

Learning in a Compiler for SAT and MINSAT Algorithms – A Summary*

Anja Remshagen and Klaus Truemper

University of Texas at Dallas, Computer Science Program EC31, Box 830688, Richardson, TX
75083-0688, USA
remshage@utdallas.edu, klaus@utdallas.edu

Abstract. In many logic programming applications, minimum-cost solutions have to be determined for the minimization problem MINSAT, where costs are associated with the assignment of *True/False* values. Each instance is derived from a given logic formula in conjunctive normal form by fixing of some variables. Thus, the instances are closely related, and one should be able to learn how to solve them quickly. This paper describes a scheme for such learning that can be applied to classes of MINSAT as well as SAT. The learning process has been implemented and tested on a number of SAT and MINSAT cases.

1 Introduction

Learning has been widely used in solution algorithms for the satisfiability problem of propositional logic (SAT) and the constraint satisfaction problem (CSP). The basic idea is as follows. Whenever a solution algorithm concludes that values assigned to some of the variables cannot possibly be extended to a solution, then the algorithm analyzes which of the assigned values is needed for that conclusion, and then adds to the problem instance a lemma that rules out future assignments of such values. One may call this learning process *dynamic* if it is carried out while the solution algorithm processes a problem instance. Learning can also be employed during preprocessing of a problem instance. Such learning may be called *static*. The classification becomes useful when an algorithm is compiled for a problem class and later is repeatedly applied to the instances of that class. *Static learning* is then learning during the compilation of an algorithm for the class. In this paper, we describe static learning in such a compiler for the logic minimization problem MINSAT as well as for SAT and show first results for the implementation.

2 Definitions

An instance of SAT is a propositional logic formula S in conjunctive normal form (CNF). Thus, S is a conjunction of *clauses*. In turn, each clause is a disjunction of *literals*, which are instances of possibly negated variables. An assignment for a set of variables maps truth values *True/False* to each variable in the set. The SAT problem demands that one either determines S to be unsatisfiable – that is, there do not exist *True/False* for the variables so that S evaluates to *True* – or produces a satisfying solution. An instance of MINSAT consists of a CNF formula S and a

* This research was supported in part by ONR grant N00014-93-1-0096.

nonnegative rational vector c . For each variable x of S , the element $c(x)$ of c is the cost incurred when x takes on the value *True*. The MINSAT problem demands that one either determines S to be unsatisfiable or produces a satisfying solution whose total cost – that is, $\sum_{x \ni x = \text{True}} c(x)$ – is minimum. We represent a MINSAT instance by the pair (S, c) . A lemma obtained by learning is a CNF clause. The *length* of the lemma or of a CNF clause is the number of literals of the clause. Due to this definition, we can use qualitative terms such as *short* or *long* in connection with lemmas or clauses.

The typical class \mathcal{C} of SAT or MINSAT that is treated here consists of an instance S or a MINSAT instance (S, c) , plus all instances that may be derived from that instance by fixing some variables of S to *True/False* and deleting the clauses that become satisfied by these values. Classes \mathcal{C} arise from applications where one must decide whether some CNF clause is a logic consequence of the set of clauses of a given SAT instance S , or where one must find a least-cost satisfying solution for a MINSAT instance (S, c) when some variables take on specified *True/False* values. The applications of such classes \mathcal{C} often demand that the requested answer is supplied within a very short response time period. For some applications – for example, traffic control – the time period is prescribed and must not be exceeded regardless of circumstances.

We review prior related work on learning.

3 Prior Work

Early references are Dechter (1990) and Prosser (1993). They enhance backtracking search algorithms for CSP by a dynamic learning process as follows. Whenever an assignment of values to variables violates a constraint, the reason for the violation is determined and added to the CSP instance as a lemma which becomes a constraint of the CSP instance. The same ideas are used by effective SAT algorithms such as GRASP (Marques-Silva and Sakallah 1996), SATO3 (Zhang 1997), or relsat(4) (Bayardo and Schrag 1997). Since the required space for learned lemmas can be exponential, Marques-Silva and Sakallah (1996) and Zhang (1997) keep only clauses of bounded length. The SAT solver relsat(4) not only keeps short clauses, but also retains long clauses temporarily; see Bayardo and Schrag (1997) for details. Algorithm learn-SAT by Richards and Richards (2000) for CSP assigns values to the variables incrementally so that no constraint is violated. If such an assignment cannot be extended without violating a constraint, a lemma that invalidates the current partial assignment is added to the CSP instance, and learn-SAT tries to find another assignment. Van Gelder and Okushi (1999) are using lemmas to prune refutation trees in the SAT solver Modoc. Static learning is used in the SAT algorithm Satz (Li and Anbulagan 1997), where short clauses are computed by resolution before the solution process begins. In the preprocessing step of Marques-Silva (2000), lemmas of at most length 2 are inferred from small subsets of the CNF clauses with length 2 and 3 in the given SAT instance. The algorithm of Dransfield, Franco, Price, and Vanfleet (2000) employs both static and dynamic learning. The method does not just solve the satisfiability problem of CNF formulas but also decides the satisfiability of formulas that are a conjunction of general propositional formulas with bounded

number of variables. In this more general setting, learning has proved to be very effective.

Other compilation techniques that are applied to classes of SAT instances try to obtain computationally more attractive logic formulations that preserve equivalence; see, for example, del Val (1994). Kautz and Selman (1994) compute tractable formulations that approximate the original SAT instance. For further references on compilation techniques, see the survey of Cadoli and Donini (1997).

4 Learning Process

The learning process for classes of MINSAT is composed of two steps. The first step treats a class \mathcal{C} obtained from a MINSAT instance (S, c) as a class of SAT instances obtained by S . Thus, the first step of the learning process can be applied for classes of SAT as well. In the second step, the costs for the variables are considered. We first describe the solution algorithm for an instance S .

We derive a *subinstance* from S by deleting from the clauses of S all literals arising from a specified set of variables. The compiler carries out a number of preprocessing and decomposition steps that we may ignore here. For each subproblem obtained by decomposition, the compiler partitions the variables of S into two sets that induce two subinstances S_E and S_N . The partition is so done that the subinstance S_E has one of several well-known special properties – for example, has hidden Horn form or is a 2SAT instance – and, subject to that condition, has as many variables as possible. Each of the special properties is maintained under deletion of variables or clauses, and it permits linear-time solution of the satisfiability problem for any instance derived from S_E by deletion of variables and clauses. Let X_N be the set of variables of S_N .

The solution algorithm for S consists of two parts: an enumerative subroutine that chooses values for the variables of X_N , and the linear-time subroutine for S_E . Specifically, the enumerative subroutine implicitly tries out all possible *True/False* values for the variables of X_N , evaluates which clauses of S become satisfied by these values, and uses the linear-time subroutine to find a satisfying solution for the remaining clauses or to decide that no such solution exists. The growth of the search tree is controlled by a modified version of a rule described in Böhm (1996). Throughout the paper, we refer to the modified rule as *Böhm's Rule*. The rule aims at fixing variables in such a sequence that each branch of the search tree quickly reaches provable unsatisfiability. Böhm's rule achieves this goal very well by, roughly speaking, fixing variables that occur in a maximum number of the currently shortest clauses of S for which the corresponding clauses of S_E contain at most one literal.

The first step of the learning process determines lemmas of a SAT instance S . If S is unsatisfiable, then, mathematically speaking, only one lemma, which is the empty clause, needs to be learned; in applications, that situation signals a formulation error. So let us assume that S is found to be satisfiable. When the algorithm stops, the search tree has been pruned to a path Q whose tip node has led to a satisfying solution. Starting at the root node of Q , let us number the nodes $1, 2, \dots, m$, for some $m \geq 1$. Suppose at node i the variable x_i was fixed to *True/False* value α_i . At that node, one of two cases applies. Either the algorithm fixed x_i to α_i without

trying the opposite value $\neg\alpha_i$ first, or the algorithm first tried the opposite value $\neg\alpha_i$, discovered unsatisfiability, and then assigned α_i . The latter case implies that $x_1 = \alpha_1, x_2 = \alpha_2, \dots, x_{i-1} = \alpha_{i-1}, x_i = \neg\alpha_i$ produces unsatisfiability. Hence, we may add to S a lemma that rules out that assignment. For example, if $x_1 = \text{True}$, $x_2 = \text{False}$, and $x_3 = \text{True}$ produce unsatisfiability, then the lemma is $\neg x_1 \vee x_2 \vee \neg x_3$. At this point, we begin a time-consuming process that is acceptable for static learning but would not be reasonable for dynamic learning. That is, we sharpen the lemma by removing from the lemma the literals corresponding to x_1, x_2, \dots, x_{i-1} one at the time. For each such removal, we check whether the reduced lemma is still valid and add the literal again if this is not so. When that effort stops, we have a minimal lemma, that is, a lemma that becomes invalid if any literal is removed.

Up to this point, we have considered learning of lemmas that help the solution algorithm to solve S . We extend this now to an instance of \mathcal{C} that is derived from S by fixing some variables. Correspondingly, we start the enumerative search by first fixing these variables and then proceeding as before. Effectively, we begin the search tree with a path P representing the initial fixing instead of just with the root node. The algorithm either finds a satisfying solution, or it stops and declares S to be unsatisfiable. In the first case, we determine minimal lemmas and adjoin them to S . Due to the path P , a lemma added to S may involve variables of S_E and thus may destroy the special property of S_E . Nevertheless, one can prove that the partition of S into two subinstances effectively does not have to be revised.

We have completed the discussion of learning lemmas for SAT and turn to the second step. Here we learn cost-dependent lemmas for the MINSAT instance (S, c) and for all instances derived from (S, c) by fixing some variables to *True/False* values in all possible ways. The solution algorithm for MINSAT not only prunes unsatisfiable assignments as in the SAT case, but also eliminates assignments resulting in nonoptimal total costs. Learning is possible for both cases, as follows. At some point, the solution algorithm for MINSAT finds a fixing of variables that eventually turns out to be part of an optimal solution. Say, x_1, x_2, \dots, x_n fixed to $\alpha_1, \alpha_2, \dots, \alpha_n$ induce an optimal solution with total cost z_{\min} . When the algorithm terminates, we know the following for each $k \leq n$: The fixing of x_1, x_2, \dots, x_k to the values $\alpha_1, \alpha_2, \dots, \alpha_{k-1}, \neg\alpha_k$ results into unsatisfiability, or that fixing can be extended to a solution that at best has total cost $z_{\min}^k \geq z_{\min}$. The first case is treated exactly as before. That is, we define lemma $L = l_1 \vee l_2 \vee \dots \vee l_k$ where, for $j = 1, 2, \dots, k-1$, $l_j = x_j$ (resp. $l_j = \neg x_j$) if $\alpha_j = \text{False}$ (resp. $\alpha_j = \text{True}$) and where $l_k = x_k$ (resp. $l_k = \neg x_k$) if $\alpha_k = \text{True}$ (resp. $\alpha_k = \text{False}$). In the second case, we define the same lemma L and combine it with z_{\min}^k to the pair (L, z_{\min}^k) . In the solution algorithm, the pair (L, z_{\min}^k) becomes a valid clause and is activated if we have already a solution with total cost not exceeding z_{\min}^k .

We want to add minimal lemmas to S that improve the effectiveness of Böhm's Rule when that rule, unassisted by lemmas, would perform badly. Since Böhm's Rule selects variables based on short clauses, we discard all minimal lemmas of length greater than some constant. From our experiments, we determined that constant to be 3. That is, we only retain the minimal lemmas of length 1, 2, or 3. Moreover, we want to achieve improvement across the full range of instances of \mathcal{C} . How much learning might be required to reach that goal? We do not have a good answer for

that question. One can show that, if one could achieve that goal reliably by learning from a number of instances that is bounded by a polynomial in the size of S , then $\Pi_2^P = \Sigma_2^P$ for the polynomial hierarchy. For details of that hierarchy, see, for example, Chap. 17 of Papadimitriou (1994). This negative result makes it unlikely that we can learn enough from a polynomial subset of \mathcal{C} . On the other hand, we may be able to carry out such learning for specific classes \mathcal{C} . In this paper, we demonstrate experimentally that this is indeed possible for some nontrivial classes when instances of \mathcal{C} are selected for learning in the following manner. For $i = 1, 2, \dots$, let \mathcal{C}_i be the subset of \mathcal{C} where each instance is obtained by fixing i arbitrarily selected variables in S . Let q_i be the average time required to solve an instance of \mathcal{C}_i . Since the algorithm produced by the compiler solves instances of \mathcal{C} very rapidly if all or almost all variables of X_N have been fixed, the values q_i are small when i is close to or equal to $|X_N|$. Correspondingly, there is no need to learn lemmas from those instances. On the other hand, large q_i values point to sets \mathcal{C}_i with instances where learning of lemmas would be useful. Let k be the index of the largest estimate for q_i . By experimentation we found that significant learning took place when we used all \mathcal{C}_i with $i \leq k + 1$. In contrast, learning from any \mathcal{C}_i with $i > k + 1$ turned out to have little effect.

5 Computational Results

We have added the learning process for SAT and MINSAT classes to logic programming software (Leibniz System (2000)) that is based on Truemper (1998). All computational results were obtained on a Sun UltraSPARC III (333MHz) workstation.

We chose standard test problems that previously had proved to be difficult for the software. Since learning for any unsatisfiable instance produces the empty clause and thus results in a trivial instance, we selected classes \mathcal{C} derived from satisfiable instances. However, note that the classes \mathcal{C} contain satisfiable and unsatisfiable instances. We give a short description of the selected instances in the order in which they are listed in Table 1. The first four instances in Table 1, sat100-4.3 through sat200-4.0, are randomly generated to contain exactly three literals in each clause. Of course, these and some of the following problems are artificial. Nevertheless, we consider them useful for the evaluation of the learning process as they have well-known properties. Since the ratio between the number of clauses and variables is 4.3, the instances sat100-4.3 and sat200-4.3 have a small solution space. Thus, we expect that many unit clauses can be learned and that significant improvement is achieved. The situation is different for instances with the clause/variable ratio 4.0. Here, we cannot hope to learn many unit clauses. As we shall see, even for these problems learning turned out to be effective. The next instances, jnh201, par8-3-c, and par16-1-c, in Table 1 are taken from the benchmark collection of the Second DIMACS Challenge (Trick 1996). Problem jnh201 is a random instance generated to be difficult by rejecting unit clauses and setting the clause/variable ratio to a hard value. The two problems par8-3-c and par16-1-c arise from a problem in learning the parity function. Instances medium and bw_large.a are block-world planning problems (Kautz and Selman 1996). The last instance, ochem, has been used in

a practical application for industrial chemical exposure management (Straach and Truemper 1999) and is a MINSAT case. We ignore the costs in the tests for the SAT case. In the experiments for the MINSAT case, we introduced a cost of 1 for *True* for each variable in the SAT instances.

Table 1. Test Instances

Instance	No. of Variables	No. of Clauses Before Learning	No. of Clauses After Learning	
			SAT case	MINSAT case
sat100-4_3	100	430	397	366
sat200-4_3	200	860	1075	2583
sat100-4_0	100	400	800	1202
sat200-4_0	200	800	1600	2404
aim-100-3_4-yes1-1	100	340	302	311
aim-200-3_4-yes1-1	200	680	644	640
jnh201	100	800	794	2403
par8-3-c	75	298	259	216
par16-1-c	317	1264	1001	1001
medium	116	953	734	696
bw_large.a	459	4675	4629	4629
ochem	154	233	233	700

Table 1 displays for each instance the number of variables and the original number of clauses as well as the number of clauses after learning for both the SAT case and the MINSAT case. Observe that for several instances the number of clauses has been reduced by learning due to the replacement of some of the original clauses by learned lemmas.

Table 2 summarizes the timing results. The second column displays the time used for the entire learning process. The third and fourth columns show the guaranteed solution time bounds computed by the compiler before and after learning. The last two columns display the estimated worst-case run times of all \mathcal{C}_i before and after learning.

Table 2. Results of learning for the SAT case

Instance	Learning Time (min)	Guaranteed	Guaranteed	Worst-case	Worst-case
		Time Bound Before (sec)	Time Bound After (sec)	Time Before (sec)	Time After (sec)
sat100-4_3	0.04	> 1000	0.0020	0.5249	0.0016
sat200-4_3	14.11	> 1000	> 1000	50.0838	0.0540
sat100-4_0	5.26	> 1000	> 1000	0.3765	0.0354
sat200-4_0	28.33	> 1000	> 1000	52.7058	0.1561
jnh201	1.99	> 1000	> 1000	0.1772	0.1169
par8-3-c	0.01	> 1000	0.0014	0.0216	0.0010
par16-1-c	8.88	> 1000	0.0050	179.5371	0.0038
medium	0.34	> 1000	0.0260	0.0146	0.0043
bw_large.a	0.57	> 1000	0.0232	0.9368	0.0162
ochem	0.15	35.6000	35.6000	0.0046	0.0046

The learning times range from 1 sec to almost 30 min. The worst-case times after learning, in the last column of Table 2, indicate that the learning effort does pay dividends. Indeed, these times are uniformly small when compared with the worst times before learning. To assess the reduction factor, we focus on the problems that originally were difficult, say with solution time greater than 0.02 sec. The problems so defined are sat100-4_3, sat200-4_3, sat100-4_0, sat200-4_0, par8-3-c, par16-1-c, and bw_large.a. For these problems, the ratios of worst time before learning divided by worst time after learning range from 1.5 to 47247. If we focus on the subset of these problems that model some practical application (par8-3-c, par16-1-c, bw_large.a), we have reduction factors 22, 47247, and 58.

Table 3. Results of learning for the MINSAT case

Instance	Learning Time (min)	Guaranteed Time Bound Before (sec)	Guaranteed Time Bound After (sec)	Worst-case Time Before (sec)	Worst-case Time After (sec)
sat100-4_3	0.07	> 1000	0.0040	0.5006	0.0019
sat200-4_3	135.98	> 1000	> 1000	117.1800	0.5507
sat100-4_0	8.63	> 1000	> 1000	1.4682	0.2040
sat200-4_0	102.32	> 1000	> 1000	210.2571	5.0776
jnh201	477.47	> 1000	> 1000	21.7862	4.8142
par8-3-c	0.01	> 1000	0.0012	0.0408	0.0009
par16-1-c	14.56	> 1000	0.0050	247.3071	0.0041
medium	0.47	> 1000	0.6008	0.0199	0.0052
bw_large.a	0.44	> 1000	0.0232	0.9197	0.0164
ochem	39.52	> 1000	> 1000	6.5255	4.3697

We discuss the experiments for the MINSAT case. For all problems, except for ochem, the learning process terminates early for the following reason. Either a low worst-case time bound is determined, or the number of clauses becomes too large.

Table 3 displays the computational results for MINSAT learning. The interpretation is as for Table 2. The times for the learning process range from 1 sec to almost 8 hrs, a rather substantial investment. To evaluate the effect of the learning, we apply the same evaluation criteria as for Table 2. That is, we look at the problems that have worst time before learning greater than 0.02 sec. The problems are sat100-4_3, sat200-4_3, sat100-4_0, sat200-4_0, jnh201, par8-3-c, par16-1-c, bw_large.a, and ochem. For these problems, the reduction factor for worst times ranges from 1.5 to 60319. The reduction factors for problems arising from practical applications (par8-3-c, par16-1-c, bw_large.a, and ochem) are 45, 60319, 56, and 1.5.

References

- Bayardo Jr., R. J. and R. Schrag (1997). Using CSP Look-Back Techniques to Solve Real-World SAT Instances. *Proceedings of the 14th National Conference on Artificial Intelligence* (1997) 203-208
- Böhm, M. (1996). *Verteilte Lösung harter Probleme: Schneller Lastenausgleich*. Ph. D. thesis, Universität zu Köln, 1996

- Cadoli, M. and F. M. Donini (1997). A Survey on Knowledge Compilation. *AI Communications-The European Journal for Artificial Intelligence* 10 (1997) 137-150
- Dechter, R. (1990). Enhancement Schemes for Constraint Processing: Backjumping, Learning, and Cutset Decomposition. *Artificial Intelligence* 41 (1990) 273-312
- del Val, A. (1994). Tractable Databases: How to Make Propositional Unit Resolution Complete Through Compilation. *Proceedings of Fourth International Conference on Principles of Knowledge Representation and Reasoning (KR-94)* (1994) 551-561
- Dransfield, M., J. Franco, R. Price, and M. Vanfleet (2000). A solver for non-linear boolean functions, 2000
- Kautz, H. and B. Selman (1994). An Empirical Evaluation of Knowledge Compilation by Theory Approximation. *Proceedings of AAAI-94* (1994) 155-161
- Kautz, H. and B. Selman (1996). Pushing the Envelope: Planning, Propositional Logic, and Stochastic Search. *Proceedings of AAAI-96* (1996) 1194-1201
- Leibniz System (2000). Version 5.0, Leibniz, Plano, Texas, 2000
- Li, C. M. and Anbulagan (1997). Look-Ahead Versus Look-Back for Satisfiability Problems. *Proceedings of third International Conference on Principles and Practice of Constraint Programming* (1997) 342-356
- Marques-Silva, J. P. (2000). Algebraic Simplification Techniques for Propositional Satisfiability. *Proceedings of the 6th International Conference on Principles and Practice of Constraint Programming* (2000)
- Marques Silva, J. P. and K. A. Sakallah (1996). GRASP—A New Search Algorithm for Satisfiability. Technical Report CSE-TR-292-96, The University of Michigan, 1996
- Papadimitriou, C. H. (1994). *Computational Complexity*. Addison-Wesley, Reading, Massachusetts, 1994
- Prosser, P. (1993). Hybrid Algorithms for the Constraint Satisfaction Problem. *Computational Intelligence* 9(3) (1993) 268-299
- Richards, E. T. and B. Richards (2000). Nonsystematic Search and No-Good Learning. *Journal of Automated Reasoning* 24 (2000) 483-533
- Straach J. and K. Truemper (1999). Learning to ask relevant questions. *Artificial Intelligence* 111 (1999) 301-327
- Trick, M. A. (1996). Second DIMACS challenge test problems. In D. S. Johnson and M. A. Trick (Eds.), *Cliques, Coloring and Satisfiability: Second DIMACS implementation challenge*, Volume 26 of DIMACS series in Discrete Mathematics and Computer Science, American Mathematical Society, 1996, pp. 653-657
- Truemper, K. (1998). *Effective Logic Computation*. Wiley, New York, 1998
- Van Gelder, A. and F. Okushi (1999). Lemma and Cut Strategies for Propositional Model Elimination. *Annals of Mathematics and Artificial Intelligence* 26 (1999) 113-132
- Zhang, H. (1997). SATO: an Efficient Propositional Prover. *Proceedings of the International Conference on Automated Deduction* (1997) 272-275