Constrained GA optimization

in *Proceedings of the 5th International Conference on Genetic Algorithms, Urbana Champaign, 1993.*

Marc Schoenauer
CMAP – CNRS URA 756
École Polytechnique, F-91128 PALAISEAU
e-mail : marc@cmapx.polytechnique.fr

Spyros Xanthakis
Research Department
OPL
10 rue Alfred Kastler, F-14000 CAEN

Abstract

We present a general method of handling constraints in genetic optimization, based on the Behavioural Memory paradigm. Instead of requiring the problem-dependent design of either repair operators (projection on the feasible region) or penalty function (weighted sum of the constraints violations and the objective function), we sample the feasible region by evolving from an initially random population, successively applying a series of different fitness functions which embody constraint satisfaction. The final step is the optimization of the objective function restricted to the feasible region. The success of the whole process is highly dependent on the genetic diversity maintained during the first steps, ensuring a uniform sampling of the feasible region.

This method succeeded on some truss structure optimization problems, where the other genetic techniques for handling the constraints failed to give good results. Moreover in some domains, as in automatic generation of software test data, no other technique can be easily applied, as some constraints are not even computable until others are satisfied.

1 INTRODUCTION

Most optimization problems are constrained problems, i.e. the search space is restricted to some subspace of the defining space of the function to optimize. Let us suppose we want to maximize a non negative real-valued function $\mathcal{F}$, called the *objective function*, defined on some space E, called the *search space*.

The *constraints* are equalities or inequalities the solution is required to satisfy, involving some real-valued functions C_i defined on E. In many cases, the main difficulty of this problem lies in identifying the *feasible region* E', the subspace of E where the constraints are satisfied. The feasible region can have any shape: It can be neither convex nor connected.

The classical numerical algorithms (see (Fletcher 87) for instance) quickly fail to give the right solution (or to give any solution at all) when the problem lacks "regularity" (like linearity, convexity, differentiability).

Genetic Algorithms (GAs) (Holland 75, Goldberg 89) have now been used successfully to solve optimization problems in many domains, even though they are not function optimizers (De Jong 92). Their ability to converge to the fittest points of the search space, in a finite, though often very large, number of generations, has been studied in many works: Theoretically in (Zhigljavski 91, Davis 91) with uncheckable hypotheses, heuristically with the well-known *Schema Theorem* (Holland 75) and its generalizations (Radcliffe 91) and of course experimentally (see papers in (ICGA 89, ICGA 91, PPSN 92) for instance). But most of the works on GAs address the general optimization problem, rarely mentionning explicitely constrained problems.

We present in this paper a general-purpose technique for handling constraints in GA optimization processes. It is based on the notion of Behavioural Memory (de Garis 90), which takes into account the information contained in the whole population after some genetic evolution. The first steps of the whole process are devoted to just sampling the feasible region. The last step is then the genetic evolution of that sample, to optimize the final objective function.

In section 2, we review some existing works in the field of GAs devoted to constrained optimization. In Section 3 we introduce and discuss our method on a simple example. In Section 4 we give the first results obtained on some Truss Structures Optimization problem, for which other GA approaches failed to succeed. Section 5 presents the problem of Automatic Software Test Data Generation, where other classical optimization approaches have encountered major combinatorial limitations. The particularity of this problem is that constraint C_i can only be computed after constraints $C_1, \ldots, C_{i-1}$ are satisfied.

2 GAs AND CONSTRAINTS

As constrained problems are quite numerous among optimization problems, there has been many attempts to solve them using GAs. Nevertheless, as pointed out in (Davis 87), the methods used are either based on some penalty function, or problem dependent, or restricted to some particular objective functions and/or constraints. A review of the different ways GAs handle constraints can be found in (Michalewicz 91).

We shall briefly discuss the weaknesses or the limitations of these approaches.

2.1 ADJUSTING THE WEIGHTS, OR THE PENALTY FUNCTION METHOD

The most widely used method to treat constraints is to incorporate them in the objective function, and to use standard methods for unconstrained optimization: In GAs, the fitness function usually becomes some weighted sum of the original objective function minus some *penalty* for every constraint violation. The problem then becomes to adjust the penalty function, and the relative weight of the objective function as well as the different constraint violation penalizations.

There is no general solution to this problem, neither in classical numerical methods, nor in the GAs field. Some guidelines for the penalty functions design are given in (Richardson 89). The authors unfortunately conclude that their method can hardly be generalized. From our experience, the penalty function method is robust when the feasible region is "large" [1] or when the problem is "smooth". It then remains the easiest and best way to treat constraints.

On more difficult problems, designing a reasonnable penalty function can become a domain dependent task.

2.2 DOMAIN SPECIFIC GAs

The best results obtained by GAs on constrained problems use problem dependent methods, where the genetic operators, the fitness function and the search space itself are tailored to take the constraints into account. When possible, this approach, which uses as much domain-specific knowledge as possible, is probably the best way to tackle constraints.

[1] depending for instance on the ratio Cardinality(E')/Cardinality(E) for discrete problems, and on measure(E')/measure(E) for continuous problems.

Note that the constrained problem (P) on E can always be posed as the unconstrained problem on the feasible region E'. But, as already mentioned, we are concerned by constrained problems where the feasible region is not computable by direct methods. In particular, in order to apply successfully GAs to the unconstrained problem, one should be able to both generate an initial random population in E', and design recombination and mutation operators closed in E', suitable for genetic optimization.

Theoretical studies adressing the design of good recombination operators are yet incomplete (Radcliffe 91). It is nevertheless well established that initial repartitions of the population and mutation operators must have continuous density of probability on E (Zhigljavski 91). And all the widely used (when available) projection operators, and other repair algorithms certainly don't respect these conditions, as they induce a discontinuity on the boundary of the feasible region. Of course this warning does not prevent such approaches from succeeding.

By domain-specific GAs we don't exclusively mean the above mentioned family of repair algorithms, but also the following approaches:

- the TSP as solved in (Grefenstette 87) ;
- the handling of linear constraints where the feasible region is convex, ensuring that the solution lies on its boundary (Michalewicz 91) ;
- the general method based on constraint propagation (Paredis 92), which applies on problems like job shop scheduling, where the constraints are analytically defined, and the propagation of the constraints is symbolically or numerically possible.

3 BEHAVIOURAL MEMORY

We propose to address the general problem of genetic constrained optimization by a multi-steps process: The initial steps are devoted to sampling the feasible region, i.e. initializing a population on which, in the sole last step, the objective function is optimized using standard GAs.

The scheme we use is based on the Behavioural Memory paradigm: the population resulting from an evolution under genetic presssure can be viewed in a whole as a memory containing some essential information about the context it evolved in, that is the fitness function used in the GA. Such scheme, despite the fact that it has already been proved to be helpful on some difficult optimization problems (de Garis 90, Desquilbet 92), has not yet, as far as we know, been systematically applied to constrained optimization.

In the simplest case, the whole optimization process is a two phases process:

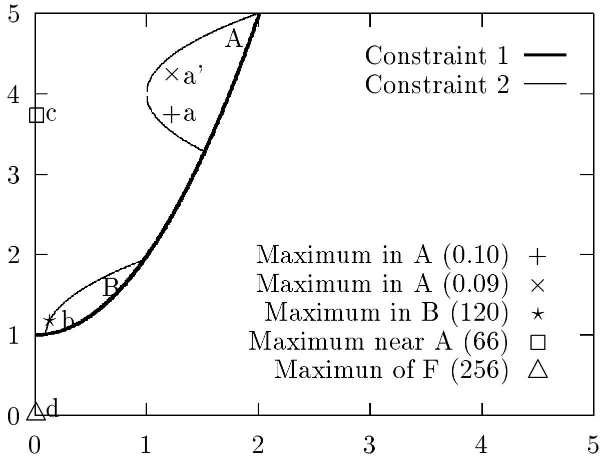

Figure 1: The feasible regions.

- Evolve an initial random population with some standard GA, the fitness function being related to the constraint satisfaction.
- Take the final population resulting from this evolution, and use it as initial population for a GA with the objective cost function as fitness function, which we override by assigning zero fitness whenever the constraints are not satisfied.

3.1 AN ARTIFICIAL PROBLEM

Consider the following function of two real variables:

$$\mathcal{F}(x,y) = \frac{\sin(x)^3 \times \sin(y)}{x^3 \times (x+y)},$$

on the space $E = \{(x,y)\ ;\ 0 < x \leq 10\ ,\ 0 < y \leq 10\}$

This function has many local optima in E. Let us treat two optimization problems involving $\mathcal{F}$:

$$(P1) \begin{cases} \text{Max } \{\ \mathcal{F}(\text{x,y})\ ;\ (\text{x,y}) \in \text{E}\ \} \\ \text{x}^2 - \text{y} + 1 \leq 0 \\ \text{x} - (\text{y}-4)^2 \leq 1 \end{cases}$$

$$(P2) \begin{cases} \text{Max } \{\ \mathcal{F}(\text{x,y})\ ;\ (\text{x,y}) \in \text{E}\ \} \\ \text{x}^2 - \text{y} + 1 \leq 0 \\ \text{x} - (\text{y}-4)^2 \leq 1 \ \textbf{OR}\ \text{x} - (\text{y}-1)^2 \leq 0.1 \end{cases}$$

The characteristics of the problems can be found on Fig. 1: Domain A (resp $A \bigcup B$) is the feasible region of $P1$ (resp. $P2$) ; there are in A two maxima (points a and a') with very similar values for $\mathcal{F}$; but many other local maxima take considerably higher values, including the global maximum for $P2$ (point b in B), and the overall global maximum of $\mathcal{F}$ (point d).

This explains why the design of a penalty function is difficult here [2]: with low weights, the global maximum of the penalized function is still outside the feasible region, near (0,0) ; with high weights, region A appears in the fitness landscape like some *plateau*, making hard for the GAs to distinguish between different maxima.

The experiments reported are done with a lab-made GA package based on standards: real encoding, stochastic remainder selection with fitness scaling factor of 2.0, crossover at rate 0.2 performed by random barycentric combination, both offsprings replacing the parents, and mutation at rate 0.2 by addition of gaussian noise of standard deviation 0.5. The algorithm stops after 50 generations without improvement.

3.2 SAMPLING THE FEASIBLE REGION

The scheme we propose is contained in Figure 2 for problem $P1$:

- A randomly initialized population evolves to minimize the violation of the first constraint, until a given percentage of the population (we call the *flip threshold*, and denote by ϕ) is feasible for the first constraint.
- This population is then the starting point for the second phase of evolution, minimizing the violation of the second constraint. During that phase, points that are not feasible for the first constraint have zero fitness, and thus disappear due to selection. The stop criterion is again the satisfaction of the second constraint by the flip threshold percentage ϕ of the population.
- We now have a feasible population to start optimizing function $\mathcal{F}$. During this last step, non feasible points are in turn eliminated through selection.

Of course, this is the ideal case. Let us now detail some of the key features.

3.3 FITNESS FUNCTIONS

It is wellknown that the shape of the fitness landscape is of utter importance for the behaviour of GAs. During the minimization of constraint C_i, we set the fitness function to $M - C_i$, for a "sufficiently large" positive number M. But two reasons prohibit an absolute choice of M:

- It is not always easy to get even an approximation of what the maximum of the violation of some

[2] though we did find sucessful weights and *a priori* maximum values of the constraint violations for these simple and smooth problems.

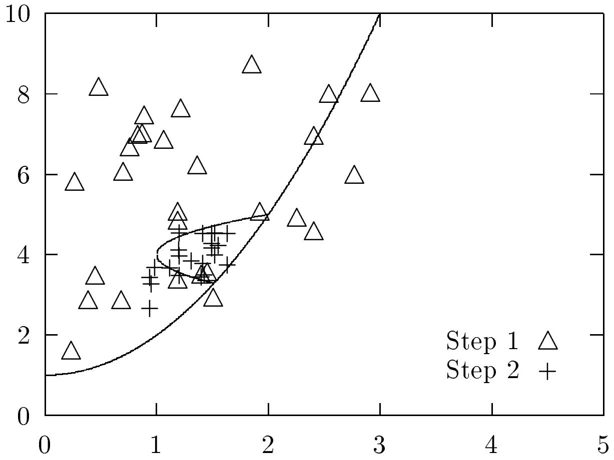

Figure 2: End of first steps (Problem $P1$).

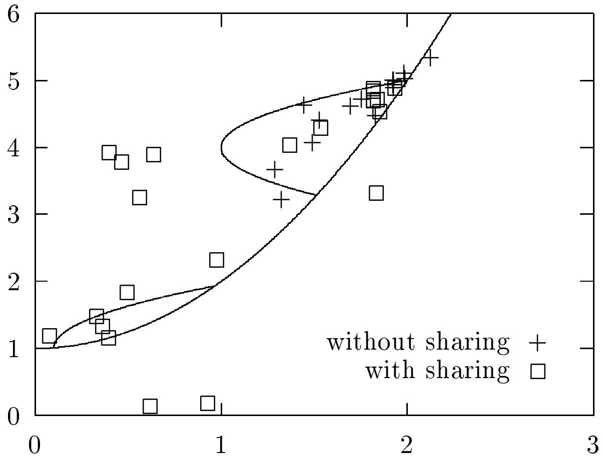

Figure 3: Effect of sharing (Problem $P2$).

constraint will be (see section 4 for instance) ; and

- choosing a too large constant leads to forbid any distinction between "nearly feasible" and "really feasible" points for the constraint C_i at hand, as the whole fitness landscape nearby the feasible region - with low constraint violation - will be flattened by the zero fitness assigned to the points failing to satisfy constraints $C_0, \ldots, C_{i-1}$.

This is why parameter M is adjusted dynamically at each generation.

At generation t, the maximum value of the constraint violation Γ_t is computed. But some fluke mutation can suddenly give very high constraint violation, in which case setting parameter M directly to Γ_t can have the same disastrous effect than choosing a too large constant value for M. So we want parameter M to be non-decreasing along generations.

The fitness function we used throughout the following experiments is defined at generation t by $(M_t - C_i)^+$, where M_t is the minimum of Γ_t and M_{t-1}.

3.4 GENETIC DIVERSITY

All experiments on problem $P1$ made with the preceeding algorithms converged to the global maximum (with pop. size 30 to 100, flip threshold from 0.5 to 0.9). The two first steps were achieved in 10 to 30 generations, the optimal point being reached after 100 to 150 more generations.

But the first experiments on problem $P2$ showed that the key feature in the Behavioural Memory paradigm is to sustain genetic diversity within the population: We wish the final population of each intermediate step to be the initial population of the following step, *sampled as uniformly as possible* over the feasible region of the current constraint.

- All the points of the target feasible region must admit the same fitness value, to avoid *any* convergence inside that region. In particular, the final objective function must not be taken into account before the final step.
- Premature convergence toward the first feasible points found by the algorithm must be avoided. And this is specially important when the feasible region is not a connected domain, as in $P2$. To this end, we use the sharing scheme as described in (Goldberg 89, Deb 89), together with restricted mating.

Figure 3 shows the repartition of the population at the end of step 2 for problem $P2$, with and without sharing: Without sharing, the smaller domain B is not sampled at all and the initial population of the third phase is completely contained in domain A. The right solution, which lies in B, will only be found depending on a lucky mutation, as the whole space between A and B gets zero fitness in the last step.

The choice of the sharing factor σ is important, but that of the flip thershold ϕ is as well, as demonstrates Table 1. For a population size of 50 on problem $P2$, it reports the results of ten independent runs: First the number of successes of the algorithm in finding the maximum in B and not in A, then the average number of generations it took to get the required percentage of feasible points for the second constraint [3].

[3]where *** indicates at least one run could not reach that percentage.

Table 1 : Effects of the Flip Threshold and the Sharing Factor

$\sigma \setminus \phi$	50 %	60 %	70 %	80 %
0	1 - 18	1 - 26	2 - 32	0 - 32
.01	6 - 26	8 - 35	7 - 40	5 - 71
.05	6 - 24	9 - 32	7 - 45	5 - 61
.1	7 - 33	9 - 36	6 - 54	8 - 87
.2	6 - 35	5 - 46	3 - 75	3 - 358
.3	8 - 47	5 - 55	3 - 144	3 - ***
.5	9 - 64	6 - 81	6 - 133	4 - ***

The time to correctly sample the feasible region increases with the flip threshold ϕ, which is natural, but also with the sharing factor σ.

Using sharing, every point can be thought of as a small elastic ball of radius σ, repelling other points of the population, and - possibly - sending them in other components of the feasible region. Therefore, the "amount" of feasible region so sampled increases with σ.

But, in order to satisfy the second constraint, a certain amount of such "balls" must get in the feasible region, whose size is finite - small in our case. So, as σ goes on increasing, it becomes more and more difficult to meet the requirement of constraint satisfaction for the same number of points. In which case the unavoidable genetic drift due to so numerous useless generations spoils the possible genetic diversity this sharing factor could have brought. And, of course, when σ and ϕ are too large, the required constraint satisfaction simply becomes impossible.

The sharing factor σ and the flip threshold ϕ must be adjusted together: the order of magnitude of σ can be approximated from below using large ϕ, and increasing σ until the required percentage of feasible points cannot be reached any more. Slightly decreasing σ should then allow to find good values for both σ and ϕ.

4 TRUSS STRUCTURE OPTIMIZATION

The previous scheme has been sucessfully applied to *the* test problems of truss structure optimization (Haug 79): The 10-bar (2D) and the 25-bar (3D) truss structures.

The design variables to optimize are here the section areas of the bars, the objective function to minimize is the weight of the structure, and the constraints are maximum values of the stress in the bars, to avoid collapsing of the structure.

These problems can be solved by many gradient-like numerical methods when the sections take continuous real values. But such methods cannot be used when some of the design variables are discrete: It could be the material the bars are made of, or even the possible section areas (if limited to manufactured items). And GAs indeed can solve *both* discrete and continuous problems, as we shall now briefly report (more details can be found in (Schoenauer 93)).

Only the section areas are taken here as design variables, taking values in some given real interval (*the continuous problem*) or in a given set of 36 preset values (*the discrete problem*). The genetic operators are those of section 3.1, "naturally" discretized in the discrete case.

The constraints are maximal values for the stress in every bar when some given loading is applied to the structure. There is no analytical way to express such stress values ; they have to be computed using some finite element method (which takes up to a few tenth of second on a 68040-based computer). Moreover, it is not easy to guess some upper bound for these stress values: The mechanical model will go on giving huge values long after the structure has in fact collapsed.

We use 100 structures in the population ; the given results express averages on 5 runs with different initial populations. In the discrete case, around 30 generations are needed to satisfy all the stress constraints in 70% of the population (being of the same nature, the 10 constraints are considered in one single step). The convergence toward the solution is reached in 500 to 800 generations. In the continuous case, about 100 generations are necessary to sample the feasible region, and about 5000 generations are necessary to reach a good approximation of the solution[4].

On both problems, most of the penalty functions, first tried with GAs, failed - converging towards local maxima, or failing to stabilize.

To end this section, it must be stated that GAs have a huge field of application in Structure Optimization: Problems involving qualitative variables (e.g. the material), and even the topological optimization problem, where the number and connections of the elements are unknown. And all these problems involve constraints.

5 GENERATING SOFTWARE TEST DATA

Software testing is generally considered to be the most significant and labour intensive phase in the software lifecycle because of both of its economic consequences and technical complexity. The major part of the testing effort is the Test Data Generation process (TDG), nowadays poorly automated, which consists in choosing a representative subset of inputs then executing

[4] which is more than 50 times slower than classical gradient methods!

the program and verifying that the results are in accordance with the specifications. Our concern is the automation of the structural test data generation task, that is the generation of test data (TD) which are able to execute (we also say "cover" or "sensitize") selected substructures of the software (i.e. statements, crucial paths, etc). It has been established that providing a general automated tool for TDG is formally impossible (Howden 77). In fact the sensitization of a software substructure can be considered as the satisfaction of a set of conditions, that is the conditions which lead the execution flow to visit the specific statement. For instance if we wish to execute the `Do_exception` statement of the following program (a and b are inputs, f_1 and f_2 are arbitrary functions) :

```
read(a,b);
a = f1(a,b);
if (a < b) {
   b = f2(a,b);
   if (a > b) Do_Exception;
...
```

we have to satisfy the two conditional `if` statements. Moreover the second condition cannot be established before the first one has been satisfied (we need to know the new values of variables a and b after the execution of function f_2).

Three main approaches tackle the automatic TDG problem :

- The predominant approach is random testing (Duran 80): Input values are randomly generated and the substructures covered are observed by means of instrumentation (additional statements tracking program execution). The limitation of this approach comes from its blindness: Too nested structures are never executed and equality conditions are rarely satisfied.
- In symbolic execution, variables are assigned symbolic values and program statements are executed symbolically by means of algebraic manipulations (Clarke 83). However, dynamic informations (like pointers, arrays or number of iterations) must be hypothesized during symbolic execution, leading to an unavoidable combinatorial explosion when non toy programs are considered.
- In the dynamic approach (Korel 90), an initial random set of TD is generated and progressively adapted during consecutive executions with the goal of sensitizing the substructure chosen by a test strategy.

The condition satisfaction task (e.g. satisfy statement if $(a < b)$ ...) is expressed in terms of optimization (e.g. minimize the expression $a - b$ where a and b are the observed values just before the first if statement). Classic gradient-like optimization methods meet a number of difficulties. First there is no analytic way (unless we adopt symbolic execution) to express the condition to be optimized. Secondly, a problem of lack of orthogonality comes from the interdependency between parameters. Moreover, the values may evolve discontinuously during the execution of the program. Lastly is the problem of local optima.

All these observations lead to adopt GA for the automatic TDG using a dynamic approach. Given the stepwise nature of the constraint satisfaction process, we adopted the Behavioral Memory paradigm as previously described. The constrained GA optimisation algorithm is embedded into an automatic test data generator named TAGGER. The inner loop is the following: Given a structure,

1. A simple Data Flow Analysis algorithm as described in (Xanthakis 92) is used to determine the subset of inputs which affect the conditions appearing at the structure to cover.
2. An initial random population of values for this subset is produced.
3. For each constraint, the popuulation evolves until some given percentage of the population satisfies it, allowing to compute the next one. When one individual satisfies this last constraint, the given structure is covered. The loop exits, asking the generator for the next structure to cover.

The TAGGER prototype has been used on a number of critical programs, though not yet in an industrial environment. Nonetheless the results so far are extremely promising. Coverage metrics of 100 % of branches are often achieved with a performance well beyond by random testing (5 - 35 times faster when a solution can be found by random testing).

6 CONCLUSION

We presented a general method to handle constraints in GAs: First sample the feasible region by genetically evolving a population in the whole search space minimizing some constraint violation ; then, evolve the resulting population to maximise the initial objective function on the feasible region so sampled.

Our method is problem independent. We emphasize it can handle any computable constraints. It also allows GAs to be independant of the fitness landscape outside the feasible region: It can be non defined at some points, or have numerous local optima, ... Moreover, it optimizes in its last step the exact objective function, not some artificial transformation.

The counterparts are an increased computational cost as each step is a partial GA optimization in itself, and the need to maintain genetic diversity during evolution, whatever scheme is used. And when using the sharing scheme, the sharing factor must be adjusted very carefully.

We do not claim to outperform all other methods for constraints handling using GAs. In particular when feasible region is large, using penalty function may be a cheaper strategy. And designing problem-specific operators will probably, when possible, give better results than any general method.

But in many problems, like in engineering optimization for instance, the feasible region is small and quite sparse in the whole search space, and the constraints are available only through some heavy numerical computation. These restrictions forbid the use of standard methods (the feasible region is not convex, nor does the solution lie on its boundary) as well as the design of specific closed genetic operators. Moreover in problems like the generation of software test data, the method we propose seems to be the only one to be able to handle "hierarchically computable" constraints.

References

(Clarke 83) L.A. Clarke, D.J. Richardson, Symbolically evaluation - an aid to testing and evaluation, University of Massachussets Technical Report, 83-41, 1983.

(Davis 87) L. Davis, M. Steenstrup, Genetic algorithms and simulated annealing : an overview, in (Davis ed. 87), pp 1-11.

(Davis 91) T. E. Davis, J. C. Principe, A simulated annealing-like convergence theory for the simple genetic algorithm, in (ICGA 91) pp 174-181.

(Deb 89) K. Deb, D. E. Goldberg, An investigation of niche and species formation in genetic function optimization, in (ICGA 89), pp 42-50.

(Fletcher 87) R. Fletcher, *Practical Methods of Optimization*, second edition, John Wiley and Sons, Inc., New York, 1987.

(de Garis 90) H. de Garis, Genetic Programming : building artificial nervous systems using genetically programmed neural networks modules, in *Proceedings of the 7^{th} International Conference on Machine Learning*, R. Porter B. Mooney Eds, Morgan Kaufmann, 1990, pp 132-139.

(Haug 79) E.J. Haug, J.S. Arrora, *Applied Optimal Design,* John Wiley and Sons, Inc., New York, 1979.

(De Jong 92) K. A. De Jong, Are genetic algorithms function optimizers ?, in (PPSN 92), pp 3-13.

(Desquilbet 92) C. Desquilbet, F. Sassus, sous la direction de M. Schoenauer, Reconnaissance d'un detail d'une image par algorithmes génétiques, technical report, École Polytechnique, Palaiseau, Mars 1992.

(Duran 80) J.W. Duran, S.C. Naftos, An evaluation of random testing, *IEEE Transactions in Software Engineering*, 10 (4), pp 438-444, 1980.

(Goldberg 89) D. E. Goldberg, *Genetic algorithms in search, optimization and machine learning*, Addison Wesley, 1989.

(Grefenstette 87) J. J. Grefenstette, Incorporating domain specific knowledge into genetic algorithms, in (Davis ed. 87), pp 42-60.

(Holland 75) J. Holland, *Adaptation in natural and artificial systems*, University of Michigan Press, Ann Harbor, 1975.

(Howden 77) W.E. Howden, Symbolic testing and the DISSECT symbolic evaluation system, *IEEE Transactions in Software Engineering*, 3 (4), pp 266-278, 1977.

(Korel 90) B. Korel, Automated software test data generation, *IEEE Transactions in Software Engineering*, 16 (8), 1990.

(Michalewicz 91) Z. Michalewicz, C. Z. Janikov, Handling constraints in genetic algorithms, in (ICGA 91), pp 151-157.

(Paredis 92) J. Paredis, Exploiting constraints as background knowledge for genetic algorithms : a case study, in (PPSN 92), pp 229-238.

(Radcliffe 91) N. J. Radcliffe, Equivalence Class Analysis of Genetic Algorithms, in *Complex Systems* **5**, pp 183-205, 1991.

(Richardson 89) J. T. Richardson, M. R. Palmer, G. Liepins, M. Hilliard, Some guidelines for genetic algorithms with penalty functions, in (ICGA 89) pp 191-197.

(Schoenauer 93) M. Schoenauer, Z. Wu, Optimisation discrète de structures par Algorithmes Génétiques, *Actes du Colloque National en Calcul de Structures*, Giens, Mai 1993.

(Xanthakis 92) S. Xanthakis, C. Skourlas, An automatic tool for data flow verification of structured program, *Proceedings of the 2^{nd} International Conference on Software Quality*, Toulouse, 1992.

(Zhigljavski 91) A. A. Zhigljavski, *Theory of global random search*, Chap. 5, Kluwer Academic Publishers, 1991.

(Davis ed. 87) L. Davis Editor, *Genetic algorithms and simulated annealing*, Morgan Kauffman Publishers, 1987.

(ICGA 89) J. Shaffer Editor, *Proceedings of the 3^{rd} International Conference on Genetic Algoriothms*, George Mason University, Morgan Kauffman Publishers, June 4-7 1989.

(ICGA 91) R. K. Belew, L. B. Booker Editors, *Proceedings of the 4^{th} International Conference on Genetic Algoriothms*, University of California, San Diego, Morgan Kauffman Publishers, June 13-16 1991.

(PPSN 92) R. Manner, B. Manderick, *Proceeding of the second conference on Parallel Problem Solving from Nature*, Free University of Brussel, North Holland Publishers, 1992.