The Log-Support Encoding of CSP into SAT

Marco Gavanelli

Dept. of Engineering, Ferrara University
http://www.ing.unife.it/docenti/MarcoGavanelli/

Abstract. Various encodings have been proposed to convert Constraint Satisfaction Problems (CSP) into Boolean Satisfiability problems (SAT). Some of them use a logical variable for each element in each domain: among these very successful are the *direct* and the *support* encodings.

Other methods, such as the *log*-encoding, use a logarithmic number of logical variables to encode domains. However, they lack the propagation power of the direct and support encodings, so many SAT solvers perform poorly on log-encoded CSPs.

In this paper, we propose a new encoding, called *log-support*, that combines the log and support encodings. It has a logarithmic number of variables, and uses support clauses to improve propagation. We also extend the encoding using a Gray code. We provide experimental results on Job-Shop scheduling and randomly-generated problems.

1 Introduction

One methodology for solving Constraint Satisfaction Problems (CSP) relies on the conversion into boolean satisfiability (SAT) problems. The advantage is the wide availability of free, efficient, SAT solvers, and the possibility to exploit advances in SAT solvers without reimplementing them in CP. SAT solvers have reached significant levels of efficiency, and new solvers are proposed, tested and compared every year in annual competitions [2,20]. There are both complete SAT solvers, based on systematic search (typically, variants of the DPLL procedure [6]), and incomplete solvers, often based on local search.

Very popular encodings [23,18] assign a SAT variable to each element of a CSP domain, i.e., for each CSP variable i and each value v in its domain, there is a logical variable $x_{i,v}$ that is true iff i takes value v. The reason for such a representation is that it lets the SAT solver achieve *pruning*: if the SAT solver infers that $x_{i,v}$ is $false$, then the corresponding CSP variable i cannot take value v. The most popular CSP-SAT encoding is the *direct* [23]; DPLL applied to a SAT encoded CSP mimics the Forward Checking on the original CSP [11,23]. The *support encoding* [12] has the same representation of domains, but a different representation of constraints. Unit propagation (used in DPLL solvers) applied to the support-encoded CSP achieves the same pruning of arc-consistency on the original CSP. Stronger types of consistency are proven in [3,7].

On the other hand, using a SAT variable for each value in a domain generates a huge search space. Indeed, it lets the SAT solver perform powerful propagation, but at a cost: the search space is exponential in the number of SAT variables.

In logarithmic encodings, each domain is represented by $\lceil \log_2 d \rceil$ SAT variables [15,14,9,23,18,1]. Such encodings can be tailored for specific constraints, such as *not equal* [10]. In general, however, they lack the ability to remove single values from domains, which yields less powerful propagation when compared to CSP solvers. Constraints are usually represented as in the direct encoding.

In this paper, we propose a new encoding, called *Log-Support*, that uses support clauses in a logarithmic encoding. The codification of domains can be either the usual binary representation, or based on a Gray code, in order to maximise the propagation power of support clauses. We apply the new encodings on randomly generated problems and on benchmark job-shop scheduling problems, and compare the performances of the SAT solvers Chaff [17] and MiniSat [8] on the encoded problems.

2 Preliminaries and Notation

A *CSP* consists of a set of variables, ranging on domains, and subject to constraints. We focus on *binary CSPs*, where constraints involve at most two variables. We call n the number of variables, and d the maximum domain cardinality. The symbols i and j refer to variables, while v and w are domain values.

A *SAT problem* contains a formula built on a set of variables, which can take only values $true$ (or 1) and $false$ (or 0). We call them *logical variables* or *SAT variables* to distinguish them from CSP variables. The formula is often required to be in conjunctive normal form, i.e., a set of *clauses*, i.e., disjunctions of literals of the logical variables. A solution to a SAT problem is an assignment of values $true/false$ to the logical variables, such that all clauses are satisfied.

3 A Survey on Encodings

The Direct encoding. [23] uses a logical variable $x_{i,v}$ for each CSP variable i and domain value v. For each CSP variable i, a clause (called *at-least-one*) imposes that i takes at least one of the values in its domain: $x_{i,1} \vee x_{i,2} \vee \ldots \vee x_{i,d}$. *At-most-one* clauses forbid the variable i to take two values: $\forall j_1 \neq j_2$ we add $\neg x_{i,j_1} \vee \neg x_{i,j_2}$. Constraints are encoded with *conflict clauses*: for each pair of inconsistent assignments $i \leftarrow v$, $j \leftarrow w$ s.t. $(v,w) \notin c_{i,j}$, we have $\neg x_{i,v} \vee \neg x_{j,w}$.

For example, consider the CSP: $A \leq B$, with A and B ranging on $\{0,1,2\}$. The direct encoding produces the clauses:

at-least-one		$a_0 \vee a_1 \vee a_2$		$b_0 \vee b_1 \vee b_2$
at-most-one	$\neg a_0 \vee \neg a_1$	$\neg a_0 \vee \neg a_2$	$\neg b_0 \vee \neg b_1$	$\neg b_0 \vee \neg b_2$
		$\neg a_1 \vee \neg a_2$		$\neg b_1 \vee \neg b_2$
conflict	$\neg a_1 \vee \neg b_0$	$\neg a_2 \vee \neg b_0$	$\neg a_2 \vee \neg b_1$	

The Support Encoding. [16,12] represents domains as in the direct encoding, i.e., we have *at-least-one* and *at-most-one* clauses. Constraints are based on the notion of support. If an assignment $i \leftarrow v$ supports the assignments $j \leftarrow w_1$, $j \leftarrow w_2$, ..., $j \leftarrow w_k$, we impose that $x_{i,v} \rightarrow x_{j,w_1} \vee x_{j,w_2} \vee \ldots \vee x_{j,w_k}$ i.e., we impose a *support clause*: $\neg x_{i,v} \vee x_{j,w_1} \vee x_{j,w_2} \vee \ldots \vee x_{j,w_k}$.

The constraints of the previous example are represented as the support clauses: $\neg a_1 \vee b_1 \vee b_2$, $\neg b_0 \vee a_0$, $\neg a_2 \vee b_2$ and $\neg b_1 \vee a_0 \vee a_1$.

The Log Encoding. [15,23,10] uses $m = \lceil \log_2 d \rceil$ logical variables to represent domains: each of the 2^m combinations represents an assignment. For each CSP variable i we have logical variables x_i^b, where $x_i^b = 1$ iff bit b of the value assigned to i is 1. *At-least-one* and *at-most-one* clauses are not necessary; however, in case the cardinality of domains is not a power of two, we need to exclude the values in excess, with the so-called *prohibited-value clauses* [18] (although the number of these clauses can be reduced [10]). If v is not in the domain of i, and v is represented with the binary digits $\langle v_{m-1}, \dots, v_0 \rangle$, we impose $\neg \left(\bigwedge_{b=0}^{m-1} \neg(v_b \oplus x_i^b) \right)$ where $\oplus$ means exclusive-or. Intuitively, $\neg(s \oplus b)$ is the literal b if s is *true*, and $\neg b$ if s is false. We obtain the *prohibited-value* clause $\bigvee_{b=0}^{m-1} v_b \oplus x_i^b$.

Constraints can be encoded with conflict clauses. If two assignments $i \leftarrow v$, $j \leftarrow w$ are in conflict, and v_b and w_b are the binary representations of v and w, we impose a clause of length $2m$: $\left(\bigvee_{b=0}^{m-1} v_b \oplus x_i^b \right) \vee \left(\bigvee_{b=0}^{m-1} w_b \oplus x_j^b \right)$.

In the running example, we will have:

prohibited-value	$\neg a_1 \vee \neg a_0$	$\neg b_1 \vee \neg b_0$
conflict	$a_1 \vee \neg a_0 \vee b_1 \vee b_0$	$\neg a_1 \vee a_0 \vee b_1 \vee b_0$
	$\neg a_1 \vee a_0 \vee b_1 \vee \neg b_0$	

4 The Log-Support Encoding

In the log-encoding, conflict clauses consist of $2m$ literals; unluckily, the length of clauses typically influences negatively the performance of a SAT solver.

The DPLL applied to a support-encoded CSP performs a propagation equivalent to arc-consistency on the original CSP [12]. One could think of applying support clauses to log encodings; an intuitive formulation is the following. If an assignment $i \leftarrow v$ supports the assignments $j \leftarrow w_1, \dots, j \leftarrow w_k$, we could impose, as in the support encoding, that $v \rightarrow w_1 \vee \dots \vee w_k$, and then encode in binary form the values v and w_i. However, the binary form of a value is a conjunction, so the formula becomes $\left(\bigwedge_b \neg(v^b \oplus x_i^b)\right) \rightarrow \left(\bigwedge_b \neg(w_1^b \oplus x_j^b)\right) \vee \dots \vee \left(\bigwedge_b \neg(w_k^b \oplus x_j^b)\right)$ which, in conjunctive normal form, generates an exponential number of clauses[1].

We convert in clausal form only implications that have exactly one literal in the conclusion. Let us consider, as a first case, only the most significant bit. In our running example, the assignment $A \leftarrow 2$ supports only $B \leftarrow 2$. We can say that, whenever A takes value 2, the most significant bit of B must be 1: $a_1 \wedge \neg a_0 \rightarrow b_1$. We add the support clause $\neg a_1 \vee a_0 \vee b_1$, that is enough to rule out two conflicting assignments:

prohibited-values	$\neg a_1 \vee \neg a_0$	$\neg b_1 \vee \neg b_0$
support	$\neg a_1 \vee a_0 \vee b_1$	
conflict	$a_1 \vee \neg a_0 \vee b_1 \vee b_0$	

Note that this transformation is not always possible: we can substitute some of the conflict clauses with one support clause only if all the binary form of supported values agrees on the most significant bit.

[1] One could reduce the number of clauses by introducing new variables, as in [3].

Each support clause has length $m+1$ and removes $d/2$ conflict clauses (of length $2m$). This is a significant reduction of the number of conflict clauses when the assignments that satisfy the constraint are all grouped in the same half of the domain. This happens in many significant constraints (e.g, $>$, $\leq$, $=$).

To sum-up, this encoding has the same number ($n\lceil\log_2 d\rceil$) of logical variables required by the log-encoding, with a reduced number of conflict clauses (of length $2\lceil\log_2 d\rceil$), which are substituted by support clauses (of length $\lceil\log_2 d\rceil + 1$).

Improvements. The same scheme can be applied to the other direction (from B to A), and to other bits (not just to the most significant one). In the running example, we can add the support clauses $b_1 \vee b_0 \vee \neg a_0$, $b_1 \vee \neg b_0 \vee a_1$, $b_1 \vee \neg b_0 \vee \neg a_1$ and, in this case, remove all the conflict clauses.

Suppose that a value v in the domain of variable i conflicts with two consecutive values w and $w{+}1$ in the domain of j. Suppose that the binary representation of the numbers w and $w+1$ differs only for the least significant bit b_0. In this case, we can represent both the values w and $w+1$ using only the $m-1$ most significant bits, so we can impose one single conflict clause of length $2m-1$. This simple optimization can be considered as applying binary resolution [7] to the two conflict clauses, and can be extended to sets of consecutive conflicting values whose cardinality is a power of two.

4.1 Gray Code

The *Gray code* [13] uses a logarithmic number of bits, as the binary code; however, any two consecutive numbers differ only for one bit. So, by encoding the values in the CSP domains with a Gray code, all intervals of size 2 are representable, while in the classical binary code only half of them are representable. For instance, suppose that a CSP variable A has a domain represented in 4-bit binary code, and that during DPLL search its state is 001U, i.e., the first three bits have been assigned, while the last has not been assigned yet. We can interpret this situation as the current domain of A being $\{2, 3\}$. However, there is no combination that can represent the domain $\{3, 4\}$. In the 4-bit Gray code, $\{2, 3\}$ is represented by configuration 001U and $\{3, 4\}$ by 0U10.

With a Gray representation, the running example is encoded as follows:

prohibited-values	$\neg a_1 \vee a_0$	$\neg b_1 \vee b_0$
support	$\neg a_1 \vee a_0 \vee b_0$	$b_1 \vee b_0 \vee \neg a_0$
	$\neg a_1 \vee \neg a_0 \vee b_0$	$b_1 \vee b_0 \vee \neg a_1$
	$\neg a_1 \vee \neg a_0 \vee b_1$	$b_1 \vee \neg b_0 \vee \neg a_1$

By using a Gray code, the number of support clauses has increased from 4 to 6 (50%), while (in this case) no conflict clauses are necessary. The intuition is that a higher number of support clauses should allow for more powerful propagation, but in some cases it could also increase the size of the SAT problem. However, each support clause has one CSP value in the antecedent and one of the bits in the conclusion, so for each constraint there are at most $2d\lceil\log_2 d\rceil$ support clauses. The number of conflict clauses in the log-encoding cannot be higher than the number of pairs of elements in a domain, so d^2. Recall also that conflict clauses are longer than support clauses, so we can estimate the size of the (Gray) Log-Support encoding to be smaller than that of the log-encoding, when d is large.

5 Experimental Results

5.1 Randomly Generated Problems

The first set of experiments is based on randomly generated CSPs. A random CSP is often generated given four parameters [21]: the number n of variables, the size d of the domains, the probability p that there is a constraint on a given pair of variables, and the conditional probability q that a pair of assignments is consistent, given that there is a constraint linking the two variables.

In order to exploit the compact representation of log-encodings, we focussed on CSPs with a high number of domain values. In order to keep the running time within reasonable bounds, we had to keep small the number of CSP variables.

The Log-Support encoding was developed for constraints in which the set of satisfying assignments is connected, and we can easily foresee that a Gray code will have no impact on randomly generated constraint matrices. Thus, we used a different generation scheme, in which satisfying assignments have a high probability to be grouped in clusters. Note that also real-life constraints typically have their satisfying assignments grouped together, and not completely sparse.

For each constraint (selected with independent probability p) on variables A and B, we randomly selected a pair of values v and w respectively from the domains of A and B. The pair (v, w) works as an "attractor": the probability that a constraint is satisfied will be higher near (v, w) and smaller far from that point. Precisely, the probability that a pair of assignments (a, b) is satisfied is $q = 1-\alpha\sqrt{(a-v)^2+(b-w)^2}$, where α is a coefficient that normalises the value of q in the interval 0..1. A posteriori, we grouped the experiments with a same frequency of satisfied assignments, and plotted them in the graph of Figure 1.

These experiments were performed running zChaff 2004.5.13 [17] and MiniSat 1.14 [8] on a Pentium M715 processor 1.5GHz, with 512MB RAM. A memory limit was set at 150MB, and a timeout at 1000s. Each point is the geometric mean of at least 25 experiments, where the conditions of timeout or out of memory are represented by 1000s. Timing results include both the time spent for the encoding and for solving the problem; however, the encoding time was always negligible. Note that to perform the experiments we did not generate a DIMACS file, because the time for loading the DIMACS could have been large (see also the discussion in the next section).

From the graphs, we see that the log encoding is the slowest when the constraints are tight (q is small). This could be due to the fact that in the log-encoding we have limited propagation of constraints, which makes hard proving unsatisfiability. On the other hand, when the constraints are loose (q near 80-90%), the log encoding performs better than the direct and support encodings.

The support encoding is often the best option for MiniSat, while Gray was the best in the zChaff experiments. Moreover, the Log-Support/Gray encodings are

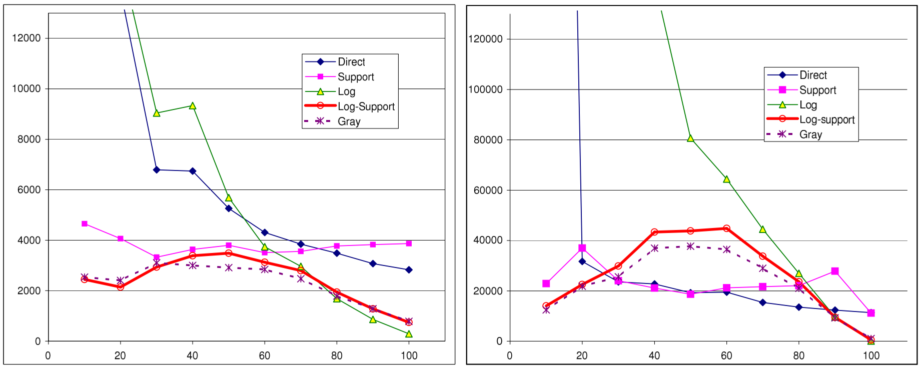

Fig. 1. Experiments on randomly-generated problems ($n = 7$, $d = 512$, $p = 30$ and q from 10 to 100). Times in ms. Left: Chaff, Right: MiniSat.

often competitive. For high values of q, the Log-Support/Gray encodings keep the same behaviour of the log-encoding. This is reasonable, because when q is high, few support clauses are inserted. On the other hand, when q is small, support clauses have a strong influence, and allow the SAT solver to detect infeasibility orders of magnitude faster than in the log-encoding. Both the Log-Support and the Gray encodings are typically faster than the direct encoding.

Finally, the Gray encoding is slightly faster than the Log-Support, probably due to the fact that more support clauses are present.

5.2 Job-Shop Scheduling Problems

We applied the encodings to the set of Job-Shop Scheduling Problems taken from the CSP competition 2006 [22] (and originally studied in [19]). These problems involve 50 CSP variables with variable domain sizes, from 114 to 159.

The results are in Figure 2: the plots show the number of problems that were solvable within a time limit given in abscissa (the higher the graph, the better).

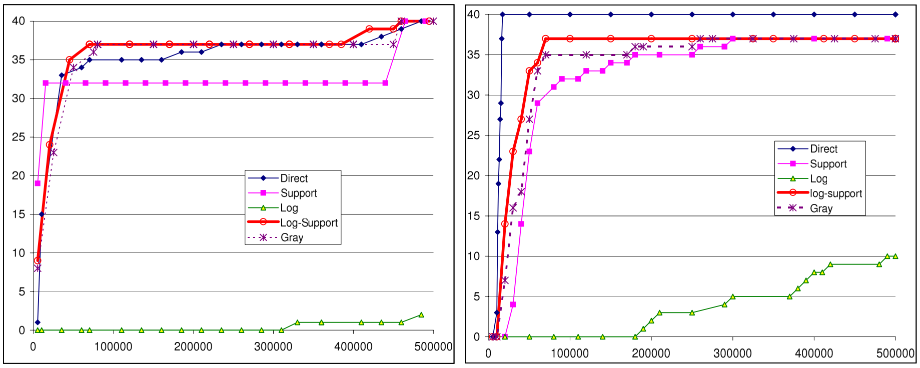

Fig. 2. Experiments on Job-Shop scheduling problems. Left: Chaff, right: MiniSAT.

The log encoding performed worst, and both Chaff and MiniSat were able to solve only a limited subset of the problems within 500s.

In the experiments performed with Chaff, the support encoding was able to solve some of the problems very quickly; however, given more time, the Log-Support was typically the best choice (it was able to solve more problems). In the experiments with MiniSat, the best encoding was the direct, possibly because of the special handling of binary clauses implemented in MiniSat. Notice that for both solvers the support encoding performed worse than the Log-Support and the Gray encodings. In these instances, the Gray encoding did not provide improvements with respect to the Log-Support.

Chaff required on average 65MB of RAM to solve a direct-encoded CSP, 56MB to solve a support-encoded CSP, and only 19MB to solve a problem encoded with Log-Support or Gray. The size of the generated SAT instance is also interesting. On average, a log-encoded CSP used 10^7 literals, from which we can estimate a DIMACS file of about 55MB. The Log-Support and Grey encodings needed on average $1.7 \cdot 10^6$ literals, with a DIMACS of about 9.5MB. The direct encoding used $2.3 \cdot 10^6$ literals (14MB), and the support $7 \cdot 10^6$ (45MB).

We can conclude that the Log-Support and Gray encodings are significant improvements with respect to the log encoding, both in terms of solution time and size of the generated SAT problem. The direct encoding is often faster than the Log-Support, but it requires more memory for Chaff to solve them, and the DIMACS file is much larger. Thus the Log-Support and Gray encodings could be interesting solution methods in cases with limited memory.

6 Conclusions and Future Work

We proposed two new encodings, called *Log-Support*, and *Gray*, for mapping CSPs into SAT. The Log-Support and Gray encodings use a logarithmic number of SAT variables for representing CSP domains, as in the well-known log-encoding. Experiments show that the new encodings outperform the traditional log-encoding, and is competitive with the direct and support encodings. Moreover, the size of the encoded SAT is typically a fraction of the size required by other encodings.

In future work, we plan to define a platform for defining CSPs, in the line of [5,4]. Such architecture could be populated with a variety of the many encodings proposed in recent years [1], and with the Log-Support/Gray encodings.

Other optimisations could be performed on the log encodings. We cite the *binary encoding* [9], that uses a logarithmic number of logical variables to encode domains, and it avoids imposing *prohibited value* clauses by encoding a domain value with a variable number of SAT variables. In future work, we plan to experiment with a variation of the Log-Support that exploits the same idea. Finally, we plan to experiment the various encodings with other solvers, in particular, local-search based.

Acknowledgements. This work has been partially supported by the MIUR PRIN 2005 project *Constraints and preferences as a unifying formalism for system analysis and solution of real-life problems.*

References

1. Ansótegui, C., Manyà, F.: Mapping problems with finite-domain variables to problems with boolean variables. In: Hoos, H.H., Mitchell, D.G. (eds.) SAT 2004. LNCS, vol. 3542. Springer, Heidelberg (2005)
2. Le Berre, D., Simon, L.: SAT competition (2005), `www.satcompetition.org/2005/`
3. Bessière, C., Herbrard, E., Walsh, T.: Local consistencies in SAT. In: Giunchiglia, E., Tacchella, A. (eds.) SAT 2003. LNCS, vol. 2919. Springer, Heidelberg (2004)
4. Cadoli, M., Mancini, T., Patrizi, F.: SAT as an effective solving technology for constraint problems. In: Esposito, F., Raś, Z.W., Malerba, D., Semeraro, G. (eds.) ISMIS 2006. LNCS (LNAI), vol. 4203, pp. 540–549. Springer, Heidelberg (2006)
5. Cadoli, M., Schaerf, A.: Compiling problem specifications into SAT. Artificial Intelligence 162(1-2), 89–120 (2005)
6. Davis, M., Logemann, G., Loveland, D.: A machine program for theorem-proving. Communications of the ACM 5, 394–397 (1962)
7. Dimopoulos, Y., Stergiou, K.: Propagation in CSP and SAT. In: Benhamou, F. (ed.) CP 2006. LNCS, vol. 4204, pp. 137–151. Springer, Heidelberg (2006)
8. Eén, N., Sörensson, N.: An extensible SAT-solver. In: Giunchiglia, E., Tacchella, A. (eds.) SAT 2003. LNCS, vol. 2919, pp. 502–518. Springer, Heidelberg (2004)
9. Frisch, A., Peugniez, T.: Solving non-boolean satisfiability problems with stochastic local search. In: Nebel, B. (ed.) IJCAI 2001, pp. 282–290 (2001)
10. Van Gelder, A.: Another look at graph coloring via propositional satisfiability. In: Computational Symposium on Graph Coloring and its Generalizations (2002)
11. Genisson, R., Jegou, P.: Davis and Putnam were already forward checking. In: Proc. of the 12th ECAI, pp. 180–184. Wiley, Chichester (1996)
12. Gent, I.P.: Arc consistency in SAT. In: van Harmelen, F. (ed.) ECAI'2002 (2002)
13. Gray, F.: Pulse code communication, U. S. Patent 2 632 058 (March 1953)
14. Hoos, H.: SAT-encodings, search space structure, and local search performance. In: Dean, T. (ed.) IJCAI 99, pp. 296–303. Morgan Kaufmann, San Francisco (1999)
15. Iwama, K., Miyazaki, S.: SAT-variable complexity of hard combinatorial problems. In: IFIP World Computer Congress, North-Holland, pp. 253–258 (1994)
16. Kasif, S.: On the parallel complexity of discrete relaxation in constraint satisfaction networks. Artificial Intelligence 45, 275–286 (1990)
17. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient SAT solver. In: Proc. of DAC 2001, pp. 530–535. ACM Press, New York (2001)
18. Prestwich, S.: Local search on SAT-encoded colouring problems. In: Giunchiglia, E., Tacchella, A. (eds.) SAT 2003. LNCS, vol. 2919. Springer, Heidelberg (2004)
19. Sadeh, N.M., Fox, M.S.: Variable and value ordering heuristics for the job shop scheduling constraint satisfaction problem. Artificial Intelligence 86(1), 1–41 (1996)
20. Sinz, C.: The SAT race (2006), `http://fmv.jku.at/sat-race-2006/`
21. Smith, B.M., Dyer, M.E.: Locating the phase transition in binary constraint satisfaction problems. Artificial Intelligence 81(1-2), 155–181 (1996)
22. van Dongen, M., Lecoutre, C., Roussel, O.: Second international competition of CSP and Max-CSP solvers (2006), `http://www.cril.univ-artois.fr/CPAI06/`
23. Walsh, T.: SAT v CSP. In: Dechter, R. (ed.) CP 2000. LNCS, vol. 1894, pp. 441–456. Springer, Heidelberg (2000)