A criteria to select genetic operators for solving CSP

María Cristina Riff Rojas
Universidad Técnica Federico Santa María
Casilla 110-V
Valparaíso
CHILE
E-mail: `mcriff@inf.utfsm.cl`

Abstract

Our interest is to define evolutionary algorithms to solve Constraint Satisfaction Problems (CSP), which include benefits of traditional resolution methods of CSPs as well as inherent characteristics of these kind of problems.
In this paper we propose a criterion to be able to evaluate the performance of genetic operators within evolutionary algorithms that solve CSP.

Keywords: *Evolutionary Algorithms, Constraint Satisfaction, Specialized genetic operators*

A criteria to select genetic operators for solving CSP

1 Introduction

This paper involves two research areas: Constraint Satisfaction Problems (CSP) with finite domains and evolutionary methods. We focus our attention on genetic operators. CSPs are NP-hard problems. A finite domains CSP is a set of variables, their related domains and a set of constraints between these variables. The aim is to find a value for each variable from its respective domains, such that all the CSP constraints are satisfied, [13], [14], [11].

Evolutionary methods are based on the evolutionary theory and they are a particular class of stochastic search methods to solve optimization problems, [15]. They have been applied to solve Constraint Satisfaction Optimization Problems [23], and CSP [3], [6], [7], [8], [9], [21], [22], [10], arguing that they are efficient methods for large scale constraint problems resolution.

The methods currently proposed in the literature have focused their study on the genetic representation and reproduction mechanisms (specially in genetic operators). One of the most current problems for this kind of methods is about the performance evaluation of the genetic operators.

On the other hand, the constraint research community has more focused on developing techniques to improve the algorithms performance, using the constraints knowledge, [5], [13], [11], [2], for instance reducing the search tree. The algorithms that use a systematic search are usually evaluated according to the number of constraint checks needed to find a solution or to decide that the problem doesn't have one.

This paper is organized as follows: Section 2 defines some CSPs and Evolutionary Algorithms(EA) notions. We introduce in section 3 two genetic operators. In section 4 we propose a suitable model for comparing the genetic operators performance. Section 5 presents a set of tests for solving randomly generated CSPs which have at least one solution. Finally, section 6 gives some conclusions.

2 CSP y EA

In this section we talk about the CSP basic notions, such as: constraint matrix, instantiation, and partial instantiation. Then we present the evolutionary algorithm structure and its components that we will use afterwards. For simplicity we restrict our attention here to binary CSPs, where the constraints involve two variables. N-ary CSP has an equivalent binary one [19].

2.1 CSP

A *Constraint Satisfaction Problem* (CSP) is a set of *variables* $V = \{X_1, \ldots, X_n\}$, their related *domains* $D_1, \ldots, D_n$ and θ a set of η *constraints* between these variables. A variable must be instanciated from values within its domain. The domain sizes are respectively $m_1, \ldots, m_n$, with $\mathbf{m}$ equal to the maximum of m_i. Each variable X_j is *relevant* (denoted by *"be relevant to"* $\triangleright$), for a subset $C_{j_1}, \ldots, C_{j_k}$ of constraints where $\{j_1, \ldots, j_k\}$ is some subsequence of $\{1, 2, \ldots, \eta\}$. A binary constraint has exactly two relevant variables. A binary CSP can be represented by a constraints graph where nodes are the variables and arcs are the constraints. We talk about inconsistency or constraint violation when the relevant variables for a given constraint don't have values that can satisfy the constraint. An instantiation of the variables that satisfies all the constraints is a solution of the CSP. Generate & Test is the simplest algorithm, it only tries every possible values for the variables.

Definition 2.1 *(Constraint Matrix)*
A Constraint Matrix $\mathbf{R}$ *is a* $\eta \times n$ *matrix, such as:*

$$\mathbf{R}_{\alpha j} = \mathbf{R}[\alpha, j] = \begin{cases} 1 & \textit{if variable } X_j \triangleright C_\alpha \\ 0 & \textit{otherwise} \end{cases}$$

Remark 2.1 *With binary constraints, each row of* $\mathbf{R}$ *has only two values equal to one.*

Definition 2.2 *(Instantiation)*
An Instantiation $\mathbf{I}$ *is an assignment from a n-tuple of the variables* $(X_1, \ldots, X_n) \rightarrow D_1 \times \ldots \times D_n$*, such that it gives a value from its domain to each variable in* V*.*

Definition 2.3 *(Partial Instantiation)*
Given $V_p \subseteq V$*, an Partial Instantiation* $\mathbf{I_p}$ *is an assignment from a j-tuple of variables* $(X_{p_1}, \ldots, X_{p_j}) \rightarrow D_{p_1} \times \ldots \times D_{p_j}$*, such that it gives a value from its domain to each variable in* V_p*.*

Remark: We speak about satisfaction (or insatisfaction) of C_α *in* $\mathbf{I_p}$ *iff all the relevant variables of* C_α *are instantiated*

2.2 Evolutionary Algorithms

The structure of an evolutionary algorithm is shown in figure 1.
The evolutionary algorithm used in this paper generates in a random way the initial population. To construct a chromosome the variables values are selected from their domains with a uniform probability. We have used a non-binary genetic representation because it is the most natural representation for this kind of problems. Finally the selection algorithm is biased to the best chromosomes.

```
Begin /* Evolutionary Algorithm */
 t = 0
 inicialise population P(t)               (1)
 evaluate the individuals in P(t)         (4)
 while (not end-condition) do
       begin
       t=t+1
      Parents = select-parents-from P(t-1)  (5)
       Childs = alter Parents               (3)
       P(t) = Childs
      evaluate P(t)                         (4)
       end
 endwhile
End /* Evolutionary Algorithm */
```

Figure 1: Structure of the EA and its components

Remark 2.2 *An instantiation* **I** *corresponds to a chromosome (individual) in our evolutionary algorithm.*

2.2.1 Evaluation Function

Evolutionary Methods have usually been applied to solve optimization problems. They search a solution guided by the evaluation function. A CSP doesn't have a function to be optimized, thus in order to solve it with an evolutionary algorithm we require to define an ad-hoc fitness function. For instance, the number of satisfied constraints [12].

In [20] we proposed a fitness function specially defined for CSPs. We roughly describe it in the following paragraph:

Definition 2.4 *(Involved Variables)*
For a CSP and an Instantiation **I** *of its variables, we define a set of involved variables* $\mathbf{E}_{\alpha(\mathbf{I})} \subseteq V$ *for each constraint* C_α $(\alpha = 1, \ldots, \eta)$*, as follows:*

- $\mathbf{E}_{\alpha(\mathbf{I})} = \phi$ *iff* C_α *is satisfied*
- $X_i \in \mathbf{E}_{\alpha(\mathbf{I})}$ *if* $X_i \triangleright C_\alpha$ *and* C_α *id not satisfied under* **I**
- $X_j \in \mathbf{E}_{\alpha(\mathbf{I})}$ *if* $X_i \triangleright C_\alpha$ *and* $\exists C_\beta \neq C_\alpha$ *such that both* X_i *and* $X_j \triangleright C_\beta$*, and* C_α *is not satisfied under* **I**

This definition shows that there exists variables that are relevant for many constraints. When C_α is violated the involved variables are: the variables directly connected by C_α, and the other variables connected to them. More precisely, for an instantiation **I**, the consequence of changing a value of a variable can affect many constraints. We have incorporated this idea in the fitness function. Before to define the fitness function we need the next definition:

Definition 2.5 *(Error Evaluation)*
For a CSP with a constraint matrix **R** *and an Instantiation* **I**, *we define the Error Evaluation* $\mathbf{e}(C_\alpha, I)$ *for a constraint* C_α *as:*

$$\mathbf{e}(C_\alpha, \mathbf{I}) = \textit{Number of variables in } \mathbf{E}_{\alpha(\mathbf{I})}$$

$$\mathbf{e}(C_\alpha, \mathbf{I}) = \begin{cases} \sum_{w=1}^{n} \mathbf{R}[\alpha, w] \left(\sum_{\beta=1}^{\eta} \mathbf{R}[\beta, w] \right) & \textit{if } C_\alpha \textit{ is violated} \\ 0 & \textit{otherwise} \end{cases} \quad (1)$$

Remark 2.3 *If a binary constraint* C_α *has* X_k *and* X_l *as relevants variables and the constraint is violated, then:*

$\mathbf{e}(C_\alpha, \mathbf{I})$ *= (Number of relevant variables of* C_α*) + (Propagation effect* X_k *and* X_l*)*
where the propagation effect X_k and X_l *in a binary network constraint is defined by the number of constraints* C_β, $\beta = 1, \ldots, \eta, \beta \neq \alpha$ *that have either* X_k *or* X_l *as relevant variables.*

Definition 2.6 *(Contribution to the evaluation)*
For a CSP, we define the contribution of C_α *to the* evaluation function $\mathbf{c}(C_\alpha)$ *as:*

$$\mathbf{c}(C_\alpha) = \mathbf{e}(C_\alpha, \mathbf{I_j}), \textit{ when } C_\alpha \textit{ is violated on } \mathbf{I_j}.$$

Remark 2.4 *The value of* $\mathbf{c}(C_\alpha)$ *does not depend on the values of the instantiation but of the fact that the instantiation violates the constraint.*

Finally, the evaluation function is the sum of the *evaluation of the error* (equation 1) of all the constraints in the CSP, that means:

Definition 2.7 *(Fitness Function)*
For a CSP with a constraint matrix **R** *and an instantiation* **I**, *and the error evaluation* $\mathbf{e}(C_\alpha, \mathbf{I})$ *for each constraint* $C_\alpha, (\alpha = 1, \ldots, \eta)$, *we define a Fitness function* $\mathbf{Z}(\mathbf{I})$ *as:*

$$\mathbf{Z}(\mathbf{I}) = \sum_{\alpha=1}^{\eta} \mathbf{e}(C_\alpha, \mathbf{I}) \quad (2)$$

The goal is to minimize the fitness function $\mathbf{Z}(\mathbf{I})$, which will be equal to zero when all the constraints will be satisfied. The evaluation function reflects that it is more important to satisfy those constraints that involve more variables, i.e., those that are more strongly connected with other constraints.

3 Genetic Operators

In this section we present a comparison between two evolutionary algorithms that only differ in their genetic operators. Therefore the genetic operators used in this paper are roughly presented in the following sections:

3.1 Asexual Operator: (n, p, g)

It is an specialized operator defined by Eiben in [8]: The operator selects a number of positions from its parent and then it selects new values for these positions. The parameters to be defined are the number of values to be modified, the criteria to identify the positions of the variables to be modified and the criteria to define the new values for the child. The operator is denoted by (n,p,g) where n is the number of modified values and p,g values that will be chosen from the set {r,b}, where r is for a random uniform selection and b is for a biased selection using some heuristics. The best parameters found for this operator are (n,p,g) $= (\#, r, b)$ where # means that the number of values to be modified is randomly chosen but less or equal than $\frac{1}{4}$ of the number of total positions.

3.2 Arc-operators

In [22] we proposed two operators denoted in a generic way as *arc-operators*. They are based on the constraint network.

3.2.1 Arc-crossover

This operator makes a crossover between two randomly selected individuals from the population, and creates a new individual (the son). The son inherits its values using a greedy procedure which analyzes each constraint according to a pre-established order. The first arc which it analyzes is the more difficult to satisfy, that is the one which is stronger connected. According to the functions defined in the previous section, the first constraint to take into account is the one which has the higher value for the error evaluation function, when the constraint is violated. We need the following definitions:

Definition 3.1 *(S_α)*

Given a CSP and a partial instanciation $\mathbf{I_p}$. *For a fully instanciated constraint* C_α *(i.e. all its relevant variables are instanciated) we define a set* $\mathbf{S}_\alpha \subseteq \theta$ *for* $C_\beta \in \mathbf{S}_\alpha$ *iff*

- $\exists X_i : X_i \rhd C_\alpha$ *and* $X_i \rhd C_\beta$ *(C_α, C_β have a commun variable)*
- $\forall X_j$ *relevant for* C_β, X_j *is instantiated*

This definition shows that changing a variable value could affect other constraints. This is used when the algorithm is creating the son, that is when it is instanciating the son's variables.

Remark: $C_\alpha \in \mathbf{S}_\alpha$

Definition 3.2 *(Partial Crossover Evaluation Function) Given a CSP with a partial instanciation* $\mathbf{I_p}$, *the sets* $\mathbf{S}_\alpha$, *and the functions* $\mathbf{e}(C_\alpha, \mathbf{I_p})$.*For all fully instanciated constraint* C_α, *we define the crossover partial evaluation function* $\mathbf{cff}(C_\alpha)$ *for* C_α *as:*

$$\mathbf{cff}(C_\alpha, \mathbf{I_p}) = \sum_{C_\gamma \in \mathbf{S}_\alpha} \mathbf{e}(C_\gamma, \mathbf{I_p}) \tag{3}$$

Remark 3.1 *We extend the domain of the previously defined functions for* $\mathbf{I}$ *to* $\mathbf{I}$ *and* $\mathbf{I_p}$ *computing these functions only considering involved constraints (related with* C_α*) which variables are instanciated in* $\mathbf{I_p}$

Definition 3.3 *(*$\mathbf{M_j}$*)*
Given a CSP with a partial instanciation $\mathbf{I_p}$. *For an instantiated variable* X_j *we define a set of constraints* $\mathbf{M_j} \subseteq \theta$ *by* $C_\alpha \in \mathbf{M_j}$ *iff:*

- $X_j \rhd C_\alpha$
- C_α *is fully instantiated on* $\mathbf{I_p}$ *(i.e.* $\forall X_k \rhd C_\alpha : X_k$ *is instantiated)*

Definition 3.4 *(Partial mutation evaluation function)*
Given a CSp with a partial instanciation $\mathbf{I_p}$,*the sets* $\mathbf{M_j}$*for all variables instanciated in* $\mathbf{I_p}$, *and the functions* $\mathbf{e}(C_\alpha, \mathbf{I_p})$.*For all fully instanciated constraint* C_α, *we define the* mutation partial evaluation function $\mathbf{mff}$ *for* X_j *as:*

$$\mathbf{mff}(X_j, \mathbf{I_p}) = \sum_{C_\gamma \in \mathbf{M_j}} \mathbf{e}(C_\gamma, \mathbf{I_p}) \tag{4}$$

Remark 3.2 *his function is computed considering only the involved constraints (related con* X_j*) which variables are instanciated in* $\mathbf{I_p}$

Definition 3.5 *(Number of violations)*
Given a CSP and the error evaluation functions $\mathbf{e}(C_\alpha, \mathbf{I_1})$ *associated to each constraint* C_α, *under an instanciation* $\mathbf{I_1}$, *and* $\mathbf{e}(C_\alpha, \mathbf{I_2})$ *for each constraint* C_α, *under an instantiation* $\mathbf{I_2}$. *We define for each constraint* C_α *the* number of *violations* $\mathbf{nv}(C_\alpha, \mathbf{I_1}, \mathbf{I_2})$ *as:*

$$\mathbf{nv}(C_\alpha, \mathbf{I_1}, \mathbf{I_2}) = \begin{cases} 0 & \mathbf{e}(C_\alpha, \mathbf{I_1}) = \mathbf{e}(C_\alpha, \mathbf{I_2}) = 0 \\ 1 & \textit{either } \mathbf{e}(C_\alpha, \mathbf{I_1}) \neq 0 \textit{ or } \mathbf{e}(C_\alpha, \mathbf{I_2}) \neq 0 \\ 2 & \mathbf{e}(C_\alpha, \mathbf{I_1}) \neq 0, \textit{ and } \mathbf{e}(C_\alpha, \mathbf{I_2}) \neq 0 \end{cases}$$

Figure 2 shows the *arc-crossover* algorithm.
The idea is the following: we select two individuals from the population and we will use their values to create a son, which is expected to satisfy more constraints than his parents. The algorithm starts with the more connected constraint in the graph and checks if it is satisfied or violated by the parents. If it is satisfied for both parents, the son will take the two relevant variables of this constraint from the parent which has the best evaluation . If only one parent satisfies the constraint, then the son inherits its values from the two variables of this parent. If both parents violate the constraint, then the algorithm crosses the parents' values for each variable which belongs to the constraint (with two possibilities). Within these two possibilities for the son's values, we select the one that will give it a better evaluation (taking into account the variables that have already been instanciated), that is a partial revision of the son is made before instanciating the variables. Once almost all the constraints have been analyzed, they are going to be left some constraints to consider. It is probable that some constraints, without being yet analyzed, already have the values of their variables instanciated in the son (due to the connectivity with previously analyzed constraints). If a constraints only lacks a variable to be instanciated during it analysis, the selection of its value is made taking into account the evaluation that would have the son with each value coming from both parents. The best values for the son is selected according to already instanciated variables.

4 Comparing Genetic Operators

4.1 Number of constraints checks

In this section we estimate the number of constraints checks made by an algorithm which uses arc-mutation and arc-crossover, and compare it with an algorithm based on $(\#, r, b)$ and mutation. Given an individual, arc-mutation randomly selects the variable to modify. This variable's value is chosen computing the partial evaluation mutation function for each value from its domain (given by definition 3.4. The selected value is the one with the best value for **mff**.

4.1.1 The comparison model

Parameter of the model:

- p_c crossover probability
- p_m mutation probability
- t_p population size
- n number of variables

```
Algorithm Arc-Crossover ($Parent_1, Parent_2$)
Begin
For each $C_\alpha$ using the evaluation-error order
Analise $C_\alpha$ $(x_i, x_j)$ in both $Parent_1$ and $Parent_2$
if $((X_i \dashv \mathbf{I_p})^a$ y $(X_j \dashv \mathbf{I_p}))$ then
   if $(\mathbf{nv}(C_\alpha, \mathbf{Parent_1}, \mathbf{Parent_2}) = 0)$ then
      if $(\mathbf{Z}(Parent_2) > \mathbf{Z}(Parent_1))$ then
         $\mathbf{I_p}(X_i, X_j) = (x_{i_1}, x_{j_1})$
      else
        if $(\mathbf{Z}(Parent_1) > \mathbf{Z}(Parent_2))$ then
            $\mathbf{I_p}(X_i, X_j) = (x_{i_2}, x_{j_2})$
        else
           $\mathbf{I_p}(X_i, X_j) = random((x_{i_1}, x_{j_1}), (x_{i_2}, x_{j_2}))$
   else
      if $(\mathbf{nv}(C_\alpha, \mathbf{Parent_1}, \mathbf{Parent_2}) = 1)$ then
         if $(C_\alpha \models Parent_1)^b$ then
          $\mathbf{I_p}(X_i, X_j) = (x_{i_1}, x_{j_1})$
         else $\mathbf{I_p}(X_i, X_j) = (x_{i_2}, x_{j_2})$
      else
        $\mathbf{I_p}(X_i, X_j) = argmin_{s_1 \in S_1}(\mathbf{cff}(C_\alpha, (\mathbf{I_p} \cup (x_{i_1}, x_{j_2}))),$
                                 $\mathbf{cff}(C_\alpha, (\mathbf{I_p} \cup (x_{i_2}, x_{j_1}))))^c$
else
   if $((X_i \dashv \mathbf{I_p})$ or $(X_j \dashv \mathbf{I_p}))$ then
      if $(X_i \dashv \mathbf{I_p})$ then k=i else k=j
     $\mathbf{I_p}(X_k) = argmin_{s_2 \in S_2}(\mathbf{mff}(X_k, (\mathbf{I_p} \cup x_{k_1})), \mathbf{mff}(X_k, (\mathbf{I_p} \cup x_{k_2})))^d$
End
```

[a] X_i is not instantiated in $\mathbf{I_p}$

[b] $\models$: satisfied by

[c] $argmin_{s \in S}\{a_s\}$ gives s^* such that $a_{s^*} \leq a_s, \forall s \in S$.
$S_1 = \{(x_{i_1}, x_{j_2}), (x_{i_2}, x_{j_1})\}$

[d] $S_2 = \{x_{k_1}, x_{k_2}\}$

Figure 2: Structure of Arc-crossover

- m domain size
- p_1 connectivity probability, i.e., the probability to have a constraint between two variables.

Model Consequences:

- Average number of constraints $= \frac{n(n-1)p_1}{2}$
- Average connectivity of each variables $= p_1(n-1)$

arc-crossover: we are going to analyze the worst case for arc-crossover, when the constraint C_α is violated by both parents, and the two involved variables in C_α have not been yet instanciated in the son. In this case, arc-crossover has to select between the two combinations of values from the parents, with two evaluation of cff. Each constraint is only verified once, with combinations of values. Therefore we have the following:

- $2\eta \leq$ number of constraint checks of arc-crossover$\leq 4\eta$

Thus, the minimum number of constraints checks for arc-crossover is 2η, when at least one of the parents satisfies all of the constraints in the graph.

arc-mutation: Evaluates with **mff** all other values of the variable domain (saving the actual). Therefore:

- Number of constraint checks of arc-mutation $= 2\eta(m-1)p_m$

Thus, the number of constraints checks for the whole algorithm which uses the arc-operators (N_{valgo}) is:

- $N_{valgo} \leq \left(4\eta\frac{p_c}{2-p_c} + 2\eta(m-1)p_m\right) t_p$

$(\#, \mathbf{r}, \mathbf{b})$ is the asexual operator. Therefore the number of constraint checks for $(\#, r, b)$ is equal to $\frac{\eta m}{2}$. For the whole algorithm which uses $(\#, r, b)$ and mutation, the number of constraint checks is:

- $N'valgo = \frac{t_p p_c \eta m}{2}$

because the standard mutation operator does not verify any constraint.

Obviously, the number of constraint checks for arc-operators is much more than other algorithms. However, the performance of an algorithm is related to:

- The ratio of cases where the algorithm finds a solution.
- The number of generations to find a solution.

5 Tests

We have executed the two algorithms to solve the classic constraints satisfaction problem of coloring a graph with three colours (3-colours). The 3-colours problem consists in assigning a color to each node such that every adjacent nodes will have a different colour. The tests was executed on a SUN Sparc Station IPX running Solaris 2.5. We have randomly generated 100 problems (with solution) according to their connectivity, with a connectivity ratio between [10%..90%].We fixed a maximum number of iterations of 500 for the arc-operators algorithm, and a maximum of 1500 for the other one. The following table shows the average CPU time to solve 100 graphs, and the average number of constraint checks.

Constraints	N'_{valgo}	CPU'time	N_{valgo}	CPU time
30	11141	67	7609	18
45	185671	1144	44967	110
60	843709	4891	422426	1053
90	1401437	6372	465359	917
120	1311153	5908	250505	632
150	1060255	5938	219715	452
180	1070010	5961	219847	580
210	1022748	5693	201619	419
240	936505	5296	220700	419
270	1153107	5087	252820	673

The performance of the arc-operators algorithm is much better than the one which uses $(\#, r, b)$ and mutation. This behavior comes from the fact that the first algorithm converges faster to a solution, and also because it has been able to solve more problems.

6 Conclusion

The main objective of systematic algorithms which solve CSPs is to minimize the number of constraint checks. However, evaluating methods based on stochastic searches is not an easy task. This comes from the fact that we deal with evolutionary algorithms where we don't a priori know the number of iterations that will be completed, neither the number of times that genetic operator(s) will be applied.

In this paper we proposed a model to evaluate the number of constraint checks completed by operators, to compare the efficiency of genetic operators for CSP solving. Moreover, we propose to evaluate the performance of the algorithm according to the CPU time and the number of constraint checks. This allows to have a vision of the algorithm for solving CSPs.

Acknowledgements

I wish to gratefully acknowledge the discussions with Bertrand Neveu. I thank Xavier Bonnaire for his technical support on writing this paper.

References

[1] Adorf H.M. and Johnston M.D., A discrete stochastic neural network algorithm for constraint satisfaction problems, in: Proceedings International Joint Conference on Neural Networks, San Diego, CA, 1990.

[2] Affane M.S. and Bennaceur H., A labelling Arc Consistency Method for Functional Constraints. Constraint Processing (CP96), Ed. Eugene Freuder, pp. 16-30, 1996.

[3] Bowen J., Dozier G., Solving Constraint Satisfaction Problems Using A Genetic/Systematic Search Hybrid That Realizes When to Quit. Proceedings of the Sixth International Conference on Genetic Algorithms pp. 122-129, 1995.

[4] Cheeseman P.,Kanefsky B., Taylor W., Where the Really Hard Problems Are. Proceedings of IJCAI-91, pp. 163-169, 1991

[5] Dechter R., Enhancement schemes for constraint processing: backjumping, learning, and cutset decomposition. Artificial Intelligence 41, pp. 273-312, 1990.

[6] Dozier G., Bowen J., Bahler D., Solving Small and Large Scale Constraint Satisfaction Problems Using a Heuristic-Based Microgenetic Algorithm. Proc. of the First IEEE Conf on Evolutionary Computation, Orlando, pp 306-311, 1994.

[7] Eiben A.E., Raué P-E, Ruttkay Zs, Solving Constraint Satisfaction Problems Using Genetic Algorithms. Proc. of the First IEEE Conf on Evolutionary Computation, Orlando, pp 542-547, 1994.

[8] Eiben A.E., Raué P-E, Ruttkay Zs, GA-easy and GA-hard Constraint Satisfaction Problems. Constraint Processing, Ed. Manfred Meyer, CAPringer-Verlag LNCS 923, pp. 267-283, 1995.

[9] Eiben A.E., Raué P-E, Ruttkay Zs, Self-adaptivity for Constraint Satisfaction: Learning Penalty Functions. Proc. of the Third IEEE Conf on Evolutionary Computation, Nagoya, pp 258-261, 1996.

[10] Fleurent C., Ferland J., Genetic Algorithms and Hybrids for Graph Coloring, Annals of Operations Research, 1995.

[11] Freuder E., A sufficient condition of backtrack-free search. J. ACM. 29, pp. 24-32, 1982.

[12] Freuder E., The Many Paths to Satisfaction. Constraint Processing, Ed. Manfred Meyer, CAPringer-Verlag LNCS 923, pp. 103-119, 1995.

[13] Haralick and Elliott, Increasing tree search efficiency for Constraint Satisfaction Problems. Artificial Intelligence 14, pp. 263-313, 1980.

[14] Kumar. Algorithms for constraint satisfaction problems:a survey. AI Magazine, 13(1):32-44, 1992.

[15] Michalewicz Z., Genetic Algorithms + Data Structures = Evolution Programs. Ed. CAPringer-Verlag, Artifial Intelligence Series, 1994.

[16] Minton, Johnston, Philips, Laird, Minimizing conflicts: a heuristic repair method for constraint satisfaction and scheduling problems. Artificial Intelligence 58, pp. 161-205, 1992.

[17] Minton S., Integrating heuristics for constraint satisfaction problems: A case study. Proceedings of the Eleventh National Conference on Artificial Intelligence, 1993.

[18] Minton S., Automatically Configuring Constraint Satisfaction Programs: A case study. Constraints, Vol1, Number 1, 1996.

[19] Rossi, F; Petrie, C.; and Dhar, V., On the equivalence of Constraint-Satisfaction Problems, Technical Report ACT-AI-222-89, MCC Corp., Austin, Texas, 1989.

[20] Riff M.-C., From Quasi-solutions to Solution: An Evolutionary Algorithm to Solve CSP. Constraint Processing (CP96), Ed. Eugene Freuder, pp. 367-381, 1996.

[21] Riff M.-C., Using the knowledge of the Constraints Network to design an evolutionary algorithm that solves CSP. Proc. of the Third IEEE Conf on Evolutionary Computation, Nagoya, pp 279-284, 1996.

[22] Riff M.-C., Evolutionary Search guided by the Constraint Network to solve CSP. Proc. of the Fourth IEEE Conf on Evolutionary Computation, Indianopolis, pp. 337-342, 1997.

[23] Tsang E., Applying Genetic Algorithms to Constraint Satisfaction Optimization Problems. Proc. of ECAI-90, Pitman Publishing, pp 649-654 , 1990

[24] Warwick T., Tsang E., Using a Genetic Algorithm to Tackle the Processors Configuration Problem. Proc. of ACM Symposium on Applied Computing (SAC), pp. 217-221, 1994.