Comparing Classical Methods for Solving Binary Constraint Satisfaction Problems with State of the Art Evolutionary Computation

J.I. van Hemert

Leiden Institute of Advanced Computer Science, Leiden University
jvhemert@cs.leidenuniv.nl

Abstract. Constraint Satisfaction Problems form a class of problems that are generally computationally difficult and have been addressed with many complete and heuristic algorithms. We present two complete algorithms, as well as two evolutionary algorithms, and compare them on randomly generated instances of binary constraint satisfaction problems. We find that the evolutionary algorithms are less effective than the classical techniques.

1 Introduction

A *constraint satisfaction problem* (CSP) is a set of variables all of which have a corresponding domain. As well as variables, it contains a set of constraints that restricts certain simultaneous value assignments to occur. The objective is to assign each variable one of the values from its domain without violating any of the restricted simultaneous assignments as set by the constraints.

Constraint satisfaction has been a topic of research in many different forms. Lately, research has been concentrating on binary constraint satisfaction problems, which is a more abstract form of constraint satisfaction. This general model has resulted in a quest for faster algorithms that solve CSPs. Just like particular constraint satisfaction problems, such as k-graph colouring and 3-SAT, binary CSPs have a characteristic phase-transition, which is the transition from a region in which almost all problems have many solutions to a region in which almost all problems have no solutions. The problem instances found in this phase-transition are among the hardest to solve, making them interesting candidates for experimental research [15].

Backtracking kind of algorithms were the first to be used for tackling CSPs. One of the main advantages of these *classical algorithms* is that they are sound (i.e., every result is indeed a solution) and complete (i.e., they make it possible to find all solutions) [14]. Unfortunately, such algorithms need enormous amounts of constraint checks to solve the problem instances inside the phase-transition. To counter for this, many techniques have been developed and most of these keep the properties of soundness and completeness intact.

Compared with the classical algorithms, the evolutionary computation technique is a newcomer in constraint satisfaction. During the past ten years much study has gone into improving the speed and accuracy of evolutionary algorithms

on solving CSPs. Since evolutionary algorithms are stochastic, they are neither sound nor complete.

In this paper we first give more details on binary constraint satisfaction problems. Then, in Section 3 and Section 4 we present the algorithms. Section 5 provides the experimental setup and the results, followed by conclusions in Section 6.

2 Binary Constraint Satisfaction Problems

In general, a CSP has a set of constraints where each constraint can be over any subset of the variables. Here, we focus on binary CSPs: a model that only allows constraints over a maximum of two variables. At first, this seems a restriction, but Tsang has shown [14] this is not the case by proving that any CSP can be rewritten into a binary CSP. Solving the general CSP corresponds then to finding a solution in the binary form. Although Bacchus and van Beek [1] argue that the binary form may, depending on the problem, not be the most efficient way of handling a CSP, it still remains a popular object of study.

The binary model has led to a study where the object is to find problem instances that are difficult to solve. Collections of these problem instances serve as a good test bed for experimental research on solving methods. The idea behind finding difficult problem instances is that these are most likely to be the problem instances having only one solution, yet without being over-constrained and without any other structure that would make the solution easy to find. The first models were made using four parameters in the model, the number of variables n, the overall domain size m, the density of the constraints p_1 and the average tightness of the constraints p_2. Smith [13] estimated the number of solutions by using these parameters in a predictor shown in Equation 1.

$$E(solutions) = m^n(1 - p_2)^{\frac{n(n-1)p_1}{2}} \tag{1}$$

The predictor helps us in visualising the total space of problem instances. If we fix two of the four parameters in the model, in our case the number of variables and the domain size, we are left with the parameters density and tightness. Using Equation 1 we can plot these latter parameters against the predicted number of solvable problem instances $E(solutions)$. Figure 1 shows this plot for the case where we set the number of variables and domain size to fifteen. To make the plot easier to read we cut off $E(solutions)$ above one.

Figure 1 clearly shows three regions. First, it shows a region where the number of solvable problem instances is equal to or higher than one ($E(solutions) \geq 1$). Second, it shows a region where the number of solvable problem instances is equal to zero ($E(solutions) = 0$) and third, it shows a region in between the other regions, which is called the *mushy region* ($0 < E(solutions) < 1$). Some problem instances that have the density and tightness settings within the mushy region have no solutions, while others have. To make the region more visible a contour plot is drawn in the x,y-plane. This is the area of parameter settings where we expect to find the most difficult to solve problem instances.

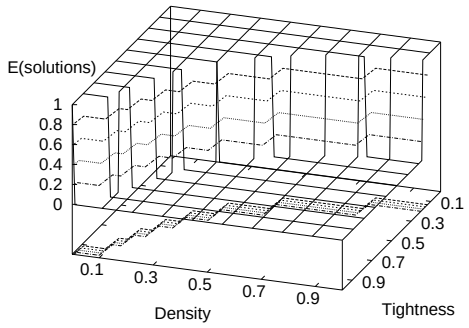


Fig. 1. The expected number of solutions with $E(solutions) \leq 1$ for fixed $n = 15$ and $m = 15$ plotted against density and tightness

3 Classical Algorithms

The term *classical algorithm* is used in this paper for algorithms that are not based on evolutionary computation. Algorithms in this section have been tested with an adapted version of *csplib*[1] where we have added a model for randomly generating binary CSPs.

3.1 Chronological Backtracking

Chronological backtracking (BT) is a simple algorithm dating back to 1965 [7] which is easy to construct using recursion. This algorithm is selected because it is simple and most algorithms studied in the field of constraint programming are based on some form of backtracking. These algorithms rely on recursion to move them through the search space, guided by all sorts of heuristics to find solutions as quickly as possible.

```
bool Backtrack(solution, current)
  if current > number_of_variables then
    return true;
  else
    foreach value ∈ domain_current do
      solution[current] = value;
      if Consistent(solution, current) then
        if Backtrack(solution, current+1) then
          return true;
  return false;
```

```
bool Consistent(solution, current)
  for i = 1 ... current − 1 do
    constraint_checks++;
    if Conflict(current, i, solution[current],
      solution[i]) then
      return false;
  return true;
```

Fig. 2. Pseudo code for chronological backtracking

The chronological backtracking algorithm is shown in pseudo code in Figure 2. This code uses the function `Conflict` to check whether the value assign-

[1] available at `http://www.lpaig.uwaterloo.ca/~vanbeek/software/csplib.tar.gz`

ment of two variables (*current* and i) raises a constraint conflict. A global variable *constraint_checks* is used to keep track of the number of constraint checks performed during a run of the algorithm. Chronological backtracking consists of two functions. First, the recursive function `Backtrack` that tries, for every variable, all values of the corresponding variable's domain. Second, a function called `Consistent` is used to check if a partial solution causes a constraint violation involving any of the previous assigned variables.

3.2 Forward Checking with Conflict-Directed Backjumping

The method *forward checking with conflict-directed backjumping* (FC-CBJ) is constructed using two techniques. Forward checking, which originates from 1980 [8] and constraint-directed backjumping, which was added in 1993 [12].

Forward checking instantiates a current variable and then checks forwards all the uninstantiated (future) variables. During this process all values incompatible with the current variable are removed from the domains of the corresponding uninstantiated variables. The process continues until it has instantiated all variables, i.e., found a solution, or until it checks a variable that has an empty domain left. In the last case the effects of the forward checking, i.e., the shrinking of domains, is undone and a new value for the current variable is tried.

Conflict-directed backjumping tries to improve the speed of forward checking by jumping over previous conflict checks that are unnecessary to repeat. To make this possible more bookkeeping is needed. Every variable is assigned its own conflict set which contains future variables that have failed consistency checks with the value assigned to the current variable. More precisely, every time a consistency check fails between an instantiation of the current variable and an instantiation of a future variable, the future variable is added to the conflict set of the current variable. When a domain annihilation of a future variable occurs the variables in the conflict set are added to the current variable's conflict set. When we run out of values to try for our current variable we turn to its conflict set joined with the conflict set of the annihilated variable and pick the variable that has been assigned earliest in the search. Joining the conflict sets of the new current and old current variable, and the past set of the old current (minus the new current) ensures that we keep our information up-to-date.

4 Evolutionary Algorithms

Two evolutionary algorithms will represent the current state of evolutionary computation on solving binary CSPs. These two algorithms are at this moment reported as performing the best on solving binary CSPs. The algorithms presented here all try to minimise the fitness function, where a fitness of zero equals to finding a solution.

Before we present the two algorithms in detail we want to point out a potential problem. Before an evolutionary algorithm can solve a constraint satisfaction problem another problem has to be overcome. Basically, every evolutionary algorithm is a stochastic optimisation algorithm that can be applied to minimise a

(fitness) function. However, a constraint satisfaction problem is a decision problem which offers no help to our evolutionary algorithms. To bypass this problem we often measure another property, such as the number of violated constraints. By minimising this property we arrive at a solution for our decision problem when we arrive at zero constraint violations.

The algorithms here share a common feature. They both adapt the fitness function during the run. This means that over time the same candidate solutions can have different fitness values. The rationale behind this adaptive scheme is to let the algorithms gain information on the problem instance they are currently solving, in order to better guide their search for a solution.

4.1 Microgenetic Iterative Descent Method

The *microgenetic iterative descent* (MID) method is a rather technical evolutionary algorithm. It uses a small population, hence the term microgenetic. The original idea is from Dozier et al. [3, 4]. This study involves gradually improving the general idea of using heuristics and breakout mechanisms to solve CSPs. The implementation of Eiben et al. is a re-implementation of the versions described in [3, 4].

The core of MID is the breakout mechanism used to reduce the chance of getting stuck in local optima. Basically the breakout mechanism is a bookkeeper for pairs of values that have been involved in a constraint violation when the algorithm previously got stuck in a local optimum. The collection of breakouts with corresponding weights that is constructed during the search is used in the fitness function of the evolutionary algorithm. Equation 2 shows that the fitness function comprises of two parts. First, the sum of all violated constraints is counted per variable. Second, the sum of all weights of breakouts that are used by the solution. This discourages the appearance of solutions that make use of pairs of values known to appear in local optima:

$$\text{fitness}(\textit{solution}) = \sum_{v \in \text{variables}} V(v) + \sum_{b \in \textit{breakouts}} B(b), \tag{2}$$

where $V(v)$ is the number of violated constraints corresponding with variable v and where $B(b)$ is the weight corresponding with breakout b or 0 if b does not exist or when b is not used in the *solution*.

4.2 Stepwise Adaptation of Weights

The *stepwise adaptation of weights* (SAW) method [5, 6] is an extension for evolutionary algorithms that is intended to increase the efficiency of the search process. The earlier evolutionary algorithms used a fitness function that only counted the number of constraint violations. Obviously, this is a blind way of searching that ignores possible knowledge of, for instance, the internal structure of a problem. The SAW method is a technique for adding this kind of knowledge to the fitness function.

The SAW method works by assigning a weight w_c to each constraint c of the CSP instance to solve. These weights are all initialised to one. When the evolutionary algorithm is running, its main loop will be paused every 250 evaluations

to update the weights. Within an update each weight w_c is incremented with one if the corresponding constraint c is violated by the best solution in the current population.

By having this weight vector included into the fitness function of the evolutionary algorithm we hope to force the evolutionary algorithm into focusing more on constraints that seem hard to satisfy during a search. Equation 3 shows how the weights are included in the fitness function. Note that keeping the weights w_c set to one during the run of an evolutionary algorithm would result in a standard evolutionary algorithm, that is, without SAW. Hence,

$$\text{fitness}(\textit{solution}) = \sum_{c \in \textit{constraints}} w_c \cdot C(c) \tag{3}$$

$$\text{where } C(c) = \begin{cases} 1 & \text{if } c \text{ is violated by } \textit{solution,} \\ 0 & \text{otherwise.} \end{cases}$$

The evolutionary algorithm in [5, 6] uses an order-based representation, i.e., a permutation of the variables of the binary CSP. A greedy decoder is used to decode this permutation into a value assignment.

5 Experiments and Results

We randomly generate problem instances of binary CSPs, using RandomCsp [9]. The process whereby we create these instances is called Model B [11], which first determines how many constraints and conflicts an instance will contain and then distributes both of these uniform randomly over the instance. Every instance is generated with the same domain size for each of its variables, this domain size is fixed to fifteen. In the first experiment we will vary the constraint density and the constraint tightness, while in the second experiment we will vary the number of variables.

The results for the evolutionary algorithms used in this paper are taken from a study by Eiben et al. [6], where the same set of randomly generated binary CSPs is used.

We measure the *success rate* (SR) and *the average number of constraint checks* (ACS) for each pair of density and tightness. The classical algorithms are sound and therefore the percentage of solutions presented is the actual percentage of solutions in the test set. When an evolutionary algorithm reaches the same SR it has actually found a solution to every CSP that had one. Whenever an evolutionary algorithm is not able to find a solution within a fixed number of generated proposed solutions it is terminated. This number is fixed to 100,000 for both evolutionary algorithms.

5.1 Fixed Variables and Domain Sizes

The test set consists of problem instances created with two parameters fixed and two parameters that vary: the number of variables and the overall domain size are set to fifteen and the constraint density and average tightness are varied

from 0.1 to 0.9 with a step size of 0.2. Thereby, creating 25 different pairs of density and tightness. For each of these pairs we randomly generate 25 different problem instances. Because of the stochastic nature of evolutionary algorithms we let each of them do ten independent runs on each instance, which follows the procedure in [6].

Table 1. Success ratio and average number of constraint checks, with standard deviations within brackets for 25 pairs of density and tightness settings. Every result is averaged over 25 instances

density	algorithm	tightness									
		0.1		0.3		0.5		0.7		0.9	
0.1	BT	1.00	1.08e2 (3e0)	1.00	1.18e2 (9e0)	1.00	1.37e2 (1e1)	1.00	3.52e2 (9e2)	**0.84**	**7.99e3 (2e4)**
	FC-CBJ	1.00	1.53e3 (2e1)	1.00	1.44e3 (4e1)	1.00	1.35e3 (6e1)	1.00	1.25e3 (1e2)	**0.84**	**8.64e3 (2e4)**
	MID	1.00	1.01e1 (3e-1)	1.00	5.07e1 (2e1)	1.00	2.25e2 (6e1)	1.00	0.48e2 (3e2)	**0.84**	**2.09e5 (4e5)**
	SAW	1.00	1.00e1 (0)	1.00	1.00e1 (0)	1.00	1.75e1 (2e1)	1.00	8.91e1 (8e1)	**0.36**	**6.74e5 (5e5)**
0.3	BT	1.00	1.21e2 (1e1)	1.00	1.03e3 (4e3)	1.00	6.86e3 (1e4)	**0.56**	**1.60e5 (4e5)**	0.00	3.41e4 (7e3)
	FC-CBJ	1.00	1.4e3 (3e1)	1.00	1.08e3 (7e1)	1.00	9.42e2 (3e2)	**0.56**	**1.48e4 (2e4)**	0.00	3.41e4 (7e3)
	MID	1.00	1.01e1 (3e1)	1.00	1.74e3 (3e2)	1.00	1.2e4 (4e3)	**0.41**	**2.35e6 (8e5)**	0.00	3.15e6 (0)
	SAW	1.00	3.10e1 (0)	1.00	4.48e1 (4e1)	1.00	1.26e3 (7e2)	**0.17**	**2.78e6 (8e5)**	0.00	3.15e6 (0)
0.5	BT	1.00	1.31e2 (1e1)	1.00	1.07e3 (2e3)	**1.00**	**2.11e5 (5e5)**	0.00	1.59e4 (1e4)	0.00	2.64e4 (4e3)
	FC-CBJ	1.00	1.27e3 (4e1)	1.00	8.66e2 (6e1)	**1.00**	**2.29e4 (4e4)**	0.00	7.3e3 (2e3)	0.00	2.64e4 (4e3)
	MID	1.00	5.43e2 (1e2)	1.00	9.33e3 (2e3)	**0.93**	**1.65e6 (7e5)**	0.00	5.25e6 (0)	0.00	5.25e6 (0)
	SAW	1.00	5.22e1 (1e0)	1.00	3.82e2 (3e2)	**0.69**	**2.01e6 (2e6)**	0.00	5.25e6 (0)	0.00	5.25e6 (0)
0.7	BT	1.00	1.49e2 (2e1)	**1.00**	**2.43e4 (9e4)**	0.00	1.64e5 (9e4)	0.00	6.38e3 (3e3)	0.00	2.49e4 (3e3)
	FC-CBJ	1.00	1.16e3 (5e1)	**1.00**	**9.92e2 (5e2)**	0.00	3.97e4 (1e4)	0.00	4.42e3 (8e2)	0.00	2.49e4 (3e3)
	MID	1.00	1.55e3 (2e2)	**1.00**	**5.49e4 (2e4)**	0.00	7.35e6 (0)	0.00	7.35e6 (0)	0.00	7.35e6 (0)
	SAW	1.00	7.48e1 (9e0)	**1.00**	**5.28e3 (3e3)**	0.00	7.35e6 (0)	0.00	7.35e6 (0)	0.00	7.35e6 (0)
0.9	BT	1.00	3.29e3 (2e4)	**1.00**	**1.94e5 (2e5)**	0.00	7.40e4 (9e3)	0.00	3.38e3 (5e2)	0.00	2.28e4 (3e3)
	FC-CBJ	1.00	1.09e3 (5e1)	**1.00**	**1.47e4 (2e4)**	0.00	2.42e4 (2e3)	0.00	3.67e3 (4e2)	0.00	2.28e4 (3e3)
	MID	1.00	3.29e3 (5e2)	**1.00**	**7.4e5 (3e5)**	0.00	9.45e6 (0)	0.00	9.45e6 (0)	0.00	9.45e6 (0)
	SAW	1.00	1.02e2 (4e1)	**1.00**	**2.54e5 (1e5)**	0.00	9.45e6 (0)	0.00	9.45e6 (0)	0.00	9.45e6 (0)

A quick look at the results in Table 1 shows three distinct "regions" of parameter settings, which correspond to the theoretical expectation presented in the x,y-plane in Figure 1. First, the upper left where the density and tightness are low shows that all problem instances have solutions that may easily be found by every algorithm. Second, opposite of the left corner where density and tightness are high we see that none of the problem instances can be solved. Third, between these two regions a mushy region exists where not for every pair of density and tightness we may solve all of the problem instances. Furthermore, here the algorithms need significantly more constraint checks to determine the solution or to determine that no solution exists. Also we see that for those settings not every instance is solvable as is the case for density/tightness pairs 0.1/0.9 and 0.3/0.7.

Looking at the lower right corner we see the large difference in the number of constraint checks for the classical algorithms and for the evolutionary algorithms. The explanation for this difference is twofold. First, the large numbers for evolutionary algorithms occur because, in general, evolutionary algorithms do not

know when to quit. Second, the small numbers for the classical algorithms are caused by the problem instances becoming over-constrained, making it easy to reduce the search space.

The results of SAW in the upper left region seem a little strange. As evolutionary algorithms are generally started with a random population we would not expect them to find solutions quickly. Nevertheless, SAW is able to find a solution (on average) with fewer than 400 constraint checks for all of these pairs. Again the explanation is twofold. Firstly, this version of SAW uses a very small population size, thus losing little at the first evaluation of the whole population. Secondly, its greedy decoder is successful for easy problems that have many solutions, thus able to solve them in a few attempts.

When observing the results we see that except for easy problems classical algorithms are better. When we focus on the classical algorithms we see that the winner is FC-CBJ. Although it does not beat the other algorithms in the upper left and lower right regions it makes up for this small loss in the mushy region.

To verify the significance of our results we performed a number of statistical tests [10]. When we look at the results and take these as data for statistical analysis, we are confronted with a problem. The data is not normally distributed according to KS-Liliefors tests. Even if they were normally distributed the standard deviations are too far apart. This implies that the spread in the results is too large to perform statistical tests like student or ANNOVA. We turn towards a Wilcoxon Rank test to do our analysis as this test is able to handle data that is not normally distributed.

Table 2. Results of the Wilcoxon Rank test for paired up algorithms, where H_0^1 is the one-sided hypothesis, H_0^2 the two-sided hypothesis and z the normally distributed variable

	FC-CBJ	MID	SAW		MID	SAW	SAW
H_0^1	0.0005	0.0000	0.0000		0.0000	0.0000	0.0000
H_0^2 BT	0.0011	0.0001	0.0000	FC-CBJ	0.0000	0.0000	MID 0.0000
z	-3.2642	-4.0314	-13.0.99		-9.7333	-13.153	-19.634

We list the results of each instance, averaging the 10 runs for the evolutionary algorithms. This gives us 625 lines. Then we pair up two algorithms and perform the Wilcoxon's Rank Pairs Signed Rank Test [2] for pairwise ascertained data sets. Here the null-hypothesis (H_0) is that the both results are to be treated equally. We test the null-hypothesis for one-side (H_0^1) and for two-sides (H_0^2). In Table 2 we see that this test shows the results of our comparison is significant for every pair as the largest chance that the null-hypothesis is true is only 0.0011.

5.2 Scale-up tests

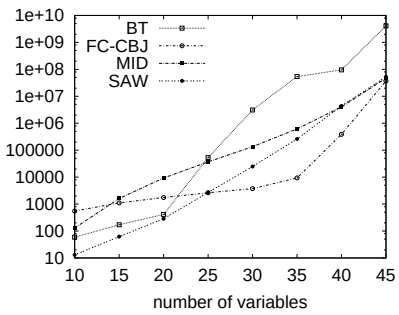
Besides a large comparison for a fixed number of variables and domain size, we also perform a scale-up test whereby we increase the number of variables. We

want to see whether evolutionary algorithms are able to outperform classical methods when the problem size increases. Again we randomly generate 25 different problem instances per settings. We start with 10 variables and increase with steps of 5 up until 45. Every instance used in this experiment has a solution.

Because for $n \leq 45$ every run of a classical algorithm produces a solution, we only present results on the average number of constraint checks. The graph on the right in Figure 3 shows, on a logarithmic scale, the results from the table on the left. Chronological backtracking scales up very poorly. Although, for a small number of variables chronological backtracking is able to compete with the other three algorithms, we see that from $n = 25$ onwards the number of constraint checks increases exponentially. The two evolutionary algorithms are able to keep up a good performance for small number of n, but eventually they have to give in. Very different is FC-CBJ that shows an almost constant performance until the number of variables has reached as much as 35.

Fig. 3. Results for the scale-up test where the number of variables is varied, while the domain size is fixed to 15 and the constraintness and tightness are both set to 0.3. Note that for $n = 45$ the maximum number of evaluations is set to 200,000

n	BT	FC-CBJ	MID		SAW	
	ACS	ACS	SR	ACS	SR	ACS
10	5.90e1	5.49e2	1.00	1.30e2	1.00	1.30e1
15	1.71e2	1.10e3	1.00	1.61e3	1.00	6.20e1
20	4.14e2	1.75e3	1.00	9.29e3	1.00	2.85e2
25	5.31e4	2.63e3	1.00	3.69e4	1.00	2.70e3
30	3.10e6	3.75e3	1.00	1.35e5	1.00	2.47e4
35	5.43e7	9.32e3	1.00	6.16e5	1.00	2.61e5
40	9.57e7	3.88e5	1.00	4.04e6	1.00	4.37e6
45	4.10e9	3.70e7	0.44	4.70e7	0.24	5.30e7



The last value ($n = 45$) is the first time that the evolutionary algorithms have trouble finding a solution within the maximum of 100,000 evaluations. MID has a success rate of 0.21 and SAW has 0.12. We increase this maximum to 200,000 because FC-CBJ is using more constraint checks than the previous maximum allows. MID's success rate rises to 0.44 and SAW's success rate rises to 0.24. The number of evaluations of both is larger than that of FC-CBJ.

6 Conclusions

In this study we have made a comparison based on a test set of randomly generated binary constraint satisfaction problems between two classical algorithms and two evolutionary algorithms. This comparison shows that evolutionary algorithms have to improve their speed by a considerable factor if they want to compete with an algorithm as simple as chronological backtracking.

In our comparison, the idea that evolutionary algorithms might outperform classical methods when the problem at hand is scaled-up holds only partially. Although both evolutionary algorithms easily outperform the simple backtracking algorithm, they are not fast enough to take on the more modern FC-CBJ.

One reason evolutionary algorithms are not promising is that they fail to realize when a problem has no solution — unlike the classical algorithms that, aided by preprocessing methods such as arc-consistency, are very efficient in pointing out the impossible.

References

[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*. Morgan Kaufmann, 1998.

[2] R.J. Barlow. *Statistics: a guide to the use of statistical methods in the physical sciences*. John Wiley & Sons, 1995.

[3] 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.

[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] 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.

[6] 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 *Proceedings of the 5th Conference on Parallel Problem Solving from Nature*, number 1498 in LNCS, pages 196–205, Berlin, 1998. Springer.

[7] S.W. Golomb and L.D. Baumert. Backtrack programming. *A.C.M.*, 12(4):516–524, October 1965.

[8] R. Haralick and G. Elliot. Increasing tree search efficiency for constraint-satisfaction problems. *Artificial Intelligence*, 14(3rd):263–313, 1980.

[9] J.I. van Hemert. Randomcsp, a library for generating and handling binary constraint satisfaction problems, Version 1.5, 2001. `http://www.liacs.nl/~jvhemert/randomcsp`.

[10] D.S. Moore and G.P. McCabe. *Introduction to the Practice of Statistics*. W.H. Freeman and Company, New York, 3rd edition, 1998.

[11] E. M. Palmer. *Graphical Evolution*. John-Wiley & Sons, New York, 1985.

[12] P. Prosser. Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence*, 9(3):268–299, August 1993.

[13] 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.

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

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