Solving Constraint Satisfaction Problems with Heuristic-based Evolutionary Algorithms

B.G.W. Craenen
Vrije Universiteit
Faculty of Exact Sciences
De Boelelaan 1081
1081 HV Amsterdam
The Netherlands

A.E. Eiben

Vrije Universiteit
Faculty of Exact Sciences
De Boelelaan 1081
1081 HV Amsterdam
The Netherlands

Leiden University
Leiden Institute for Advanced Computer Science
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

E. Marchiori
Vrije Universiteit
Faculty of Exact Sciences
De Boelelaan 1081
1081 HV Amsterdam
The Netherlands

Abstract- Evolutionary algorithms (EAs) for solving constraint satisfaction problems (CSPs) can be roughly divided into two classes: EAs with adaptive fitness functions and heuristic based EAs. In [10] effective EAs of the first class have been compared experimentally using a large set of benchmark instances consisting of randomly generated binary CSPs. In this paper we complete this comparison by performing the same experiments using three of the most effective heuristic based EAs. The results of the experiments indicate that the three heuristic based EAs have similar performance on random binary CSPs. Comparing these results with those in [10], we are able to identify the best EA for binary CSPs as the algorithm introduced in [5] which uses a heuristic as well as an adaptive fitness function.

1 Introduction

Constraint satisfaction is a fundamental topic in artificial intelligence with relevant applications in planning, default reasoning, scheduling, etc. Informally, a constraint satisfaction problem consists of finding an assignment of values to variables in such a way that the restrictions imposed by the constraints are satisfied. CSPs are, in general, computationally intractable (NP-hard): as a consequence a number of heuristic algorithms have been developed for the approximated solution of CSPs. In particular, in the last decade various methods based on evolutionary algorithms have been introduced. Evolutionary algorithms (EAs) for CSPs can be roughly divided into two classes: EAs using adaptive fitness functions [2, 5, 6, 8, 9, **?**, 18, **?**] and EAs using heuristics [11, 13, 21, 20]. In [10], an experimental comparison of EAs of the first class was done using a test suit consisting of randomly generated binary CSPs. In this paper we perform a comparative study of three EAs of the second class using the same benchmark instances as in [10]. The results of the experiments indicate that `H-GA.1` by Eiben et al. [11] slightly outperforms the other two algorithms suggesting that this version of `H-GA` encompasses a good balance between the avoidance of premature convergence and guidance of the search process. However, when considering also the results from [10], the best EA for random binary CSPs turns out to be the algorithm by Dozier et al. [5, 7] which employs a heuristic based on the Iterative Descent Method as well as an adaptive fitness function. This seems to indicate that the integration of adaptive operators with heuristics are required in order to obtain an effective EA for solving binary CSPs. The paper is organized as follows. Section 2 contains the problem specification. In Section 3 we describe the main features of the three heuristic based EAs we intend to compare. Section 4 presents the results of the experiments. In Section 5 we compare the results with those reported in other experimental studies. Finally, in Section 6 we draw some conclusions.

2 Random Binary CSPs over Finite Domains

We consider binary CSPs over finite domains, where constraints act between pairs of variables. This is not restrictive since every CSP can be transformed into an equivalent binary CSP (c.f. [23]). A *binary CSP* is a triple $(V, \mathcal{D}, C)$ where $V = \{v_1, \ldots, v_n\}$ is a set of variables, $\mathcal{D} = (D_1, \ldots, D_n)$ is a sequence of finite domains, such that v_i takes values 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 (v_i, v_j). The pairs of values which are not compatible, also called incompatible pairs, are used in the generator of random binary CSPs we will use in the experiments. For simplicity, in the sequel we assume all domains are equal ($D_i = D$ for $i \in \{1, n\}$). An *instantiation* ι is a mapping $\iota : V \rightarrow D$, where $\iota(v_i)$ is the value associated to v_i. A *solution* σ of a CSP is an instantiation such that $(\sigma(v_i), \sigma(v_j))$ is in c_{ij}, for every v_i, v_j in V with $i \neq j$. A class of random binary CSPs can be specified by four parameters $\langle n, m, d, t \rangle$ with n the number of variables, m the (uniform) domain size, d the *constraint density* and t the *constraint tightness*. The constraint density is the proba-

bility that a constraint exists between two variables, while the constraint tightness is the probability of conflict between two values across a constraint. When one of the parameters is varied, CSPs exhibit a *phase transition* where problems change from being relatively easy to solve to being very easy to prove unsolvable. The region where this phenomenon occurs is also called mushy region [22]. Problems in the mushy region are identified as the most difficult to solve or prove unsatisfiable (cf., e.g., [3, 19, 22, 25]). For instance, for $n = 20$, $m = 10$ and $d = 0.5$ the mushy region consists of those instances with $0.34 < t < 0.4$: it contains a mixture of soluble and insoluble problems. All problems with $t \leq 0.34$ are soluble, while all problems with $t \geq 0.4$ are insoluble. The test suit used for the experiments conducted in this paper consists of problem instances produced by the generator[1] by J. van Hemert, which is loosely based on the generator by G. Dozier [2, 7]. The generator produces a random binary CSP by assigning $\frac{n(n-1)}{2} \cdot d$ constraints between two randomly selected variables (v_i and v_j) and then assigning $m \cdot m \cdot t$ incompatible pairs to the constraint.

3 Heuristic EAs for CSPs

We consider three heuristic based EAs: ESP-GA by E. Marchiori [13], H-GA by Eiben et al. [11] and Arc-GA by M. C. Riff Rojas [21, 20]. The three EAs were selected because of the different heuristics they use: ESP-GA employs a simple repair rule as separate module (hence the genetic operators are blind); H-GA incorporates an heuristic in its genetic operators; Arc-GA incorporates information on the constraint network into the genetic operators and fitness function. All algorithms use the integer representation: an individual is a sequence of integers where integer p in the i-th entry indicates that the i-th variable is instantiated to value p.

3.1 ESP-GA

In [13] E. Marchiori introduces an EA for solving CSPs which transforms constraints into a canonical form in such a way that there is only one single (type of) primitive constraint. This approach, called glass box approach, is used in constraint programming [24], where CSPs are given in implicit form by means of formulas of a given specification language. For instance, for the N-Queens Problem, we have the well known formulation in terms of the following constraints:

two queens cannot be on the same row:
$v_i \neq v_j$ for all $i \neq j$

two queens cannot be on the same diagonal:
$abs(v_i - v_j) \neq abst(i - j)$ for all $i \neq j$

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, called *constraint processing*, is done in two steps: elimination of functional constraints (as in GENOCOP [16]) and decomposition of the CSP into primitive constraints. The choice of primitive constraints depends on the specification language. The primitive constraints chosen in the examples considered in [13], the N-Queens Problem and the Five Houses Puzzle, are linear inequalities of the form: $\alpha \cdot v_i - \beta \cdot v_j \neq \gamma$. When all constraints are reduced to the same form, a single probabilistic repair rule is applied, called *dependency propagation*. The repair rule used in the examples is of the form **if** $\alpha \cdot p_i - \beta \cdot p_j = \gamma$ **then** change p_i or p_j. The violated constraints are processed in random order. Repairing a violated constraint can result in the production of new violated constraints, which will not be repaired. Thus at the end of the repairing process the chromosome will not in general be a solution. ESP-GA is designed under the implicit assumption that CSPs are given in implicit form by means of formulas in some specification language.

In order to investigate the performance of ESP-GA on the random binary CSPs, which are given explicitly by means of a table of incompatible values, we translate the table into constraints of the form $\alpha \cdot v_i - \beta \cdot v_j \neq \gamma$, by setting $\gamma = |D| \cdot p_i - p_j$ (with p_i, p_j the values of v_i, v_j) and $\alpha = |D|$ and $\beta = 1$. Violation of such constraints is detected by entering the values of the specified variables and checking if the result is the calculated γ-value. This transformation produces constraints in canonical form, hence the constraint processing of ESP-GA becomes unnecessary and the ESP-GA becomes a simple EA with repair rule. Moreover, we modify slightly the repair rule by selecting the variable whose value has to be changed as the one which occurs in the largest number of constraints, and by setting its value to a different value in D. The genetic operators we use are defined as follows. The crossover operator is the standard one-point crossover. The mutation is the random mutation which sets the value of a randomly chosen variable to a randomly selected value from its domain. The main features of ESP-GA are summarized in Table 1.

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

Table 1: Specific features of ESP-GA

3.2 H-GA

In [**?**, 11], Eiben et al. propose to incorporate existing CSP heuristics into genetic operators. Two heuristic based genetic operators are specified: an asexual operator that transforms one individual into a new one and a multi-parent operator that generates one offspring using two or more parents. In the next two subsections we discuss both heuristic based genetic

[1] see http://www.wi.leidenuniv.nl/home/jvhemert/csp-ea/

operators in more detail.

3.2.1 Asexual heuristic based genetic operator

The asexual heuristic based genetic operator selects a number of variables in a given individual, and then selects new values for these variables. We consider the operator that changes up to one fourth of the variables, selects the variables that are involved in the largest number of violated constraints, and selects the values for these variables which maximize the number of constraints that become satisfied.

3.2.2 Multi-parent heuristic crossover

The basic mechanism of this crossover operator is scanning: for each position, the values of the variables of the parents in that position are used to determine the value of the variable in that position in the child. The selection of the value is done using the heuristic employed in the asexual operator. The difference with the asexual heuristic operator is that the heuristic does not evaluate all possible values but only those of the variables in the parents. The multi-parent crossover is applied with 5 parents and produces one child.

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

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

We consider three EAs based on this approach, and call them `H-GA.1`, `H-GA.2`, and `H-GA.3`. As seen in Table 2, 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 normally filled in by mutation. The same random mutation operator used in `ESP-GA` is used in `H-GA.1` and `H-GA.2`.

3.3 `Arc-GA`

In [21, 20] M. C. Riff Rojas introduces an EA for solving CSPs which uses information about the constraint network in the fitness function and in the genetic operators (crossover and mutation). The fitness function is based on the notion of *error evaluation* of a constraint. The error evaluation of a constraint is the sum of the number of variables of the constraint[2] and the number of variables that are connected to these variables in the CSP network. The fitness function of an individual, called *arc-fitness*, is the sum of error evaluations of all the violated constraints in the individual. The mutation operator, called *arc-mutation*, selects randomly a variable of an individual and assigns to that variable the value that minimizes the sum of the error-evaluations of the constraints involving that variable. The crossover operator, called *arc-crossover*, selects randomly two parents and builds an offspring by means of the following iterative procedure over all the constraints of the considered CSP. Constraints are ordered according to their error-evaluation with respect to instantiations of the variables that violate the constraints. For the two variables of a selected constraint c, say v_i, v_j, the following cases are distinguished. If none of the two variables are instantiated in the offspring under construction, and none of the parents satisfies c, then a pair of values for v_i, v_j from the parents is selected which minimizes the sum of the error evaluations of the constraints containing v_i or v_j whose other variables are already instantiated in the offspring. If there is one parent which satisfies c, then that parent supplies the value for the child. If both parents satisfy c, then the parent which has the higher fitness provides its values for v_i, v_j. If only one variable, say v_i, is not instantiated in the offspring under construction, then the value for v_i is selected from the parent minimizing the sum of the error-evaluations of the constraints involving v_i. If both variables are instantiated in the offspring under construction, then the next constraint (in the ordering described above) is selected.

[2] In a binary CSP there are just two variables

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

Table 3: Specific features of `Arc-GA`

4 Experimental Comparison

All three algorithms use a steady state model with a population of 10 individuals. The choice of such a small population is justified by computational testing (see also [10, **?**]). At each generation two new individuals are created using the crossover (or the main genetic operator), and both new individuals are mutated. Linear ranking with bias $b = 1.5$ is used as parent selection while the replacement strategy removes the two individuals in the population that have the lowest fitness. The results in Tables 4 and 5 are obtained by testing the three methods (five algorithms) on random binary CSPs with 15 variables and a uniform domain size of 15. We generate 25 classes of instances by considering the combinations of 5 different constraint tightness and 5 different density values. For each class 10 instances are generated and 10 independent runs are performed on each instance; thus the results for each class are the averages over 100 runs. All the algorithms stop if they find a solution or if a maximum of $100,000$ fitness eval-

uations is reached. In order to compare the algorithms two performance measures are used: the percentage of runs that found a solution, called success rate (SR), and the average number of fitness evaluations to solution (AES) in successful runs[3].

density	alg.	tightness				
		0.1	0.3	0.5	0.7	0.9
0.1	Esp-GA	1	1	1	1	0.68
	H-GA.1	1	1	1	1	0.49
	H-GA.2	1	1	1	1	0.46
	H-GA.3	1	1	1	1	0.43
	Arc-GA	1	1	1	1	0.30
0.3	Esp-GA	1	1	1	0.02	0
	H-GA.1	1	1	1	0.30	0
	H-GA.2	1	1	1	0.06	0
	H-GA.3	1	1	1	0.05	0
	Arc-GA	1	1	0.99	0	0
0.5	Esp-GA	1	1	0.04	0	0
	H-GA.1	1	1	0.18	0	0
	H-GA.2	1	1	0.15	0	0
	H-GA.3	1	1	0.14	0	0
	Arc-GA	1	1	0.04	0	0
0.7	Esp-GA	1	1	0	0	0
	H-GA.1	1	1	0	0	0
	H-GA.2	1	1	0	0	0
	H-GA.3	1	1	0	0	0
	Arc-GA	1	0.97	0	0	0
0.9	Esp-GA	1	0	0	0	0
	H-GA.1	1	0.49	0	0	0
	H-GA.2	1	0.36	0	0	0
	H-GA.3	1	0.35	0	0	0
	Arc-GA	1	0.17	0	0	0

Table 4: SR of Esp-GA, H-GA.{1,2,3}, and Arc-GA

Tables 4 and 5 give some indication of the *landscape of solvability* for the different EAs. This landscape of solvability typically has a high SR for binary CSP instances that have low density and/or tightness, with SRs dropping as density and/or tightness becomes higher. The region where the algorithm exhibits a *phase transition* is of particular interest since it contains hard problem instances. The *mushy region* of the algorithms consists of the binary CSPs with the following density-tightness combinations: $(0.1, 0.9)$, $(0.3, 0.7)$, $(0.5, 0.5)$, $(0.7, 0.3)$ and $(0.9, 0.3)$. This is in accordance with the theoretical predictions of phase transitions for binary CSPs (cf. e.g., [22]). When looking at the SR of the algorithms on hard instances we found that Arc-GA has the worst success rate while both H-GA and ESP-GA find more solutions. The only exception to this is in density-tightness combination $(0.9, 0.3)$ where ESP-GA finds no solutions and Arc-GA still finds 17 solutions out of a hundred experiments. The results indicate that H-GA.1 outperforms the other algorithms when looking at SR, with a single exception for density-tightness combination $(0.1, 0.9)$. When looking at the AES of the algorithms in the *mushy region* we found a tie between H-GA.1 and Arc-GA: in density-tightness combinations $(0.1, 0.9)$, $(0.3, 0.7)$ and $(0.7, 0.3)$ H-GA performs better while in density-tightness combinations $(0.5, 0.5)$ and $(0.9, 0.3)$ Arc-GA has the best performance. Concerning the three versions of H-GA, we conclude that the heuristic asexual version outperforms the multi-parent crossover operator and that the replacement in the latter algorithm of the random mutation operator with an heuristic mutation operator based on the asexual crossover operator does not improve the performance. Based on the good performance of H-GA.1 when looking at SR and the fair performance when looking at AES we conclude that H-GA.1 is the best algorithm of the five tested. We suspect that the success of H-GA.1 lies in the fact that it uses heuristic information in such a way that premature convergence of the population is avoided while still providing guidance for finding solutions.

[3]If $SR = 0$ then AES is undefined

density	alg.	tightness				
		0.1	0.3	0.5	0.7	0.9
0.1	Esp-GA	10	17	28	68	2858
	H-GA.1	10	12	14	23	190
	H-GA.2	11	292	907	1942	10988
	H-GA.3	11	261	956	1989	13111
	Arc-GA	10	18	32	77	319
0.3	Esp-GA	14	52	667	81891	-
	H-GA.1	11	19	63	272	-
	H-GA.2	279	2381	6567	24123	-
	H-GA.3	293	2400	7087	24226	-
	Arc-GA	16	50	452	-	-
0.5	Esp-GA	23	268	18648	-	-
	H-GA.1	13	34	4205	-	-
	H-GA.2	998	4826	24455	-	-
	H-GA.3	897	4885	21430	-	-
	Arc-GA	92	88	955	-	-
0.7	Esp-GA	31	22218	-	-	-
	H-GA.1	17	179	-	-	-
	H-GA.2	1621	10259	-	-	-
	H-GA.3	1637	10284	-	-	-
	Arc-GA	37	367	-	-	-
0.9	Esp-GA	43	-	-	-	-
	H-GA.1	19	1776	-	-	-
	H-GA.2	2310	30443	-	-	-
	H-GA.3	2314	32095	-	-	-
	Arc-GA	46	1439	-	-	-

Table 5: AES of Esp-GA, H-GA.{1,2,3}, and Arc-GA

5 Discussion

It is interesting to compare the results with those reported in [10], where three EAs using adaptive fitness functions have been tested on the same benchmark instances as used here.

The best results in that article were obtained by the microgenetic iterative descendent genetic algorithm (MID) of Dozier et al [5]. MID incorporates heuristics in the reproduction mechanism and in the fitness function in order to direct the search towards better individuals. More precisely, MID works on a pool of 8 individuals. It uses a roulette-wheel based selection mechanism; however, it is not generational, but has a steady state reproduction mechanism where at each generation an offspring is created by mutating a specific gene of the selected chromosome, called pivot gene, and that offspring replaces the worse individual of the actual population.

Roughly, the fitness function of a chromosome is determined by adding a suitable penalty term to the number of constraint violations the chromosome is involved in. The penalty term depends on the set of break-outs whose values occur in the chromosome. A break-out consists of two parts: 1) a pair of values that violates a constraint; 2) a weight associated to that pair. The set of break-outs is initially empty and it is modified during the execution by increasing the weights of 1 and by adding new break-outs according to the technique used in the Iterative Descent Method [17].

In [5] it is shown that MID outperforms the Iterative Descent Method algorithm [17].

density	tightness				
	0.1	0.3	0.5	0.7	0.9
0.1	1	1	1	1	0.96
0.3	1	1	1	0.52	0
0.5	1	1	0.9	0	0
0.7	1	1	0	0	0
0.9	1	1	0	0	0

Table 6: SR for MID

density	tightness				
	0.1	0.3	0.5	0.7	0.9
0.1	1	4	21	87	2923
0.3	3	50	323	32412	-
0.5	10	177	26792	-	-
0.7	20	604	-	-	-
0.9	33	8136	-	-	-

Table 7: AES for MID

Tables 6 and 7 report SR and AES obtained by MID: the results show that MID has better performance than the algorithms considered in this paper, both in terms of SR and AES.

The success of MID can be explained from the fact that it belongs to both classes of EAs mentioned in the introduction: it uses a heuristic method incorporated into the mutation operator and an adaptive mechanism which changes the fitness function during the run. The results of the experiments seem to indicate that the search for a solution does profit from the combination of heuristic information and dynamic adaptation of the fitness function.

An alternative evolutionary approach for solving CSPs is the so-called genetic local search. In the genetic local search approach, genetic operators act on a population of local optima resulting from the application of a local search procedure to each chromosome. This approach has been shown to be rather effective for tackling NP-hard combinatorial optimization problems (cf., e.g. [14]). In particular, in [15] the authors show that a genetic local search algorithm obtained by incorporating local search into a simple GA yields equal or better results than MID on the same benchmark instances as used here. The resulting evolutionary algorithm is called Repair Improve Genetic Algorithm. More specifically, RIGA is a generational genetic algorithm with elitist selection mechanism which copies the best individual of a population to the population of the next generation [**?**]. A chromosome after application of local search represents a partial solution (that is some variables may not be instantiated, and all the constraints whose variables are both instantiated are satisfied). The fitness of a chromosome is equal to the number of instantiated variables in the chromosome. The genetic operators are blind: uniform crossover and random mutation which adds or deletes randomly selected values from the genes (thus genes before application of local search may contain more than one value; see [15] for more details).

Tables 8 and 9 report SR and AES obtained by RIGA.

density	tightness				
	0.1	0.3	0.5	0.7	0.9
0.1	1	1	1	1	0.96
0.3	1	1	1	0.72	0
0.5	1	1	1	0	0
0.7	1	1	0	0	0
0.9	1	1	0	0	0

Table 8: SR for RIGA

density	tightness				
	0.1	0.3	0.5	0.7	0.9
0.1	10	10	10	17	197
0.3	10	10	24	15604	-
0.5	10	10	6809	-	-
0.7	10	42	-	-	-
0.9	10	588	-	-	-

Table 9: AES for RIGA

Observe that AES for RIGA does not take into account the computational effort of local search, hence it is much lower than AES of MID. In terms of running time, MID is about five times faster than RIGA. Thus the computational effort of MID can be fairly considered less than the one of RIGA.

In [4] it is shown that the performance of RIGA does not improve when the SAW-ing method is incorporated in the selection mechanism of the GA: SAW-ing uses an on-line fitness adjusting mechanism adaptively raising penalties of

variables that are often involved in constraint violations. This seems to indicate that the search guidance provided by SAWing is already present in the genetic local search algorithm.

6 Conclusion

The experimental study conducted in this paper together with those reported in [10, 15] indicate that effective methods based on evolutionary algorithms for solving random binary CSPs need to incorporate problem knowledge, either in the form of ad hoc genetic operators and fitness function, or in the form of a local search procedure.

In particular, the best performance is obtained by those evolutionary algorithms incorporating local search, either in the genetic operators and fitness function [5], or as an external module [15].

Bibliography

[1] R.K. Belew and L.B. Booker, editors. *Proceedings of the 4th International Conference on Genetic Algorithms*. Morgan Kaufmann Publishers, Inc., 1991.

[2] 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 Publishers, Inc., 1995.

[3] P. Cheeseman, B. Kenefsky, and W.M. Taylor. Where the really hard problems are. In J. Mylopoulos and R. Reiter, editors, *Proceedings of the 12th IJCAI*, pages 331–337. Morgan Kaufmann Publishers, Inc., 1991.

[4] B.G.W. Craenen, A.E Eiben, E. Marchiori, and A. Steenbeek. Combining local search and fitness function adaptation in a GA for solving binary constraint satisfaction problems. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2000)*, 2000.

[5] G. Dozier, J. Bowen, and D. Bahler. Solving small and large constraint satisfaction problems using a heuristic-based microgenetic algorithm. In IEEE [12], pages 306–311.

[6] 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 Computer Society Press, 1995.

[7] G. Dozier, J. Bowen, and A. Homaifar. Solving constraint satisfaction problems using hybrid evolutionary search. *Transactions on Evolutionary Computation*, 2(1):23–33, 1998.

[8] 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 Lecture Notes in Computer Science, pages 95–106, Berlin, 1998. Springer-Verlag.

[9] 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.

[10] 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 Lecture Notes in Computer Science, pages 196–205, Berlin, 1998. Springer-Verlag.

[11] A.E. Eiben, P-E. Raué, and Zs. Ruttkay. Solving constraint satisfaction problems using genetic algorithms. In IEEE [12], pages 542–547.

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

[13] 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, San Francisco, CA, 1997. Morgan Kaufmann Publishers, Inc.

[14] E. Marchiori. A simple heuristic based genetic algorithm for the maximum clique problem. In *Proceedings of the ACM Symposium on Applied Computing*, pages 366–373. ACM Press, 1998.

[15] E. Marchiori and A. Steenbeek. Genetic local search algorithm for random binary constraint satisfaction problems. In *Proceedings of the ACM Symposium on Applied Computing*, 2000. to appear.

[16] Z. Michalewicz and M. Schoenauer. Evolutionary algorithms for constrained parameter optimization problems. *Evolutionary Computation*, 4(1):1–32, 1996.

[17] P. Morris. The breakout method for escaping from local minima. In *Proceedings of the 11th National Conference on Artificial Intelligence, AAAI-93*, pages 40–45. AAAI Press/The MIT Press, 1993.

[18] J. Paredis. Coevolutionary constraint satisfaction. In 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, pages 46–55, Berlin, 1994. Springer-Verlag.

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

[20] M.C. Riff-Rojas. Evolutionary search guided by the constraint network to solve CSP. In Belew and Booker [1], pages 337–348.

[21] M.C. Riff-Rojas. Using the knowledge of the constraint network to design an evolutionary algorithm that solves CSP. In Belew and Booker [1], pages 279–284.

[22] 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, New York, NY, 1994. John Wiley & Sons.

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

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

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