Improving First-order Model Searching by Propositional Reasoning and Lemma Learning

Zhuo Huang[1], Hantao Zhang*[2], and Jian Zhang**[1]

[1] Laboratory of Computer Science
Institute of Software, Chinese Academy of Sciences
Beijing 100080, China
{hz,zj}@ios.ac.cn
[2] Department of Computer Science
University of Iowa Iowa City, IA 52242, U.S.A.
hzhang@cs.uiowa.edu

Abstract. The finite model generation problem in the first-order logic is a generalization of the propositional satisfiability (SAT) problem. An essential algorithm for solving the problem is backtracking search. In this paper, we show how to improve such a search procedure by lemma learning. For efficiency reasons, we represent the lemmas by propositional formulas and use a SAT solver to perform the necessary reasoning. We have extended the first-order model generator SEM, combining it with the SAT solver SATO. Experimental results show that the search time may be reduced significantly on many problems.

1 Introduction

The satisfiability (SAT) problem in the propositional logic has been studied by many researchers in the past 40 years. A number of efficient algorithms and data structures have been proposed, which lead to powerful SAT solvers like GRASP [11], SATO [16], Chaff [9], BerkMin [4], Siege [10], to name a few. In contrast, the satisfiability problem in the first-order logic has not received much attention. One reason is that the problem is undecidable in general.

Since the early 1990's, several researchers have made serious attempts to solving the *finite* domain version of the problem. More specifically, the problem becomes deciding whether the formula is satisfiable in a given finite domain. Several model generation programs have been constructed. Some of them are based on first-order reasoning (e.g., SATCHMO [7] and MGTP [3]); some of them are based on constraint satisfaction (e.g., FINDER [12], FALCON [17] and SEM [18]); some of them are based on the propositional logic (e.g., ModGen [6] and MACE [8]). These tools have been used to solve quite a number of challenging problems in discrete mathematics [13, 8, 17]. Despite such successes, the efficiency of such tools is still not so satisfactory [13].

One proven technique in the development of SAT solvers is the intelligent backtracking with conflict-driven learning (also called *lemma learning* or *conflict analysis*) [11, 20]. In this paper, we study how to use lemma learning to improve the performance of first-order finite model generators. We shall discuss the related issues (e.g., how to generate and use lemmas), and report our experiences with a prototype tool which combines SATO with SEM.

2 Background

The model generation problem studied in this paper is stated as follows. Given a set of first order clauses and a non-empty domain, find an interpretation of all the function symbols and predicate symbols appearing in the clauses such that all the clauses are true under this interpretation. Such an interpretation is called a *model*. Note that we assume that the input formulas are all clauses. Every variable in a clause is (implicitly) universally quantified.

Without loss of generality, an n-element domain is assumed to be $D_n = \{ 0, 1, \ldots, n-1 \}$. The Boolean domain is { FALSE, TRUE }. When the involved domain is finite and the arity of each

* Supported in part by NSF under grant CCR-0098093.
** Supported in part by NSFC under grant 60125207.

function/predicate is at most 2, a model can be conveniently represented by a set of *multiplication tables*, one for each function/predicate. For example, a 3-element model of the clause $f(x,x) = x$ is like the following:

f	0	1	2
0	0	1	0
1	1	1	0
2	0	1	2

Here f is a binary function symbol and its interpretation is given by the above 2-dimensional matrix. Each entry in the matrix is called a *cell*.

A model can also be represented by a set of formulas, with each formula describing a cell's value. To represent the main diagonal of the above matrix, we can use the following three formulas:

$$f(0,0) = 0; \quad f(1,1) = 1; \quad f(2,2) = 2.$$

From them, we know that the clause $f(x,x) = x$ is true and the matrix is indeed a model.

When the size of the model is fixed, we usually instantiate the input clauses and get a set of ground clauses (i.e., clauses containing no variables). For example, if the input formula is $f(x,y) = f(y,x)$ and the size of the model is 2, then we get the following ground clauses:

$$f(0,0) = f(0,0); \quad f(0,1) = f(1,0); \quad f(1,0) = f(0,1); \quad f(1,1) = f(1,1).$$

Let us denote such a set of ground clauses by Ψ.

3 Finite Model Searching Methods

Currently, there are essentially two classes of approaches for finding finite models of given first-order formulas: the complete approaches and incomplete approaches. The complete approaches often use backtrack search methods which exhaust a search space, and can show that a set of clauses do not have models. The incomplete approaches apply local search methods and are very attractive when the search space of the problem is too large for a backtracking based procedure and the solution set to the problem is very dense (the n-queen problems and the randomly generated SAT problems are such problems). Although there is no guarantee that any solution will be found by the incomplete approaches, they can indeed solve some large scale problems which are beyond the reach of complete search methods. However, this class of approaches is less successful for some structured problems whose solution set is sparse. For instance, several backtracking based programs have been reported to solve open quasigroup problems [12, 8]. However, incomplete methods have difficulty in solving these problems.

There are two complete approaches to finding finite models of given first-order formulas. The first approach translates the problem into a SAT problem and then uses a SAT solver to solve it. For details about this approach, see for example, [6, 8, 5]. A model can be represented by a set of assignments to propositional variables. Suppose there are m cells (c_0, c_1, ..., c_{m-1}) and the domain size is n. We introduce mn propositional variables: p_{ij} ($0 \le i < m$, $0 \le j < n$), where p_{ij} is true if the i'th cell has the value j. While there are many high-performance SAT solvers available, the major problem with this approach is that the translated SAT clauses for many real problems are long and huge in numbers. In recent years, some researchers proposed the lazy translation approaches (see for example, [2]). These methods abstract each atom of the first-order formula as a propositional variable, use a SAT solver to find a propositional model and then check the model against the theory. One benefit of the lazy approaches is that the number of generated propositional formulas is reduced.

The second approach to finding first-order models is searching for the values of the cells directly. The finite model generation problem may be viewed as a constraint satisfaction problem (CSP). The variables of the CSP are the *cell terms* (i.e., ground terms like $f(0,0)$, $f(0,1)$, etc.). The domain of each variable is D_n, and the constraints are the set of ground instances of the clauses, namely, Ψ. The goal is to find a set of assignments to the cells (e.g., $f(0,1) = 2$) such that all the ground clauses hold.

To solve the above problem, we can use backtracking search. We assume that, in this paper, the procedure stops when one model is found. That is, we do not try to find all models, even though the procedure can find all the models. The basic idea of the search procedure is roughly like the following: Repeatedly extend a partial model (denoted by *Pmod*) until it becomes a complete model (in which every cell gets a value). Initially *Pmod* is empty. *Pmod* is extended by selecting an unassigned cell and trying to find a value for it. Of course, when no value is appropriate for the cell, backtracking is needed and *Pmod* becomes smaller.

Similar to the DPLL procedure for solving SAT, the above procedure may also be depicted as a search tree, but the tree may not be binary. Each edge of the tree corresponds to choosing a value for some cell: $c_i = v$. As we discussed previously, this is equivalent to assigning TRUE to the propositional variable p_{iv}. Such a cell assignment usually results a set of further cell assignments, e.g., assigning FALSE to p_{iu} for every u ($0 \leq u < n$, $u \neq v$). For convenience, we call the assignment of FALSE to p_{iu} a *negative cell assignment*, and the assignment of TRUE to p_{iv} a *positive cell assignment*. The former excludes a value from the cell's domain, while the latter assigns a specific value to the cell. In the sequel, $c = v$ represents a positive cell assignment, while $c \neq v$ represents a negative cell assignment.

We define the *level* of a rooted tree as usual: The level of the root is 0 and the level of the children of a level n node is $n+1$. The level of a cell assignment such as $c_i = v$ whose truth values are decided at level n is n; and undefined if the truth value of this cell assignment is not decided. That is, the level of a cell assignment is the depth of the search tree at which its truth value is decided. For instance, if a cell is the first chosen one to be assigned, the level of that cell assignment is 1. If, as a result of the propagation of that cell assignment, another cell c_j cannot have value k, then the level of the cell assignment $c_j \neq k$ will also be 1.

Each node of the search tree corresponds to a partial model, in which the truth values of some cell assignments are decided. Among the truth values of these cell assignments, some are decided by heuristic selection, while others are decided by propagation (or deduction). In the former case, the *reason* for the cell assignment is defined to be the empty set. In the latter case, the *reason* of that cell assignment is defined to be the set of all the cell assignments sharing a ground input clause and it is that ground clause which forces the truth value of the cell assignment. For instance, suppose $f(g(0,1),2) \neq 3$ is a ground clause (instantiated from some input clause) and $g(0,1) = 2$ becomes true at level 2, then the level of the cell assignment $f(2,2) \neq 3$ is also 2 and reason$(f(2,2) \neq 3) = \{g(0,1) = 2\}$.

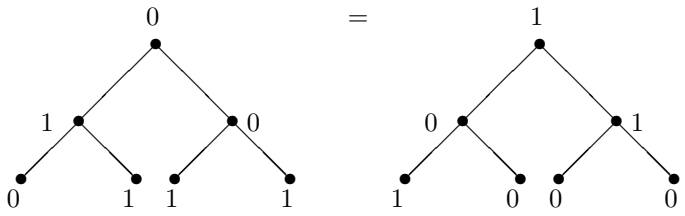
4 Lemma Learning

Lemma learning has been quite successful in improving SAT algorithms [11, 16, 19]. In this paper, we study how to incorporate such a mechanism into first-order model generators. As we know, the leaves of the search tree can be SUCCESS (denoting that a model is found) or FAILURE (denoting that a contradiction is encountered).

One problem with the current first-oder model searchers like SEM [18] is that, they may fail for the same reason in more than one branches of the search tree. To avoid this problem, we add *lemma learning* to the search procedure. Specifically, suppose at the leaf node L, when we evaluate the clause $\varphi_1 \in \Psi$, we find that its value is FALSE. Then we try to find a set of cell assignments that contribute to the falsity of φ_1. This set is denoted by FAS(L, φ_1), where FAS stands for False Assignment Set.

The set FAS is decided in the following way. At node L, all (sub)terms and literals in φ_1 should get specific values. Suppose there are l literals in φ_1. Then $FAS = FAS_1 \cup FAS_2 \cup \ldots \cup FAS_l$, where each FAS_i is the false assignment set of literal l_i. For each i ($1 \leq i \leq l$), the i'th literal is a tree structure. We visit all its internal nodes recursively, recording all the cell assignments. For example, suppose one internal node corresponds to the function symbol 'f', and the node has two children. If the current values of the three nodes are 1, 0, 2, respectively, we add the cell assignment $f(0,2) = 1$ to the set FAS.

Let us look at a slightly more complex example. Suppose the clause φ_1 is $f(f(0,1), f(1,1)) = f(f(1,0), f(0,0))$, as depicted below. Every internal node is associated with the function symbol 'f', and thus it is omitted in the picture. Instead, the current value at each node is shown. From these values, we obtain that FAS = $\{\ f(0,1) = 1,\ f(1,1) = 0,\ f(1,0) = 0,\ f(0,0) = 1\ \}$.



Suppose FAS = { $x_1 \neq e_1, x_2 = e_2, \ldots, x_k = e_k$ }. Then we may generate the following lemma

$$x_1 = e_1 \vee x_2 \neq e_2 \vee \ldots \vee x_k \neq e_k.$$

In many cases, there are more than one cell assignment of FAS whose level is at the bottom level. To produce a high quality lemma which is more likely to eliminate futile assignments and reduce the size of the remaining search space, we may use the following procedure to change FAS and thus produce a different lemma from FAS.

FAS_enhancement(FAS)
while TRUE **do**
pick *ca* in FAS whose truth value is decided last
if reason(*ca*) is empty **return** FAS
FAS = FAS − {*ca*} ∪ reason(*ca*)
end while

Theorem 1. (*a*) FAS_enhancement(FAS) *will terminate for any set* FAS *of cell assignments;* (*b*) *Let* FAS′ = FAS_enhancement(FAS)*, where the lemma created from* FAS *is a valid consequence of the input, so is the lemma created from* FAS′.

5 The Modified Procedure

The lemmas are quite simple syntactically. Thus we choose to use propositional reasoning to deduce useful information from them.

We keep two sets of clauses, one is the original input, namely Ψ, and the other is a set of propositional clauses, denoted by *Lma*. The latter is a set of lemmas learned during the search.

Initially, the set *Lma* consists of the domain closure axioms for each cell *ce*:

$$ce = 0 \vee ce = 1 \vee \ldots \vee ce = n - 1$$

Now we modify the basic backtracking procedure. The modification lies mainly in the following two points:

- Adding a lemma when a failure is encountered, as described previously.
- Performing propositional reasoning on *Lma*, and add the deduced assignments to *Pmod*, if no contradiction occurs.

The above two steps involve translating certain first-order clauses to propositional clauses and vice versa. This is straightforward as the form of the lemmas is a propositional clause.

6 Implementation and Experimental Results

Based on the above ideas, we have extended the tool SEM [18], using the SAT solver SATO [16] to perform propositional reasoning. Table 1 compares the performances of SEM with (new) and without (old) lemma learning on some mathematical problems, along with the experimental results of two versions of Mace [8]. All of the problems come from TPTP [14], except for LatinZ which is a problem in combinatorics. The experiments were carried out on a SUN UltraSparc workstation

Table 1. Performance comparison on TPTP problems

Problem	Size	New SEM		Old SEM		MACE2	MACE4
		Round	Time	Round	Time	Time	Time
ALG 8-1	5	22	11.34	21	12.31	12.33	27.65
BOO 8-3	5	65	5.69	65	6.10	2.50	15.76
BOO19-1	5	41	0.07	41	0.05	12.02	0.13
BOO020	12	3694	84.26	3458	75.88	140.49	>1800
BOO30-1	5	25	0.01	25	0.00	0.18	0.02
BOO32-1	8	56	0.04	56	0.07	6.03	0.09
CAT19-4	20	420	0.43	420	0.28	166.79	0.04
COL73-1	10	69	0.04	69	0.05	>600	1.23
LatinZ	6	25372	7.71	48446	10.32	559.38	12.28
LCL137-1	6	2205	0.21	608	0.06	75.42	0.08
NUM14-1	15	204	0.84	212	0.80	5.40	2.08
RNG25-8	5	857	0.32	867	0.30	31.72	0.87
RNG31-6	6	2960	1.12	3578	1.63	>1200	0.80
ROB12-1	3	32916	1.45	123458	24.64	12.03	9.01
ROB12-2	3	36762	2.26	1202834	33.77	19.02	9.96
SYN305-1	12	838	0.02	1630	0.02	>600	0.02

(ENTERPRISC 450, two 400MHz CPUs, 1GB Memory). In the table, the running time is given in seconds, and "size" refers to the domain size. For SEM, a "round" refers to the number of times when we try to find an appropriate value for a selected cell.

The experimental results show that introducing lemma learning increases the performance of SEM on many problems, such as ROB (Robbins Algebra), and LatinZ. It appears that the lemma learning rule is not so effective on easy problems (which are not the focus of this research).

For some problems like GRP and RNG, the new version of SEM can reduce the number of branches in the search tree, but not necessarily the overall running time. There are probably two reasons: one is the effectiveness of the current lemma learning method, and the other is the efficiency of the implementation. In the future, we need to study the best way to find and use the lemmas, and to use more advanced implementation techniques.

On some satisfiable problems, the new version of SEM takes longer to find a solution than the older version. The first solution found by the new SEM may be different from the first solution found by the old SEM.

We also tried to solve the problems by translating them into propositional clauses and then using zChaff [9]. For some problems like COL73-1, LCL137-1 and RNG 31-6, there are too many propositional clauses and the memory is not enough. The other problems can be solved within a few seconds. For instance, the LatinZ problem is translated by SAGE [5] in 1.35 seconds, and then the propositional clauses are solved by zChaff [9] in 4.65 seconds.

7 Concluding Remarks

The finite model generation problem for the first-order logic can be solved either directly (by searching for the cells' values) or indirectly (by translating to SAT). For some problems, the translation approach may result in too many clauses. For the other problems, a SAT solver is usually quite efficient. The benefits of the direct search method include that the structure (in particular the symmetries) of the problem description can be used to reduce the search space, and the reasoning can be performed in larger steps.

The conflict-directed lemma learning is a way of generating new valid clauses (called lemmas) in response to a contradictory set of assignments. The new clauses eliminate futile assignments and reduce the size of the remaining search space. This is a proven technique for the success of today's high performance SAT solvers. We showed in this paper that the same technique can be used for finite model generation in first-order logic.

We have implemented the lemma learning in the first-order model searcher SEM and used the SAT solver SATO to handle propositional clauses produced by lemma learning. We conducted some preliminary experiments and showed that the performance has been improved for some problems. We observed in our experiments that the lemma learning rule may interfere with the variable choosing heuristic [18]. We shall look into the issue in the future. We also plan to conduct more experiments in order to have a better understanding of the new system and further improve its performance.

In the recent years, there are many attempts to combine a SAT solver with another system, for instance [1, 20]. The work presented in this paper is one of such examples. A systematic approach for combining SAT solvers with other theories has been proposed in [15]. However, the proposed approach does not address the issue of reusing existing SAT solvers for these theories. As a direction for further research, we will study a modular interface for the required operations of a SAT solver in such applications.

References

1. F.A. Aloul *et al.*, Generic ILP versus specialized 0-1 ILP, an update. *Proc. ICCAD*, 450–457, 2002.
2. C.W. Barrett, D.L. Dill and A. Stump, Checking satisfiability of first-Order formulas by incremental translation to SAT, *Proc. 14th Int'l Conf. on Computer-Aided Verification*, LNCS 2404, 236–249, 2002.
3. M. Fujita, J. Slaney and F. Bennett, Automatic generation of some results in finite algebra, *Proc. IJCAI-93*, 52–57, Chambéry, France.
4. E. Goldberg and Y. Novikov, BerkMin: A fast and robust SAT solver. In *Design, Automation, and Test in Europe* (DATE'02), 142–149, 2002
5. Z. Huang and J. Zhang, Generating SAT instances from first-order formulas, *J. of Software*, to appear.
6. S. Kim and H. Zhang, ModGen: Theorem proving by model generation, *Proc. 12th AAAI*, 162–167, 1994.
7. R. Manthey and F. Bry, SATCHMO: A theorem prover implemented in Prolog, *Proc. CADE-9*, 415–434, 1998.
8. W. McCune, A Davis-Putnam program and its application to finite first-order model search: quasigroup existence problems. Technical Report ANL/MCS-TM-194, Argonne National Laboratory, 1994.
9. M. Moskewicz *et al*, Chaff: Engineering an efficient SAT solver, *Proc. 38th Design Automation Conference*, 530–535, 2001.
10. L. O. Ryan, The siege satisfiability solver. `http://www.cs.sfu.ca/~loryan/personal/`
11. J.P.M. Silva and K.A. Sakallah, Conflict analysis in search algorithms for propositional satisfiability, Technical Reports, Cadence European Laboratories, ALGOS, INESC, Lisboa, Portugal, May 1996.
12. J. Slaney, FINDER: Finite domain enumerator – system description, *Proc. CADE-12*, 798–801, 1994.
13. J. Slaney, M. Fujita and M. Stickel, Automated reasoning and exhaustive search: Quasigroup existence problems, *Computers and Mathematics with Applications* 29(2): 115–132, 1995.
14. G. Sutcliffe and C. Suttner, The TPTP problem library for automated theorem proving, 2001. `http://www.cs.miami.edu/~tptp/`.
15. C. Tinelli. A DPLL-based calculus for ground satisfiability modulo theories. *Proc. of 8th European Conf. on Logics in Artificial Intelligence*, LNAI 2424, pages 308–319, 2002.
16. H. Zhang, SATO: An efficient propositional prover, *Proc. CADE-14*, LNAI 1249, 272–275, 1997.
17. J. Zhang, Constructing finite algebras with FALCON, *J. Automated Reasoning* 17(1): 1–22, 1996.
18. J. Zhang and H. Zhang, SEM: a system for enumerating models, *Proc. 14th IJCAI*, 298–303, 1995.
19. L. Zhang and S. Malik, The quest for efficient Boolean satisfiability solvers, *Proc. CADE-18*, LNAI 2392, 295–313, 2002.
20. L. Zhang, C.F. Madigan, M.W. Moskewicz and S. Malik, Efficient conflict driven learning in a boolean satisfiability solver, *Proc. ICCAD*, 279–285, 2001.