evolutionary techniques is perfect (or even robust) across the problem spectrum; only the whole family of algorithms based on evolutionary computation concepts (i.e., evolutionary algorithms) have this property of robustness. But the main key to successful applications is in heuristics methods, which are mixed skillfully with evolutionary techniques (e.g., recent work on search strategies for constrained design spaces by Bilchev and Parmee 1995; Parmee, Johnson and Burt 1994).

4 Feasible and infeasible solutions

In evolutionary computation methods the evaluation function serves as the only link between the problem and the algorithm. The evaluation function rates individuals in the population: better individuals have better chances for survival and reproduction. Hence it is essential to define an evaluation function which characterize the problem in a 'perfect way'. In particular, the issue of handling feasible and infeasible individuals should be addressed very carefully: very often a population contains infeasible individuals but we search for a *feasible* optimal. Finding a proper evaluation measures for feasible and infeasible individuals is of great importance; it directly influences the outcome (success or failure) of the algorithm.

The issue of processing infeasible individuals is very important for solving constrained optimization problems using evolutionary techniques. For example, in continuous domains, the general nonlinear programming problem[10] is to find $\overline{X}$ so as to

$$\text{optimize } f(\overline{X}),\ \overline{X} = (x_1, \ldots, x_n) \in R^n,$$

where $\overline{X} \in \mathcal{F} \subseteq \mathcal{S}$. The set $\mathcal{S} \subseteq R^n$ defines the search space and the set $\mathcal{F} \subseteq S$ defines a *feasible* search space. Usually, the search space $\mathcal{S}$ is defined as a n-dimensional rectangle in R^n (domains of variables defined by their lower and upper bounds):

$$l(i) \leq x_i \leq u(i), \quad 1 \leq i \leq n,$$

whereas the feasible set $\mathcal{F}$ is defined by an intersection of $\mathcal{S}$ and a set of additional $m \geq 0$ constraints:

$$g_j(\overline{X}) \leq 0, \text{ for } j = 1, \ldots, q, \text{ and } h_j(\overline{X}) = 0, \text{ for } j = q+1, \ldots, m.$$

Most research on applications of evolutionary computation techniques to nonlinear programming problems was concerned with complex objective functions with $\mathcal{F} = \mathcal{S}$. Several test functions used by various researchers during the last 20 years consider only

[10]We consider here only continuous variables.

domains of n variables; this was the case with five test functions F1–F5 proposed by De Jong (1975), as well as with many other test cases proposed since then.

In discrete domains the problem of constraints was acknowledged much earlier. Knapsack problem, set covering problem, all types of scheduling and timetabling problems are constrained. Several heuristic methods emerged to handle constraints; however, these methods have not been studied in a systematic way.

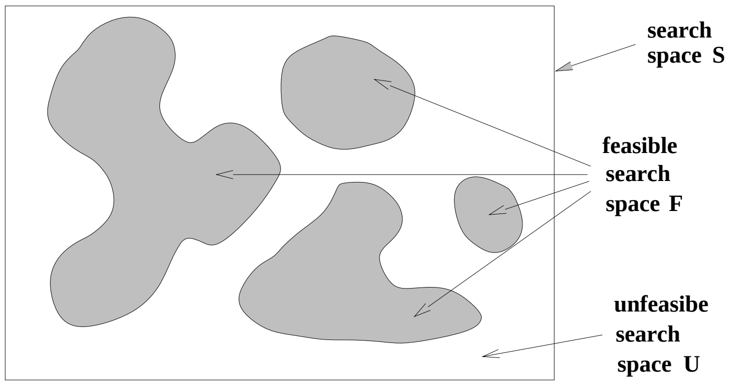

Figure 4: A search space and its feasible and infeasible parts

In general, a search space $\mathcal{S}$ consists of two disjoint subsets of feasible and infeasible subspaces, $\mathcal{F}$ and $\mathcal{U}$, respectively (see Figure 4). We do not make any assumptions about these subspaces; in particular, they need not be convex and they need not be connected (e.g., as it is the case in the example in Figure 4 where feasible part $\mathcal{F}$ of the search space consists of four disjoined subsets). In solving optimization problems we search for a *feasible* optimum. During the search process we have to deal with various feasible and infeasible individuals; for example (see Figure 5), at some stage of the evolution process, a population may contain some feasible (b, c, d, e, i, j, k, p) and infeasible individuals (a, f, g, h, l, m, n, o), while the (global) optimum solution is marked by 'X'.

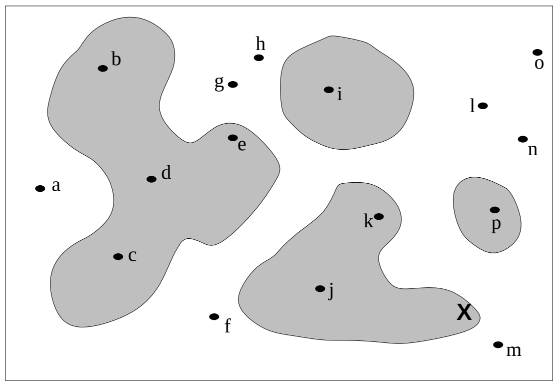

Figure 5: A population of 16 individuals, a – o

The presence of feasible and infeasible individuals in the population influences other parts of the evolutionary algorithm; for example, should the elitist selection method consider a possibility of preserving the best *feasible* individual, or just the best individual overall? Further, some operators might be applicable to feasible individuals only. However, the major aspect of such a scenario is the need for evaluation of feasible and infeasible individuals. The problem of how to evaluate individuals in the population is far from trivial. In general, we have to design two evaluation functions, $eval_f$ and $eval_u$, for feasible and infeasible domains, respectively. There are many important questions to be addressed (we discuss them in detail in the next section):

A. How should two feasible individuals be compared, e.g., 'c' and 'j' from Figure 5? In other words, how should the function $eval_f$ be designed?

B. How should two infeasible individuals be compared, e.g., 'a' and 'n'? In other words, how should the function $eval_u$ be designed?

C. How are the functions $eval_f$ and $eval_u$ related to each other? Should we assume, for example, that $eval_f(s) \succ eval_u(r)$ for any $s \in \mathcal{F}$ and any $r \in \mathcal{U}$ (the symbol $\succ$ is interpreted as 'is better than', i.e., 'greater than' for maximization and 'smaller than' for minimization problems)?

D. Should we consider infeasible individuals harmful and eliminate them from the population?

E. Should we 'repair' infeasible solutions by moving them into the closest point of the feasible space (e.g., the repaired version of 'm' might be the optimum 'X', Figure 5)?

F. If we repair infeasible individuals, should we replace an infeasible individual by its repaired version in the population or rather should we use a repair procedure for evaluation purpose only?

G. Since our aim is to find a feasible optimum solution, should we choose to penalize infeasible individuals?

H. Should we start with initial population of feasible individuals and maintain the feasibility of offspring by using specialized operators?

I. Should we change the topology of the search space by using decoders?

J. Should we extract a set of constraints which define feasible search space and process individuals and constraints separately?

K. Should we concentrate on searching a boundary between feasible and infeasible parts of the search space?

Several trends for handling infeasible solutions have emerged in the area of evolutionary computation. We discuss them in the following section using examples from discrete and continuous domains.

5 Heuristics for evaluating individuals

In this section we discuss several methods for handling feasible and infeasible solutions in a population; most of these methods emerged quite recently. Only a few years ago Richardson et al. (1989) claimed: "Attempts to apply GA's with constrained optimization problems follow two different paradigms (1) modification of the genetic operators; and (2) penalizing strings which fail to satisfy all the constraints." This is no longer the case as a variety of heuristics have been proposed. Even the category of penalty functions consists of several methods which differ in many important details on how the penalty function is designed and applied to infeasible solutions. Other methods maintain the feasibility of the individuals in the population by means of specialized operators or decoders, impose a restriction that any feasible solution is 'better' than any infeasible solution, consider constraints one at the time in a particular linear order, repair infeasible solutions, use multiobjective optimization techniques, are based on cultural algorithms, or rate solutions using a particular co-evolutionary model. We discuss these techniques in turn by addressing questions A – K from the previous section.

A. Design of $eval_f$

This is usually the easiest issue: for most optimization problems, the evaluation function f for feasible solutions is given. This is the case for numerical optimization problems and for most operation research problems (knapsack problems, traveling salesman problems, set covering problems, etc.) However, for some problems the selection of evaluation function might be far from trivial. For example, in building an evolutionary system to control a mobile robot (Michalewicz and Xiao 1995) there is a need to evaluate a robot's paths. It is unclear, whether path #1 or path #2 (Figure 6) should have better evaluation (taking into account their total distance, clearance from obstacles, and smoothness): path #1 is shorter, but path #2 is smoother. For such problems there is a need for some heuristic measures to be incorporated into the evaluation function. Note, that even the subtask of measuring the smoothness or clearance of a path is not simple.

This is also the case in many design problems, where there are no clear formulae for comparing two feasible designs. Clearly, some problem-dependent heuristics are necessary in such cases, which should provide with a numerical measure $eval_f(x)$ of a feasible individual x.

One of the best examples to illustrate the problem of necessity of evaluating feasible individuals is the satisfiability (SAT) problem. For a given conjunctive normal form formula, say

$$F(x) = (x_1 \vee \overline{x_2} \vee x_3) \wedge (\overline{x_1} \vee \overline{x_3}) \wedge (x_2 \vee x_3),$$

it is hard to compare two feasible individuals $p = (0, 0, 0)$ and $q = (1, 0, 0)$ (in both cases $F(p) = F(q) = 0$). De Jong and Spears (1989) examined a few possibilities. For example,

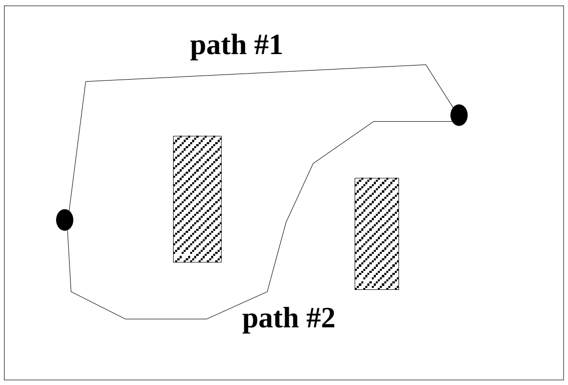

Figure 6: Paths in an environment

it is possible to define $eval_u$ to be a ratio of the number of conjuncts which evaluate to true; in that case

$$eval_f(p) = 0.666 \text{ and } eval_f(q) = 0.333.$$

It is also possible (Pardalos 1994) to change the Boolean variables x_i into floating point numbers y_i and to assign:

$$eval_f(y) = |y_1 - 1||y_2 + 1||y_3 - 1| + |y_1 + 1||y_3 + 1| + |y_2 - 1||y_3 - 1|,$$

or

$$eval'_f(y) = (y_1 - 1)^2(y_2 + 1)^2(y_3 - 1)^2 + (y_1 + 1)^2(y_3 + 1)^2 + (y_2 - 1)^2(y_3 - 1)^2.$$

In the above cases the solution to the SAT problem corresponds to a set of global minimum points of the objective function: the *true* value of $F(x)$ is equivalent to the global minimum value 0 of $eval_u(y)$.

There is also another possibility: in some cases we do not need to define the evaluation function $eval_f$ at all! This function is necessary only if the evolutionary algorithm uses proportional selection (see Section 2). For other types of selection routines it is possible to establish only a linear ordering relation on individuals in the population. If a linear ordering relation ρ handles decisions of the type "is a feasible individual x better than a feasible individual y?",[11] then such a relation ρ is sufficient for tournament and ranking selections methods, which require either a selection of the best individual out of some number of individuals, or linear ordering of all individuals, respectively.

Of course, it might be necessary to use some heuristics to build such a linear ordering relation ρ. For example, for multi-objective optimization problems it is relatively easy to establish a partial ordering between individual solutions; additional heuristics might be necessary to order individuals which are not comparable by the partial relation.

[11]The statement $\rho(x, y)$ is interpreted as x is better than y, for feasible x and y.

In summary, it seems that tournament and ranking selections give some additional flexibility to the user: sometimes it is easier to compare two solutions than to provide their evaluation values as numbers. However, in these methods it is necessary to resolve additional problems of comparing two infeasible individuals (see part B) as well as comparing feasible and infeasible individuals (see part C).

B. Design of $eval_u$

This is a quite hard problem. We can avoid it altogether by rejecting infeasible individuals (see part D). Sometimes it is possible to extend the domain of function $eval_f$ to handle infeasible individuals, i.e., $eval_u(x) = eval_f(x) \pm Q(x)$, where $Q(x)$ represents either a penalty for infeasible individual x, or a cost for repairing such individual (see part G). Another option is to design a separate evaluation function $eval_u$, independent of $eval_f$, however, in a such case we have to establish some relationship between these two functions (see part C).

It is difficult to evaluate infeasible individuals. This is the case for knapsack problem, where the amount of violation of capacity need not be a good measure of the individual's 'fitness' (see part G). This is also the case for many scheduling and timetable problems as well as the path planning problem: it is unclear whether path #1 or path #2 is better (Figure 7), since path #2 has more intersection points with obstacles and is longer than path #1; on the other hand most infeasible paths are "worse" using the above criteria than the straight line (path #1).

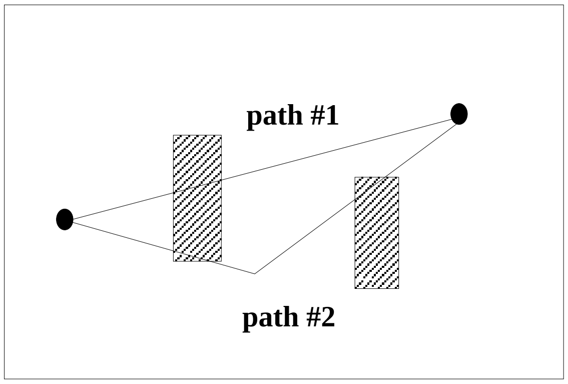

Figure 7: Infeasible paths in an environment

As was the case with feasible solutions (part A), it is possible to develop an ordering relation for infeasible individuals (as opposed the construction of $eval_u$); in both cases it is necessary to establish a relationship between evaluations of feasible and infeasible individuals (part C).

C. Relationship between $eval_f$ and $eval_u$

Assume that we process both feasible and infeasible individuals in the population and that we evaluate them using two evaluation functions, $eval_f$ and $eval_u$, respectively. In other words, evaluations of a feasible individual x and infeasible individual y are $eval_f(x)$ and $eval_u(y)$, respectively. Now it is of great importance to establish a relationship between these two evaluation functions.

One possibility (as mentioned already in part B) is to design $eval_u$ by means of $eval_f$, i.e., $eval_u(y) = eval_f(y) \pm Q(y)$, where $Q(y)$ represents either a penalty for infeasible individual y, or a cost for repairing such an individual (we discuss this option in part G).

Another possibility is as follows. We can construct a global evaluation function $eval$ as

$$eval(p) = \begin{cases} q_1 \cdot eval_f(p) & if \;\; p \in \mathcal{F} \\ q_2 \cdot eval_u(p) & if \;\; p \in \mathcal{U} \end{cases}$$

In other words, two weights, q_1 and q_2, are used to scale the relative importance of $eval_f$ and $eval_u$.

Both above methods allow infeasible individuals to be "better" than feasible individuals. In general, it is possible to have a feasible individual x and an infeasible one, y, such that $eval(y) \succ eval(x)$.[12] This may lead the algorithm to converge to an infeasible solution; it is why several researchers experimented with dynamic penalties Q (see part G) which increase pressure on infeasible individuals with respect to the current state of the search. An additional weakness of these methods lies in their problem dependence; often the problem of selecting $Q(x)$ (or weights q_1 and q_2) is almost as difficult as solving the original problem.

On the other hand, some researchers (Powell and Skolnick 1993, Michalewicz and Xiao 1995) reported good results of their evolutionary algorithms, which worked under the assumption that any feasible individual was better than any infeasible one. Powell and Skolnick (1993) applied this heuristic rule for the numerical optimization problems: evaluations of feasible solutions were mapped into the interval $(-\infty, 1)$ and infeasible solutions—into the interval $(1, \infty)$ (for minimization problems). Michalewicz and Xiao (1995) experimented with the path planning problem and used two separate evaluation functions for feasible and infeasible individuals. The values of $eval_u$ were increased (i.e., made less attractive) by adding such a constant, so that the best infeasible individual was worse that the worst feasible one. However, it is not clear whether this should always be the case. In particular, it is doubtful whether the feasible individual 'b' (Figure 5) should have higher evaluation than infeasible individual 'm', which is "just next" to the optimal solution. A similar example can be drawn from the path planning problem: it is unclear whether a feasible path #2 (see Figure 8) deserves better evaluation than infeasible path #1!

[12]The symbol $\succ$ is interpreted as 'is better than', i.e., 'greater than' for maximization and 'smaller than' for minimization problems.

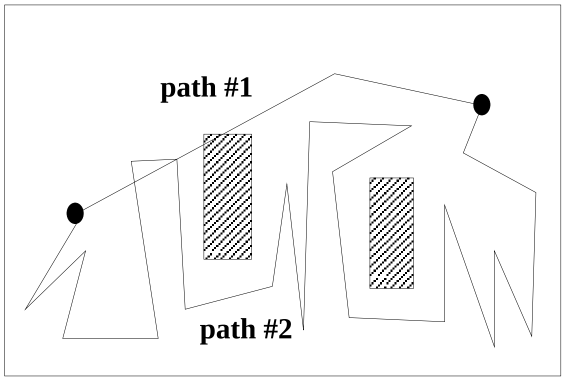

Figure 8: Infeasible and feasible paths in an environment

The issue of establishing a relationship between evaluation functions for feasible and infeasible individuals is one of the most challenging problems to resolve while applying an evolutionary algorithm to a particular problem.

D. Rejection of infeasible individuals

This "death penalty" heuristic is a popular option in many evolutionary techniques (e.g., evolution strategies). Note that rejection of infeasible individuals offers a few simplifications of the algorithm: for example, there is no need to design $eval_u$ and to compare it with $eval_f$.

The method of eliminating infeasible solutions from a population may work reasonably well when the feasible search space is convex and it constitutes a reasonable part of the whole search space (e.g., evolution strategies do not allow equality constraints since with such constraints the ratio between the sizes of feasible and infeasible search spaces is zero). Otherwise such an approach has serious limitations. For example, for many search problems where the initial population consists of infeasible individuals only, it might be essential to improve them (as opposed to rejecting them). Moreover, quite often the system can reach the optimum solution easier if it is possible to "cross" an infeasible region (especially in non-convex feasible search spaces).

E. Repair of infeasible individuals

Repair algorithms enjoy a particular popularity in the evolutionary computation community: for many combinatorial optimization problems (e.g., traveling salesman problem, knapsack problem, set covering problem, etc.) it is relatively easy to 'repair' an infeasible individual. Such a repaired version can be used either for evaluation only, i.e.,

$$eval_u(y) = eval_f(x),$$

where x is a repaired (i.e., feasible) version of y, or it can also replace (with some probability) the original individual in the population (see part F). Note, that the repaired version of solution 'm' (Figure 5) might be the optimum 'X'.

The process of repairing infeasible individuals is related to a combination of learning and evolution (so-called *Baldwin effect*, Whitley et al. 1994). Learning (as local search in general, and local search for the closest feasible solution, in particular) and evolution interact with each other: the fitness value of the improvement is transferred to the individual. In that way a local search is analogous to learning that occurs during one generation of a particular string.

The weakness of these methods is in their problem dependence. For each particular problem a specific repair algorithm should be designed. Moreover, there are no standard heuristics on design of such algorithms: usually it is possible to use a greedy repair, random repair, or any other heuristic which would guide the repair process. Also, for some problems the process of repairing infeasible individuals might be as complex as solving the original problem. This is the case for the nonlinear transportation problem (see Michalewicz 1993), most scheduling and timetable problems, and many others.

On the other hand, recently completed Genocop III system (Michalewicz and Nazhiyath 1995) for constrained numerical optimization (nonlinear constraints) is based on repair algorithms. Genocop III incorporates the original Genocop system (which handles linear constraints only; see section H), but also extends it by maintaining two separate populations, where a development in one population influences evaluations of individuals in the other population. The first population P_s consists of so-called search points which satisfy linear constraints of the problem. As in Genocop, the feasibility (in the sense of linear constraints) of these points is maintained by specialized operators. The second population P_r consists of so-called reference points from $\mathcal{F}$; these points are fully feasible, i.e., they satisfy *all* constraints. Reference points $\vec{r}$ from P_r, being feasible, are evaluated directly by the objective function (i.e., $eval_f(\vec{r}) = f(\vec{r})$). On the other hand, search points from P_s are "repaired" for evaluation and the repair process works as follows. Assume, there is a search point $\vec{s} \in P_s$. If $\vec{s} \in \mathcal{F}$, then $eval_f(\vec{s}) = f(\vec{s})$, since $\vec{s}$ is fully feasible. Otherwise (i.e., $\vec{s} \notin \mathcal{F}$), the system selects one of the reference points, say $\vec{r}$ from P_r and creates a sequence of points $\vec{z}$ from a segment between $\vec{s}$ and $\vec{r}$: $\vec{z} = a\vec{s} + (1-a)\vec{r}$. This can be done either (1) in a random way by generating random numbers a from the range $\langle 0, 1 \rangle$, or (2) in a deterministic way by setting $a_i = 1/2, 1/4, 1/8, \ldots$ until a feasible point is found. Once a fully feasible $\vec{z}$ is found, $eval_u(\vec{s}) = eval_f(\vec{z}) = f(\vec{z})$. Clearly, in different generations the same search point $\mathcal{S}$ can evaluate to different values due to the random nature of the repair process.

F. Replacement of individuals by their repaired versions

The question of replacing repaired individuals is related to so-called *Lamarckian evolution* (Whitley et al. 1994), which assumes that an individual improves during its lifetime and that the resulting improvements are coded back into the chromosome. As stated in

Whitley et al. 1994:

> "Our analytical and empirical results indicate that Lamarckian strategies are often an extremely fast form of search. However, functions exist where both the simple genetic algorithm without learning and the Lamarckian strategy used [...] converge to local optima while the simple genetic algorithm exploiting the Baldwin effect converges to a global optimum."

This is why it is necessary to use the replacement strategy very carefully.

Recently (see Orvosh and Davis 1993) a so-called 5%-rule was reported: this heuristic rule states that in many combinatorial optimization problems, an evolutionary computation technique with a repair algorithm provides the best results when 5% of repaired individuals replace their infeasible originals. However, many recent experiments (e.g., Michalewicz 1994) indicated that for many combinatorial optimization problems this rule did not apply. Either a different percentage gives better results, or there is no significant difference in the performance of the algorithm for various probabilities of replacement.

In continuous domains, a new replacement rule is emerging. The Genocop III system (see section E) for constrained numerical optimization problems with nonlinear constraints is based on repair approach. The first experiments (based on 10 test cases which have various numbers of variables, constraints, types of constraints, numbers of active constraints at the optimum, etc.) indicate that the 15% replacement rule is a clear winner: the results of the system are much better than with either lower or higher values of the replacement rate.

At present, it seems that the 'optimal' probability of replacement is problem-dependent and it may change over the evolution process as well. Further research is required for comparing different heuristics for setting this parameter, which is of great importance for all repair-based methods.

G. Penalizing infeasible individuals

This is the most common approach in the genetic algorithms community. The domain of function $eval_f$ is extended; the approach assumes that

$$eval_u(p) = eval_f(p) \pm Q(p),$$

where $Q(p)$ represents either a penalty for infeasible individual p, or a cost for repairing such an individual. The major question is, how should such a penalty function $Q(p)$ be designed? The intuition is simple: the penalty should be kept as low as possible, just above the limit below which infeasible solutions are optimal (so-called *minimal penalty rule*, see Le Riche et al. 1995). However, it is difficult to implement this rule effectively.

The relationship between infeasible individual 'p' and the feasible part $\mathcal{F}$ of the search space $\mathcal{S}$ plays a significant role in penalizing such individuals: an individual might be penalized just for being infeasible, the 'amount' of its infeasibility is measured to determine the penalty value, or the effort of 'repairing' the individual might be taken into account. For example, for the knapsack problem with capacity 99 we may have two infeasible solutions yielding the same profit, where the total weight of all items taken is 100 and 105, respectively. However, it is difficult to argue that the first individual with the total weight 100 is 'better' than the other one with the total weight 105, despite the fact that for this individual the violation of the capacity constraint is much smaller than for the other one. The reason is that the first solution may involve 5 items of the weight 20 each, and the second solution may contain (among other items) an item of a low profit and weight 6—removal of this item would yield a feasible solution, possibly much better than any repaired version of the first individual. However, in such cases a penalty function should consider the "easiness of repairing" an individual as well as the quality of its repaired version; designing such penalty functions is problem-dependent and, in general, quite hard.

Several researchers studied heuristics on design of penalty functions. Some hypotheses were formulated (Richardson et al. 1989):

- "penalties which are functions of the distance from feasibility are better performers than those which are merely functions of the number of violated constraints,
- for a problem having few constraints, and few full solutions, penalties which are solely functions of the number of violated constraints are not likely to find solutions,
- good penalty functions can be constructed from two quantities, the *maximum completion cost* and the *expected completion cost*,
- penalties should be close to the *expected completion cost*, but should not frequently fall below it. The more accurate the penalty, the better will be the solutions found. When penalty often underestimates the completion cost, then the search may not find a solution."

and (Siedlecki and Sklanski 1989):

- "the genetic algorithm with a variable penalty coefficient outperforms the fixed penalty factor algorithm,"

where a variability of penalty coefficient was determined by a heuristic rule.

This last observation was further investigated by Smith and Tate (1993). In their work they experimented with dynamic penalties, where the penalty measure depends on the number of violated constraints, the best feasible objective function found, and the best objective function value found.

For numerical optimization problems penalties usually incorporate degrees of constraint violations. Most of these methods use constraint violation measures f_j (for the j-th constraint) for the construction of the $eval_u$; these functions are defined as

$$f_j(\overline{X}) = \begin{cases} \max\{0, g_j(\overline{X})\}, & if\ 1 \leq j \leq q \\ |h_j(\overline{X})|, & if\ q+1 \leq j \leq m \end{cases}$$

For example, Homaifar et al. (1994) assume that for every constraint we establish a family of intervals that determines appropriate penalty values. The method works as follows:

- for each constraint, create several (ℓ) levels of violation,
- for each level of violation and for each constraint, create a penalty coefficient R_{ij} ($i = 1, 2, \ldots, \ell$, $j = 1, 2, \ldots, m$); higher levels of violation require larger values of this coefficient.
- start with a random population of individuals (i.e., these individuals are feasible or infeasible),
- evaluate individuals using the following formula

  $$eval(\overline{X}) = f(\overline{X}) + \sum_{j=1}^{m} R_{ij} f_j^2(\overline{X}),$$

  where R_{ij} is a penalty coefficient for the i-th level of violation and the j-th constraint.

Note, that the function $eval$ is defined on $\mathcal{S}$, i.e., it serves both feasible and infeasible solutions.

It is also possible (as suggested in Siedlecki and Sklanski 1989) to adjust penalties in a dynamic way, taking into account the current state of the search or the generation number. For example, Joines and Houck (1994) assumed dynamic penalties; individuals are evaluated (at the iteration t) by the following formula:

$$eval(\overline{X}) = f(\overline{X}) + (C \times t)^{\alpha} \sum_{j=1}^{m} f_j^{\beta}(\overline{X}),$$

where C, α and β are constants. As in Homaifar et al. (1994), the function $eval$ evaluates both feasible and infeasible solutions.

The method is quite similar to Homaifar et al. (1994), but it requires many fewer parameters (C, α and β), and this is independent of the total number of constraints. Also, the penalty component is not constant but changes with the generation number. Instead of defining several levels of violation, the pressure on infeasible solutions is increased due to the $(C \times t)^{\alpha}$ component of the penalty term: towards the end of the process (for high values of t), this component assumes large values.

Michalewicz and Attia (1994) considered the following method:

- divide all constraints into four subsets: linear equations, linear inequalities, nonlinear equations, and nonlinear inequalities,
- select a random single point as a starting point (the initial population consists of copies of this single individual). This initial point satisfies all linear constraints,
- create a set of active constraints A; include there all nonlinear equations and all violated nonlinear inequalities.

- set the initial temperature $\tau = \tau_0$,
- evolve the population using the following formula:

  $eval(\overline{X}, \tau) = f(\overline{X}) + \frac{1}{2\tau} \sum_{j \in A} f_j^2(\overline{X})$,

  (only active constraints are considered),
- if $\tau < \tau_f$, stop, otherwise
  - decrease temperature τ,
  - the best solution serves as a starting point of the next iteration,
  - update the set of active constraints A,
  - repeat the previous step of the main part.

Note that the algorithm maintains the feasibility of all linear constraints using a set of closed operators (see Michalewicz and Janikow method, part H). At every iteration the algorithm considers active constraints only, the pressure on infeasible solutions is increased due to the decreasing values of temperature τ. The method requires 'starting' and 'freezing' temperatures, τ_0 and τ_f, respectively, and the cooling scheme to decrease temperature τ.

A method of adapting penalties was developed by Bean and Hadj-Alouane (1992). As the previous method, it uses a penalty function, however, one component of the penalty function takes a feedback from the search process. Each individual is evaluated by the formula:

$$eval(\overline{X}) = f(\overline{X}) + \lambda(t) \sum_{j=1}^{m} f_j^2(\overline{X}),$$

where $\lambda(t)$ is updated every generation t in the following way:

$$\lambda(t+1) = \begin{cases} (1/\beta_1) \cdot \lambda(t), & if\ \overline{B}(i) \in \mathcal{F}\ for\ all\ t-k+1 \leq i \leq t \\ \beta_2 \cdot \lambda(t), & if\ \overline{B}(i) \in \mathcal{S} - \mathcal{F}\ for\ all\ t-k+1 \leq i \leq t \\ \lambda(t), & otherwise, \end{cases}$$

where $\overline{B}(i)$ denotes the best individual, in terms of function $eval$, in generation i, $\beta_1, \beta_2 > 1$ and $\beta_1 \neq \beta_2$ (to avoid cycling). In other words, the method (1) decreases the penalty component $\lambda(t+1)$ for the generation $t+1$, if all best individuals in the last k generations were feasible, and (2) increases penalties, if all best individuals in the last k generations were unfeasible. If there are some feasible and unfeasible individuals as best individuals in the last k generations, $\lambda(t+1)$ remains without change.

Yet another approach was proposed recently by Le Riche et al. 1995. The authors designed a (segregated) genetic algorithm which uses two values of penalty parameters (for each constraint) instead of one; these two values aim at achieving a balance between heavy and moderate penalties by maintaining two subpopulations of individuals. The population is split into two cooperating groups, where individuals in each group are evaluated using either one of the two penalty parameters.

It seems that the appropriate choice of the penalty method may depend on (1) the ratio between sizes of the feasible and the whole search space, (2) the topological properties of the feasible search space, (3) the type of the objective function, (4) the number of variables, (5) number of constraints, (6) types of constraints, and (7) number of active constraints at the optimum. Thus the use of penalty functions is not trivial and only some partial analysis of their properties is available. Also, a promising direction for applying penalty functions is the use of adaptive penalties: penalty factors can be incorporated in the chromosome structures in a similar way as some control parameters are represented in the structures of evolution strategies and evolutionary programming.

H. Maintaining feasible population by special representations and genetic operators

As indicated in Section 3, one reasonable heuristic for dealing with the issue of feasibility is to use specialized representation and operators to maintain the feasibility of individuals in the population.

We would illustrate this point using an example of the traveling salesman problem. For this problem it is possible to use various representations and various operators which would transform a feasible parent individual into a feasible offspring (see Michalewicz 1994). For example, the so-called ordinal representation represents a tour as a list of n cities; the i-th element of the list is a number in the range from 1 to $n - i + 1$. The idea behind the ordinal representation is as follows. There is some ordered list of cities C, which serves as a reference point for lists in ordinal representations. Assume, for example, that such an ordered list (reference point) is simply

$$C = (1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9).$$

A tour

$$1 - 2 - 4 - 3 - 8 - 5 - 9 - 6 - 7$$

is then represented as a list l of references,

$$l = (1\ 1\ 2\ 1\ 4\ 1\ 3\ 1\ 1),$$

and should be interpreted as follows: the first number on the list l is 1, so take the first city from the list C as the first city of the tour (city number 1), and remove it from C. At this stage the partial tour is (1). The next number on the list l is also 1, so take the first city from the current list C as the next city of the tour (city number 2), and remove it from C. At this stage the partial tour is (1, 2), etc. The main advantage of the ordinal representation is that the classical crossover works: any two tours in the ordinal representation, cut after some position and crossed together, would produce two offspring, each of them being a legal tour. For example, the two parents

p_1 = (1 1 2 1 | 4 1 3 1 1) and
p_2 = (5 1 5 5 | 5 3 3 2 1),

which correspond to the tours

1 – 2 – 4 – 3 – 8 – 5 – 9 – 6 – 7 and
5 – 1 – 7 – 8 – 9 – 4 – 6 – 3 – 2,

with the crossover point marked by '|', would produce the following offspring:

o_1 = (1 1 2 1 5 3 3 2 1) and
o_2 = (5 1 5 5 4 1 3 1 1);

these offspring correspond to

1 – 2 – 4 – 3 – 9 – 7 – 8 – 6 – 5 and
5 – 1 – 7 – 8 – 6 – 2 – 9 – 3 – 4.

On the other hand, one can use the path representation of a tour. In such representation, a tour

5 – 1 – 7 – 8 – 9 – 4 – 6 – 2 – 3

is represented simply as

(5 1 7 8 9 4 6 2 3).

Several crossovers were defined for the path representation: *partially -mapped* (PMX), *order* (OX), *cycle* (CX), *edge recombination* (ER), *enhanced edge recombination* (EER) crossovers. Each of these operators maintains feasibility of individuals. For example, PMX—proposed by Goldberg and Lingle (1985)—builds an offspring by choosing a subsequence of a tour from one parent and preserving the order and position of as many cities as possible from the other parent. A subsequence of a tour is selected by choosing two random cut points, which serve as boundaries for swapping operations. For example, the two parents (with two cut points marked by '|')

p_1 = (1 2 3 | 4 5 6 7 | 8 9) and
p_2 = (4 5 2 | 1 8 7 6 | 9 3)

would produce offspring in the following way. First, the segments between cut points are swapped (the symbol 'x' can be interpreted as 'at present unknown'):

o_1 = (x x x | 1 8 7 6 | x x) and
o_2 = (x x x | 4 5 6 7 | x x).

This swap defines also a series of mappings:

$1 \leftrightarrow 4$, $8 \leftrightarrow 5$, $7 \leftrightarrow 6$, and $6 \leftrightarrow 7$.

Then we can fill further cities (from the original parents), for which there is no conflict:

$o_1 =$ (x 2 3 | 1 8 7 6 | x 9) and
$o_2 =$ (x x 2 | 4 5 6 7 | 9 3).

Finally, the first x in the offspring o_1 (which should be 1, but there was a conflict) is replaced by 4, because of the mapping $1 \leftrightarrow 4$. Similarly, the second x in the offspring o_1 is replaced by 5, and the x and x in the offspring o_2 are 1 and 8. The offspring are

$o_1 =$ (4 2 3 | 1 8 7 6 | 5 9) and
$o_2 =$ (1 8 2 | 4 5 6 7 | 9 3).

During the last decade several specialized systems were developed for particular optimization problems; these systems use a unique chromosomal representations and specialized 'genetic' operators which alter their composition. Some of such systems were described in Davis (1991); other examples include Genocop (Michalewicz and Janikow 1991) for optimizing numerical functions with linear constraints and Genetic-2N (Michalewicz et al. 1991) for nonlinear transportation problem. For example, Genocop assumes linear constraints only and a feasible starting point (or feasible initial population). A closed set of operators maintains feasibility of solutions. For example, when a particular component x_i of a solution vector $\overline{X}$ is mutated, the system determines its current domain $dom(x_i)$ (which is a function of linear constraints and remaining values of the solution vector $\overline{X}$) and the new value of x_i is taken from this domain (either with flat probability distribution for uniform mutation, or other probability distributions for non-uniform and boundary mutations). In any case the offspring solution vector is always feasible. Similarly, arithmetic crossover[13]

$$a\overline{X} + (1-a)\overline{Y}$$

of two feasible solution vectors $\overline{X}$ and $\overline{Y}$ yields always a feasible solution (for $0 \le a \le 1$) in convex search spaces (the system assumes linear constraints only which imply convexity of the feasible search space $\mathcal{F}$). Consequently, there is no need to define the function $eval_u$; the function $eval_f$ is (as usual) the objective function f.

Such systems are much more reliable than any other evolutionary techniques based on penalty approach (Michalewicz 1994). This is a quite popular trend. Many practitioners

[13]The arithmetical crossover operator generate offspring by linear combinations of the parents. As noted, such a strategy of generating a set of diverse trial points by linear and convex combinations (and allowing the offspring to influence the search) was proposed some years ago in the scatter search approach by Glover (1977).

use problem-specific representations and specialized operators in building very successful evolutionary algorithms in many areas; these include numerical optimization, machine learning, optimal control, cognitive modeling, classic operation research problems (traveling salesman problem, knapsack problems, transportation problems, assignment problems, bin packing, scheduling, partitioning, etc.), engineering design, system integration, iterated games, robotics, signal processing, and many others.

Also, it is interesting to note, that original evolutionary programming techniques (Fogel et al. 1966) and genetic programming techniques (Koza 1992) fall into this category of evolutionary algorithms: these techniques maintain feasibility of finite state machines or hierarchically structured programs by means of specialized representations and operators.

I. Use of decoders

Decoders offer an interesting option for all practitioners of evolutionary techniques. In these techniques a chromosome "gives instructions" on how to build a feasible solution. For example, a sequence of items for the knapsack problem can be interpreted as: "take an item if possible"—such interpretation would lead always to feasible solutions. Let us consider the following scenario: we try to solve the 0–1 knapsack problem with n items; the profit and weight of the i-th item are p_i and w_i, respectively. We can sort all items in decreasing order of p_i/w_i's and interpret the binary string

(1100110001001110101001010111010101...0010)

in the following way: take the first item from the list (i.e., item with the largest ratio profit per weight) if the item fits in the knapsack. Continue with second, fifth, sixth, tenth, etc. items from the sorted list, until the knapsack is full or there are no more items available. Note that the sequence of all 1's corresponds to a greedy solution. Any sequence of bits would translate into a feasible solution, every feasible solution may have many possible codes. We can apply classical binary operators (crossover and mutation): any offspring is clearly feasible.

However, it is important to point out that several factors should be taken into account while using decoders. Each decoder imposes a relationship T between a feasible solution and decoded solution (see Figure 9).

It is important that several conditions are satisfied: (1) for each solution $s \in \mathcal{F}$ there is a decoded solution d, (2) each decoded solution d corresponds to a feasible solution s, and (3) all solutions in $\mathcal{F}$ should be represented by the same number of decodings d. Additionally, it is reasonable to request that (4) the transformation T is computationally fast and (5) it has locality feature in the sense that small changes in the decoded solution result in small changes in the solution itself. An interesting study on coding trees in genetic algorithm was reported by Palmer and Kershenbaum (1994), where the above conditions were formulated.

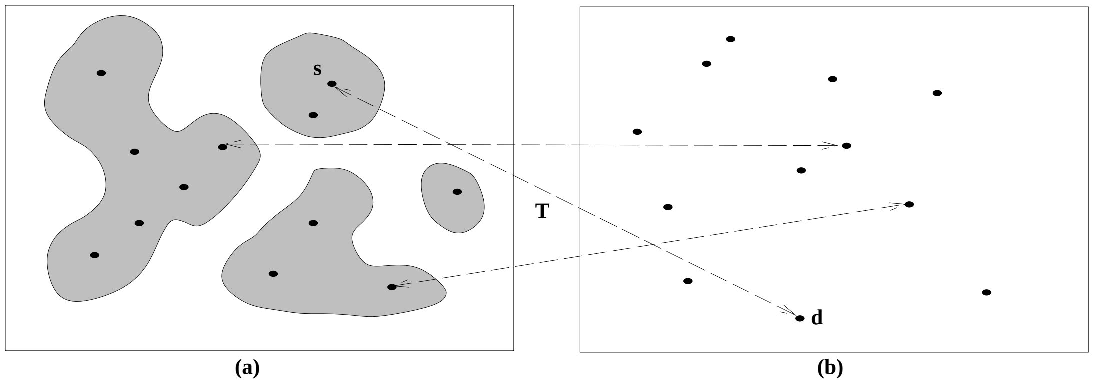

Figure 9: Transformation T between solutions in original (a) and decoder's (b) space

J. Separation of individuals and constraints

This is a general and interesting heuristic. The first possibility would include utilization of multi-objective optimization methods, where the objective function f and constraint violation measures f_j (for m constraints) constitute a $(m+1)$-dimensional vector $\vec{v}$:

$$\vec{v} = (f, f_1, \ldots, f_m).$$

Using some multi-objective optimization method, we can attempt to minimize its components: an ideal solution x would have $f_j(x) = 0$ for $1 \leq i \leq m$ and $f(x) \leq f(y)$ for all feasible y (minimization problems). A successful implementation of this approach was presented recently in Surry et al. (1995).

Another approach was recently reported by Paredis (1994). The method (described in the context of constraint satisfaction problems) is based on a co-evolutionary model, where a population of potential solutions co-evolves with a population of constraints: fitter solutions satisfy more constraints, whereas fitter constraints are violated by more solutions. It means, that individuals from the population of solutions are considered from the whole search space $\mathcal{S}$, and that there is no distinction between feasible and infeasible individuals (i.e., there is only one evaluation function $eval$ without any split into $eval_f$ or $eval_u$). The value of $eval$ is determined on the basis of constraint violations measures f_j's; however, better f_j's (e.g., active constraints) would contribute more towards the value of $eval$.

Yet another heuristic is based on the idea of handling constraints in a particular order; Schoenauer and Xanthakis (1993) called this method a "behavioral memory" approach. The initial steps of the method are devoted to sampling the feasible region; only in the final step is the objective function f optimized.

- start with a random population of individuals (i.e., these individuals are feasible or infeasible),

- set $j = 1$ (j is constraint counter),
- evolve this population to minimize the violation of the j-th constraint, until a given percentage of the population (so-called flip threshold ϕ) is feasible for this constraint. In this case

  $$eval(\overline{X}) = g_1(\overline{X}).$$

- set $j = j + 1$,
- the current population is the starting point for the next phase of the evolution, minimizing the violation of the j-th constraint:

  $$eval(\overline{X}) = g_j(\overline{X}).[14]$$

  During this phase, points that do not satisfy at least one of the 1st, 2nd, ... ,$(j-1)$-th constraints are eliminated from the population. The halting criterion is again the satisfaction of the j-th constraint by the flip threshold percentage ϕ of the population.

- if $j < m$, repeat the last two steps, otherwise ($j = m$) optimize the objective function f rejecting infeasible individuals.

The method has a few merits. One of them is that in the final step of the algorithm the objective function f is optimized (as opposed to its modified form). But for larger feasible spaces the method just provides additional computational overhead, and for very small feasible search spaces it is essential to maintain a diversity in the population.

It is also possible to incorporate the knowledge of the constraints of the problem into the belief space of cultural algorithms (Reynolds 1994); such algorithms provide a possibility of conducting an efficient search of the feasible search space (Reynolds et al. 1995). The research on cultural algorithms (Reynolds 1994) was triggered by observations that culture might be another kind of inheritance system. But it is not clear what the appropriate structures and units to represent the adaptation and transmission of cultural information are. Neither is it clear how to describe the interaction between natural evolution and culture. Reynolds developed a few models to investigate the properties of cultural algorithms; in these models, the belief space is used to constrain the combination of traits that individuals can assume. Changes in the belief space represent macro-evolutionary change and changes in the population of individuals represent micro-evolutionary change. Both changes are moderated by the communication link.

The general intuition behind belief spaces is to preserve those beliefs associated with "acceptable" behavior at the trait level (and, consequently, to prune away unacceptable beliefs). The acceptable beliefs serve as constraints that direct the population of traits. It

[14]To simplify notation, we do not distinguish between inequality constraints g_j and equations h_j; all m constraints are denoted by g_j.

seems that the cultural algorithms may serve as a very interesting tool for numerical optimization problems, where constraints influence the search in a direct way (consequently, the search in constrained spaces may be more efficient than in unconstrained ones!).

K. Exploring boundaries between feasible and infeasible parts of the search space

One of the most recently developed approach for constrained optimization is *strategic oscillation*. Strategic oscillation was originally proposed in accompaniment with the evolutionary strategy of scatter search, and more recently has been applied to a variety of problem settings in combinatorial and nonlinear optimization (see, for example, the review of Glover 1995). The approach is based on identifying a *critical level*, which for our purposes represents a boundary between feasibility and infeasibility, but which also can include such elements as a stage of construction or a chosen interval of values for a functional. In the feasibility/infeasibility context, the basic strategy is to approach and cross the feasibility boundary, by a design that is implemented either by adaptive penalties and inducements (which are progressively relaxed or tightened according to whether the current direction of search is to move deeper into a particular region or to move back toward the boundary) or by simply employing modified gradients or subgradients to progress in the desired direction. Within the context of neighborhood search, the rules for selecting moves are typically amended to take account of the region traversed and the direction of traversal. During the process of repeatedly approaching and crossing the feasibility frontier from different directions, the possibility of retracing a prior trajectory is avoided by mechanisms of memory and probability.

The application of different rules (according to region and direction) is generally accompanied by crossing a boundary to different depths on different sides. An option is to approach and retreat from the boundary while remaining on a single side, without crossing. One-sided oscillations are especially relevant in a variety of scheduling and graph theory settings, where a useful structure can be maintained up to a certain point and then is lost (as by running out of jobs to assign or by going beyond the conditions that define a tree or tour, etc.). In these cases, a constructive process for building to the critical level is accompanied by a destructive process for dismantling the structure.

It is frequently important in strategic oscillation to spend additional search time in regions close to the boundary. This may be done by inducing a sequence of tight oscillations about the boundary as a prelude to each larger oscillation to a greater depth. If greater effort is allowed for executing each move, the method may use more elaborate moves (such as various forms of “exchanges”) to stay at the boundary for longer periods. For example, such moves can be used to proceed to a local optimum each time a critical proximity to the boundary is reached. A strategy of applying such moves at additional levels is suggested by a *proximate optimality principle*, which states roughly that good constructions at one level are likely to be close to good constructions at another.

One of the useful forms of strategic oscillation operates by increasing and decreasing bounds for a function $g(x)$. Such an approach has been effective in a number of applications where $g(x)$ has represented such items as workforce assignments and function values (as well as feasibility/infeasibility levels), to guide the search to probe at various depths within the associated regions. In reference to degrees of feasibility and infeasibility, $g(x)$ may represent a vector-valued function associated with a set of problem constraints (which may expressed, for example, as $g(x) \leq b$). In this instance, controlling the search by bounding $g(x)$ can be viewed as manipulating a parameterization of the selected constraint set. An often-used alternative is to make $g(x)$ a lagrangean or surrogate constraint penalty function, avoiding vector-valued functions and allowing tradeoffs between degrees of violation of different component constraints according to their importance. Surrogate constraint approaches are particularly useful for isolating such tradeoffs, accompanied by special memory to keep track of behavior that discloses the relative influence of constraints. Approaches that embody such ideas may be found, for example, in Freville and Plateau 1986, Gendreau, Hertz and Laporte 1991, Kelly, Golden and Assad 1993, Voss 1993, Xu and Kelly 1995 and Glover and Kochenberger 1995.

6 Conclusions

The paper surveys many heuristics which support the most important step of any evolutionary technique: evaluation of the population. It is clear that further studies in this area are necessary: different problems require different "treatment". It is also possible to mix different strategies described in this paper; for example, Paechter et al. 1994 built a successful evolutionary system for a timetable problem, where "each chromosome in the population gives instructions on how to build a timetable. These instruction may or may not result in a feasible timetable", thus allowing other heuristics to be added to the proposed decoder. The author is not aware of any results which provide heuristics on relationships between categories of optimization problems and evaluation techniques in the presence of infeasible individuals; this is an important area of future research.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant IRI-9322400. The author wishes to thank Fred Glover and David W. Johnston for comments on the first draft of the paper.

References

Antonisse, H.J. and K.S. Keller (1987). Genetic Operators for High Level Knowledge Representation. In *Proceedings of the Second International Conference on Genetic Algorithms*, 69–76, Cambridge, MA: Lawrence Erlbaum.

Bäck, T., F. Hoffmeister, and H.-P. Schwefel (1991). A Survey of Evolution Strategies. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, Los Altos, CA, Morgan Kaufmann Publishers, 2–9.

Bean, J.C. and Hadj-Alouane, A.B. (1992). A Dual Genetic Algorithm for Bounded Integer Programs. Department of Industrial and Operations Engineering, The University of Michigan, TR 92-53.

Bilchev, G. and I.C. Parmee (1995). Adaptive Search Strategies for Heavily Constrained Design Spaces. In *Proceedings of the 22nd International Conference CAD '95*, Ukraine, Yalta, May 8–13, 1995.

Davis, L. (1987). *Genetic Algorithms and Simulated Annealing*, Morgan Kaufmann Publishers, Los Altos, CA, 1987.

Davis, L. (1989). Adapting Operator Probabilities in Genetic Algorithms. In *Proceedings of the Third International Conference on Genetic Algorithms*, Los Altos, CA, Morgan Kaufmann Publishers, 61–69.

Davis, L. (1991). *Handbook of Genetic Algorithms*, New York, Van Nostrand Reinhold.

De Jong, K.A. (1975). An Analysis of the Behavior of a Class of Genetic Adaptive Systems, Doctoral dissertation, University of Michigan, *Dissertation Abstract International*, 36(10), 5140B. (University Microfilms No 76-9381).

De Jong K.A. and W.M. Spears (1989). Using Genetic Algorithms to Solve NP-Complete Problems. In *Proceedings of the Third International Conference on Genetic Algorithms*, Los Altos, CA, Morgan Kaufmann Publishers, 124–132.

Fogel, L.J., A.J. Owens and M.J. Walsh (1966). *Artificial Intelligence through Simulated Evolution*, New York, Wiley.

Forrest, S. (1985). Implementing Semantic Networks Structures using the Classifier System. In *Proceedings of the First International Conference on Genetic Algorithms*, 24–44, Pittsburgh, PA: Lawrence Erlbaum.

Fox, B.R. and M.B. McMahon (1990). Genetic Operators for Sequencing Problems. In G. Rawlins (Ed.), *Foundations of Genetic Algorithms*, First Workshop on the Foundations of Genetic Algorithms and Classifier Systems, Morgan Kaufmann.

Freville, A. and G. Plateau (1986). Heuristics and Reduction Methods for Multiple Constraint 0-1 Linear Programming Problems. European Journal of Operational Research 24, 206-215.

Gendreau, M., A. Hertz and G. Laporte (1991). A Tabu Search Heuristic for Vehicle Routing. CRT-7777, Centre de Recherche sur les transports, Universite de Montreal, to appear in Management Science.

Glover, F. (1977). Heuristics for Integer Programming Using Surrogate Constraints. *Decision Sciences*, Vol.8, No.1, 156–166.

Glover, F. (1994). Genetic Algorithms and Scatter Search: Unsuspected Potentials. Statistics and Computing, 4, 131-140.

Glover, F. (1995). Tabu Search Fundamentals and Uses. Graduate School of Business, University of Colorado.

Glover, F. and G. Kochenberger (1995). Critical Event Tabu Search for Multidimensional Knapsack Problems. Metaheuristics for Optimization. Kluwer Publishing, 113-133.

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

Goldberg, D.E. and R. Lingle (1985). Alleles, Loci, and the TSP. In *Proceedings of the First International Conference on Genetic Algorithms*, Lawrence Erlbaum Associates, Hillsdale, NJ, 154–159.

Grefenstette, J.J. (1987). Incorporating Problem Specific Knowledge into Genetic Algorithms. In L. Davis (Ed.), *Genetic Algorithms and Simulated Annealing*, Morgan Kaufmann.

Hadj-Alouane, A.B. and Bean, J.C. (1992). A Genetic Algorithm for the Multiple-Choice Integer Program. Department of Industrial and Operations Engineering, The University of Michigan, TR 92-50.

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

Homaifar, A., S. H.-Y. Lai and X. Qi (1994). Constrained Optimization via Genetic Algorithms. *Simulation*, Vol.62, 242–254.

Joines, J.A. and C.R. Houck (1994). On the Use of Non-Stationary Penalty Functions to Solve Nonlinear Constrained Optimization Problems With GAs. In *Proceedings of the Evolutionary Computation Conference—Poster Sessions*, part of the IEEE World Congress on Computational Intelligence, Orlando, 27–29 June 1994, 579–584.

Kelly, J.P., B. L. Golden and A.A. Assad (1993). Large Scale Controlled Rounding Using Tabu Search with Strategic Oscillation. Annals of Operations Research, Vol. 41, 69-84.

Koza, J.R. (1992). *Genetic Programming*, Cambridge, MA, MIT Press.

Le Riche, R., C. Vayssade, R.T. Haftka (1995). A Segregated Genetic Algorithm for Constrained Optimization in Structural Mechanics. Technical Report, Universite de Technologie de Compiegne, France, 1995.

Michalewicz, Z. (1993). A Hierarchy of Evolution Programs: An Experimental Study. *Evolutionary Computation*, Vol.1, 51–76.

Michalewicz, Z. (1994). *Genetic Algorithms + Data Structures = Evolution Programs*, Springer-Verlag, 2nd edition, New York.

Michalewicz, Z. and N. Attia (1994). In Evolutionary Optimization of Constrained Problems. *Proceedings of the 3rd Annual Conference on Evolutionary Programming*, eds. A.V. Sebald and L.J. Fogel, River Edge, NJ, World Scientific Publishing, 98–108.

Michalewicz, Z. and C. Janikow (1991). Handling Constraints in Genetic Algorithms. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, Los Altos, CA, Morgan Kaufmann Publishers, 151–157.

Michalewicz, Z. and Nazhiyath, G. (1995). Genocop III: A Co-evolutionary Algorithm for Numerical Optimization Problems with Nonlinear Constraints. *Proceedings of the 2nd IEEE International Conference on Evolutionary Computation*, Perth, 29 November – 1 December 1995.

Michalewicz, Z., G.A. Vignaux, and M. Hobbs (1991). A Non-Standard Genetic Algorithm for the Nonlinear Transportation Problem. ORSA Journal on Computing, Vol.3, No.4, 1991, 307–316.

Michalewicz, Z. and J. Xiao (1995). Evaluation of Paths in Evolutionary Planner/Navigator. In *Proceedings of the 1995 International Workshop on Biologically Inspired Evolutionary Systems*, Tokyo, Japan, May 30–31, 1995, pp.45–52.

Orvosh, D. and L. Davis (1993). Shall We Repair? Genetic Algorithms, Combinatorial Optimization, and Feasibility Constraints. In *Proceedings of the Fifth International Conference on Genetic Algorithms*, Los Altos, CA, Morgan Kaufmann Publishers, 650.

Palmer, C.C. and A. Kershenbaum (1994). Representing Trees in Genetic Algorithms. In *Proceedings of the IEEE International Conference on Evolutionary Computation*, 27–29 June 1994, 379–384.

Pardalos, P. (1994). On the Passage from Local to Global in Optimization. In *Mathematical Programming*, J.R. Birge and K.G. Murty (Editors), The University of Michigan, 1994.

Paechter, B., A. Cumming, H. Luchian, and M. Petriuc (1994). Two Solutions to the General Timetable Problem Using Evolutionary Methods. In *Proceedings of the IEEE International Conference on Evolutionary Computation*, 27–29 June 1994, 300–305.

Paredis, J. (1994). Co-evolutionary Constraint Satisfaction. In *Proceedings of the 3rd Conference on Parallel Problem Solving from Nature*, New York, Springer-Verlag, 46–55,

Parmee, I.C., M. Johnson, and S. Burt (1994). Techniques to Aid Global Search in Engineering Design. In *Proceedings of the 7th International Conference IEA/AIE*, Gordon and Breach Science Publishers, 377–385.

Powell, D. and M.M. Skolnick (1993). Using Genetic Algorithms in Engineering Design Optimization with Non-linear Constraints. In *Proceedings of the Fifth International Conference on Genetic Algorithms*, Los Altos, CA, Morgan Kaufmann Publishers, 424–430.

Reynolds, R.G. (1994). An Introduction to Cultural Algorithms. In *Proceedings of the Third Annual Conference on Evolutionary Programming*, River Edge, NJ, World Scientific, 131–139.

Reynolds, R.G., Z. Michalewicz and M. Cavaretta (1995). Using Cultural Algorithms for Constraint Handling in Genocop. In *Proceedings of the 4th Annual Conference on Evolutionary Programming*, San Diego, CA, March 1–3, 1995.

Richardson, J.T., M.R. Palmer, G. Liepins and M. Hilliard (1989). Some Guidelines for Genetic Algorithms with Penalty Functions. In *Proceedings of the Third International Conference on Genetic Algorithms*, Los Altos, CA, Morgan Kaufmann Publishers, 191–197.

Sarle, W. (1993). Kangaroos. Article posted on *comp.ai.neural-nets* on 1 September 1993.

Schoenauer, M., and S. Xanthakis (1993). Constrained GA Optimization. In *Proceedings of the Fifth International Conference on Genetic Algorithms*, Los Altos, CA, Morgan Kaufmann Publishers, 573–580.

Schwefel, H.-P. (1981). *Numerical Optimization for Computer Models*. Chichester, UK, Wiley.

Siedlecki, W. and J. Sklanski (1989). Constrained Genetic Optimization via Dynamic Reward–Penalty Balancing and Its Use in Pattern Recognition. In *Proceedings of the Third International Conference on Genetic Algorithms*, Los Altos, CA, Morgan Kaufmann Publishers, 141–150.

Smith, A.E. and D.M. Tate (1993). Genetic Optimization Using a Penalty Function. In *Proceedings of the Fifth International Conference on Genetic Algorithms*, 499–503, Urbana-Champaign, CA: Morgan Kaufmann.

Starkweather, T., S. McDaniel, K. Mathias, C. Whitley, and D. Whitley (1991). A Comparison of Genetic Sequencing Operators. In *Proceedings of the Fourth International Conference on Genetic Algorithms*, 69–76, San Diego, CA: Morgan Kaufmann.

Surry, P.D., N.J. Radcliffe, and I.D. Boyd (1995). A Multi-objective Approach to Constrained Optimization of Gas Supply Networks. Presented at the *AISB-95 Workshop on Evolutionary Computing*, Sheffield, UK, April 3–4, 1995.

Voss, S. (1993). Tabu Search: Applications and Prospects. Technical report, Technische Hochshule Darmstadt.

Whitley, D., V.S. Gordon, and K. Mathias (1994). Lamarckian Evolution, the Baldwin Effect and function Optimization. In *Proceedings of the Parallel Problem Solving from Nature, 3*, Springer-Verlag, New York, 6–15.

Xu, J. and J.P. Kelly (1995). A Robust Network Flow-Based Tabu Search Approach for the Vehicle Routing Problem. Graduate School of Business, University of Colorado, Boulder.