Constraint Consistent Genetic Algorithms

Ryszard Kowalczyk

CSIRO Mathematical and Information Sciences
723 Swanston Street, Carlton 3053, Australia
richard.kowalczyk@cmis.csiro.au

Abstract-- **It has commonly been acknowledged that solving constrained problems with a variety of complex constraints is a challenging task for genetic algorithms (GA). Existing methods to handle constraints in GA are often computationally expensive, problem dependent or constraint specific. In this paper we introduce an idea of constraint consistent GA (CCGA) as an attempt to overcome those drawbacks. Constraint handling is based on general constraint consistency methods that prune the search space and thus reduce the search effort in CCGA. Unfeasible solutions are detected and eliminated from the search space at each stage of CCGA simulation process to support genetic operations in producing feasible solutions. A number of well known standard genetic operators are adapted to take advantage of provided constraint consistency during initialization, crossover and mutation. Initial experiments indicate that in the terms of the solution quality and the number of iterations the constraint consistency based approach in CCGA can outperform other constraint handling methods in GA for a number of selected test problems.**

I. INTRODUCTION

Genetic algorithms (GA) provide an adaptive method of search using principles of evolutionary simulation [3]. A basic behavior of GA is based on applying evaluation, selection, crossover and mutation on populations of chromosomes representing prospective solutions. After generation of an initial population, GA iteratively applies those four basic steps to generate a number of subsequent populations with a hope that they contain better solutions.

GAs have proven to be efficient in solving a number of search problems including numerical and combinatorial optimisation [4, 6, 12, 13, 14, 16]. However, it has commonly been acknowledged that the real challenge for GAs (as for most search algorithms) is to solve constrained problems with a variety of complex constraints (e.g. non-linear, implicit or disjunctive constraints). Constraints usually limit a number of sought solutions and thus increase the unfeasible portion of the search space, making efficiency of GAs more critical. In addition, constraints often cause that the feasible search space is not convex and forms disjoint parts, making exploration of the search space very difficult.

Many approaches to handle constraints in GAs have been proposed. They are usually based on using penalty functions [1, 11], repairing infeasible solutions [11, 12], preserving feasibility of solutions [12], emphasizing the distinction between feasible and infeasible solutions [11, 12] or using hybrid methods [12, 14]. Although many proposed methods have been successful in handling constraints, they are often computationally expensive, problem dependent or constraint specific. Applying penalty to the infeasible solutions requires a careful selection of control parameters and is often computationally expensive in more constrained problems. Methods based on repairing infeasible solutions are usually good only for handling specific explicit constraints and may be inefficient for implicit constraints. Moreover, these methods are often problem domain specific. Preserving feasibility of solutions usually requires problem specific chromosome representation and/or genetic operators. Some more general methods are able to preserve feasible solutions only for specific types of constraint (e.g. linear constraints). Method based on emphasizing the distinction between feasible and infeasible solutions may face similar problems. Therefore the quest for efficient and general methods of handling constraints in GAs still continues.

It has generally been recognized that pruning the search space to its feasible portion is beneficial in most search algorithms. This has also been acknowledged in GAs and some methods preserving feasibility of solutions have been based on that principle. For example, a genetic algorithm for numerical optimisation for constrained problems Genocop [12] has successfully been used in solving linearly constrained optimisation problems. It has also been used as the main component in non-linear optimisation with Genocop II and Genocop III [12]. Genocop assumes only linear constraints that guarantee the convex search space [10]. The main idea there is to eliminate the equality constraints and to use special genetic operators to maintain feasibility of the solutions. The inequality constraints are used to generate bounds for all variables dynamically during the search process. This specific technique is very similar to a general concept of constraint consistency commonly used in constraint satisfaction problems [7, 8] and also successfully tried in GAs (e.g. in repairing infeasible solutions [15]).

Constraint consistency is a principle used in modeling and solving a broad class of constrained problems represented as a constraint satisfaction problem (CSP) [7, 8]. CSP is defined in the terms of variables with the associated domains and constraints acting between the variables. Constraint consistency methods are often

0-7803-3949-5/97/$10.00 ©1997 IEEE

used in CSP to ensure that assignments to the problem variables satisfy all constraints posted on the variables. The idea of constraint consistency is to prune the search by preventing variable instantiation that are not consistent with the constraints of the problem. Many methods have been proposed to provide constraint consistency at different levels (e.g. arc-consistency, path-consistency, k-consistency) [7, 8]. Most popular are constraint propagation based methods which have been proven to be an efficient and flexible approach to support solving many real-world constrained problems. Constraint propagation eliminates infeasible solutions from the search space, i.e. the values that cannot satisfy the constraints at the required level of constraint consistency are removed from the domains of the variables. It seems that constraint consistency methods can also be used to handle constraints in GAs to advantage.

In this paper we present an idea of constraint consistent GA (CCGA) where the constraint consistency methods are used to handle constraints. CSP based representation and constraint consistency in CCGA are briefly presented in section 2. A number of constrained genetic operators are discussed in section 3. Some experiments with CCGA used for numerical optimisation are described in section 4. Finally, in section 5 concluding remarks are presented.

II. CSP Based Representation and Constraint Consistency in CCGA

Binary strings have traditionally been used to represent chromosomes in GAs [3]. However more recently other chromosome representations have emerged that enhance the range of possible chromosome components (e.g. character, floating point) and its structure (e.g. matrix, list, tree) [2, 12]. It allows one to reflect the structure of the problem considered and to represent the problem variables in the chromosome directly without the need for coding them in the binary form. For example, a common representation in numerical optimisation is a floating point vector consisting of the variables of the problem considered. Such a direct representation has been shown to provide many benefits to GA in solving a number of optimisation problems [2, 12].

Extension of direct representation to constrained variables leads to CSP based representation which is used in CCGA. Each chromosome in CCGA representing a vector of constrained variables can be interpreted as an instance of the same CSP defined in the following form:

- a set of variables $x = \{x_i\}$, $i = 1,\ldots,n$, represented as a vector $\bar{x} = \langle x_i \rangle$, $i = 1,\ldots,n$ in a chromosome,
- a set of domains $D = \{D_i\}$, $i = 1,\ldots,n$ associated with the corresponding variables, where each domain consists of values that each variable can assume, i.e. $x_i \in D_i, i = 1,\ldots,n$. The domains can be finite or infinite (presented CCGA uses domains consisting of floating point values),
- a set of constraints $C = \{C_k(x)\}$ between the variables.

A solution of CSP is an assignment that all constraints are satisfied at the same time. Constraint consistency may be used in CSP to ensure that an instantiation of the problem variables satisfies all constraints posted on the variables.

CSP can be interpreted as a constraint network which consists of nodes containing the variables with the associated domains and arcs between nodes representing constraints acting between these variables. A constraint network represents a feasible search space consisting only of valid solutions of the problem if it is k-consistent. A constraint network is k-consistent if for any instantiation of any k-1 variables satisfying all constraints among those variables, it is possible to instantiate any kth variable such that the assignment satisfies all constraints among the k variables. A concept of k-consistency is generalization of constraint consistency [7, 8]. Specific cases of constraint consistency are node-consistency and arc-consistency that correspond to k-consistency for k=1 and k=2, respectively. Although in general, node and arc consistency do not always ensure that a solution of a given CSP exists, those levels of constraint consistency are most practical in simplifying the original problem. They provide a good balance between the benefits gained from reducing the search space and the computational effort needed to provide constraint consistency at the required level. Therefore constraint arc-consistency is also used to support constraint handling in GAs presented in the paper. For some examples and more discussion on arc-consistency refer to literature (e.g. [7, 8]).

CSP based representation and constraint consistency methods do not make any assumption regarding the considered problem domain and search strategies used during optimization. Therefore they can also be used to support constraint handling in GAs. They can handle constraints independently checking and providing a required constraint consistency for each generated chromosome simultaneously with the GA simulation process. As opposed to other techniques to handle constraints in GA the variable domains do not need to be intervals defined by their bounds. The domains can contain a number of disjoint segments resulting in non-convex search spaces. In addition, there is no assumption made on a type of constraints considered, e.g. explicit and implicit, linear and non-linear, relational and arithmetic constraints may be used.

III. Constrained Genetic Operators

Constraint consistency can be used during all stages of the generation process in GAs to advantage. It can efficiently be provided at the level of arc-consistency with the use of constraint propagation techniques. Constraint propagation reduces the search space by removing infeasible values from the domain of any variable according to the constraint consistency definition. The initial reduction of the search space is performed when all constraints are activated (posted). The reduction is

continued when the domain of any variable changes or is instantiated in a particular case. A number of constrained genetic operators taking advantage of constraint consistency (in particular arc-consistency as used in CCGA) can be defined as extensions of well known standard genetic operators as follows:

A. Constrained Initialization

Constraint consistency can support initialization of the domains for the constrained variables and generation of an initial population. After posting the constraints on the problem variables their domains are initialized in such a way that they are arc-consistent with all constraints posted on the variables, i.e. they do not consist of values that cannot be a part of any feasible solution. Thanks to constraint consistency checking the variable domains are dynamically updated to consist of values that are considered by genetic operators as possible instantiations of problem variables during the GA simulation process.

Once the domains of the variables are initialized they are used in generation of the initial population of chromosomes representing feasible solutions. All variables in each chromosome are instantiated with random values from their domains:

$$\vec{x}^0 = \langle x_1, x_2, \ldots, x_n \rangle, \quad x_i \in D_i .$$

A number of instantiation strategies can be used to generate feasible chromosomes. The variables can be instantiated in a random order or heuristics may be used to minimize the effort required to find feasible solutions (e.g. more constrained variables or variables with smaller domains may be instantiated first). However, when using heuristics one should ensure that the initialization process provides sufficient diversity of the initial population. Constraint propagation ensures that the domains of all variables consist only of values satisfying constraints according to the arc-consistency definition. It is possible that during the variable instantiation process some domains may become empty. In such cases a limited backtracking can be performed to try other random values. If it requires a significant computational effort the currently generated chromosome can be abandoned or included in the population as a partially feasible solution.

B. Constrained Crossover

Crossover is the main reproduction mechanism creating new chromosomes from the existing ones with a hope that they represent better solutions. A number of well-known crossover operators can take advantage of provided constraint arc-consistency.

- *Constrained uniform crossover*

Standard uniform crossover works by selecting two parent individuals $\vec{x} = \langle x_1, \ldots, x_n \rangle$, $\vec{y} = \langle y_1, \ldots, y_n \rangle$ from the population and randomly exchanging elements between both individuals to form two new offspring as follows:

$$\vec{x}' = \langle x'_1, \ldots, x'_n \rangle, \quad \vec{y}' = \langle y'_1, \ldots, y'_n \rangle$$

where

$$x'_i = \begin{cases} x_i \\ y_i \end{cases} \quad and \quad y'_i = \begin{cases} y_i & \text{if a random binary digit is 0} \\ x_i & \text{if a random binary digit is 1} \end{cases}$$

for $i = 1, \ldots, n$.

However, even if crossing two feasible solutions there is a possibility that new offspring may be infeasible. Constraint consistency can be used to support uniform crossover in generating feasible offspring. Constrained uniform crossover takes into account all constraints and generates offspring that are defined as follows:

$$\vec{x}' = \vec{\delta} \cdot \vec{x} + (1 - \vec{\delta}) \cdot \vec{y}, \quad \vec{y}' = \vec{\delta} \cdot \vec{y} + (1 - \vec{\delta}) \cdot \vec{x}$$

where $\vec{\delta} = \langle \delta_1, \ldots, \delta_n \rangle$ is a vector of auxiliary constrained binary variables $\delta_i \in D_i^{\delta} = \{0,1\}$. These variables are instantiated in a random order taking into account constraint consistency requirements imposed on the offspring. Constraint propagation ensures that the domains of not yet instantiated variables do not consist of binary values that cannot produce feasible offspring. When any domain becomes empty a limited backtracking can be performed to try another random values. It should be noted that it is possible that the offspring are the same as the parents. To prevent this situation additional constraints can be posted on the constrained binary variables as follows:

$$0 < \sum_{i=1}^{n} \delta_i < n .$$

When two parents are not able to produce any feasible offspring the domains of the binary variables become empty and such a situation is identified instantly by constraint consistency checking. When it is difficult to generate two feasible offspring (e.g. high computational effort or parents are not able to produce feasible offspring) one can choose to generate partially feasible offspring or generate only one feasible offspring.

- *Constrained m-point crossover*

Standard m-point crossover works by selecting two parent individuals $\vec{x} = \langle x_1, \ldots, x_n \rangle$, $\vec{y} = \langle y_1, \ldots, y_n \rangle$ from the population, choosing m crossover points $k_i \in \{1, \ldots, n-1\}, i = 1, \ldots, m$ at random, and exchanging the segments bounded by the crossover points between both individuals to form two new offspring. Constraint consistency can support m-point crossover by providing sets of valid crossover points that can produce feasible offspring in a constrained search space.

For example let us consider one-point crossover defined as follows:

$$\vec{x}' = \langle x_1, \ldots, x_k, y_{k+1}, \ldots, y_n \rangle, \quad \vec{y}' = \langle y_1, \ldots, y_k, x_{k+1}, \ldots, x_n \rangle .$$

Similarly to constrained uniform crossover the offspring produced by constrained one-point crossover can be defined as follows:

$$\vec{x}' = \langle \delta_{1,k} x_1 + (1-\delta_{1,k}) y_1, \ldots, \delta_{n,k} x_n + (1-\delta_{n,k}) y_n \rangle$$
$$\vec{y}' = \langle \delta_{1,k} y_1 + (1-\delta_{1,k}) x_1, \ldots, \delta_{n,k} y_n + (1-\delta_{n,k}) x_n \rangle$$

where

$$\delta_{i,k} = \begin{cases} 0 & \text{for } i > k \\ 1 & \text{otherwise} \end{cases}$$

and $k \in D^k = \{1,\ldots,n-1\}$ is an auxiliary constrained variable with the domain reduced to a set of valid crossover points for one-point crossover. A crossover point is chosen in random from the domain of k. If after constraint propagation the domain becomes empty than it means that the parents cannot produce feasible offspring. In such a situation new parents can be selected or offspring can be generated with any random crossover point and included in the population as partially feasible solutions.

Similarly, constraint consistency can support other constrained m-point crossovers such as constrained two-point crossover and others.

- *Constrained arithmetic crossover*

Arithmetic crossover operators have been introduced for GAs with real valued representation applied to convex search spaces [10]. Using principles of constraint consistency they can also be applied to other search spaces.

The so-called simple arithmetic crossover chooses a random crossover point k and crosses the parents after k^{th} position applying a contraction coefficient $a \in [0,1]$ to guarantee that the offspring fall into the feasible convex search space as follows [10]:

$$\vec{x}' = \langle x_1,\ldots,x_k, ay_{k+1} + (1-a)x_{k+1},\ldots, ay_n + (1-a)x_n \rangle$$
$$\vec{y}' = \langle y_1,\ldots,y_k, ax_{k+1} + (1-a)y_{k+1},\ldots, ax_n + (1-a)y_n \rangle$$

To consider non-convex search spaces constrained arithmetic crossover treats the contraction coefficient as a variable $a \in D^a = [0,1]$ constrained to produce feasible offspring. The offspring are generated with a random value for a chosen from its feasible domain. To ensure that the offspring are different from the parents the contraction coefficient can also be constrained to be grater than 0, i.e. $a > 0$. Alternatively, the coefficient a can be instantiated with the upper bound of its domain that gives the largest information exchange between the parents.

Similarly we can define single constrained arithmetic crossover that affects only one variable in each parent and the whole constrained arithmetic crossover that creates offspring as the full linear combinations of two parents.

- *Constrained heuristic crossover*

Heuristic crossover uses values of the objective function in determining the direction of the search and produces only one offspring [10]. The operator generates a single offspring from two parents as follows:

$$\vec{x}' = r \cdot (\vec{x} - \vec{y}) + \vec{x}$$

where r is a random number $r \in D^r = [0,1]$, and the parent $\vec{x}$ is not worse than the parent $\vec{y}$. In constrained heuristic crossover, r is treated as a constrained variable and constraint consistency ensures that its domain consists only of values producing feasible offspring. A value for r is chosen randomly from its domain or may be equal to the upper bound of the domain enforcing greater search gradient.

C. Constrained Mutation

Mutation changes individual chromosomes to provide additional variability to the reproduction process. A number of mutation operators can be adapted to take advantage of constraint consistency.

- *Constrained uniform mutation*

Uniform mutation selects a parent individual $\vec{x} = \langle x_1,\ldots,x_n \rangle$ from the population, chooses a mutation point $k \in \{1,\ldots,n\}$ at random, and modifies the randomly selected element to form new offspring as follows:

$$\vec{x}' = \langle x_1,\ldots,x_k',\ldots,x_n \rangle$$

where x_k' is a random value. In constrained uniform mutation the value for x_k' is selected randomly from the domain of variable $x_k \in D_k$ consistent with the remaining instantiated variables according to all constraints posted. After a random selection of the position for mutation k (uniform probability distribution), the corresponding variable domain is reset so it consists only of feasible values for x_k'. Then it is instantiated randomly in a such way that together with the other instantiated variables represent a feasible solution in the search space.

- *Constrained boundary mutation*

Constrained boundary mutation is a specific case of constrained uniform mutation. A randomly selected variable x_k' takes either lower or upper bound of its feasible domain at random, i.e.

$$x'_k = \begin{cases} \max(D_k) & \text{if a random binary digit is 0} \\ \min(D_k) & \text{if a random binary digit is 1} \end{cases}$$

D. Handling Partially Feasible Solutions

In general the aim of constraint consistency in CCGA is to ensure that genetic operators produce feasible solutions. However one may prefer to consider partially feasible chromosomes as a good "genetic" material for further reproduction too. It seems that most existing methods to handle infeasible solutions in GA may be used in CCGA (e.g. penalizing, repairing). Some of them can also take advantage of constraint consistency.

For example simple constrained repairing can use constraint consistency to repair an infeasible solution by randomly selecting variables in the chromosome and resetting their domains until the domains are not empty (i.e. a constraint network is arc-consistency). Then the reset variables can be instantiated with random values from their domains as described for constrained uniform initialization. Because the main objective of the current CCGA is to generate feasible solutions only the constrained repairing has not been included in the reported experiments and therefore it will be not discussed further in the paper.

IV. EXPERIMENTS

A number of experiments with selected numerical optimisation problems were performed to demonstrate CCGA. The problems were simulated with the use of a steady state GA with the 25% replacement rate, the real valued chromosome representation for constrained variables and a number of constrained genetic operators discusses in the previous section. It used the roulette wheel selection and sigma truncation scaling schemes. Constraint consistency was provided at the level of arc-consistency through dynamic constraint propagation. Ten independent runs for each test problem were performed. The population size was set to 70, number iterations limited to 2000 and probabilities of cross-over/mutation set to 0.6/0.1 and 0.5/0.5, respectively. In implementation of CCGA we used a C++ based GA toolkit (GALib) available from MIT [17] and a commercial constraint programming tool (Ilog Solver) [5] which uses propagation to provide constraint arc-consistency.

In the experiments we used selected test problems for constrained numerical optimisation reported in [12] where they were solved with the use of a number of different constraint handling techniques such as penalty coefficients, dynamic penalties, penalty with constraint ordering, death penalty, Genocop and Genocop II (for more details refer to [12]). In most cases CCGA produced better results within the smaller number of iterations. Due to the hardware differences/results availability the time based performance was difficult to compare. However it should be noted that the time requirements of CCGA were sometimes worse than expected. It is due partially to additional computations required to propagate constraints and partially to the current experimental implementation of CCGA which aims in easy use rather than the optimal performance.

Below are two examples of linearly and non-linearly constrained problems tested with CCGA. The results obtained with CCGA are listed together with the best results reported in [12].

- *Linearly constrained problem*

Minimize the function

$$F1(\vec{x},y) = -10.5x_1 - 7.5x_2 - 3.5x_3 - 2.5x_4 - 1.5x_5 - 10y - 0.5\sum_{i=1}^{5} x_i^2$$

where:

$$0 \le x_i \le 1, \quad i = 1,\ldots,5, \quad 0 \le y,$$
$$6x_1 + 3x_2 + 3x_3 + 2x_4 + x_5 \le 6.5, \quad 10x_1 + 10x_3 + y \le 20$$

The function has the global minimum $F1(\vec{x},y) = -213.00$ for $(\vec{x},y) = \langle 0,1,0,1,1,20\rangle$. CCGA produced the optimum in all runs. The number of iterations required to obtain the optimum was usually about 20 whereas the best performing GA reported in [12] required about 1000 iterations. Fig. 1 shows a typical performance of CCGA for the function F1.

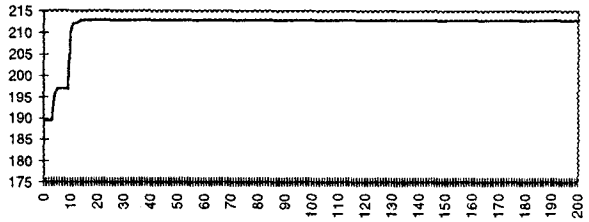

Fig. 1. Performance of CCGA (0.5/0.5) for F1

- *Non-linearly constrained problem*

Minimize the function

$$G2(\vec{x}) = x_1 + x_2 + x_3$$

where:

$$100 \le x_1 \le 10000, \quad 1000 \le x_i \le 10000, \quad i = 2,3,$$
$$10 \le x_i \le 1000, \quad i = 4,\ldots,8,$$
$$1 - 0.0025(x_4 + x_6) \ge 0, \quad 1 - 0.0025(x_5 + x_7 - x_4) \ge 0,$$
$$1 - 0.01(x_8 - x_5) \ge 0,$$
$$x_1x_6 - 833.33252x_4 - 100x_1 + 83333.333 \ge 0,$$
$$x_2x_7 - 1250x_5 - x_2x_4 + 1250x_4 \ge 0,$$
$$x_3x_8 - 1250000 - x_3x_5 + 2500x_5 \ge 0$$

The function has the global optimum $G2(\vec{x}) = 7049.330923$ for $\vec{x} = \langle 579.3167, 1359.943, 5110.071, 182.0174, 295.5985, 217.9799, 286.4162, 295.5979\rangle$. The best result reached with CCGA was $G2(\vec{x}) = 7063.95605$ for $\vec{x} = \langle 538.665649, 1313.42822, 5211.8623, 178.360245, 291.645691, 221.564026, 286.506195, 391.590393\rangle$. The results obtained during the experiments are presented in the table below.

10 runs	best	median	worst
[12]	7377.979	8206.151	9652.901
CCGA (0.1/0.6)	7102.47559	7282.26123	9588.50195
CCGA (0.5/0.5)	7063.95605	7310.10449	7854.61475

In most runs CCGA was able to provide very good solutions in the first 200 iterations. Fig. 2 shows a typical performance of CCGA for the function G2.

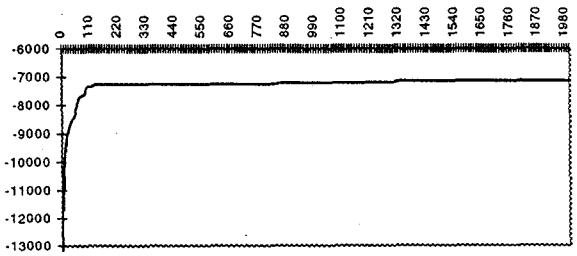

Fig. 2. Performance of CCGA (0.5/0.5) for G2

V. CONCLUSION

In the paper we have presented an idea of constraint consistent GA (CCGA) which aims to handle constraints in a general and efficient way. Constraint handling in CCGA is based on general constraint consistency methods that prune the search space and thus reduce the search effort in CCGA. A number of well known standard genetic operators are adapted to take advantage of provided constraint consistency during initialization, crossover and mutation. Initial experiments indicate that in the terms of the solution quality and the number of iterations the constraint consistency based approach in CCGA can outperform other constraint handling methods in GA for a number of selected test problems.

The main objective of further research is to analyze the overall performance of CCGA considering the additional computational effort introduced by constraint propagation. In addition, further work is to improve implementation of CCGA, experiment with other genetic operators and more comprehensive tests with other constrained problems. For example, the initial experiments with constraining genetic operators to produce better solutions during each iteration of GA are being performed and give encouraging results so far. They will be reported in a separate paper.

REFERENCES

[1] Carlson S.E. "A general method for handling constraints in genetic algorithms". *Proc. 2nd Annual Joint Conference on Information Science*, 1995, pp. 663-666.

[2] Corcoran A.L. & Sen S. "Using real-valued genetic algorithms to evolve rule sets for classification", *Proc. IEEE Conf. on Evolutionary Comp*, 1994, pp. 120-124

[3] Holland J. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, 1975.

[4] Homaifar A., Lai S. & Qi X. "Constrained optimisation via genetic algorithms". *Simulation* 62 (4), 1994, 242-254.

[5] Ilog. *Ilog Solver, ver. 3.1*. ILOG, 1996.

[6] Khuri S., Bäck T. & Heitkötter J. "An Evolutionary Approach to Combinatorial Optimization Problems". *Proc. CSC'94*, 1994.

[7] Kumar V. "Algorithms for Constraint-Satisfaction Problems: A Survey". *AI Magazine*, Spring 1992, pp. 32-44.

[8] Mackworth A.K. *Constraint satisfaction*. S.C. Shapiro (Ed.) *Encyclopedia of Artificial Intelligence*, John Wiley & Sons, 1990, pp. 205-211.

[9] Michalewicz Z. & Janikow C.Z. "Handling constraints in genetic algorithms". *Proc. 4th International Conference on Genetic Algorithms*, 1991, pp. 151-157.

[10] Michalewicz Z., Logan T. & Swaminathan S.. "Evolutionary operators for continuous convex parameter spaces". *Proc. 3rd Annual Conference on Evolutionary Programming*, 1994, pp. 84-97. World Scientific.

[11] Michalewicz Z. "A survey of constraint handling techniques in evolutionary computation methods". *Proc. of the 4th Annual Conference on Evolutionary Programming*, 1995, pp. 135-155.

[12] Michalewicz Z. *Genetic Algorithms + Data Structures = Evolution Programs*. Springer-Verlag, 3rd edition, New York, 1995.

[13] Michalewicz Z., Dasgupta D., Leriche R. & Schoenauer M.. "Evolutionary algorithms for Constrained Engineering Problems". *Computers & Industrial Engineering Journal*, vol 30(2), April 1996

[14] Le Riche R.G., Knopf-Lenoir C. & Haftka R.T. "A Segregated Genetic Algorithm for Constrained Structural Optimization". *Proc. 6th International Conference on Genetic Algorithms*, Pittsburg, USA, Morgan Kaufmann, July 1995.

[15] Paredis J. "Genetic State-Space Search for Constrained Optimisation Problems". *Proc. 13th International Joint Conference on Artificial Intelligence*, Cambery, France, 1993, pp. 967-972.

[16] Schoenauer M. & Xanthakis S. "Constrained GA optimization". *Proc. 4th International Conference on Genetic Algorithms*, Urbana Champaign, Conference on Genetic Algorithms, July 1993

[17] Wall M. *GALib: A C++ Library of Genetic Algorithm Components, ver. 2.4*. MIT, 1996.