Constraint Satisfaction Problems and Evolutionary Computation: A Reality Check

J.I. van Hemert
<jvhemert@cs.leidenuniv.nl>

Leiden University, P.O.Box 9512, 2300 RA, Leiden, The Netherlands

Abstract

Constraint satisfaction has been the subject of many studies. Different areas of research have tried to solve all kind of constraint problems. Here we will look at a general model for constraint satisfaction problems in the form of binary constraint satisfaction. The problems generated from this model are studied in the research area of constraint programming and in the research area of evolutionary computation. This paper provides an empirical comparison of two techniques from each area. Basically, this is a check on how well both areas are doing. It turns out that, although evolutionary algorithms are doing well, classic approaches are still more successful.

1 Introduction

A *constraint satisfaction problem* (CSP) is a finite set of variables all of which have a corresponding finite domain of values. Besides variables, a set of constraints exist that restrict certain simultaneous value assignments to occur. The solution to a CSP is a set of variables that all have a value from their domain assigned without violating any of the constraints.

Constraint satisfaction has been a topic of research in many different forms. Lately, science has been concentrating on a more abstract form of constraint satisfaction in the effigy of binary constraint satisfaction problems. This general model resulted in a race for the fastest algorithm to solve instances. Just as particular constraint satisfaction problems, such as k-graph colouring and 3-SAT, binary CSPs have a characteristic phase-transition. The instances that are to be found in this phase-transition are among the hardest to solve. Thereby, making them highly suitable for experimental research.

In early times the backtracking kind of algorithms were used for tackling CSPs. One of the main advantages of these *classic algorithms* is that they are sound and complete [12]. Unfortunately, it seems that such algorithms need enormous amounts of constraint checks to solve the instances inside of the phase-transition. To counter for this, many tricks are developed all of which keep the properties of soundness and completeness intact.

Compared with the classic algorithms the evolutionary computation technique is a newcomer in constraint satisfaction. Nevertheless, it has grown popular and the optimisation power of evolutionary algorithms is successfully used in the field

of constraint optimisation and constraint satisfaction. In the past years many different techniques have been invented to improve the speed and accuracy of evolutionary algorithms on solving CSPs. Especially the matter of accuracy had to be addressed as evolutionary algorithms are stochastic by nature, thus moving away from the notion of soundness.

In the next section we show more detail on binary constraint satisfaction problems. After that we present the two candidates of both research areas. Section 5 provides the experimental setup followed by results and conclusions in Section 6 and Section 7. The last section is reserved for ideas on future research.

2 Binary Constraint Satisfaction Problems

In general, a CSP has a set of constraints where each constraint can be over any of the variables. Here, we focus our study on binary CSPs. A model that only allows constraints over a maximum of two variables. Although, at first this seems a restriction Tsang has shown [12] that this is not the case by proving that any CSP can be rewritten into a binary CSP. Solving the general CSP corresponds to finding a solution to the binary form. Although Bacchus and van Beek [1] have shown 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 difficult to solve instances. Collections of these instances serve as a good test bed for experimental research on solving methods. The idea behind finding difficult instances is that these are most likely to be the instances that only have 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, the overall domain size, the density of the constraints and the average tightness of the constraints. Smith [10] estimated the number of solutions by using these parameters in a predictor. Thereby, conjecturing the presence of a mushy region where instances with only one solution would be found.

Theoretical and empirical research have provided better ways of estimating where hard instances occur, making way to more advanced methods for randomly generating instances of binary CSPs. As a consequence, this fires up the competition among those who search for the winning CSP solver.

3 In the first corner

The term *classical* is introduced in this paper to differentiate between algorithms that are based on techniques invented a while ago and those taken from evolutionary computation. The study on the classic algorithms is alive in conferences such as Constraint Programming. Before discussing the two classic algorithms, we describe two simple preprocessing algorithms that are also used. Algorithms in this section have been tested with an adapted version of *csplib*[1]

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

First an instance is checked for arc-consistency. This property states that for every variable x, for every value assignment $\langle x, a\rangle$ that satisfies constraints on x itself, there exists a value b such that $\langle y, b\rangle$ satisfies all constraints imposed on x and y. A variable's corresponding value is removed from the domain whenever a value assignment does not comply. Afterwards the arc-consistency algorithm returns whether or not the binary CSP is arc-consistent. If the arc-consistency check fails we mark this as an unsolvable CSP instance and stop. Second, just before starting the search algorithm, we sort the variables in ascending order. This ordering will be used to choose the next variable that will be assigned a value during the search process.

3.1 Chronological backtracking

Chronological backtracking (BT) is a simple algorithm dating back to 1965 [6]. It is easy to construct using recursion. We assume the reader has come across backtracking, therefore we skip the details. This algorithm is selected because it is easy to understand, and is still able to compete with algorithms in Section 4.

3.2 Forward checking with conflict-directed backjumping

The method *forward checking with conflict-directed backjumping* (FC-CBJ) is build up using two techniques. The first, forward checking originates from 1980 [7]. The second, constraint-directed backjumping was added in 1993 [9]. This combination is among the fastest in the library[1] used here.

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 its domain annihilated. 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 it's conflict set joined with the conflict set of the annihilated variable and pick the variable farthest away from the current variable. 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 In the second corner

Two evolutionary algorithms will represent the current state of evolutionary computation on solving binary CSPs. These were selected as the best out of all methods studied in [3] and [5]. Note that the EAs presented here all try to minimise the fitness function. A fitness of zero equals to finding a solution.

4.1 Microgenetic iterative descend method

The *microgenetic iterative descend* (MID) method is a rather technical evolutionary algorithm. It uses a small population of about 10 individuals. The results used in this paper are taken from a study by Eiben et al. [5] but the original idea and research is done by Dozier et al. [2, 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 [2, 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 build up during the search is used in the fitness function of the evolutionary algorithm. Equation 1 shows that the fitness function comprises of two parts. First, the sum of all violated constraints 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.

$$\text{fitness}(\textit{solution}) = \sum_{v \in \text{variables}} V(v) + \sum_{b \in \textit{breakouts}} B(b), \tag{1}$$

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.

4.2 Stepwise Adaptation of Weights

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

The SAW method works by adding 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 now and then 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 2 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 plain evolutionary algorithm, that is, without SAW.

$$\text{fitness}(\textit{solution}) = \sum_{c \in \textit{constraints}} w_c \cdot C(c) \tag{2}$$

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

5 Experiments

We randomly generate instances of binary CSPs by using model B [8]. Based on the parameter settings we generate a binary CSP with the given number of variables, all with the same given domain size. Then we use the density parameter to calculate the number of constraints as a portion of the maximum number of constraints possible. The constraints are distributed uniform randomly between the variables. The same method is applied to induce the tightness of each constraint.

The test set comprises of instances created with two parameters fixed and two parameters varied. The number of variables and the overall domain size is set fixed to fifteen. 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 instances. Because of the stochastic nature of evolutionary algorithms we let each of them do ten independent runs as in [3, 5].

We measure the percentage of solutions found and the average of constraint checks for each pair of density and tightness. Note that the classic algorithms are sound, therefore the percentage of solutions presented is the actual percentage of solutions in the test set. When an evolutionary algorithm reaches the same percentage 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.

6 Results

A quick look at the results in Table 1 shows three distinct "regions" of parameter settings. First, the upper left where the density and tightness are low reveals that all 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 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 instances. Furthermore, here the algorithms need significantly more constraint checks to determine the solution or to determine that no solution exists.

den.	alg.	tightness									
		0.1		0.3		0.5		0.7		0.9	
0.1	BT	1.00	110	1.00	114	1.00	130	1.00	584	*0.96*	*534*
	FC-CBJ	1.00	1529	1.00	1431	1.00	1346	1.00	1254	*0.96*	*1044*
	MID	1.00	10	1.00	40	1.00	210	1.00	870	*0.96*	*29230*
	SAW	1.00	10	1.00	10	1.00	20	1.00	90	*0.64*	*11590*
0.3	BT	1.00	117	1.00	167	1.00	9169	*0.68*	*150270*	0.00	42702
	FC-CBJ	1.00	1395	1.00	1085	1.00	869	*0.68*	*16750*	0.00	20008
	MID	1.00	93	1.00	1550	1.00	10013	*0.52*	*1004772*	0.00	3100000
	SAW	1.00	31	1.00	62	1.00	1116	*0.23*	*21281*	0.00	3100000
0.5	BT	1.00	131	1.00	4351	*1.00*	*103228*	0.00	22218	0.00	4392
	FC-CBJ	1.00	1285	1.00	854	*1.00*	*15444*	0.00	6813	0.00	5208
	MID	1.00	520	1.00	9204	*0.90*	*1393184*	0.00	5200000	0.00	5200000
	SAW	1.00	52	1.00	416	*0.74*	*557544*	0.00	5200000	0.00	5200000
0.7	BT	1.00	152	*1.00*	*12974*	0.00	194909	0.00	6250	0.00	4300
	FC-CBJ	1.00	1173	*1.00*	*1044*	0.00	41851	0.00	4619	0.00	3982
	MID	1.00	1460	*1.00*	*44092*	0.00	7300000	0.00	7300000	0.00	7300000
	SAW	1.00	73	*1.00*	*5329*	0.00	7300000	0.00	7300000	0.00	7300000
0.9	BT	1.00	209	*1.00*	*121187*	0.00	76826	0.00	3265	0.00	2349
	FC-CBJ	1.00	1097	*1.00*	*9454*	0.00	24563	0.00	3561	0.00	3412
	MID	1.00	3102	*1.00*	*764784*	0.00	9400000	0.00	9400000	0.00	9400000
	SAW	1.00	94	*1.00*	*361712*	0.00	9400000	0.00	9400000	0.00	9400000

Table 1: Success percentage (left) and average number of constraint checks needed (right) of two classic algorithms (BT and FC-CBJ) and two evolutionary algorithms (MID and SAW) on 25 different parameter sets of density (den.) and tightness where the number of variables and the overall domainsize is set to 15. Each result is averaged over 25 instances. In the case of the two evolutionary algorithms the number of solutions found does not imply the number of solutions possible because these algorithms are not sound. The mushy region is indicated using italics.

Looking at the lower right corner we see the large difference in the number of constraint checks for the classic algorithms and for the evolutionary algorithms. The explanation for this difference is twofold. First, the large numbers for evolutionary algorithms are due to the fact that evolutionary algorithms do, in general, not know when to quit. Second, the small numbers for the classic algorithms are caused by the 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 100 constraint checks for all, except one pair, where the density or tightness is 0.1. Again the explanation is twofold. Firstly, this version of SAW uses a population size of one, thus losing little at the first evaluation of the whole population. Secondly, it employs an order-based representation together with a greedy algorithm that is successful for easy problems, thus able to solve them in a few attempts.

When observing all the results we conclude that except for easy problems classic algorithms are in favour. Because of their speed and because of their ability to detect when a problem has no solution. When we focus on the classic algorithms we see that the obvious winner is FC-CBJ. Although it does not beat the other algorithms in the upper left and lower right regions it absolutely makes up for this small loss in the mushy region.

7 Conclusions

We have made a comparison of two classic algorithms and two evolutionary algorithms on a test set of randomly generated binary constraint satisfaction problems. This comparison clearly shows that evolutionary algorithms have to improve their speed by quite a large factor if they want to compete with an algorithm as simple as chronological backtracking.

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 small methods such as arc-consistency, are very efficient in pointing out the impossible. However grim the situation looks, evolutionary algorithms still have one advantage. In every stage the algorithm has a set of possible partial solutions which can be handed to the user when we prematurely stop the run.

8 Future research

Just as the rat race on the best CSP solver, the search continues for better ways to randomly create binary CSPs. These new ways should be used as soon as flaws are detected in older ways to make sure comparisons really are sensible.

The additional methods that are invented in the study of CSPs should not be over seen in evolutionary computation. For example, the arc-consistency check is an algorithm that may be employed before a search algorithm is started, weeding out most unsolvable problems that have a high density and high tightness.

An often employed strategy in evolutionary computation is not to fight the competition but to embrace it. By including methods or parts of them into the search process we may create a hybrid evolutionary algorithm. Results are reported that show this may boost the performance of evolutionary computation when solving specific CSPs such as graph bi-partitioning [11].

References

[1] Fahiem Bacchus and Peter 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] 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.

[3] B.G.W. Craenen, A.E. Eiben, and E. Marchiori. Solving constraint satisfaction problems with heuristic-based evolutionary algorithmes. In *Proceedings of the 2000 Congress on Evolutionary Computation*, pages 1569–1575, July 2000.

[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.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.

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

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

[8] E. M. Palmer. *Graphical Evolution.* John-Wiley & Sons, New York, 1985. *An introduction to the theory of random graphs — an important tool in modeling networks of interactions.*

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

[10] 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.

[11] A.E. Steenbeek, E. Marchiori, and A.E. Eiben. Finding balanced graph bipartitions using a hybrid genetic algorithm. In *Proceedings of the* IEEE *International Conference on Evolutionary Computation (*ICEC'98*)*, pages 90–95. IEEE press, 1998.

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