Extending Finite Model Searching with Congruence Closure Computation

Jian Zhang[1,⋆] and Hantao Zhang[2,⋆⋆]

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

Abstract. The model generation problem, regarded as a special case of the Constraint Satisfaction Problem (CSP), has many applications in AI, computer science and mathematics. In this paper, we describe how to increase propagation of constraints by using the ground congruence closure algorithm. The experimental results show that using the congruence closure algorithm can reduce the search space for some benchmark problems.

1 Introduction

Compared to the research on propositional satisfiability problem, the satisfiability of first-order formulas 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 [6, 3, 2, 10, 14, 16]. By *model generation* we mean, given a set of first order formulas as axioms, finding their models automatically. A *model* is an interpretation of the function and predicate symbols over some domain, which satisfies all the axioms. Model generation is very important to the automation of reasoning. For example, the existence of a model implies the satisfiability of an axiom set or the consistency of a theory. A suitable model can also serve as a counterexample which shows some conjecture does not follow from some premises. In this sense, model generation is complementary to classical theorem proving. Models help people understand a theory and can guide conventional theorem provers in finding proofs.

⋆ Supported by the National Science Fund for Distinguished Young Scholars of China under grant 60125207. Part of the work was done while the first author was visiting the University of Iowa.
⋆⋆ Supported in part by NSF under grant CCR-0098093.

B. Buchberger and J.A. Campbell (Eds.): AISC 2004, LNAI 3249, pp. 94–102, 2004.

Some of the model generation methods are based on first-order reasoning (e.g., SATCHMO [6] and MGTP [2, 3]); some are based on constraint satisfaction (e.g., FINDER [10], FALCON [14] and SEM [16]); while others are based on the propositional logic (e.g., ModGen [4] and MACE [7]). These tools have been used to solve a number of challenging problems in discrete mathematics [11, 7, 14]. Despite such successes, there is still space for improvement on the performance of the tools.

In this paper, we study how to improve the performance of finite model searchers by more powerful reasoning mechanisms. Specifically, we propose to incorporate congruence closure computation into the search procedure of the model generation tools.

2 Model Generation as Constraint Satisfaction

The finite model generation problem studied in this paper is stated as follows. Given a set of first order clauses and a non-empty finite 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*. Here we assume that all the input formulas are clauses. Each variable in a clause is (implicitly) universally quantified.

Without loss of generality, we assume that an n-element domain is the set $D_n = \{ 0, 1, \ldots, n-1 \}$. The Boolean domain is { FALSE, TRUE }. If the arity of each function/predicate symbol is at most 2, a finite 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*.

In this paper, we treat the problem as a constraint satisfaction problem (CSP), which has been studied by many researchers in Artificial Intelligence. 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 (except for predicates, whose domain is the Boolean domain). The constraints are the set of ground instances of the input clauses, denoted by 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.

Theoretically speaking, any approach of constraint satisfaction can be used for finite model generation. For instance, a simple backtracking algorithm can always find a finite model (if it exists). Of course, a brute-force search procedure is too inefficient to be of any practical use. There are many other search procedures and heuristics proposed in the AI literature, e.g., forward checking and lookahead. See [5] for a good survey.

In this paper, we will solve the finite model generation problem using backtrack search. The basic idea of such a 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 (from its domain). When no value is appropriate for the cell, backtracking is needed and *Pmod* becomes smaller.

The execution of the search procedure can be represented as a search tree. Each node of the tree corresponds to a partial model, and each edge corresponds to assigning a value to some cell by the heuristic. We define the *level* of a node as usual, i.e., the level of the root is 0 and the level of the children of a level n node is $n + 1$.

3 Congruence Closure for Constraint Propagation

The efficiency of the search procedure depends on many factors. One factor is how we can perform reasoning to obtain useful information when a certain number of cells are assigned values. That is, we have to address the following issue: how can we implement constraint propagation and consistency checking efficiently?

This issue may be trivial for some constraint satisfaction algorithms because the constraints they accept are often assumed to be unary or binary. It is true that n-ary constraints can be converted into an equivalent set of binary constraints; but this conversion usually entails the introduction of new variables and constraints, and hence an increase in problem size. This issue is particularly important to model generation because in this case, the constraints are represented by complicated formulas. Experience tells us that a careful implementation can improve the performance of a program significantly.

In [15], a number of inference rules are described in detail. They are quite effective on many problems. In this paper, we discuss another inference rule, namely *congruence closure*, which can be quite useful for equational problems.

3.1 Congruence Closure

An *equivalence relation* is a reflexive, symmetric and transitive binary relation. A relation $\sim$ on the terms is *monotonic* if $f(s_1, \ldots, s_n) \sim f(t_1, \ldots, t_n)$ whenever f is an n-ary function symbol and for every i $(1 \leq i \leq n)$, $s_i \sim t_i$. A *congruence relation* is a monotonic equivalence relation. A set of ground equations, E, defines a relation among the ground terms. The *congruence generated* by E, denoted by E^*, is the smallest congruence relation containing E.

There are several algorithms for computing the congruence generated by a set of ground equations, called *congruence closure* algorithms, e.g., [1, 8]. The Nelson-Oppen algorithm [8] represents terms by vertices in a directed graph. It uses UNION and FIND to operate on the partition of the vertices, to obtain the congruence relation.

A congruence closure algorithm can deduce some useful information for finite model searching. One example is given in [8]. From the following two equations:

```
f(f(f(a))) = a
f(f(f(f(f(a))))) = a
```

we can deduce that `f(a) = a`. Thus we need not "guess" a value for `f(a)`, if the above two equations are in the input clauses. Without using the congruence closure, we may have to try unnecessary assignments such as `f(a) = b, f(a) = c`, etc., before or after the assignment of `f(a) = a`.

3.2 An Example

Let us look at a problem in the combinatorial logic. A fragment of combinatory logic is an equational system defined by some equational axioms. We are interested in the fragment { B, N1 }, whose axioms are:

$$a(a(a(\mathrm{B}, x), y), z) \;=\; a(x, a(y, z))$$
$$a(a(a(\mathrm{N1}, x), y), z) \;=\; a(a(a(x, y), y), z)$$

Here B and N1 are constants, while the variables x, y and z are universally quantified. The *strong fixed point property* holds for a fragment of combinatory logic if there exists a combinator y such that for all combinators x, $a(y, x) = a(x, a(y, x))$. In other words, the fragment has the strong fixed point property if the formula φ: $\exists y \forall x\, [a(y, x) = a(x, a(y, x))]$ is a logical consequence of the equational axioms.

In [13], it is shown that the fragment { B, N1 } does not have the strong fixed point property, because there is a counterexample of size 5. It is a model of the following formulas:

```
(BN1-1)   a(a(a(0,x),y),z) = a(x,a(y,z)).
(BN1-2)   a(a(a(1,x),y),z) = a(a(a(x,y),y),z).
(BN1-3)   a(y,f(y)) != a(f(y),a(y,f(y))).
```

The last formula is the negation of the formula φ, where `f` is a Skolem function and `!=` means "not equal". In the first two formulas, we assume that B is 0 and N1 is 1. That is, B and N1 take different values. (But it is also possible to generate a counterexample in which B = N1.)

Search by SEM. When using the standard version of SEM [16], the first few steps of the search tree are the following:

(1) Choose the cell `a(0,0)` and assign the value 0 to it.
(2) Choose the cell `f(0)` and assign the value 1 to it. Note that `f(0)` cannot be `0`; otherwise the formula (BN1-3) does not hold, because we already have `a(0,0) = 0`.

(3) Choose the cell `a(0,1)` and assign the value 1 to it. After the first two choices, SEM deduces that `a(0,1)` cannot be `0`. Otherwise, suppose `a(0,1) = 0`. Let `x = 1` and `y = z = 0` in the formula (BN1-1), we have `a(a(a(0,1),0),0) = a(1,a(0,0))`, which is simplified to `0 = a(1,0)`. Then let `y = 0` in the formula (BN1-3), we shall have $0 \neq 0$.
(4) Choose the cell `a(1,1)` and assign the value 2 to it.

After the first three steps, SEM will also deduce that $\texttt{a(1,1)} \neq \texttt{1}$.

The Effect of Congruence Closure Computation. If we apply the congruence closure algorithm to the ground instances of the formulas (BN1-1) and (BN1-2) and the first three assignments

```
a(0,0) = 0;   f(0) = 1;   a(0,1) = 1;          (P1)
```

we shall get the conclusion `a(1,1) = 1`. This contradicts with the inequality $\texttt{a(1,1)} \neq \texttt{1}$ which is deduced by SEM. Thus if we extend SEM's reasoning mechanism with the congruence closure algorithm, we know that, at step 3, assigning the value 1 to the cell `a(0,1)` will lead to a dead end.

Now suppose we try `a(0,1) = 2` next. For this new branch, we add the following three equations to Ψ:

```
a(0,0) = 0;   f(0) = 1;   a(0,1) = 2.          (P2)
```

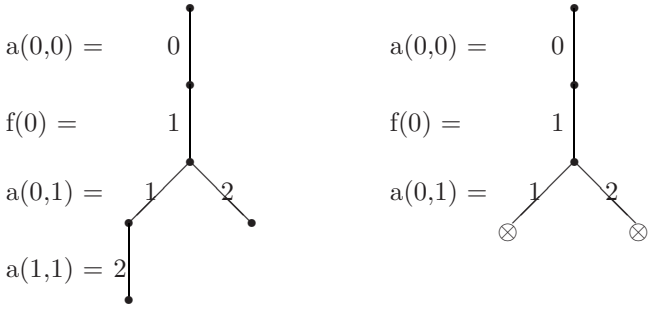
After computing the congruence closure, we can get these equations:

```
a(0,0) = a(1,0) = a(2,0) = 0;
f(0) = 1;
a(0,1) = a(0,2) = a(1,2) = a(2,2) = 2.
```

However, with the old version of SEM, after the three assignments (P2), we only deduce `a(0,2) = 2` as a new cell assignment. In other words, the partial model consists of these four assignments:

```
a(0,0) = 0;   f(0) = 1;   a(0,1) = 2;   a(0,2) = 2.
```

SEM can also deduce some negative assignments, e.g., $\texttt{a(1,2)} \neq \texttt{2}$. This contradicts with what we get from the closure computation algorithm. Thus, we have come to a dead end again. We don't need to go further and expand the search tree (as the old version of SEM does). The difference is illustrated by the following two search trees:



The left tree describes the search process of old SEM, while the right one describes the effect of congruence closure computation on the search process. We see that some branches of the search tree can be eliminated if we combine congruence closure computation with existing reasoning methods of SEM.

4 The Extended Search Algorithm

The model generation process can be described by the recursive procedure in Fig. 1, which searches for every possible model. It can be easily modified to search for only one model. The procedure uses the following parameters:

- $Pmod = \{(ce, e) \mid e \in Dom(ce)\}$: assignments (cells and their assigned values), where $Dom(ce)$ is the domain of values for ce;
- $\mathcal{D} = \{(ce, D) \mid D \subset Dom(ce)\}$: unassigned cells and their possible values;
- Ψ: constraints (i.e. the clauses).

Initially $Pmod$ is empty, and $\mathcal{D}$ contains $(ce, Dom(ce))$ for every cell ce.

```
proc search(Pmod, D, Ψ)
{
   if D = ∅ then /* a model is found */
        { print(Pmod); return; }
   choose and delete (ce_i, D_i) from D;
   if D_i = ∅ then return; /* no model */
   for e ∈ D_i do
   {
      (Pmod', D', Ψ') := propa(Pmod ∪ {(ce_i, e)}, D, Ψ);
      if Ψ' is not FALSE /* no contradiction found */
        then  search(Pmod', D', Ψ');
   }
}
```

Fig. 1. The abstract search procedure

The procedure $\texttt{propa}(Pmod, \mathcal{D}, \Psi)$ propagates assignment $Pmod$ in Ψ: it simplifies Ψ and may force some variables in $\mathcal{D}$ to be assigned. The procedure $\texttt{propa}(Pmod, \mathcal{D}, \Psi)$ is essentially a closure operation (with respect to a set of sound inference rules). It repeatedly modifies $Pmod, \mathcal{D}$, and Ψ until no further changes can be made. When it exits, it returns the modified triple $(Pmod, \mathcal{D}, \Psi)$. The basic steps of this procedure can be described as follows.

(1) For each new assignment (ce, e) in $Pmod$, replace the occurrence of ce in Ψ by e.
(2) If there exists an empty clause in Ψ (i.e., each of its literals becomes `FALSE` during the propagation), replace Ψ by `FALSE`, and exit from the procedure. Otherwise, for every unit clause in Ψ (i.e. all but one of its literals become `FALSE`), examine the remaining literal l.

- If l is a Boolean cell term ce, and $(ce, D) \in \mathcal{D}$, then delete (ce, D) from $\mathcal{D}$ and add $(ce, \texttt{TRUE})$ to $Pmod$; similarly, if l is the negation of a Boolean cell term in $\mathcal{D}$, i.e. $\neg ce$, delete (ce, D) from $\mathcal{D}$ and add $(ce, \texttt{FALSE})$ to $Pmod$.
- If l is of the form $\texttt{EQ}(ce, e)$ (or $\texttt{EQ}(e, ce)$), and $(ce, D) \in \mathcal{D}$, delete (ce, D) from $\mathcal{D}$ and add (ce, e) to $Pmod$; similarly, if l is of the form $\neg\texttt{EQ}(ce, e)$ (or $\neg\texttt{EQ}(e, ce)$), and $(ce, D) \in \mathcal{D}$, then delete e from D.

(3) For each pair $(ce, D) \in \mathcal{D}$, if $D = \emptyset$, then replace Ψ by FALSE, and exit from the procedure; if $D = \{e\}$ (i.e. $|D| = 1$), then delete the pair (ce, D) from $\mathcal{D}$, and add the assignment (ce, e) to $Pmod$.

(4) Let $E^* = \texttt{groundCC}(Pmod \cup E(\Psi))$, where $E(X)$ is the set of all equations in X and $\texttt{groundCC}(Q)$ returns the consequences of the congruence closure of ground equations Q. If E^* and $Pmod$ are inconsistent, replace Ψ by FALSE and exit from the procedure; otherwise extend $Pmod$ with all cell assignments in E^*.

The last item in the above list is an addition to the original constraint propagation procedure implemented in SEM. Basically, the extended search algorithm tries to deduce useful information using congruence closure computation, from all the equations in Ψ and all the cell assignments in $Pmod$.

When will E^* and $Pmod$ be inconsistent? Firstly, if E^* puts two domain elements in the same equivalence class (e.g., `1 = 2`, or TRUE = FALSE), we get an inconsistency. Secondly, if E^* contains $ce = v$ but v is not in $Dom(ce)$, then it is also inconsistent. When consistent, we extend the current partial assignment $Pmod$ with all cell assignments in E^*, i.e. equations like `a(0,1) = 2`.

5 Implementation and Experiments

We have extended SEM [16] with the Nelson-Oppen algorithm [8] for computing congruences. We call the new version SEMc.

The implementation of the congruence closure algorithm is straightforward. No advanced data structures are used. As a result, the efficiency is not so good. But this still allows us to experiment with the new search procedure.

Since congruence computation may not offer new information at all nodes of the search tree, we add a control parameter (denoted by Lvl) to SEMc. It means that congruence is only computed at nodes whose levels are less than Lvl.

We have experimented with SEMc on some model generation problems. Table 1 compares the performances of SEMc with that of SEM. The results were obtained on a Dell Optiplex GX270 (Pentium 4, 2.8 GHz, 2G memory), running RedHat Linux. In the table, the running times are given in seconds. A "round" refers to the number of times when we try to find an appropriate value for a selected cell. This can be done either when extending a partial solution (i.e., trying to assign a value for a new cell) or during backtracking (i.e., trying to assign a new value to a cell which already has a value).

Here `BN1` refers to the combinatorial logic problem described in Section 3. The problem `g1x` has the following two clauses:

Table 1. Performance comparison

Problem	Size	Satisfiable	SEM		SEMc	
			Round	Time	Round	Time
BN1	4	No	888	0.00	502	0.22
BN1	5	Yes	34	0.00	26	0.02
g1x	4	Yes	11803	0.05	675	0.11
BOO033-1	5	Yes	29	0.00	27	0.05
LCL137-1	6	Yes	609	0.01	569	0.03
LCL137-1	8	Yes	5662	0.09	5466	0.15
ROB012-1	3	No	983694	2.55	982652	2.69
ROB015-1	3	No	1004638	2.25	1001420	2.35

```
f(z,f(g(f(y,z)),f(g(f(y,g(f(y,x)))),y))) = x
f(0,g(0)) != f(1,g(1))
```

The existence of a model implies that the first equation is not a single axiom for group theory. The other problems are from TPTP [12].

For some problems, there are non-unit clauses as well as negated equations. Thus the use of congruence computation does not make much difference. For problems like `g1x` where complex equations dominate in the input, the new version of SEM can reduce the number of branches greatly (by 17 times).

In the above experiments, *Lvl* is set to 10 (an arbitrary value). Without the restriction, the overhead of computing congruence closure might be significant.

6 Concluding Remarks

Many factors can affect the efficiency of a backtrack procedure. In the context of finite model searching, these include the heuristics for choosing the next cell and the inference rules for deducing new information from existing assignments. In this paper, we have demonstrated that adding congruence computation as a new inference rule can reduce the number of branches of the search tree, especially when most clauses are complex equations.

The current implementation is not so efficient because we have implemented only the original Nelson-Oppen algorithm [8] for computing congruences. With more efficient data structures and algorithms (e.g. [9]), we expect that the running time of SEMc can be reduced. Another way of improvement is that the congruence should be computed incrementally. In the current version of SEMc, each time the congruence is computed, the algorithm starts from all ground instances of the input equations. However, during the search, many of them can be neglected (when both the left-hand side and the right-hand side are reduced to the same value).

Acknowledgements

We are grateful to the anonymous reviewers for their detailed comments and suggestions.

References

1. P.J. Downey, R. Sethi and R.E. Tarjan, Variations on the common subexpression problem, *J. ACM* 27(4): 758–771, 1980.
2. M. Fujita, J. Slaney and F. Bennett, Automatic generation of some results in finite algebra, *Proc. 13th IJCAI*, 52–57, 1993.
3. R. Hasegawa, M. Koshimura and H. Fujita, MGTP: A parallel theorem prover based on lazy model generation, *Proc. CADE-11*, LNAI 607, 776–780, 1992.
4. S. Kim and H. Zhang, ModGen: Theorem proving by model generation, *Proc. 12th AAAI*, 162–167, 1994.
5. V. Kumar, Algorithms for constraint satisfaction problems: A survey, *AI Magazine* 13(1): 32–44, 1992.
6. R. Manthey and F. Bry, SATCHMO: A theorem prover implemented in Prolog, *Proc. CADE-9*, LNCS 310, 415–434, 1988.
7. 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.
8. G. Nelson and D.C. Oppen, Fast decision procedures based on congruence closure, *J. ACM* 27(2): 356–364, 1980.
9. R. Nieuwenhuis and A. Oliveras, Congruence closure with integer offsets, *Proc. 10th LPAR*, LNAI 2850, 78–90, 2003.
10. J. Slaney, FINDER: Finite domain enumerator – system description, *Proc. CADE-12*, LNCS 814, 798–801, 1994.
11. 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.
12. G. Sutcliffe and C. Suttner, The TPTP problem library for automated theorem proving. `http://www.cs.miami.edu/~tptp/`
13. J. Zhang, Problems on the generation of finite models, *Proc. CADE-12*, LNCS 814, 753–757, 1994.
14. J. Zhang, Constructing finite algebras with FALCON, *J. Automated Reasoning* 17(1): 1–22, 1996.
15. J. Zhang and H. Zhang, Constraint propagation in model generation, *Proc. Int'l Conf. on Principles and Practice of Constraint Programming*, LNCS 976, 398–414, 1995.
16. J. Zhang and H. Zhang, SEM: A system for enumerating models, *Proc. 14th IJCAI*, 298–303, 1995.