Hybrid Evolutionary Algorithms for Constraint Satisfaction Problems: Memetic Overkill?

B.G.W. Craenen
Napier University
10 Colinton Road
Edinburgh, EH10 5DT
United Kingdom
b.craenen@napier.ac.uk

A.E. Eiben
Vrije Universiteit Amsterdam
De Boelelaan 1081a
1081 HV, Amsterdam
The Netherlands
gusz@cs.vu.nl

Abstract- We study a selected group of hybrid EAs for solving CSPs, consisting of the best performing EAs from the literature. We investigate the contribution of the evolutionary component to their performance by comparing the hybrid EAs with their "de-evolutionarised" variants. The experiments show that "de-evolutionarising" can increase performance, in some cases doubling it. Considering that the problem domain and the algorithms are arbitrarily selected from the "memetic niche", it seems likely that the same effect occurs for other problems and algorithms. Therefore, our conclusion is that after designing and building a memetic algorithm, one should perform a verification by comparing this algorithm with its "de-evolutionarised" variant.

1 Introduction

During the last decade, many researchers have adopted the use of heuristics within an evolutionary algorithm (EA) because of the positive effect on algorithm performance. Advocated already in the mid 90ies (cf. [22]), such algorithms, called hybrid EAs or memetic algorithms, offer the best of both words: the robustness of the EA because of the unbiased population-based search and the directed search implied by the heuristic bias. As for algorithm performance, it is assumed and expected that the hybrid EA performs better than the EA alone and the heuristic alone. Supported by significant practical evidence, the contemporary view within the EC community considers this memetic approach the most successful in treating challenging (combinatorial) optimisation problems.

In this paper we add a critical note to this opinion. In particular, we design and perform targeted experiments to assess the contribution of the evolutionary component of hybrid EAs to good results. The way to test this is to "de-evolutionarise" the EAs and see whether the results get better or worse. Technically speaking the question is how to "remove evolution" from an EA. For a solid answer one should identify the essential features of EAs for which it holds that after removing or switching off these features, the resulting algorithm would not qualify for being evolutionary. To this end, there are three obvious candidates for belonging to these essential features: Namely, the usage of:

- a population of candidate solutions;
- variation operators, crossover and mutation; and
- natural selection, that is selection based on fitness.

Our work, as reported here, is based on the third answer for the following reasons. Considering the role of the population, it is true that, in general, EAs use a population of more than one candidate solution. However, there are many successful variants, where the population size is only one, think for instance of evolution strategies [2, 10, 27].

As for the variation operators, we can observe that some move-operator in the search space is always necessary for generate-and-test methods. Hence, the sole presence of the variation operators is not EA specific. The speciality of EAs is often related to the use of crossover for mixing information of two or more candidate solutions. Nevertheless, there are many EAs that do not use crossover, or any other form of recombination, for instance in evolutionary programming, cf. [10, 15, 14].

Considering natural selection, recall that there are two selection steps in the general EA framework: parent selection and survivor selection. For either of them we say that it represents natural selection if a fitness-based bias is incorporated. Note, that an EA does not need to have natural selection in both steps. For instance, generational GAs use only parent selection (and all children survive), while ES use only survivor selection (and parents are chosen uniform randomly). However, an EA must have fitness-bias in at least one of these steps. If neither parent selection nor survivor selection are performed by using fitness-bias (e.g., by uniform random selection) then we have no natural selection and obtain random walk.

Based on these considerations we "de-evolutionarise" the EAs by switching off natural selection. Technically, we set all selection operators to uniform random, that is, candidate solutions are selected by selecting them randomly while each candidate solution has an equal probability of being selected.

2 CSPs and our generator

The *Constraint Satisfaction Problem* (CSP) is a well-known satisfiability problem that is NP-complete ([26]). Informally, the CSP is defined as a set of *variables* X and a set *constraints* C between these variables. Variables are only assigned values from their respective *domains*, denoted as D. Assigning a value to a variable is called *labelling* a variable and a label is a variable-value pair, denoted: $\langle x, d\rangle$. The simultaneous assignment of several values to their variables is called a *compound label*. A constraint is then a set of compound labels, this set determines when a constraint is *violated*. If a compound label is not in a constraint, it *satisfies* the constraint. A compound label that violates a constraint is called a *conflict*. A *solution* of the CSP is defined as the compound label containing all variables in such a way that no constraint is violated. The number of distinct variables in the compound labels of a constraint is called the *arity* of the constraint and these variables as said to be relevant to the constraint. The arity of a CSP is the maximum arity of its constraints, this is denoted with the letter k.

In this paper we consider only CSPs with an arity of two ($k = 2$), called *binary CSPs*. All constraints of a binary CSP have arity two. Although the restriction to binary constraints appears to be a serious limitation to the CSP, E. Tsang showed that every CSP can be transformed to an equivalent binary CSP ([29]). Two methods for translating CSPs have been proposed: the dual graph translation ([6]) and the hidden variable translation ([5]). Both methods were discussed in [1] and it was found that the choice of the transformation method had a large impact on the performance of the algorithm used to solve the resulting binary CSPs. However, in this paper, we will used randomly generated binary CSPs so this problem does not affect the presented outcomes.

In this paper we will consider CSPs with a *uniform domain size* only. The *number of variables* and the uniform domain size of the CSP are two complexity measures of the CSP. They are denoted with n and m respectively. The larger the number of variables and/or the larger the uniform domain size, the more difficult the CSP will be to solve. There are two more complexity measures that will be used: density and average tightness. *Density* is defined as the ratio between the maximum number of constraints of a CSP ($\binom{|X|}{2}$ for a binary CSP) and the actual number of constraints ($|C|$) and is denoted as a real number between 0.0 and 1.0, inclusive. The *tightness* of a constraint is one minus the ratio between the maximum number of compound labels possible ($|D_x \times D_y|$ for a binary constraint over variables x and y) and the actual number of compound labels in the constraint. The *average tightness* of a CSP is then the average tightness of all constraints in the CSP. Density is denoted as p_1 and average tightness as $\overline{p_2}$. All four complexity measures together form the *parameter vector* of a CSP: $\langle n, m, p_1, \overline{p_2}\rangle$.

Finding more efficient algorithms to solve CSPs has been an important driving force behind the study of CSPs. The lack of a good set of CSP-instances was seen as a major obstacle and has lead to research in ways of generating these randomly. It was soon realised that an algorithm that solves a particular set of CSP-instances efficiently may have disappointing performance on other CSP-instances. This in turn lead to research on how to produce sets of randomly generated CSP-instances that qualify as a reasonable representation of the whole class.

In the last two decades, several models for randomly generating CSP-instances have been designed ([23, 18, 19]). These models use some or all parameters of the parameter vector of a CSP to control the complexity of the instances generated. By analysing the performance of algorithms on instances generated with different parameter settings, the behaviour of the algorithms throughout the parameter space of the CSP can be studied. A set of CSP-instances for empirically testing the performance of an algorithm is called a *testset*.

Simply put, generating a CSP-instance involves choosing which constraint to add to the instance and which compound labels to remove from these constraints. Two methods for making these choices exist: the *ratio*-method and the *probability*-method. In the ratio-method a predetermined ratio of constraints are added to the CSP and a predetermined ratio of compound labels is then added to these constraints (constraints are assumed to be initialised empty). These ratios are based on the p_1 and $\overline{p_2}$ parameters of the CSP respectively. The probability-method considers each constraint and each compound label in the constraint separately and, based on the p_1 parameter for the constraints and the $\overline{p_2}$ parameter for the compound labels, determines if it is added to the CSP. In the end there are two methods for adding constraints and two methods for adding compound labels to these constraint. These can be combined into four models for generating CSP-instances randomly, called A, B, C, and D ([23, 19]).

In [18] it was found that when the number of variables (n) of a CSP is large, almost all instances generated by these models become unsolvable because of the existence of flawed variables. A *flawed variable* is a variable for which all values in its domain violate a relevant constraint. This is the result of models A to D's two-step approach for generating CSP-instances. To overcome this unwanted behaviour, a new model, called model E, was introduced. Model E combines both steps and generates CSP-instances by adding $(1 - p_e)\binom{n}{2}m^2$ compound labels out of the $\binom{n}{2}m^2$ possible ones. The p_e parameter of model E is then a combination of the p_1 and $\overline{p_2}$ parameters of models A to D. However, in [19], it was found that even for small values of p_e (e.g. $p_e < 0.05$), all possible constraints of the CSP-instance will have been added by the model E generator. In the same paper, a new model, model F, was proposed, in which first a model E generator was used to generate a CSP-instance and then a number of constraints are removed (using the ratio-method). The parameter vector of the model F random CSP generator is then: $\langle n, m, p_1, p_e\rangle$. Because the generator uses the p_e parameter of model E and because some compound labels will be removed as well, some experimental tweaking of the p_e parameter is needed to generate CSP-instance with a certain $\overline{p_2}$ value.

3 EAs for solving CSPs

In the last two decades many EAs have been proposed for solving the CSP ([8, 9, 25, 24, 20, 16, 21, 7, 11]). In [4], the performance of a representative sample of these EAs was compared on a large testset of CSP-instances generated via model E. In [3], another comparison of a larger

	HEA	**LSEA**	**ESPEA**	**rSAWEA**
Evolutionary Model	Steady state	Steady state	Steady state	Steady state
Representation	Ordered set of values	Domain sets	Ordered set of values	Permutation of variables
Fitness Function	no. violated constraints	Special	no. violated constraints	SAW
Crossover	Heuristic multi-parent	Special	Uniform random	None
Mutation	Heuristic	Special	Uniform random	Swap
Parent selection	Biased Ranking	Biased Ranking	Biased Ranking	Biased Ranking
Survivor selection	Replace worst	Replace worst	Replace worst	Replace worst
Other	None	Special repair	Special repair	Decoder

Table 1: Characteristics of the HEA, LSEA, ESPEA, and the rSAWEA

number of these EAs, including a larger number of algorithm variants, was included and this time the comparison was done on a testset that was generated with a model F generator. In [3] it was found that the Heuristic EA (HEA), the Local-Search EA (LSEA), the Eliminate-Split-Propagate EA, and the Stepwise-Adaptation-of-Weights EA (rSAWEA) outperformed all the other EAs.

Space limitations preclude us to include a full description of these four algorithms but [3] describes these algorithms fully and the original articles of the authors of these algorithms can be used as well: [8, 9] for HEA, [21] for LSEA, [20] for ESPEA, and [12, 13] for SAWEA.[1] A table showing the characteristics for these four algorithms is included in table 1.

4 Experimental setup

As stated in the introduction, we propose to de-evolutionarise the HEA, LSEA, ESPEA, and rSAWEA by removing natural selection. Natural selection is implemented in the two selection operators of the EAs: the parent selection operator and the survivor selection operator. To remove natural selection, both operators have to be changed. This is done by uniform randomly selecting parents for offspring in the parent selection operator and by uniform randomly selecting the survivors that will be added to the new population in the survivor selection operator. By using uniform selection in both operators, no bias is applied through selection and in theory, the EAs should perform a random walk through the search space.

To show the difference between the performance in these experiments we will run two experiments for all four algorithms and show their results back-to-back.

4.1 Testset

For the experiments in this paper we use the same testset as in [3]. The testset consists only of model F generated solvable CSP-instances and each instance has 10 variables ($n = 10$), and a uniform domain size of 10 ($m = 10$). For nine density-tightness combinations in the so called mushy region, 25 CSP-instances were selected from a population of 1000 generated CSP-instances. The mushy region is a region in the density-tightness parameter space where the CSP-instances generated change from being solvable to being unsolvable. The mushy region can be determined exactly by calculating the number of solutions, using a formula provided by Smith in [28]: $m^n(1-\overline{p_2}^{\binom{n}{2}p_1}$ (for binary CSPs). Smith predicted that the mushy region can be found were the number of solutions of the generated CSPs would be one, assuming that this solution will be hard to find among all other possible compound labels. The nine density-tightness combinations used are $1 : (0.1, 0.9)$, $2 : (0.2, 0.9)$, $3 : (0.3, 0.8)$, $4 : (0.4, 0.7)$, $5 : (0.5, 0.7)$, $6 : (0.6, 0.6)$, $7 : (0.7, 0.5)$, $8 : (0.8, 0.5)$, and $9 : (0.9, 0.4)$. We identify the density-tightness combinations in the mushy region by the numbers given above.

The CSP-instances in the testset are selected by a method in four steps: parameter adjustment, sample sizing, formula correction, and instance selection. In the parameter adjustment step, a sample of CSP-instances are generated and the parameters used to generate these instances are compared to the complexity measures calculated for these instances. The parameters are adjusted to remove any difference between the parameters and the complexity measures. In the sample sizing step, the size of the CSP-instance sample is determined by comparing the found average number of solutions in the sample with the calculated number of solutions from Smith's formula. The size of the sample is increased when the difference between the two is significant, with a (practical) maximum of 1000 instances for each density-tightness combination. In the formula correction step the calculated number of solutions is corrected for any remaining difference. For the instance selection step, a new sample of only solvable CSP instances equal to the size of the sample sizing step is generated. For each density-tightness combination in the mushy region, 25 CSP-instances are selected for the testset that are the closest to the corrected number of solutions found in the formula correction step.

In total the testset includes $9 \cdot 25 = 225$ CSP instances. The testset can be downloaded at: `http://www.xs4all.nl/~bcraenen/resources/testset_mushy.zip`.

[1]One technical note on this latter algorithm, however, is necessary. Here we use a slightly modified version of the original SAWEA, where for each variable the domain is randomly shuffled before the decoder is applied. We denote this algorithm by rSAWEA. A full description of this paper can be found in [3]

4.2 Performance measures

Three measures are used to measure the performance of the algorithms in this paper: the success rate (SR), the average number of evaluations to solution (AES), and the average number of conflict checks to solution (ACCS). The SR will be used to describe the effectiveness of the algorithms, the AES and ACCS will be used to describe the efficiency of the algorithms.

The SR measure is calculated by dividing the number of successful runs, that is the number of runs in which the algorithm found a solution to the CSP, by the total number of runs. The measure is given as a percentage, 100% meaning all runs were successful. The SR is the most important performance measure to compare two algorithms with. An algorithm with a higher SR finds more solutions than an algorithm with a lower SR. The accuracy of the SR measure is influenced by the total number of runs.

The AES measure is defined as the average number of fitness evaluations needed by an algorithm over all successful runs. If a run is unsuccessful, it will not show in the AES measure, if all runs are unsuccessful (SR=0), the AES is undefined. The AES measure is a secondary measure for comparing two algorithms and its accuracy is affected by the number of successful runs of an algorithm. It should be noted that counting fitness evaluations is a standard way of measuring efficiency on EC. However, in our case, much work performed by the heuristics remains hidden from this measure, for instance by being done in a mutation operator. This motivates the usage of the third measure.

The ACCS measure is calculated by the average number of conflict checks needed by an algorithm over all successful runs. A conflict check is the check made to see if a certain compound label is in a constraint. As with the AES measure, the ACCS measure is undefined when all runs are unsuccessful and its accuracy is affected by the number of successful runs of an algorithm. The ACCS measure is a more fine grained measure than the AES and also measures the so-called hidden work done by the algorithm.

4.3 EA setup

The EAs were setup with as little difference between the parameter setups as possible. Table 2 shows the parameter setup of all four algorithms. All algorithms use a population of 10 individuals, from with 10 individuals are selected using a biased ranking parent selection operator with a bias of 1.5. The HEA, LSEA, and ESPEA have a crossover operator which is always applied (crossover rate). The HEA, LSEA, and rSAWEA need extra parameters, the values for these parameters are shown in table 2. How these extra parameters are used can be seen in [3] or in the original papers of these algorithms. The ESPEA does not have any extra parameters.

	HEA	**LSEA**	**ESPEA**	**rSAWEA**
Population size	10	10	10	10
Selection size	10	10	10	10
Max. Evaluations	100000	100000	100000	100000
Ranking Bias	1.5	1.5	1.5	1.5
Crossover Rate	1.0	1.0	1.0	-
Mutation Rate	1.0	1.0	0.1	1.0
HEA no. Variables	3	-	-	-
HEA no. Parents	5	-	-	-
LS Add Rate	-	0.1	-	-
LS Remove Rate	-	0.05	-	-
LS Delete Rate	-	0.9	-	-
SAW Interval	-	-	-	25
SAW Δ	-	-	-	1

Table 2: Parameter setup of the HEA, LSEA, ESPEA, and the rSAWEA

5 Results and analysis

The results of the experiments are summarised in Tables 3, 4, 5, and 6, for the HEA, LSEA, ESPEA, and rSAWEA, respectively. The results for the real hybrid EAs are given in the left half of these tables. These figures show great differences between the algorithms. For example, on instance number 6 the success rates vary between poor (HEA: 44%), medium (75% and 80% for LSEA and ESPEA), and excellent (rSAWEA: 100%). The same holds for AES values, where the differences can be of a factor 10, e.g., HEA around one thousand, where LSEA is in the range of ten thousand.

	HEA			HEA w/o selection		
inst.	SR	AES	ACCS	SR	AES	ACCS
1	100	26	24	100	27	25
2	98	419	621	100	221	321
3	69	1635	2489	100	952	1436
4	71	1404	2110	100	404	604
5	69	2382	3647	69	717	1083
6	44	988	1493	96	1618	2468
7	59	969	1473	99	1960	2982
8	49	1258	1933	98	3601	5539
9	76	1563	2405	100	912	1393

Table 3: Performance of HEA and HEA without selection. SR in percentages, ACCS in thousands, rounded up

These results indicate a clear looser and a clear winner. The HEA is obviously the least performing algorithm, it can find a solution in fewer runs than the others, shown clearly by a lower SR. The LSEA and the ESPEA are quite close to each other in this respect, their success rates do not differ so much. Furthermore, on three out of the nine instances (1,7,9) they have an identical SR, and on the other six the two algorithms score the same: LSEA wins three times (instances 2,3,5) and so does ESPEA (instances 4,6,8). We could distinguish the two algorithms based on their efficiency: ESPEA is able to archive these success rates with less computational effort (AES and ACCS). The rSAWEA algorithm is the clear winner here as it can solve almost every problem instance and the amount of work it needs for this is significantly less than what the other algorithms need.

	LSEA			LSEA w/o selection		
inst.	SR	AES	ACCS	SR	AES	ACCS
1	100	13	9	100	13	9
2	99	540	300	99	540	300
3	81	9825	4714	81	9825	4714
4	81	5935	2642	81	5935	2642
5	92	10124	4307	92	10124	4307
6	75	12080	4573	75	12080	4573
7	78	11562	4674	78	11562	4674
8	80	11422	4280	80	11422	4280
9	94	4097	1690	94	4097	1690

Table 4: Performance of LSEA and LSEA without selection. SR in percentages, ACCS in thousands, rounded up

Analysing the results from the perspective of our main research goal we can observe rather surprising outcomes. The comparison of the hybrid EAs and their de-evolutionarised variants discloses that HEA and ESPEA become *better* if we do not use fitness information anywhere, i.e., neither within parent selection, nor within survivor selection. These results are almost ironic, considering that both algorithms originate from a pure EA, where the heuristics are the extra add-ons to improve the base algorithms. However, as the experiments show, the add-on can be worth more than the main algorithm. Or, turning the argument around, we could say that natural selection is only harmful here. The case of the LSEA is also somewhat surprising in that the results of the two algorithm variants are fully identical. In this case, fitness-bias in the selection operators seems to have no effect at all. The rSAWEA shows a different picture. Removing the evolution from this EA compromises performance. In terms of success rates the effects are not too negative, 1 % decrease maximum (on 3 instances), and on one instance the SR increases 1%. Nevertheless, the variant without natural selection is slower, in terms of AES as well as in terms of ACCS. These results, that is, the fact that de-evolutionarising rSAWEA makes it worse, indicate that the good performance of the rSAWEA is not simply the consequence of using a strong heuristic that exploits the properties of CSPs. The rSAWEA is actually very generic, it is only the decoder where a weak heuristic is applied: if all possible values for a variable would cause constraint violation, the variable is left unassigned. For this reason it is quite plausible that the rSAWEA is so successful because of the **combination** of the weak heuristic in the decoder (for deep search) and the adaptive fitness function in the SAW-mechanism (for wide search). This latter enables the algorithm to emphasise different constraints in different stages of the search and continuously redirect the "attention" of the EA.

Considering the results from the pure problem solving perspective we need to compare all eight algorithms based on their performance. The overall winner is then the ESPEA without natural selection, beating the really evolutionary rSAWEA. Their success rates are not that different, the rSAWEA looses only on 3 instances and only by a small margin. However, the de-evolutionarised ESPEA is much faster in terms of fitness evaluations (AES). It is also faster in terms of conflict checks (ACCS), but the differences regarding this measure are not that big.

	ESPEA			ESPEA w/o selection		
inst.	SR	AES	ACCS	SR	AES	ACCS
1	100	45	15	100	48	18
2	95	2404	925	100	275	179
3	73	6165	2671	100	629	423
4	84	6021	2785	100	529	347
5	84	4839	2416	100	442	297
6	80	6015	3040	100	736	492
7	78	9241	4739	100	839	558
8	84	9241	2498	100	1218	789
9	94	3589	2085	100	374	272

Table 5: Performance of ESPEA and ESPEA without selection. SR in percentages, ACCS in thousands, rounded up

	rSAWEA			rSAWEA w/o selection		
inst.	SR	AES	ACCS	SR	AES	ACCS
1	100	64	10	100	103	16
2	99	1750	351	100	5646	1044
3	96	3986	764	95	9801	1761
4	98	3598	652	97	5088	897
5	100	3166	557	100	3859	670
6	100	4024	715	99	5298	921
7	100	4878	864	100	7153	1250
8	100	5762	1012	100	7139	1240
9	100	2333	408	100	2609	462

Table 6: Performance of rSAWEA and rSAWEA without selection. SR in percentages, ACCS in thousands, rounded up

6 Conclusions

In this paper we have compared the best four heuristic EAs for solving randomly generated binary CSPs on instances from the mushy region. Such heuristic EAs, or memetic algorithms, supposedly obtain their good performance from two sources: the evolutionary, and the heuristic component. In order to assess the contribution of the evolutionary component, we also implemented a de-evolutionarised version of all of these EAs and tested them on the same test suite. We de-evolutionarised EAs by removing any fitness-based bias from selection and making all choices based on drawings from a uniform distribution. The results showed that two EAs became better, one became worse, and one remained the same. The overall winner of the whole pool of algorithms turned out to be one where natural selection was switched off. In this case it can be argued that the algorithm is not evolutionary at all. These outcomes hint on a "memetic overkill" in the sense that adding too much heuristics to an EA to increase its performance might make evolutionary component of the hybrid EA or memetic algorithm superfluous.

A remaining question is the possible role of the population. As we listed in the Introduction, there are more options for removing the evolution from an EA. In particular, one could set the population size at one (and consequently get rid of crossover as a variation operator). Testing this option could show if using the heuristics in a population-based manner offers advantages over simply using them in an iterative improvement scheme. This could shed further light on the issue of memetic overkill.

In summary, here we have shown that simply de-evolutionarising a hybrid EA can greatly increase its performance. This means that, even though one arrived to the algorithm design from an evolutionary starting point,[2] the best algorithm variant is not necessarily evolutionary. Strictly speaking, we have observed this effect only on one problem (randomly generated binary CSPs) and a few algorithms, hence we cannot simply generalise our findings without a risk. However, considering that the problem domain and the algorithms are arbitrarily selected from the "memetic niche", it seems very likely that the same effect occurs for other problems and algorithms. Therefore, our conclusion is that after designing and building a memetic algorithm, one should always perform a verification step by comparing this algorithm with its de-evolutionarised variant.

Bibliography

[1] F. Bacchus and P. van Beek. On the conversion between non-binary and binary constraint satisfaction problems. In *Proceedings of the 15th International Conference on Artificial Intelligence – ICAI98*, pages 311–318, Madison, Wisconsin, July 1998. Morgan Kaufmann.

[2] T. Bäck. *Evolutionary Algorithms in Theory and Practice*. Oxford University Press, New York, NY, 1996.

[3] B.G.W. Craenen. *Solving Constraint Satisfaction Problems with Evolutionary Algorithms*. Doctoral dissertation, Vrije Universiteit Amsterdam, Amsterdam, The Netherlands, 2005. In Press.

[4] B.G.W. Craenen, A.E. Eiben, and J.I. van Hemert. Comparing evolutionary algorithms on binary constraint satisfaction problems. *IEEE Transactions on Evolutionary Computing*, 7(5):424–445, Oct 2003.

[5] R. Dechter. On the expressiveness of networks with hidden variables. In T. Dietterich and W. Swartout, editors, *Proceedings of the 8th National Conference on Artificial Intelligence*, pages 556–562, Hynes Convention Centre, 1990. MIT Press.

[6] R. Dechter and J. Pearl. Tree clustering for constraint networks. *Artificial Intelligence*, 38(3):353–366, 1989.

[2]That is, starting with a general EA and adding heuristics to it for increasing its performance.

[7] G. Dozier, J. Bowen, and D. Bahler. Solving small and large constraint satisfaction problems using a heuristic-based micro-genetic algorithm. In ICEC94 [17], pages 306–311.

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

[9] A.E. Eiben, P-E. Raué, and Zs. Ruttkay. Solving constraint satisfaction problems using genetic algorithms. In ICEC94 [17], pages 542–547.

[10] A.E. Eiben and J.E. Smith. *Introduction to Evolutionary Computing*. Springer, 2003. ISBN 3-540-40184-9.

[11] A.E. Eiben and J.K. van der Hauw. Adaptive penalities for evolutionary graph-coloring. In J.-K. Hao, E. Lutton, E. Ronald, M. Schoenauer, and D. Snyers, editors, *Artificial Evolution '97 – AE97*, volume 1363 of *Lecture Notes on Computer Science*, pages 95–106. Springer-Verlag, Berlin, 1998.

[12] 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.

[13] A.E. Eiben and J.I. van Hemert. Saw-ing eas: Adapting the fitness function for solving constrained problems. In D. Corne, M. Dorigo, and F. Glover, editors, *New Ideas in Optimization*, pages 389–402. McGraw-Hill, 1999.

[14] D.B. Fogel. *Evolutionary Computation*. IEEE Computer Society Press, 1995.

[15] L.J. Fogel, A.J. Owens, and M.J. Walsh. *Artificial Intelligence through Simulated Evolution*. John Wiley & Sons, 1966.

[16] H. Handa, N. Baba, O. Katai, T. Sawaragi, and T. Horiuchi. Genetic algorithm involving coevolution mechanism to search for effective genetic information. In *Proceedings of the 4th Conference on Evolutionary Computation – ICEC97*, pages 709–714. IEEE Society Press, 1997.

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

[18] D. Ach/ liop/ tas, L.M. Kir/ ou/ sis, E. Kra/ na/ kis, D. Kri/-zanc, M.S. Mol/-loy, and Y.C. Sta/ ma/ tiou. Random constraint satisfaction a more accurate picture. In G. Smolka, editor, *Principles and Practice of Constraint Programming – CP97*, pages 107–120. Springer Verlag, 1997.

[19] E. MacIntyre, P. Prosser, B.M. Smith, and T. Walsh. Random constraint satisfaction: theory meets practice. In M. Maher and J.-F. Puget, editors, *Principles and Practice of Constraint Programming – CP98*, pages 325–339. Springer Verlag, 1998.

[20] 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.

[21] E. Marchiori and A. Steenbeek. A genetic local search algorithm for random binary constraint satisfaction problem. In *Proceedings of the 14th Annual Symposium on Applied Computing*, pages 463–469, 2000.

[22] Z. Michalewicz. *Genetic Algorithms + Data Structures = Evolutionary Programs*. Springer-Verlag, Berlin, 3rd edition, 1996.

[23] E.M. Palmer. *Graphical Evolution. An introduction to the theory of random graphs.* Wiley-Interscience Series in Discrete Mathematics. John Wiley & Sons, Ltd., Chichester, 1985.

[24] 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 – PPSN94*, volume 886 of *Lecture Notes in Computer Science*, pages 46–55. Springer Verlag, 1994.

[25] M.-C. Riff Rojas. Using the knowledge of the constraint network to design an evolutionary algorithm that solves csp. In *Proceedings of the 3rd IEEE Conference on Evolutionary Computation – ICEC96*, pages 279–284. IEEE Computer Society Press, 1996.

[26] F. Rossi, C. Petrie, and V. Dhar. On the equivalence of constrain satisfaction problems. In L.C. Aiello, editor, *Proceedings of the 9th European Conference on Artificial Intelligence (ECAI'90)*, pages 550–556, Stockholm, 1990. Pitman.

[27] H.-P. Schwefel. *Evolution and Optimum Seeking*. John Wiley & Sons, New York, NY, 1995.

[28] 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.

[29] E. Tsang. *Foundations of Constraint Satisfaction*. Academic Press, 1993.