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 using adaptive fitness functions and EAs using heuristics. In [8] the most 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 studying the most effective EAs of the second class. We test three heuristic based EAs on the same benchmark instances used in [8]. The results of our experiments indicate that the three heuristic based EAs have similar performance on random binary CSPs. Moreover, comparing these results with those in [8], we are able to identify the best EA for binary CSPs as the algorithm introduced in [3] 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 (CSP) 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) and the algorithms that solve them can be divided into two classes: the ones that are tailored to solve a specific CSP and the ones that use 'rules-of-thumb' or heuristics to solve them. Although heuristics do not guarantee successful performance, they are able to produce an answer in a very short time and are used to guide the algorithm through the search space. Evolutionary algorithms (EAs) for CSPs can be divided into two classes: EAs using adaptive fitness functions ([1, 3, 4, 6, 7, 11, 17, 18]) and EAs using heuristics ([10, 15, 20, 21]). In [8], an experimental comparison of EAs of the first class was done using a test suite consisting of randomly generated binary CSPs. In this paper we perform a comparative study on three EAs of the second class ([10, 15, 20, 21]) using the same benchmark instances as in [8]. A large number of experiments were done and they indicate that `H-GA.1` outperforms the other algorithms suggesting that this version of `H-GA` strikes the best balance between the avoidance of premature convergence and guidance of the search process. However, when considering the results from [8], the best EA for random binary CSPs is the algorithm by Dozier et al. [3, 5] which uses a heuristic as well as an adaptive fitness function. This seems to indicate that both the adaptive operators as well as heuristics are required for an effective EA for solving binary CSPs. The paper is organized as follows. Section 2 contains the definition of CSPs. 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. Finally, in Section 5 we conclude with a discussion of the results.

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 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 (v_i, v_j). In the sequel, we shall often use the incompatible pairs of values when dealing with constraints, like, e.g., in the generator of random binary CSPs. For simplicity we assume all domains equal ($D_i = D$ for $i \in \{1, n\}$). An *instantiation* α is a mapping $\alpha : V \to D$, where $\alpha(v_i)$ 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*. Constraint

density is the probability of a constraint between two variables,while constraint tightness is the probability of conflict between two values. When the density or the tightness is varied, CSPs exhibit a *phase transition* where problems change from being relatively easy to solve to being very easy to prove unsolvable. Problems in the phase transition are identified as the most difficult to solve or prove unsatisfiable (cf., e.g., [2, 19, 22, 25]). The test suite used for the experiments consists of problem instances produced by a generator[1] loosely based on the generator of G. Dozier [1, 5]. The generator produces a CSP by assigning $\frac{n(n-1)}{2} \cdot d$ constraints between two randomly selected variables (v_i and v_j) and then assigning $|D_i| \cdot |D_j| \cdot t$ conflicts to the constraint.

3 Heuristic EAs for CSPs

We consider three heuristic based EAs: ESP-GA by E. Marchiori [15], H-GA by Eiben et al.[10] and Arc-GA by M. C. Riff Rojas [20, 21]. The three EAs were selected because of their different use of heuristics: ESP-GA uses heuristics in a repair rule combined with blind genetic operators, H-GA uses heuristics in its genetic operators and Arc-GA uses heuristics guided by the constraint network in two novel genetic operators and a new 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 set to value p.

3.1 ESP-GA

In [15], E. Marchiori introduces an EA for solving CSPs which adjusts the CSP in such a way that there is only one single (type of) primitive constraint. This algorithm is loosely based on the *glass box* approach from [24]. 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 into constraints of a single canonical form. These primitive constraints are linear inequalities of the form: $\alpha \cdot v_i - \beta \cdot v_j \neq \gamma$. When all constraints share the same form a single repair rule can be used to enforce *dependency propagation*. The repair of an individual is done locally by applying the repair rule to every violated constraint. The repair rule of the form **if** $\alpha \cdot p_i - \beta \cdot p_j = \gamma$ **then** modify p_i or p_j is applied to all individuals in the population. In the repair rule we select the variable which occurs in the largest number of constraints, and set its value to a new value in the domain of that variable. The violated constraints to be repaired are selected in a random order. The representation of the constraints as generated by the CSP generator is a table of incompatible values. ESP-GA, on the other hand, was devised with the implicit assumption that the CSP is syntactically by means of a formula. Therefore we translate the tables of the CSP generator into constraints of the form $\alpha \cdot v_i - \beta \cdot v_j \neq \gamma$, by setting $\gamma = |D_j| \cdot p_i - p_j$ (with p_i, p_j the values of v_i, v_j) and $\alpha = |D_j|$ and $\beta = 1$. Violation of such a constraint is detected by entering the values of the specified variables and checking if the result is the calculated γ-value. The above mentioned translation produces constraints in canonical form, hence, the constraint processing of ESP-GA becomes unnecessary. This reduces ESP-GA to an EA with a repair rule. The genetic operators we use are defined as follows. The crossover operator is the standard one-point crossover: a randomly chosen position divides each parents in two parts. The two children are constructed by taking the one part from the first (respectively second) parent and the other part from the second (respectively first) parent. The mutation is the random mutation which set 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 [9, 10], Eiben et al. propose to incorporate existing CSP heuristics into genetic operators operators. 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. In the next two subsections we will discuss both heuristic operators in more detail.

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. 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

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

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 [20, 21] M. C. Riff Rojas introduces a 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 number of variables of the constraint[2] and the number of variables that are connected to these variables in the CSP network. It is used as a measure of the connectivity of the network and as an indicator of how important it is to satisfy the variable. 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 assign 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 a child by means of an 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 yet in the offspring under construction, and none of the parents satisfies c, then a combination 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.

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 [12] or to a lesser extend [8]). Per generation two new individuals are created using the crossover or main operator, both new individuals are mutated. Linear ranking with bias $b = 1.5$ is used as parent selection while the elitist 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 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 constraints tightness and 5 different density values. In each class 10 instances are generated and 10 independent runs are performed on each instance, the results for each class are the averages over 100 runs. All the algorithms stop if they find a solution or after a maximum of $100,000$ fitness evaluations. In order to compare the algorithms, two performance measures are used: the percentage of runs that found a solution, the success rate (SR), and the average number of fitness evaluations to solution (AES) in successful runs[3].

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 and is called the *mushy region*. The *mushy region* of the algorithms consists of the binary CSPs with 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 ([22]). When looking at the SR of the algorithms in the *mushy region* we

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

[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	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

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

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 $(0.9, 0.1)$ where ESP-GA finds no solutions and Arc-GA still finds 17 solutions out of a hundred experiments. In general one can also conclude that H-GA.1 outperforms all other algorithms when looking at SR again with a single exception in 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 as 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. About the three versions of H-GA we conclude that the heuristic asexual version outperforms the multi-parent crossover operator and that the addition of an heuristic mutation operator, based on the asexual crossover operator does not improve 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 heuristics in such a way that premature convergence of the population is avoided while still providing guidance the is strong enough to find a solution.

5 Conclusion and Further Research

It is interesting to compare the results with those reported in [8], where three EAs using adaptive fitness functions have been tested on the same benchmark instances as used here. The best success rates in that article were obtained by the microgenetic iterative descendent genetic algorithm (MID) of Dozier et al [3]. This algorithm employs heuristic information in the reproduction operator as well as an adaptive penalty mechanism in the fitness function.

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: Success rates for MID

Table 6 reports the success rates obtained by MID, it indicates that MID can solve random binary CSPs much better than the algorithms considered in this paper. MID is also faster with respect to AES in all cases (cf. [8]). The suc-

cess 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 redefining the fitness function during the run. It is reasonable to assume that the search for a solution does profit from the combination of these two features. Future work is directed to assess the performance of the combination of the heuristics applied in H-GA.1 and the fastest method from [8, 7], called SAW-ing EA, that uses an on-line fitness adjusting mechanism adaptively raising penalties of variables that are often involved in constraint violations. Furthermore, the use of a restart-strategy for 'fast' algorithms with low SR and, different (combinations of) heuristics will also be studied.

Bibliography

[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] G. Dozier, J. Bowen, and D. Bahler. Solving small and large constraint satisfaction problems using a heuristic-based microgenetic algorithm. In IEEE [13], pages 306–311.

[4] 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.

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

[6] 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, 1998.

[7] 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.

[8] 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.

[9] 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.

[10] A.E. Eiben, P-E. Raué, and Zs. Ruttkay. Solving constraint satisfaction problems using genetic algorithms. In IEEE [13], pages 542–547.

[11] 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.

[12] J.K. van der Hauw. Evaluating and improving steady state evolutionary algorithms on constraint satisfaction problems. Master's thesis, Leiden University, 1996. Also available as http://www.wi.leidenuniv.nl/MScThesis/IR96-21.html.

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

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

[15] 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.

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

[17] J. Paredis. Co-evolutionary 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 46–56. Springer-Verlag, 1994.

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

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

[20] M.C. Riff-Rojas. Using the knowledge of the constraint network to design an evolutionary algorithm that solves CSP. In IEEE [14], pages 279–284.

[21] M.C. Riff-Rojas. Evolutionary search guided by the constraint network to solve CSP. In IEEE [14], pages 337–348.

[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. Wiley, 1994.

[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, 1995.

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