Solving Constraint Satisfaction Problems with Heuristic-based Evolutionary Algorithms

B. Craenen
Leiden University
bcraenen@cs.leidenuniv.nl

A.E. Eiben
Leiden University
gusz@cs.leidenuniv.nl

E. Marchiori
Leiden University
elena@cs.leidenuniv.nl

Abstract

During the last decade several approaches to solving constraint satisfaction problems (CSPs) by evolutionary algorithms (EAs) have been proposed. A number of EAs for solving CSPs employ heuristics based on information over the structure of the constraints. In this paper we perform a comparative study of heuristic based EAs. To this aim, we conduct extensive experiments on a test suite consisting of randomly generated binary CSPs. By the systematic setup of the test suite we can draw conclusions on the (dis)advatages of the type and the usage of heuristics in genetic algorithms for solving CSPs.

1 Introduction

Constraint satisfaction is a fundamental topic in artificial intelligence with relevant applications in different areas, like planning, default reasoning, scheduling, etc. Informally, a constraint satisfaction problem (CSP) consists of finding an assignment of values to variables in such a way that the restrictions imposed by the constraints are satisfied.

In general, constraint satisfaction problems are computationally intractable (NP-hard). Therefore, besides trying to improve the efficiency of complete algorithms by means of pruning techniques for reducing the size of the search space, much effort has been spent in the design of heuristic algorithms which have not guaranteed performance yet are able to give an answer in very short time. In particular, in the last decade various approaches based on evolutionary algorithms have been introduced. Evolutionary algorithms for solving constraint satisfaction problems can be roughly divided into two classes: EAs using adaptive fitness functions ([1, 4, 5, 12, 13, 6, 18, 19]) and EAs utilizing heuristics ([11, 16, 21, 22]). An experimental comparison of EAs of the first kind is given in [7], using a large test suite consisting of randomly

generated binary CSPs, where the hardness of the problem instances is influenced by two parameters: constraint density and constraint tightness. In this paper we describe heuristic based evolutionary algorithms and perform experiments on the above mentioned test suite. This allows us to assess the performance of the two classes of algorithms, and to draw conclusions on the effectiveness of different GA based methods.

2 Random Binary CSPs over Finite Domains

We consider binary constraint satisfaction problems over finite domains, where constraints act between pairs of variables. This is not restrictive since any CSP can be reduced to a binary CSP by means of a suit able transformation which involves the definition of more complex domains (cf. [24]). A *binary CSP* is a triple $(V, \mathcal{D}, C)$ where $V = \{x_1, \ldots, x_n\}$ is a set of variables, $\mathcal{D} = (D_1, \ldots, D_n)$ is a sequence of finite domains, such that x_i takes value from D_i, and C is a set of binary constraints. A *binary constraint* c_{ij} is a subset of the cartesian product $D_i \times D_j$ consisting of the compatible pairs of values for (x_i, x_j). For simplicity here and in the sequel we shall assume that all the domains D_i are equal ($D_i = D$ for $i \in [1, n]$). An *instantiation* α is a mapping $\alpha : V \to D$, where $\alpha(x_i)$ is the value associated to x_i. A *solution* σ of a CSP is an instantiation such that $(\sigma(x_i), \sigma(x_j))$ is in c_{ij}, for every x_i, x_j in V with $i \neq j$.

A class of random binary CSPs can be specified by means of four parameters $\langle n, m, d, t \rangle$, where n is the number of variables, m is the uniform domain size, d is the probability that a constraint exists between two variables, and t is the probability of a conflict between two values in a constraint. CSPs exhibit a *phase transition* when a parameter is varied. At the phase transition, problems change from being relatively easy to solve (i.e., almost all problems have many solutions) to being very easy to prove unsolvable (i.e., almost all problems have no solutions). Problems in the phase transition are identified as the most difficult to solve or to prove unsatisfiable ([2, 20, 23, 26]), and occur for higher density/tightness of the constraint networks.

The test suite we have used consists of problem instances produced by a generator (see http://www.wi.leidenuniv.nl/home/jvhemert/csp-ea/) loosely based on the generator of Gerry Dozier [1]. The generator first calculates the number of constraints that will be produced using the equation $\frac{n(n-1)}{2} \cdot d$. It then starts producing constraints by randomly choosing two variables and assigning a constraint between them. When a constraint is assigned between variable v_i and v_j, a table of conflicting values is generated. To produce a conflict two values are chosen randomly, one for the first and one for the second variable. When no conflict is present between the two values for the variables, a conflict is produced. The number of conflicts in

this table is determined in advance by the equation $m(v_i) \cdot m(v_j) \cdot t$ where $m(v_i)$ is the domain size of variable i.

3 Heuristic EAs for CSPs

Heuristic EAs for CSPs share the same rationale. The uniform random part of the EA and the heuristic-based components are used to counterbalance each others deficiencies. The application of heuristics can improve the performance of the blind random mechanism, while the random component can compensate the strong bias that is introduced by the heuristics. We consider three different EAs using these techniques: `ESP-GA` by E. Marchiori, which uses a constraint processing phase and a probabilistic repair rule, `H-GA` by Eiben et al., using heuristic genetic operators, and `Arc-GA` by M. C. Riff Rojas, which uses two novel genetic operators and a new fitness function that are guided by information from the constraint network. All algorithms use the staightforward integer representation of chromosomes, where each gene corresponds to one variable and the set of alleles for v_i equals the domain D_i, that is, a chromosome is a sequence of integers where integer p in the i-th entry indicates that the i-th variable is set to value p.

3.1 ESP-GAs

In [16], E. Marchiori suggests an approach for solving CSPs using GAs. The idea is based on the '*glass-box*' approach [25] because it adjusts the CSP in such a way that there is only one single (type of) primitive constraint. By decomposing more complex constraints into primitive ones, the resulting constraints have the same granularity and therefore the same intrinsic difficulty. This rewriting of constraints is done in two steps and is called *constraint processing*. In the first step (elimination step), functional constraints are eliminated in order to reduce the number of variables in the problem. This is done analogously to the method used, e.g., in `GENOCOP` [17]. In the second step (splitting step), the resulting constraints are decomposed into a set of constraints in canonical form called primitive constraints. The primitive constraints chosen in [16] are of the form $\alpha \cdot v_i - \beta \cdot v_j \neq \gamma$.

After the constraint processing phase all constraints have the same form, hence a single repair rule can be used in the GA to enforce *dependency propagation*. Because all constraints share a single repair rule, repairing an individual can be performed locally by applying the repair rule to every violated constraint. This is done in the second phase (dependency propagation), where the resulting CSP is solved using a GA that incorporates a form of probabilistic *repair rule* of the form **if** $\alpha \cdot p_i - \beta \cdot p_j = \gamma$ **then** modify p_i or p_j.

Finally, a GA with dependency propagation is used to find a solution.

The elimination of functional constraints reduces the number of variables hence the complexity of the search space. Moreover, the transformation of

constraints into primitive ones smoothens the relative difficulty of the constraints, thus helping the GA to escape from those local optima derived from the concentration of the search towards chromosomes that satisfy easier constraints. Finally, the dependency propagation phase seems to direct further the search beyond local optima by small random improvements of the chromosomes.

This method requires the setting of a number of parameters. In the splitting step, one has to choose the class of primitive constraints. In the propagation step, one has to decide which and how many chromosomes to select for dependency propagation. This choice depends on the problem. Moreover, one has to decide which rules to use for repairing chromosomes that violate a primitive constraint. This choice depends on the form of the primitive constraint. Moreover, there are various ways to change the value of the selected variable.

3.1.1 Implementation Details

As mentioned in Section 2 the representation of the constraints is fixed, where a constraint is characterized by a table of conflicting values. This is a problem for the applicability of the `ESP-GA` technique, whose constraint processing phase relies on the implicit assumption that CSPs are described implicitely as formulas in a logic language. As expected, it turns out that the use of a table of conflicting values for representing a constraint renders unnecessary the application of the constraint processing phase. Therefore, the `ESP-GA` reduces to the GA with dependency propagation. The results of the experiments will indicate that this specialization of the `ESP-GA` technique is not very effective for solving binary constraint satisfaction problems.

In order to remain as close as possible with the original GA based approach in [16], we convert conflict tables into constraints of the form $\alpha \cdot v_i - \beta \cdot v_j \neq \gamma$, by setting $\gamma = |D_j| \cdot p_i - p_j$, where p_i, p_j are the values of v_i and v_j respectively, and by setting $\alpha = D_j$ and $\beta = 1$. To check violation of a constraint of this form one enters the values for the specific variables. If the result is the calculated γ-value, the constraint is violated. It is not difficult to prove that the above mentioned technique transforms a CSP into an equivalent one.

The implementation of dependency propagation we consider is based on the following choices:
- variable selection: The most restricted variable is chosen, that is, the one with the smallest number of zeros on its row or column is selected. Counting the zeros in the conflict table for every value of the two variables is enough to determine which one of the variables is to be changed. With an equal number of zeros, a random choice is made.
- value selection: A random value among those that do not cause conflict.
- constraints order: A random permutation of the constraints is chosen,

every time an individual is repaired. In this way each individual is repaired with a different random constraint order and therefore population diversity should be ensured. The other main features of the implemented GA are illustrated in Table 1.

Crossover operator	One-point crossover
Mutation operator	Random mutation
Fitness function	Number of violated constraints
Extra	Repair rule

Table 1: GA features of `ESP-GA`

3.2 H-GAs

In [8, 10], Eiben et al. propose the possibility of using existing CSP heuristics within GAs. Heuristics are commonly used in CSP solvers and are already available for most classical CSPs, therefore, it is a natural idea to use these heuristics in a GA. In [10], two heuristic operators are specified: an asexual operator that transforms one individual into a new one and a multi-parent operator that introduces a new individual based on two or more parents. Both have been tested on the n-queens and the graph 4-coloring problem. In this paper we will call the GAs as suggested by Eiben et al. heuristic GAs: `H-GA`s.

Thorough this paper we maintain a subtle difference between two types of unary search operators, i.e. operators that utilize one parent to create (one) offspring. The name *mutation* stands for a unary operator that is completely random, without any bias. The term *asexual (heuristic) operator* is used for a unary operator that applies a bias when creating the offspring.

3.2.1 Asexual heuristic operator

The asexual heuristic operator selects a number of variables in a given individual, and then selects new values for these variables. There are three defining parameters of the possible asexual operators: the number of variables to be modified, the criteria for selecting these variables, and the criteria for the new values of these variables. Thus, different asexual operators can be denoted by the triple (n, x, y) where n indicates the number of variables to be modified — being either 1,2 or # (with # meaning that the number of variables to be altered is chosen randomly but is at most one-fourth of all variables in the individual) — and x and y indicate the selection criteria for variable and value selection respectively. Both x and y can have two values: r standing for random selection and b for a heuristic biased selection ((n, r, r) would thus be termed as 'mutation', while all other variants not). In this paper we study the operator $(\#, b, b)$, meaning that each time the

operator is used, up to one fourth of the variables is changed, its variables to be changed and the values they will be changed to are chosen using a heuristic.

In [8] several measures or heuristics for selecting a variable to reinstatiate as well as heuristics for selecting a (new) value for the selected variable are considered. In accordance with [10] we implement a heuristic bias system that changes the variable that is involved in the largest number of constraint violations. It is expected that by changing this variable the largest improvement to the individual can be made. The heuristic for selecting a new value is based on the number of satisfied constraints under the given instantiation. The heuristic counts the number of satisfied constraints for each different value in the domain of the given variable and chooses one maximizing this measure. It is expected that by using this measure on value selection, the possibility of introducing a (new) conflict in the individual is the smallest.

3.3 Multi-parent heuristic crossover

The crossover operator of our `H-GA` is a multi-parent operator that uses a heuristic to determine which values of the parents are selected for a child. The basic mechanism of this crossover operators is scanning [9]. This operator examines all positions of the parents consecutively and per position it chooses one of the values present in the parents to be included in the child at the given position. Clearly, heuristics for value selection in the asexual operator can be used in the scanning crossover too. Conform with the setup in [10] the heuristics based on the number of satisfied constraints is used in the scanning crossover during our experiments. The difference with the asexual heuristic operator is that the heuristic will not evaluate all possible values but only ose ones that are represented in the parents at the given position. The multi-parent crossover is applied with 5 parents.

	Version 1	Version 2	Version 3
Main operator	Asexual heuristic operator	Multi parent operator	Multi parent operator
Secondary operator	Random mutation	Random mutation	Asexual heuristic operator
Fitness function	Number of violated constraints		
Extra	None		

Table 2: Features of the three implemented versions of `H-GA`

As Table 2 discloses we use the asexual heuristic operator in a double role. In the `H-GA.1` version it serves as the main search operator assisted by

(random) mutation. In `H-GA.3` it accompanies the multi-parent crossover in a role which is mostly filled in by mutation.

3.4 Arc-GAs

In [21, 22] M.C. Riff-Rojas introduces a GA that solves CSPs by integrating information about the constraint network in the fitness function and in the genetic operators (crossover and mutation).

The basic idea of the *arc-fitness function* used in [21] is to use a penalty function which measures for each unsatisfied constraint the so called error evaluation, that is the number of variables occurring in that constraint plus the number of those variables that are directly connected to these variables in the constraint network. The error evaluation gives an indication of how hard a constraint is, relatively to the instantiation represented by the chromosome. The arc-fitness function of a chromosome is then the sum of the error evaluations of all constraints that the chromosome does violate. We denote by $E_c(chrom)$ the error evaluation of constraint c with respect to the instantiation described by the chromosome $chrom$.

In *arc-mutation*, a gene of a chromosome is randomly selected for mutation, and a new value for the corresponding variable, say v, is chosen, namely the value which minimizes the sum of the error-evaluations of the constraints involving v.

For using *arc-crossover* first the constraints have to be ordered, which is done in decreasing order according to their error-evaluation w.r.t. an instantiation of the variables that violates all constraints. Thus $c \succ c'$ if $E_c(unsat) \geq E_{c'}(unsat)$, where $unsat$ denotes an instantiation (chromosome) violating all constraints. Given two parents, this ordering is used for selecting constraints by enumerating them according to $\succ$. For the two variables of a selected constraint, say v, w, the following cases are distinguished:

1. If none of the two variables are instantiated yet in the offspring under construction, and
   (a) there is a parent which satisfies that constraint, then the parent having the higher fitness provides its values for v, w.
   (b) none of the parents satisfies the constraint, then a combination of values for v, w from the parents is chosen which minimizes the sum of the $E_c(unsat)$'s, for all c containing v or w whose other variables are already instantiated in the offspring.
2. If only one variable, say v, is not instantiated in the offspring under construction, then the value for v is chosen from the parent that minimizes the sum of the error-evaluations of the constraints involving v.

Although in [21] a new selection method is also proposed, it has not be implemented in our algorithm, since this method is not used in [22] for most of the additional abilities of the this selection method where incorporated in the arc-fitness function. The GA features of the implemented algorithm are summarized in Table 3.

Crossover operator	Arc-crossover operator
Mutation operator	Arc-mutation operator
Fitness function	Arc-fitness
Extra	None

Table 3: Features of `Arc-GA`

4 Experimental comparison

All three algorithms use a steady state GA, with a population of 10 individuals. Per generation one crossover operation and two mutation operations are performed, resulting in three fitness evaluations per generation. For parent selection linear ranking with bias $b = 1.5$ is used and the replacement strategy deletes the two worst members of the population.

The results in table 4 are obtained by testing the three methods (five algorithms) on 25 classes of CSPs obtained by considering the combinations of 5 different constraints tightness and 5 different density values. In each class 10 instances are generated and 10 independent runs are performed on each instance. The CSP instances have 15 variables and uniform domain size of 15. All the algorithms stop if they find a solution or after 100000 fitness evaluations. In order to compare the results, two common measures are used: the percentage of runs that found a solution the so-called success rate (SR), and the average number of fitness evaluations to solution (AES) in successful runs. Notice that in density-tightness combination (0.1, 0.1), for several methods, the entry labelled AES contains the symbol i, meaning that a solution was found in the initial population.

Entries in boldface are used to highlight the best result for the considered class of CSPs. It is worth to note that the three versions of `H-GA` seem to do the least amount of hidden work comparable to the other measures while `ESP-GA` seems to do the most amount of hidden work with `Arc-GA` somewhere between these two GAs. These because `Arc-GA` uses heuristics more often, in the crossover and mutation operator as well as in repair method.

Table 4 gives some indication of what is called the *landscape of solvability* of the different GAs. One can observe success rates SR = 1 in the upper left corner, while SR = 0 is typical in the lower right corner, separated by a 'diagonal' indicating the mushy region. Technically, for higher const raint density and tightness, all three EAs are unable to find any solution. This

den-sity	alg	tightness				
		0.1	0.3	0.5	0.7	0.9
0.1	ESP-GA	**1**(i)	**1(23)**	1(78)	0.91(600)	0.45(13559)
	H-GA.1	1(11)	1(54)	1(169)	1(643)	**0.72(10419)**
	H-GA.2	1(12)	1(88)	1(315)	1(1325)	0.61(15254)
	H-GA.3	**1**(i)	**1(23)**	**1(53)**	**1**(484)	0.64(14752)
	Arc-GA	**1**(i)	1(32)	1(79)	0.99**(211)**	0.27(14131)
0.3	ESP-GA	**1(23)**	1(132)	0.91(5699)	0.01**(8366)**	0()
	H-GA.1	1(50)	1(441)	**1**(4481)	0.02(69632)	0()
	H-GA.2	1(70)	1(704)	1(4921)	**0.05**(22954)	0()
	H-GA.3	1(26)	**1(119)**	0.97(3587)	0()	0()
	Arc-GA	1(33)	1(175)	0.91**(617)**	0.02(25802)	0()
0.5	ESP-GA	**1(36)**	1(891)	**0.19**(4371)	0()	0()
	H-GA.1	1(121)	1(1671)	0.08(43337)	0()	0()
	H-GA.2	1(188)	1(1861)	0.07(36780)	0()	0()
	H-GA.3	1(47)	1(498)	0.07(21083)	0()	0()
	Arc-GA	1(95)	**1(388)**	0.01**(554)**	0()	0()
0.7	ESP-GA	**1(52)**	0.91(8190)	0()	0()	0()
	H-GA.1	1(204)	**1**(5950)	0()	0()	0()
	H-GA.2	1(428)	1(8454)	0()	0()	0()
	H-GA.3	1(61)	0.95(8960)	0()	0()	0()
	Arc-GA	1(138)	0.71**(1230)**	0()	0()	0()
0.9	ESP-GA	**1(69)**	**0.42**(12180)	0()	0()	0()
	H-GA.1	1(338)	0.37(35593)	0()	0()	0()
	H-GA.2	1(487)	0.4(32954)	0()	0()	0()
	H-GA.3	1(92)	0.13(21457)	0()	0()	0()
	Arc-GA	1(164)	0.04**(1193)**	0()	0()	0()

Table 4: SR and AES (within parenthesis) for ESP-GA, Arc-GA and the three versions of H-GA

is not surprising, because higher density and tightness yield prob lems that are almost always unsatisfiable. Another interesting aspect of the behavior of these algorithms is for which problem instances their performance rapidly degrade. The most difficult CSPs seem to start in the classes where $d \geq 0.3$ and $t = 0.7$, and where $d \geq 0.5$ and $t = 0.5$. The se results are in accordance with theoretical predictions of the phase tra nsition for binary CSP problems ([23, 26]).

In the mushy region Arc-GA has worse performance. Only in density-tightness combinations (0.1,0.7) and (0.3,0.7) does Arc-GA have a larger SR than ESP-GA although this difference is small. Compared to the third version of H-GA in density-tightness combination (0.3,0.7), Arc-GA still finds a few solutions while the third version does not.

ESP-GA has a mixed performance. In density-tightness combinations (0.1,0.7), (0.1,0.9), (0.3,0.5), (0.3,0.7) and (0.7,0.3), ESP-GA does not perform as well as the other GAs, always being out-performed by at least one of the

other GAs. In density-tightness combinations (0.5,0.5) and (0.9,0.3), the opposite is the case. In these two combinations `ESP-GA` performs best while these two combinations are hard to solve for all GAs. However, in general `ESP-GA` performs slightly better than `Arc-GA`. The third version of `H-GA` fails to find any solution in density-tightness combinations (0.3,0.7), (0.3,0.5) and (0.7,0.3). This indicates that the third version of `H-GA` is not effective for solving hard binary CSP instances. The difference between the first and the second version of `H-GA` is small. In density-tightness combinations (0.1,0.9) and (0.5,0.5) the first version has a slightly better SR than the second version while in density-tightness combinations (0.3,0.9) and (0.9,0.3) the second version of `H-GA` is slightly better than the first version. When reviewing AES, the first version performs slightly better than the second version. Therefore we can conclude that the first version `H-GA` performs slightly better that the second version.

In summary, the results of our experiments seem to indicate that the algorithms have comparable performance, where first two versions of `H-GA` perform slightly better than the other GAs.

5 Conclusions

This paper contains an experimental study on binary constraint satisfaction problems of three different heuristic based GAs. The results of the experiments indicate that the three methods have comparable performance.

It is interesting to compare the results with those reported in [7], where three GA based algorithms using adaptive fitness functions have been tested on the same benchmark instances used in our experiments. The best performance is there obtained by a GA-based algorithm by Dozier et al [4], there called `MIDA` (microgenetic iterative descendent genetic algorithm) which employs heuristic information in the reproduction operator as well as an adaptive mechanism in the fitness function that increases the penalty of constraints that are more often unsatisfied.

In Table 5 we report the success rates obtained by `MIDA` (the winner in [7]) and those for `H-GA.1` (the winner in the present study). The figures indicate that `MIDA` can solve CSPs much better than the algorithms studied in this paper. The `MIDA` is also faster w.r.t. AES in all cases (figures not reported here). The success of `MIDA` most probably comes from the fact that it actually belongs to both classes of EAs mentioned in the introduction: it uses a heuristic method incorporated into the mutation operator and an adaptive mechanism redefining the fitness function during the run. It is reasonable to assume that the search for a solution does profit from the combination of the heuristic mutation and the adaptive fitness function. Future work is directed to assess the performance of combination of the heuristics applied in `H-GA.1` and the adaptive fitness adjusting mechanism

den-sity	alg	tightness				
		0.1	0.3	0.5	0.7	0.9
0.1	MIDA	1	1	1	1	0.96
	H-GA.1	1	1	1	1	0.72
0.3	MIDA	1	1	1	0.52	0
	H-GA.1	1	1	1	0.02	0
0.5	MIDA	1	1	0.9	0	0
	H-GA.1	1	1	0.08	0	0
0.7	MIDA	1	1	0	0	0
	H-GA.1	1	1	0	0	0
0.9	MIDA	1	1	0	0	0
	H-GA.1	1	0.37	0	0	0

Table 5: Success rates for MIDA and H-GA.1

called SAW-ing from [7].

References

[1] J. Bowen and G. Dozier. Solving constraint satisfaction problems using a genetic/systematic search hybride that realizes when to quit. In L.J. Eshelman, editor, *Proceedings of the 6th International Conference on Genetic Algorithms*, pages 122–129. Morgan Kaufmann, 1995.

[2] P. Cheeseman, B. Kanefsky, and W.M. Taylor. Where the really hard problems are. In J. Mylopoulos and R. Reiter, editors, *Proceedings of the 12th IJCAI-91*, volume 1, pages 331–337, Morgan Kaufmann, 1991. Morgan Kaufmann.

[3] Y. Davidor, H.-P. Schwefel, and R. Männer, editors. *Proceedings of the 3rd Conference on Parallel Problem Solving from Nature*, number 866 in Lecture Notes in Computer Science. Springer-Verlag, 1994.

[4] G. Dozier, J. Bowen, and D. Bahler. Solving small and large constraint satisfaction problems using a heuristic-based microgenetic algorithm. In IEEE [14], pages 306–311.

[5] G. Dozier, J. Bowen, and D. Bahler. Solving randomly generated constraint satisfaction problems using a micro-evolutionary hybrid that evolves a population of hill-climbers. In *Proceedings of the 2nd IEEE Conference on Evolutionary Computation*, pages 614–619. IEEE Press, 1995.

[6] A.E. Eiben, J.K. van der Hauw, and J.I. van Hemert. Graph coloring with adaptive evolutionary algorithms. *Journal of Heuristics*, 4(1):25–46, 1998.

[7] A.E. Eiben, J.I. van Hemert, E. Marchiori, and A.G. Steenbeek. Solving binary constraint satisfaction problems using evolutionary algorithms with an adaptive fitness function. In A.E. Eiben, Th. Bäck, M. Schoenauer, and H.-P. Schwefel, editors, *Proceedings of the 5th Conference on Parallel Problem Solving from Nature*, number 1498 in LNCS, pages 196–205, Berlin, 1998. Springer.

[8] A.E. Eiben, P-E. Raué, and Zs. Ruttkay. Heuristic genetic algorithms for constrained problems, part i: Principles. Technical Report IR-337, Free University Amsterdam, 1993.

[9] A.E. Eiben, P-E. Raué, and Zs. Ruttkay. Genetic algorithms with multi-parent recombination. In Davidor et al. [3], pages 78–87.

[10] A.E. Eiben, P-E. Raué, and Zs. Ruttkay. Solving constraint satisfaction problems using genetic algorithms. In IEEE [14], pages 542–547.

[11] A.E. Eiben, P.-E. Raué, and Zs. Ruttkay. Constrained problems. In L. Chambers, editor, *Practical Handbook of Genetic Algorithms*, pages 307–365. CRC Press, 1995.

[12] A.E. Eiben and Zs. Ruttkay. Self-adaptivity for constraint satisfaction: Learning penalty functions. In *Proceedings of the 3rd IEEE Conference on Evolutionary Computation*, pages 258–261. IEEE Press, 1996.

[13] A.E. Eiben and J.K. van der Hauw. Adaptive penalties for evolutionary graph-coloring. In J.-K. Hao, E. Lutton, E. Ronald, M. Schoenauer, and D. Snyers, editors, *Artificial Evolution'97*, number 1363 in LNCS, pages 95–106. Springer, Berlin, 1997.

[14] *Proceedings of the 1st IEEE Conference on Evolutionary Computation.* IEEE Press, 1994.

[15] *Proceedings of the 4th IEEE Conference on Evolutionary Computation.* IEEE Press, 1997.

[16] E. Marchiori. Combining constraint processing and genetic algorithms for constraint satisfaction problems. In Th. Bäck, editor, *Proceedings of the 7th International Conference on Genetic Algorithms*, pages 330–337. Morgan Kaufmann, 1997.

[17] Z. Michalewicz. *Genetic Algorithms + Data structures = Evolution programs.* Springer, Berlin, 3rd edition, 1996.

[18] J. Paredis. Co-evolutionary constraint satisfaction. In Davidor et al. [3], pages 46–56.

[19] J. Paredis. Co-evolutionary computation. *Artificial Life*, 2(4):355–375, 1995.

[20] P. Prosser. An empirical study of phase transitions in binary constraint satisfaction problems. *Artificial Intelligence*, 81:81–109, 1996.

[21] M.C. Riff-Rojas. Using the knowledge of the constraint network to design an evolutionary algorithm that solves CSP. In IEEE [15], pages 279–284.

[22] M.C. Riff-Rojas. Evolutionary search guided by the constraint network to solve CSP. In IEEE [15], pages 337–348.

[23] B.M. Smith. Phase transition and the mushy region in constraint satisfaction problems. In A. G. Cohn, editor, *Proceedings of the 11th European Conference on Artificial Intelligence*, pages 100–104. Wiley, 1994.

[24] E.P.K. Tsang. *Foundations of Constraint Satisfaction*. Academic Press Limited, 1993.

[25] P. van Hentenryck, V. Saraswat, and Y. Deville. Constraint processing in cc(fd). In A. Podelski, editor, *Constraint Programming: Basics and Trends*. Springer-Verlag, 1995.

[26] C.P. Williams and T. Hogg. Exploiting the deep structure of constraint problems. *Artificial Intelligence*, 70:73–117, 1994.