Solving Over-Constrained Problems with SAT[*]

Student: Josep Argelich[1]
Supervisor: Felip Manyà[2]

[1] Universitat de Lleida, Jaume II, 69, E-25001 Lleida, Spain
[2] IIIA-CSIC, Campus UAB, 08193 Bellaterra, Spain

Abstract. We present a new generic problem solving approach for over-constrained problems based on Max-SAT. We first define a clausal form formalism that deals with blocks of clauses and distinguish between hard and soft blocks. We then present two Max-SAT-based solvers for our formalism and an experimental investigation.

1 Introduction

The SAT-based problem solving approach presents some limitations due to the fact that it only provides a solution when the input formula is satisfiable. Nevertheless, in many combinatorial problems, some potential solutions could be acceptable even when they violate some constraints. This is our motivation to extend the SAT formalism to solve over-constrained problems. In such problems, the goal is to find the *solution* that *best respects* the constraints of the problem.

In this paper we present a new generic problem solving approach for over-constrained problems based on Max-SAT, where we consider that all the constraints are *crisp* (i.e., they are either completely satisfied or completely violated), but constraints can be either *hard* (i.e., must be satisfied by any solution) or *soft* (i.e., can be violated by some solution). A solution *best respects* the constraints of the problem if it satisfies all the hard constraints and the maximum number of soft constraints. First, we define a clausal form formalism that deals with blocks of clauses instead of individual clauses, and that allows one to declare each block either as *hard* (i.e., must be satisfied by any solution) or *soft* (i.e., can be violated by some solution). We call *soft CNF formulas* to this new kind of formulas. Then, we present two Max-SAT solvers that find a truth assignment that satisfies all the hard blocks of clauses and the maximum number of soft blocks of clauses. Our solvers are branch and bound algorithms equipped with original lazy data structures; the first one, Soft-SAT-S, incorporates static variable selection heuristic while the second one, Soft-SAT-D, incorporates dynamic variable selection heuristic. Finally, we present an experimental investigation to assess the performance of our approach on graph coloring instances.

[*] Research partially supported by projects TIN2004-07933-C03-03 and TIC2003-00950 funded by the *Ministerio de Educación y Ciencia*. The second author is supported by a grant *Ramón y Cajal*.

2 Soft CNF Formulas

Definition 1. *A soft CNF formula is formed by a set of pairs* $(clause, label)$*, where clause is a Boolean clause and label is either* h_i *or* s_i *for some* $i \in \mathbb{N}$*. A hard block of a soft CNF formula is formed by all the pairs* $(clause, label)$ *with the same label* h_i*, and a soft block is formed by all the pairs* $(clause, label)$ *with the same label* s_i*.*

Definition 2. *A truth assignment satisfies a hard block of a soft CNF formula if it satisfies all the clauses of the block. A truth assignment satisfies a soft CNF formula* ϕ *if it satisfies all the hard blocks of* ϕ*. We say then that* ϕ *is satisfiable. A soft CNF formula* ϕ *is unsatisfiable if there is no truth assignment that satisfies all the the hard blocks of* ϕ*. A truth assignment satisfies a soft block if it satisfies all the clauses of the block. A truth assignment is a solution to a soft CNF formula* ϕ *if it satisfies all the hard blocks of* ϕ *and the maximum number of soft blocks. The Soft-SAT problem is the problem of finding a solution to a Soft CNF formula.*

The use of blocks is relevant for two reasons: (i) it provides to the user information about constraint violations in a more natural way; (ii) it allows us to get more propagation at certain nodes; for example, unit clauses in hard blocks can be propagated.

Example 1. We want to color a graph with two colors in such a way that the minimum number of adjacent vertices are colored with the same color. If we consider the graph with vertices $\{v_1, v_2, v_3\}$ and with edges $\{(v_1, v_2), (v_1, v_3), (v_2, v_3)\}$, that problem is encoded as a Soft-SAT instance as follows: (i) the set of propositional variables is $\{v_1^1, v_1^2, v_2^1, v_2^2, v_3^1, v_3^2\}$; the intended meaning of variable v_i^j is that vertex v_i is colored with color j; (ii) there is one hard block formed by the following at-least-one and at-most-one clauses: $(v_1^1 \vee v_1^2, h_1), (\neg v_1^1 \vee \neg v_1^2, h_1), (v_2^1 \vee v_2^2, h_1), (\neg v_2^1 \vee \neg v_2^2, h_1), (v_3^1 \vee v_3^2, h_1), (\neg v_3^1 \vee \neg v_3^2, h_1)$; and (iii) there is a soft block, with two clauses, for every edge: $(\neg v_1^1 \vee \neg v_2^1, s_1), (\neg v_1^2 \vee \neg v_2^2, s_1), (\neg v_1^1 \vee \neg v_3^1, s_2), (\neg v_1^2 \vee \neg v_3^2, s_2), (\neg v_2^1 \vee \neg v_3^1, s_3), (\neg v_2^2 \vee \neg v_3^2, s_3)$.

3 Description of solver Soft-SAT-S

The space of all possible assignments for a soft CNF formula ϕ can be represented as a search tree, where internal nodes represent partial assignments and leaf nodes represent complete assignments. Our algorithm for the Max-SAT problem of soft CNF formulas, called Soft-SAT-S, explores that search tree in a depth-first manner. At each node, the algorithm backtracks if the current partial assignment violates some clause of the hard blocks, and applies the one-literal rule[1] to the literals that occur in unit clauses of hard blocks. If the current partial assignment does not violate any clause of the hard blocks, the algorithm

[1] Given a literal $\neg p$ (p), the one-literal rule deletes all the clauses containing the literal $\neg p$ (p) and removes all the occurrences of the literal p $(\neg p)$

compares the number of soft blocks unsatisfied by the best complete assignment found so far, called upper bound (ub), with the number of soft blocks unsatisfied by the current partial assignment, called lower bound (lb). Obviously, if $ub \leq lb$, a better assignment cannot be found from this point in search. In that case, the algorithm prunes the subtree below the current node and backtracks to a higher level in the search tree. If $ub > lb$, it extends the current partial assignment by instantiating one more variable, say p, which leads to create two branches from the current branch: the left branch corresponds to instantiate p to false, and the right branch corresponds to instantiate p to true. In that case, the formula associated with the left (right) branch is obtained from the formula of the current node by applying the one-literal rule using the literal $\neg p$ (p). The value that ub takes after exploring the entire search tree is the minimum number of soft blocks that cannot be satisfied by a complete assignment.

In branch and bound Max-SAT algorithms, the lower bound is the sum of the number of unsatisfied clauses by the current partial assignment plus an underestimation of the number of clauses that will become unsatisfied if we extend the current partial assignment into a complete assignment, which is calculated taking into account the inconsistency counts of the variables not yet instantiated. The concept of inconsistency counts cannot be easily extended to soft blocks[2] and our lower bound is not so powerful as the lower bounds of current Max-SAT solvers. In Soft-SAT-S, the initial lower bound is obtained with a GSAT-like local search algorithm. For instances with CSP variables with domain size greater than 2, we defined a lower bound that incorporates an underestimation of the number of soft blocks that will become unsatisfied if we extend the current partial assignment into a complete assignment. Each CSP variable with a domain of size k is represented by a set of k Boolean variables $x_1, \ldots, x_k$ in a SAT encoding. The inconsistency count associated with a Boolean variable x_i ($1 \leq i \leq k$) is the number of soft blocks violated when x_i is set to true. The inconsistency count associated with a CSP variable X, which is encoded by the Boolean variables $x_1, \ldots, x_k$, is the minimum of the inconsistency counts of x_i ($1 \leq i \leq k$). As underestimation for the lower bound, we consider exactly one CSP variable for each soft block and take the sum of the inconsistency counts of such variables.

When branching is done, algorithms for Max-SAT apply the one-literal rule (simplifying with the branching literal) instead of applying unit propagation (i.e., the repeated application of the one-literal rule until a saturation state is reached) as in the Davis-Putnam-style solvers for SAT. If unit propagation is applied at each node, the algorithm can return a non-optimal solution. For example, if we apply unit propagation to $\{p, \neg q, \neg p \vee q, \neg p\}$ using the unit clause $\neg p$, we derive one empty clause while if we use the unit clause p, we derive two empty clauses. However, when the difference between the lower bound and the upper bound is one, unit propagation can be safely applied, because otherwise by fixing to false any literal of any unit clause we reach the upper bound. Soft-SAT-S performs unit propagation in that case too. Moreover, as pointed out before, Soft-SAT-S

[2] This is due to the fact that inconsistency counts deal with clauses but not with blocks of clauses.

applies the one-literal rule when a clause of a hard block becomes unit. This propagation, which leads to substantial performance improvements, cannot be safely applied in Max-SAT solvers, and is a key feature of our approach.

Our current version of Soft-SAT-S incorporates the following static variable selection heuristic: In SAT encodings that model CSP variables, each CSP variable with a domain of size k is represented by a set of k Boolean variables $x_1, \ldots, x_k$. We associate a weight to each one of these sets: the sum of the total number of occurrences of each variable of the set. We order the sets according to such weight. It instantiates, first and in lexicographical order, the Boolean variables of the set with the highest weight. Then, it instantiates, in lexicographical order, the Boolean variables of the set with the second highest weight, and so on. The idea behind this heuristic is to instantiate first the CSP variables that occur most often. This way, we emulate an n-ary CSP branching by means of a binary branching (i.e., we consider all the possible values of the CSP variable under consideration before instantiating another CSP variable).

The fact of using static variable selection heuristics allows us to implement extremely efficient data structures for representing and manipulating soft CNF formulas. Our data structures take into account the following fact: we are only interested in knowing when a clause has become unit or empty. Thus, if we have a clause with four variables, we do not perform any operation in that clause until three of the variables appearing in the clause have been instantiated; i.e., we delay the evaluation of a clause with k variables until $k-1$ variables have been instantiated. In our case, as we instantiate the variables using a static order, we do not have to evaluate a clause until the penultimate variable of the clause in the static order has been instantiated.

The data structures are defined as follows: For each clause we have a pointer to the penultimate variable of the clause in the static order, and the clauses of a soft CNF formula are ordered by that pointer. We also have a pointer to the last variable of the clause. When a variable p is fixed to true (false), only the clauses whose penultimate variable in the static order is $\neg p$ (p) are evaluated. This approach has two advantages: the cost of backtracking is constant (we do not have to undo pointers like in adjacency lists) and, at each step, we evaluate a minimum number of clauses.

4 Description of solver Soft-SAT-D

The second solver we have designed and implemented is Soft-SAT-D, which is like Soft-SAT-S except for the fact that its variable selection heuristic is dynamic. This fact, in turn, did not allow us to implement the data structures we have described in the previous section. The data structures implemented in Soft-SAT-D are the two-watched literal data structures of Chaff [4]. Our current version of Soft-SAT-D incorporates a dynamic variable selection heuristic: Is the dynamic version of the heuristic of Soft-SAT-S. We associate a weight to each set of free Boolean variables that encode a same CSP variable: the sum of the total number of occurrences of each variable of the set that has not been yet instantiated. We select the set with the highest weight and instantiate its variables in lexicographical order.

	Soft-SAT-S		Soft-SAT-D		Toolbar		PFC-MPRDAC	
$\langle n,k,c\rangle$	mean	median	mean	median	mean	median	mean	median
$\langle 15,15,8\rangle$	**56.35**	**5.54**	325.00	27.19	98.14	19.46	102.81	15.87
$\langle 15,15,10\rangle$	**56.23**	**0.03**	266.83	0.03	146.65	0.05	105.81	0.13
$\langle 16,14,6\rangle$	153.64	36.53	1068.80	243.54	**110.22**	**29.92**	186.40	64.43
$\langle 16,14,8\rangle$	**88.92**	**10.27**	464.33	33.93	144.15	23.02	157.84	20.75
$\langle 16,16,6\rangle$	162.34	101.10	1142.79	682.78	**110.50**	**63.65**	199.33	146.87
$\langle 16,16,8\rangle$	**48.63**	**12.68**	229.35	37.77	105.31	28.12	114.27	29.99

Table 1. Comparison between Soft-SAT-S, Soft-SAT-D, PFC-MPRDAC and Toolbar on randomly generated graph coloring instances. Experiments performed on a 1.6 Ghz AMD64-Opteron with 1 Gbyte RAM. Time in seconds.

5 Experimental investigation

We next report the experimental investigation we conducted to evaluate the performance of our problem solving approach. The solvers used were the following ones: Soft-SAT-S, Soft-SAT-D, PFC-MPRDAC[3] and Toolbar [3][4].

We used unsatisfiable graph coloring instances and the problem we solved was to find a coloring that minimizes the number of adjacent vertices with the same color. We used individual instances from the graph coloring symposium celebrated in CP-2002, and randomly generated instances using the generator of Culberson [1]. We use the generator with option IID (independent random edge assignment). The parameters of the generator are: number of vertices (n), optimum number of colors to get a valid coloring (k), and number of colors we use to color the graph (c).

In the first experiment we considered 6 sets of randomly generated instances, where each set had 100 instances. We solved the instances with Soft-SAT-S, Soft-SAT-D, PFC-MPRDAC and Toolbar. The results obtained are shown in Table 1: the first column displays the parameters given to the generator, and the rest of columns display the mean and median time needed to solve an instance of the set with each one of the solvers used. We repeated the previous experiments but using a representative sample of individual instances from the graph coloring symposium celebrated in CP-2002. The results obtained are shown in Table 2: the first column displays the name of the instance, the optimum number of colors to get a valid coloring (k), and the number of colors we used to color the graph

[3] This is a highly optimized solver from the Constraint Programming community for solving binary Max-CSP problems [2].

[4] http://carlit.toulouse.inra.fr/cgi-bin/awki.cgi/ToolBarIntro (version July, 2005)

$\langle$Instance, $k, c\rangle$	vc	Soft-SAT-S	Soft-SAT-D	Toolbar	PFC-MPRDAC
$\langle$myciel5.col, $6, 3\rangle$	16	11.04	46.39	**0.66**	12.11
$\langle$myciel5.col, $6, 4\rangle$	4	78.50	226.59	**6.28**	96.41
$\langle$myciel5.col, $6, 5\rangle$	1	3177.85	31.87	**26.02**	44.34
$\langle$GEOM30a.col, $6, 3\rangle$	11	9.31	27.22	**0.87**	14.33
$\langle$GEOM30a.col, $6, 4\rangle$	4	4.48	**2.35**	2.42	22.89
$\langle$GEOM30a.col, $6, 5\rangle$	1	0.49	**0.15**	0.17	0.18
$\langle$GEOM40.col, $6, 2\rangle$	22	3.89	20.58	**0.08**	4.42
$\langle$GEOM40.col, $6, 3\rangle$	7	**10.83**	30.63	25.20	770.46
$\langle$GEOM40.col, $6, 4\rangle$	3	95.18	**14.67**	1981.17	>7200.00
$\langle$GEOM40.col, $6, 5\rangle$	1	1.58	**0.51**	1186.22	1186.22
$\langle$queen5_5.col, $5, 3\rangle$	29	57.60	167.60	**9.22**	27.27
$\langle$queen5_5.col, $5, 4\rangle$	12	37.50	124.24	**13.18**	73.67

Table 2. Comparison between Soft-SAT-S, Soft-SAT-D, PFC-MPRDAC and Toolbar on individual graph coloring instances. Experiments performed on a 2GHz Pentium IV with 512 Mb of RAM. Time in seconds.

(c); the second column displays the number of violated constraints; and the rest of columns display the time needed to solve the instance with each one of the solvers used.

We observe that, in both cases, our approach is very competitive. For all the sets of randomly generated instances, Soft-SAT-S outperforms PFC-MPRDAC and outperforms Toolbar on 4 sets. For individual instances some times is better Soft-SAT and sometimes Toolbar. These results indicate that soft CNF formulas are a suitable formalism for representing and solving over-constrained problems with SAT technology.

References

1. J. Culberson. Graph coloring page: The flat graph generator. See http://web.cs.ualberta.ca/~ joe/Coloring/Generators/flat.html, 1995.
2. J. Larrosa. *Algorithms and Heuristics for Total and Partial Constraint Satisfaction.* PhD thesis, FIB, Universitat Politècnica de Catalunya, Barcelona, 1998.
3. J. Larrosa and T. Schiex. In the quest of the best form of local consistency for weighted CSP. In *Proceedings of the International Joint Conference on Artificial Intelligence, IJCAI'03, Acapulco, México*, pages 239–244, 2003.
4. M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient sat solver. In *39th Design Automation Conference*, 2001.