A Study into Ant Colony Optimization, Evolutionary Computation and Constraint Programming on Binary Constraint Satisfaction Problems

Jano van Hemert, Christine Solnon

A Study into Ant Colony Optimisation, Evolutionary Computation and Constraint Programming on Binary Constraint Satisfaction Problems

Jano I. van Hemert[1] and Christine Solnon[2]

[1] CWI, P.O. Box 94079, NL-1090 GB Amsterdam, The Netherlands
jvhemert@cwi.nl
[2] LIRIS, CNRS FRE 2672, University of Lyon 1, Nautibus,
43 bd du 11 novembre, 69 622 Villeurbanne cedex, France
csolnon@liris.cnrs.fr

Abstract. We compare two heuristic approaches, evolutionary computation and ant colony optimisation, and a complete tree-search approach, constraint programming, for solving binary constraint satisfaction problems. We experimentally show that, if evolutionary computation is far from being able to compete with the two other approaches, ant colony optimisation nearly always succeeds in finding a solution, so that it can actually compete with constraint programming. The resampling ratio is used to provide insight into heuristic algorithms performances. Regarding efficiency, we show that if constraint programming is the fastest when instances have a low number of variables, ant colony optimisation becomes faster when increasing the number of variables.

1 Introduction

Solving *constraint satisfaction problems* (CSPs) involves finding an assignment of values to variables that satisfies a set of constraints. This general problem has many real-life applications, such as time-tabling, resource allocation, pattern recognition, and machine vision.

To solve CSPs, one may explore the search space in a systematic and complete way, until either a solution is found, or the problem is proven to have no solution [1]. In order to reduce the search space, this kind of complete approach is usually combined with filtering techniques that narrow the variables domains with respect to some partial consistencies. Completeness is a very desirable feature, but it may become intractable on hard combinatorial problems. Hence, incomplete approaches, such as evolutionary computation and ant colony optimisation, have been proposed. These approaches do not perform an exhaustive search, but try to quickly find approximately optimal solutions in an opportunistic way: the search space is explored stochastically, using heuristics to guide the search towards the most-promising areas.

[2] and [3] provide experimental comparisons of different evolutionary algorithms with complete methods for solving binary constraint satisfaction problems. Both studies show that evolutionary computation is far from being able

to compete with complete methods. A first motivation of this paper is to go on these comparative studies by integrating another heuristic approach, based on the Ant Colony Optimisation metaheuristic. Another motivation is to propose an explanation of the reasons for the differences of performances of the three approaches. Indeed, [4] introduces the notion of resampling ratio, that allows one to measure how often an algorithm re-samples the search space, i.e., generates a candidate solution already generated before. This measure is used in this paper to get insight into search efficiency of the three compared approaches.

In the next section we explain what constraint satisfaction problems are and what makes them an interesting object of study. Then in Section 3 we introduce the three algorithms involved in the empirical comparison, which is presented in Section 4. Concluding remarks are given in Section 5.

2 Constraint Satisfaction

A *constraint satisfaction problem* (CSP) [1] is defined by a triple (X, D, C) such that X is a finite set of variables, D is a function that maps every variable to its finite domain and C is a set of constraints. A *label*, denoted by $\langle X_i, v_i \rangle$, is a variable-value pair that represents the assignment of value v_i to variable X_i. An *assignment*, denoted by $\mathcal{A} = \{\langle X_1, v_1 \rangle, \ldots, \langle X_k, v_k \rangle\}$, is a set of labels and corresponds to the simultaneous assignment of values $v_1, \ldots, v_k$ to variables $X_1, \ldots, X_k$ respectively. An assignment is *complete* if it contains a label for each variable of the CSP. A *solution of a* CSP is a complete assignment that satisfies all the constraints.

2.1 Random binary CSPs

Binary CSPs only have binary constraints, that is, each constraint involves two variables exactly. Binary CSPs may be generated at random. A class of randomly generated CSPs is characterized by four components $\langle n, m, p_1, p_2 \rangle$ where n is the number of variables, m is the number of values in each variable's domain, $p_1 \in [0, 1]$ is a measure of the density of the constraints, i.e., the number of constraints, and $p_2 \in [0, 1]$ is a measure of the tightness of the constraints, i.e., the number of inconsistent pairs of values for each constraint.

Experiments reported in this paper were obtained with random binary CSPs generated according to Model E as described in [5], that is, the set of conflicting value pairs is created by uniformly, independently and with repetitions selecting $p_e \binom{n}{2} m^2$ edges out of the $\binom{n}{2} m^2$ possible ones. This process guarantees that no flawed variables are generated, which would otherwise make instances easy to solve.

After generating a problem instance with Model E, we can measure the density and the tightness of the constraints for this instance. The density of the constraints p_1 is often equal to one, except for very small (< 0.05) values of p_e. The tightness of the constraints p_2 is always smaller than or equal to p_e because of the possible repetitions. With sufficiently high p_e values (> 0.11) the p_2 value will be lower than p_e.

2.2 Phase-transitions

When considering a class of combinatorial problems, rapid transitions in solvability may be observed as an order parameter is changed [6]. These "phase-transitions" occur when evolving from instances that are under-constrained, and therefore solved rather easily, to instances that are over-constrained, whose inconsistency is thus proven rather easily. Harder instances usually occur between these two kinds of "easy" instances, when approximately 50% of the instances are satisfiable.

In order to predict the phase-transition region, [7] introduces the notion of "constrainedness" of a class of problems, noted κ, and shows that when κ is close to 1, instances are critically constrained, and belong to the phase-transition region. For a class of random binary CSPs $\langle n, m, p_1, p_2 \rangle$, [7] defines this constrainedness by $\kappa = \frac{n-1}{2} p_1 \log_m(\frac{1}{1-p_2})$.

One might think that phase-transitions only concern complete approaches, as they are usually associated with transitions from solvable to unsolvable instances, and incomplete approaches cannot detect whether a problem is not solvable. However, different studies (e.g., [8, 9]) have shown that very similar phase-transition phenomena may also be observed with incomplete approaches.

3 Algorithms

3.1 Ant Colony Optimisation

The main idea of the Ant Colony Optimization (ACO) metaheuristic [10] is to model the problem as the search of a best path in a "construction graph" that represents the states of the problem. Artificial ants walk through this graph, looking for good paths. They communicate by laying pheromone trails on edges of the graph, and they choose their path with respect to probabilities that depend on the amount of pheromone previously left.

The ACO algorithm considered in our comparative study is called Ant-solver, and it is described in [11]; we briefly recall below the main features of this algorithm.

Construction graph and pheromone trails: The construction graph associates a vertex with each variable/value pair $\langle X_i, v \rangle$ such that $X_i \in X$ and $v \in D(X_i)$. There is a non oriented edge between any pair of vertices corresponding to two different variables. Ants lay pheromone trails on edges of the construction graph. Intuitively, the amount of pheromone laying on an edge $(\langle X_i, v \rangle, \langle X_j, w \rangle)$ represents the learned desirability of assigning simultaneously value v to variable X_i and value w to variable X_j. As proposed in [12], pheromone trails are bounded between τ_{min} and τ_{max}, and they are initialized at τ_{max}.

Construction of an assignment by an ant: At each cycle, each ant constructs an assignment, starting from an empty assignment $\mathcal{A} = \emptyset$, by iteratively adding labels to $\mathcal{A}$ until $\mathcal{A}$ is complete. At each step, to select a label, the ant first chooses a variable $X_j \in X$ that is not yet assigned in $\mathcal{A}$. This choice is performed with

respect to the smallest-domain ordering heuristic, i.e., the ant selects a variable that has the smallest number of consistent values with respect to the partial assignment $\mathcal{A}$ under construction. Then, the ant chooses a value $v \in D(X_j)$ to be assigned to X_j. The choice of v is done with respect to a probability which depends on two factors, respectively weighted by two parameters α and β: a pheromone factor —which corresponds to the sum of all pheromone trails laid on all edges between $\langle X_j, v\rangle$ and the labels in $\mathcal{A}$— and a heuristic factor — which is inversely proportional to the number of new violated constraints when assigning value v to variable X_j.

Local improvement of assignments: Once a complete assignment has been constructed by an ant, it is improved by performing some local search, i.e., by iteratively changing some variable-value assignments. Different heuristics can be used to choose the variable to be repaired and the new value to be assigned to this variable. For all experiments reported below, we have used the min-conflict heuristics [13], i.e., we randomly select a variable involved in some violated constraint, and then we assign this variable with the value that minimizes the number of constraint violations.

Pheromone trails update: Once every ant has constructed an assignment, and improved it by local search, the amount of pheromone laying on each edge is updated according to the ACO metaheuristic, i.e., all pheromone trails are uniformly decreased —in order to simulate some kind of evaporation that allows ants to progressively forget worse paths— and then pheromone is added on edges participating to the construction of the best assignment of the cycle —in order to further attract ants towards the corresponding area of the search space.

3.2 Evolutionary Computation

In the most basic form, an evolutionary algorithm that solves constraint satisfaction problems uses a population of candidate solutions, i.e., complete assignments. The fitness is equal to the number of violated constraints, which needs to be minimised to zero. Such a basic approach does not yield a powerful constraint solver. To improve effectiveness and efficiency, many additional mechanisms such as heuristic operators, repair mechanisms, adaptive schemes, and different representations exist. Craenen et al. [3] have performed a large scale empirical comparison on 11 evolutionary algorithms that employ a variety of mechanisms. They conclude that the three best algorithms, Heuristics GA version 3, Stepwise Adaptation of Weights and the Glass-Box approach, have performances that are statistically not significantly different. However, all three are significantly better than the other 8 algorithms. This conclusion differs from [2] because the newer study considers even harder and larger problem instances.

We have selected the Glass-Box approach in our study and we use the same implementation as in [3], which is called JavaEa2 [14]. It can be downloaded for free and used under the GNU General Public License [15]. We provide a short overview of the algorithm to provide insight into its concept, for a more detailed description of the algorithm and all its parameters we refer to [16, 3].

The Glass-Box approach is proposed by Marchiori [16]. By rewriting all constraints into one form the algorithm can then process these constraints one at a time. For each constraint that is violated the corresponding candidate solution is repaired with a local search such that this constraint no longer is violated. However, previously repaired constraints are not revisited, thus a repaired candidate solutions not necessarily yields a feasible solution. The evolutionary algorithm keeps the local search going by providing new candidate solutions. A heuristic is added to the repair mechanism by letting it start with the constraint involved in the most violations.

3.3 Constraint Programming

Constraint programming dates back to 1965 when Chronological Backtracking was proposed [17]. Since then many variants have been proposed that rely on the basic principle of using recursion to move through the search space in an exhaustive way. The main interest in creating more variants is to speed up the process by making smarter moves and decisions. These involve a better order in which the variables and their domains are evaluated, as well as providing mechanisms that make it possible to take larger leaps through the search tree.

Here we use a constraint programming algorithm that uses *forward checking* (FC) of Haralick and Elliot [18] for its constraint propagation, which means that the algorithm first assigns a value to a variable and then checks all unassigned variables to see if a valid solution is still possible. The goal of forward checking is to prune the search tree by detecting inconsistencies as soon as possible. To improve upon the speed, *conflict-directed backjumping* (CBJ) [19] was added to it by Prosser [20] to form FC-CBJ. The speed improvement comes from the ability to make larger jumps backward when the algorithm gets into a state where it needs to backtrack to a previous set of assignments by jumping back over variables of which the algorithm knows that these will provide no solutions.

4 Experiments

This section reports experimental results obtained by the three algorithms Antsolver, Glass-Box, and FC-CBJ, in order to compare their effectiveness and efficiency.

To determine the effectiveness of the algorithms, we compare their success ratio, i.e., the percentage of runs that have succeeded in finding a solution. Note that, without any time limit, complete approaches always succeed in finding a solution as we only consider solvable instances.

To determine the efficiency of the algorithms, we compare the number of conflict checks they have performed. A *conflict check* is defined as querying if the assignment of two variables is allowed. Thereby we can characterize the speed of the algorithms independently from implementation and hardware issues as all algorithms have to perform conflict checks, which take the most time in execution.

Finally, the resampling ratio is used to get insight into how efficient algorithms are in sampling the search space. If we define S as the set of unique candidate solutions generated by an algorithm over a whole run and *evals* as the total number of generated candidate solutions then the resampling ratio is defined as $\frac{evals-|S|}{evals}$. Low values correspond with an efficient search, i.e., not many duplicate candidate solutions are generated. Note that complete approaches never generate twice a same candidate solution, so that their resampling ratio is always zero.

4.1 Experimental setup

The Ant-solver algorithm has been run with the following parameters setting: $\alpha = 2, \beta = 10, \rho = 0.99, \tau_{min} = 0.01, \tau_{max} = 4$, as suggested in [11]. The number of ants has been set to 15 and the maximum number of cycles to 2 000, so that the number of generated candidate solutions is limited to 30 000.

The Glass-Box algorithm has been run with a population size of 10, a generational model using linear ranking selection with a bias of 1.5 to determine the set of parents. The mutation rate was set to 0.1. The evolutionary algorithm terminates when either a solution is found or the maximum of 100 000 evaluations is reached.

For both Ant-solver and Glass-Box, we have performed 10 independent runs for each instance, and we report average results over these 10 runs.

4.2 Test suites

We compare the three algorithms on two test suites of instances, all generated according to Model E.

Test suite 1 is used to study what happens when moving towards the phase transition region —where the most difficult instances are supposed to be. It contains 250 instances, all of them having 20 variables and 20 values in each variable's domain, i.e., $n = m = 20$, and a density of constraints p_1 equals to 1, i.e., there is a constraint between any pair of different variables. These 250 instances are grouped into 10 sets of 25 problem instances per value of the tightness parameter p_2, ranging between 0.21 and 0.28, as the phase transition region occurs when $p_2 = 0.266$ (which corresponds with $p_e = 0.31$ when using Model E).

Test suite 2 is used to study the scale-up properties of the algorithms within the phase transition region. It is composed of 250 instances, grouped into 10 sets of 25 instances per number of variables n, ranging from 15 to 60. For all these instances, the domain size m of each variable is kept constant at 5. To make sure the problem instances have similar positions with respect to the phase-transition region, so that the difficulty of instances only depends on the number of variables, we keep the constrainedness $\kappa = 0.92$ on average (with a standard deviation of 0.019) by setting appropriately the order parameter p_e.

4.3 Moving towards the phase-transition region

Table 1 reports results obtained by each algorithm for each group of 25 instances of test suite 1. This table clearly shows that for the Glass-Box approach, one of the best known evolutionary algorithms for solving binary CSPs, the success ratio quickly drops from one to almost zero when moving towards the phase transition. Set against this result, Ant-solver seems a much better competition for the complete method FC-CBJ, being able to find a solution for nearly all the runs.

Table 1. Results of Glass-Box (GB), Ant-solver (AS), and FC-CBJ on the 250 $\langle 20, 20, 1, p_2 \rangle$ instances of test suite 1, grouped into 10 sets of 25 instances per value of p_2 ranging between 0.213 and 0.280.

p_2	0.213	0.221	0.228	0.236	0.244	0.251	0.259	0.266	0.273	0.280
	Success ratio									
GB	1.00	0.97	0.86	0.65	0.36	0.17	0.10	0.02	0.03	0.02
AS	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.96	0.98	1.00
FC-CBJ	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	Resampling ratio									
GB	0.10	0.23	0.40	0.60	0.78	0.87	0.90	0.94	0.93	0.93
AS	0.00	0.00	0.00	0.00	0.00	0.00	0.01	0.14	0.13	0.08
FC-CBJ	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	Average number of conflict checks									
GB	2.4e6	1.5e7	5.3e7	1.2e8	2.0e8	2.6e8	3.1e8	3.6e8	3.9e8	4.3e8
AS	2.5e5	4.4e5	1.0e6	2.5e6	6.3e6	1.8e7	5.4e7	1.2e8	1.1e8	9.0e7
FC-CBJ	5.7e4	1.2e5	2.1e5	7.0e5	1.7e6	5.2e6	2.0e7	4.1e7	4.1e7	2.6e7
	Standard deviation of the number of conflict checks									
GB	1.4e7	5.3e7	1.0e8	1.3e8	1.3e8	1.1e8	1.1e8	8.0e7	9.3e7	9.8e7
AS	2.4e5	4.7e5	9.6e5	2.4e6	6.6e6	1.8e7	5.2e7	2.8e8	2.0e8	1.1e8
FC-CBJ	5.3e4	1.5e5	2.9e5	7.5e5	1.4e6	4.2e6	2.9e7	3.1e7	2.4e7	1.5e7
	Constraint checks ratio of GB and AS w.r.t. FC-CBJ									
GB/FC-CBJ	42.11	125.00	252.38	171.43	117.65	50.00	15.50	8.78	9.51	16.54
AS/FC-CBJ	4.39	3.67	4.76	3.57	3.71	3.46	2.70	2.93	2.68	3.46

The resampling ratio reveals what happens. As the success ratio of Glass-Box drops, its resampling ratio keeps increasing, even to levels of over 90%, indicating that Glass-Box does not diversify enough its search. For Ant-solver we only observe a slight increase of the resampling ratio, up to 14% around the most difficult point at $p_2 = 0.266$, which corresponds with a slight drop in its success ratio.

However, when we compare the efficiency of the three algorithms by means of the number of conflict checks, we notice that although Ant-solver is faster than Glass-Box, it is slower than FC-CBJ. The last two lines of table 1 display the ratio

of the number of conflict checks performed by Glass-Box and Ant-solver with respect to FC-CBJ, and show that Glass-Box is from 8 to 250 times as slow as FC-CBJ whereas Ant-solver is from two to five times as slow than FC-CBJ. Note that for Ant-solver this ratio is rather constant: for Ant-solver and FC-CBJ, the number of conflict checks increases in a very similar way when getting closer to the phase transition region.

4.4 Varying the number of variables

Table 2 reports results obtained by the three algorithms for each group of 25 instances of test suite 2. In this table, one can first note that the effectiveness of the Glass-Box approach decreases when increasing the number of variables and although the rise in efficiency seems to slow down, this corresponds with success ratios less than 50%. The resampling ratio shows inverse behaviour with the success ratio. On the contrary, Ant-solver always finds a solution, while its resampling ratio is always less than 3% for each setting. The search of Ant-solver is thus extremely efficient for this scale-up experiment.

Table 2. Results of Glass-Box (GB), Ant-solver (AS), and FC-CBJ on the 250 $\langle n, 5, p_1, p_2 \rangle$ instances of test suite 2, grouped into 10 sets of 25 instances per value of n ranging between 15 and 60.

n	15	20	25	30	35	40	45	50	55	60
p_1	0.99	0.98	0.95	0.92	0.89	0.85	0.81	0.78	0.74	0.72
p_2	0.19	0.15	0.12	0.10	0.093	0.086	0.080	0.075	0.071	0.069
	Success ratio									
GB	0.86	0.70	0.58	0.57	0.38	0.28	0.23	0.20	0.11	0.14
AS	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
FC-CBJ	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	Resampling ratio									
GB	0.32	0.47	0.51	0.51	0.58	0.64	0.65	0.63	0.62	0.61
AS	0.021	0.014	0.008	0.003	0.002	0.001	0.001	0.000	0.000	0.000
FC-CBJ	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	Average number of conflict checks									
GB	1.1e7	3.6e7	7.1e7	1.0e8	1.9e8	2.7e8	3.7e8	4.4e8	6.0e8	7.0e8
AS	6.0e4	2.4e5	7.0e5	1.5e6	4.7e6	8.4e6	1.6e7	4.0e7	6.5e7	1.0e8
FC-CBJ	5.1e3	3.9e4	2.0e5	8.9e5	4.7e6	3.2e7	7.8e7	5.3e8	3.0e9	9.0e9
	Standard deviation of the number of conflict checks									
GB	2.3e7	4.8e7	7.7e7	1.1e8	1.4e8	1.6e8	1.8e8	2.1e8	2.0e8	2.7e8
AS	6.8e4	2.8e5	8.5e5	1.8e6	6.7e6	9.3e6	2.0e7	4.7e7	6.2e7	7.6e7
FC-CBJ	4.4e3	3.3e4	1.8e5	7.2e5	3.4e6	4.0e7	6.9e7	4.6e8	3.2e9	7.6e9
	Conflict checks ratio of GB and AS w.r.t. FC-CBJ									
GB/FC-CBJ	2156.86	923.08	355.00	112.36	40.43	8.44	4.74	0.83	0.20	0.08
AS/FC-CBJ	11.76	6.15	3.50	1.69	1.00	0.26	0.21	0.08	0.02	0.01

Regarding efficiency, we note that FC-CBJ starts out as the fastest algorithm at first, but at 35 variables it loses this position to Ant-solver. This difference in efficiency is clearly visible in the last line of Table 2: for instances with 15 variables, FC-CBJ is 12 times as fast than Ant-solver, whereas for instances with 60 variables, it is 100 times as slow.

Actually, the number of conflict checks performed by Ant-solver for computing one candidate solution is in $\mathcal{O}(n^2m)$ (where n is the number of variables and m the size of the domain of each variable), and the number of computed candidate solutions is always bounded to 30 000. Hence, the complexity of Ant-solver grows in a quadratic way with respect to the number of variables, and experimental results of Table 2 actually confirm this. On the contrary, the theoretical complexity of tree-search approaches such as FC-CBJ grows exponentially with respect to the number of variables. Even though FC-CBJ uses elaborate backtracking techniques, i.e., trying to jump over unsuccessful branches of the search tree, experimental results show us that the number of conflict checks performed by FC-CBJ still increases exponentially.

5 Conclusions

Experiments reported in this paper showed us that, considering effectiveness, ant colony optimisation is a worthy competitor to constraint programming, as Ant-solver almost always finds a solution, whereas evolutionary computation more often fails in finding solutions when getting closer to the phase transition region, or when increasing the size of the instances.

Scale-up experiments showed us that, on rather small instances, the constraint programming approach FC-CBJ is much faster than the two considered incomplete approaches. However, run times of FC-CBJ grow exponentially when increasing the number of variables, so that on larger instances with more than 35 variables, its efficiency becomes significantly lower than Ant-solver and Glass-Box.

The resampling ratio is a good indication of the problems that occur during the search in both the ant colony optimisation and the evolutionary algorithms. Whenever an algorithm's resampling increases this corresponds to a decrease in effectiveness. At the same time the efficiency will also drop.

Future research will incorporate experiments on over constrained instances, the goal of which is to find an assignment that maximises the number of satisfied constraints.

Acknowledgements

We thank Bart Craenen for developing the JavaEa2 platform and making it freely available to the community.

References

1. Tsang, E.: Foundations of Constraint Satisfaction. Academic Press (1993)
2. van Hemert, J.: Comparing classical methods for solving binary constraint satisfaction problems with state of the art evolutionary computation. In Cagnoni, S., Gottlieb, J., Hart, E., Middendorf, M., Raidl, G., eds.: Applications of Evolutionary Computing. Number 2279 in Springer Lecture Notes on Computer Science, Springer-Verlag, Berlin (2002) 81–90
3. Craenen, B., Eiben, A., van Hemert, J.: Comparing evolutionary algorithms on binary constraint satisfaction problems. IEEE Transactions on Evolutionary Computation **7** (2003) 424–444
4. van Hemert, J., Bäck, T.: Measuring the searched space to guide efficiency: The principle and evidence on constraint satisfaction. In Merelo, J., Panagiotis, A., Beyer, H.G., Fernández-Villacañas, J.L., Schwefel, H.P., eds.: Parallel Problem Solving from Nature — PPSN VII. Number 2439 in Springer Lecture Notes on Computer Science, Springer-Verlag, Berlin (2002) 23–32
5. Achlioptas, D., Kirousis, L., Kranakis, E., Krizanc, D., Molloy, M., Stamatiou, Y.: Random constraint satisfaction: A more accurate picture. Constraints **4** (2001) 329–344
6. Cheeseman, P., Kenefsky, B., Taylor, W.M.: Where the really hard problems are. In: Proceedings of the IJCAI'91. (1991) 331–337
7. Gent, I., MacIntyre, E., Prosser, P., Walsh, T.: The constrainedness of search. In: Proceedings of AAAI-96, AAAI Press, Menlo Park, California. (1996)
8. Davenport, A.: A comparison of complete and incomplete algorithms in the easy and hard regions. In: Proceedings of CP'95 workshop on Studying and Solving Really Hard Problems. (1995) 43–51
9. Clark, D., Frank, J., Gent, I., MacIntyre, E., Tomv, N., Walsh, T.: Local search and the number of solutions. In: Proceedings of CP'96, LNCS 1118, Springer Verlag, Berlin, Germany. (1996) 119–133
10. Dorigo, M., Caro, G.D., Gambardella, L.M.: Ant algorithms for discrete optimization. Artificial Life **5** (1999) 137–172
11. Solnon, C.: Ants can solve constraint satisfaction problems. IEEE Transactions on Evolutionary Computation **6** (2002) 347–357
12. Stützle, T., Hoos, H.: $\mathcal{MAX} - \mathcal{MIN}$ Ant System. Journal of Future Generation Computer Systems **16** (2000) 889–914
13. Minton, S., Johnston, M., Philips, A., Laird, P.: Minimizing conflicts: a heuristic repair method for constraint satistaction and scheduling problems. Artificial Intelligence **58** (1992) 161–205
14. Craenen, B.: JaveEa2: an evolutionary algorithm experimentation platform for constraint satisfaction in Java (Version 1.0.1) `http://www.xs4all.nl/~bcraenen/JavaEa2/download.html`.
15. Foundation, F.S.: The GNU general public license (Version 2, June 1991) `http://www.gnu.org/licenses/gpl.txt`.
16. Marchiori, E.: Combining constraint processing and genetic algorithms for constraint satisfaction problems. In Bäck, T., ed.: Proceedings of the 7th International Conference on Genetic Algorithms, Morgan Kaufmann (1997) 330–337
17. Golomb, S., Baumert, L.: Backtrack programming. ACM **12** (1965) 516–524
18. Haralick, R., Elliot, G.: Increasing tree search efficiency for constraint-satisfaction problems. Artificial Intelligence **14** (1980) 263–313
19. Dechter, R.: Enhancement schemes for constraint processing: Backjumping, learning, and cutset decomposition. Artificial Intelligence **41** (1990) 273–312
20. Prosser, P.: Hybrid algorithms for the constraint satisfaction problem. Computational Intelligence **9** (1993) 268–299