01 Jan 1997

Direct Finite First-Order Model Generation with Negative Constraint Propagation Heuristic

Olga Shumsky

Ralph W. Wilkerson
Missouri University of Science and Technology, ralphw@mst.edu

Fikret Ercal
Missouri University of Science and Technology, ercal@mst.edu

William W. McCune

Recommended Citation

O. Shumsky et al., "Direct Finite First-Order Model Generation with Negative Constraint Propagation Heuristic," *Proceedings of the ACM Symposium on Applied Computing*, pp. 25 - 29, Association for Computing Machinery, Jan 1997.
The definitive version is available at https://doi.org/10.1145/331697.331704

Direct Finite First-Order Model Generation with Negative Constraint Propagation Heuristic

Olga Shumsky
Department of Electrical and Computer Engineering
Northwestern University, Evanston, Illinois 60201
shumsky@ece.nwu.edu

Ralph W. Wilkerson and Fikret Ercal
Department of Computer Science, University of Missouri-Rolla, Rolla, Missouri 65401

William W. McCune
Mathematics and Computer Science Division, Argonne National Laboratory
9700 South Cass Avenue, Argonne, Illinois 60439

ABSTRACT

An automated finite first-order model generator has been developed. The problem is viewed as a first-order satisfiability problem. Most existing model generators reduce the problem to propositional satisfiability by converting the input first-order clauses into propositional clauses. This generator, unlike others, stores the input first-order clauses and solves the problem directly. It uses an exhaustive backtracking algorithm with weight-based splitting. A negative constraint propagation is implemented to reduce the number of decision points and thus to speed up the search.

PROBLEM STATEMENT AND SOLUTION METHODS

We are presented with a set of first-order sentences and a finite domain of elements. To simplify the problem, we always convert the sentences to *first-order clauses*, which are disjunctions of literals. *A literal*, in this case, is a statement about functions. The task is to determine whether the given set of clauses is satisfiable, i.e. whether there exists an assignment of function values that makes every clause true. Such set of assignments is called *a model*. We would like to find models, if they exist. Two methods have been used to solve the problem of finite first-order model generation.

Conversion-Based Model Generators

A first-order clause modified so that every n-place function is replaced by an $(n+1)$-place relation and every constant is replaced by a one-argument relation is called *a flat clause*. A flat clause may be converted to a set of propositional clauses by instantiating every variable in the clause over a given domain. The resulting propositional clauses are called *ground flat clauses*.

One approach to finite first-order model generation is to convert the input first-order sentences to flat clauses[5]. The set of flat clauses is instantiated over the specified domain. The resulting set of ground clauses is used to search for propositional models. If found, the models are converted to first-order models.

This method has several advantages. Propositional satisfiability is a well studied problem. While it is computationally intractable in general, many practically useful solutions have been developed[2, 3]. The search procedure operates on clauses of literals, each of which is of the form $P(n_0, n_1, \ldots, n_m)$ where P is a relation and n_i is an element of the domain. Because of this relatively simple structure of the search space, elegant, and therefore fast, implementations are possible. Once the problem has been converted to ground flat clauses, the original nature of the problem does not matter.

The drawbacks of the method include possibly excessive memory requirements to store ground clauses. Some formulae involving equality translate into two flat clauses. Given a domain of m elements, a flat clause with n variables produces n^m ground flat clauses. Also, the conversion from first-order to ground clauses is time-consuming.

Direct Model Generators

This method addresses the problems of the conversion-based approach. The input conversion phase is omitted by representing every input clause as an algebraic tree. Several copies of the tree are created during the instantiation process, when variables are replaced by the domain elements. The search procedure then traverses the resulting trees, reducing their depth through assignment of values to functions. For example, the input formula $f(g(x),x)=e$ is viewed as a tree with $=$ as a root, f and g as internal nodes, and the elements x, x and e as leaves. Given a three element domain, the formula is instantiated to three clauses, one for each value of x. Consider one of the resulting clauses, $f(g(1),1)=e$. The following is a possible set of assignments that will completely reduce the clause. The assignment $e=0$ results in $f(g(1),1)=0$. No new units are derived from the assignment. The next assignment $g(1)=2$ gives $f(2,1)=0$, which is a new unit. If this new information does not conflict with any other clauses in the search space, the clause is reduced.

Compared to the conversion-based method, this approach uses less memory for clause storage and spends less time processing the input. However, in this method, input encoding is not straight-forward. First-order clauses need to be stored in a very efficient manner to make processing and backtracking fast. Another difficulty lies in the fact that the search algorithm may need to be tailored to each kind of problem to be efficient. For example, if a function is specified as a bijection, extra steps need to be taken by the search algorithm to guarantee that the condition is satisfied. The third, and major, problem is in the processing of negative constraints of the form $f(x,y)\neq z$.

Related Work

First-order model generation is an old problem. However, the first practical model generators have all been created in this decade. All of these programs are based on backtracking search algorithms.

FINDER (Finite Domain Enumerator), the first practical model generator, was developed in 1993[7]. The next two implementations, SATO (SAtisfiability Testing Optimized)[8] and LDPP linear-list-based Davis-Putnam Prover)[8], came out in 1994. Another program released the same year was MACE (Models And CounterExamples)[5]. Finally, SEM (System for Enumerating finite Models) was released in 1995[9, 10, 12]. The first four programs use the conversion-based method, while SEM uses the direct approach. Recently, nonexhaustive search techniques have been successfully applied to some problems[11].

THE MODEL-GENERATION PROGRAM

The new first-order model generating program, called Stork, was designed to eliminate the time- and memory-consuming phase of the previous model generators. The phase of concern is the conversion of first-order input statements to propositional clauses. We avoid this stage by converting the first-order input statements to first-order clauses and storing the clauses directly.

Data Representation

Three major data structures were designed. One structure is used to store information about functional and relational symbols in the input formulae. This structure contains a pointer to a table, which has to be filled with values. Every cell of the table corresponds to the set of domain values on which the function has to be evaluated. Each cell contains a pointer to clauses in which the symbol and its arguments occur. The clause setup is similar to that in MACE[5]. Every clause consists of a header and several trees, one tree for every literal. A header records how many of these literals have been reduced completely.

Splitting

Selection of the next assignment for the search tree is called splitting. We combine two techniques to make splitting efficient. First, for every cell in the table, we keep a list of domain elements that can be values in the cell. We also keep a counter of those elements. A similar technique is used in SEM[12].

The second heuristic for splitting is based on the following observations from the experiments with nonabelian groups. The search is faster if more new units are derived from each assignment. The search is also faster if the conflicting assignments are detected as soon as they are processed or shortly thereafter, i.e. few other assignments after. These favorable conditions arise if the expression trees are reduced beginning at the leaves. Relational operators are always at the root of the tree, so they should be assigned last. In the problems with multiple function symbols, such as group problems, the function symbols with lower number of arguments occur deeper in the tree (constants, for example, which are functions with zero arguments are always the leaves.) Combining these observations, we introduce a method called "weight-based splitting." Every symbol has an associated weight. Symbols of lower weight are assigned

problem	Stork	MACE	SEM
group.6	0.07	3.12	0.07
.8	0.14	*	0.12
ring.3	0.03	0.29	0.03
.4	0.09	2.09	0.02
.5	0.12	10.39	0.04
.6	0.20	*	0.07
ring_unit.3	0.05	0.27	0.05
.4	0.07	1.88	0.04
.5	0.11	9.06	0.05
.6	0.30	33.71	0.08
.7	0.82	*	0.11
tba.3	0.10	1.16	0.05
.4	0.29	17.33	0.09
.5	0.79	*	0.25

Table 1: Performance Comparison on Simple Algebraic Problems — Time in Seconds

problem	Stork	MACE	SEM
group.6	516	2937	63
.8	811	*	159
ring.3	426	294	63
.4	591	882	95
.5	859	4402	159
.6	1256	*	287
ring_unit.3	430	294	63
.4	596	296	95
.5	865	2058	191
.6	1263	9681	287
.7	1815	*	447
tba.3	692	587	95
.4	1919	7918	319
.5	5198	*	926

Table 2: Performance Comparison on Simple Algebraic Problems — Memory Usage in Kilobytes

first. When a symbol is selected for assignment, the cell with the smallest possible number of values is selected from the symbol function table. The scheme for weight assignment is open to experimentation. In our approach, the weight of a function symbol is equal to the number of its arguments. The weight of a relational symbol is ten times the number of its arguments.

Search Algorithm

The search algorithm used in this program is a straightforward recursive backtracking exhaustive procedure. It can be viewed as a tree where every node corresponds to a triple of an operator, its arguments, and a value, such as $\{f,(x,y),z\}$. The edges in the tree correspond to the assignments: the right edge represents an assignment $f(x,y)=z$, while the left edge represents $f(x,y)\neq z$. Each leaf node is an assignment that completed a successful model or created a partial solution that does not satisfy at least one constraint. Each assignment is recorded in the expression trees where $f(x,y)$ occurs. This action might result in new information. For example, when assigning $e=0$ in the clause $f(e,1)=1$, we obtain a new unit $f(0,1)=1$. All new units must be processed, that is, the assignments have to be performed, before the search continues to the next level. If any assignment conflicts with a constraint, the algorithm backtracks.

EXPERIMENTAL RESULTS

The literature[5, 6, 8] suggests that the conversion-based model generators have similar time and memory usage performance. Therefore, it suffices to compare Stork to only one representative of this class of programs, namely MACE.

configuration	squares removed
1	(0,1),(7,7)
2	(1,2) ,(1,3) ,(2,3),(3,3)
3	(0,0),(7,7)
4	(0,0),(0,2)

Table 3: Checkerboard Puzzle Configurations

	cover?	MACE	Stork	SEM
1	yes	0.10	0.50	4.31
2	yes	0.11	0.49	0.23
3	no	40.52	268.74	>3 hours
4	no	0.84	5.62	>2.5 hours

Table 4: Performance Comparison on Mutilated Checkerboard Puzzle

On small algebraic problems we accomplish our goal of reduced time and memory usage, compared to MACE. At the same time, Stork performs on par with SEM timewise, but uses more memory than SEM. Tables 1 and 2 display execution statistics for small problems of finding a nonabelian group, a ring, a ring with unity, and a ternary boolean algebra. The number following the problem label is the domain size. On some larger domains MACE runs over the 16,000K memory limit, as indicated by an asterisk.

The mutilated checkerboard puzzle was presented by McCarthy[4] in 1964 as a challenge to automated reasoning programs and as a measure of their "intelligence." This problem is of interest because the finding of the cover can be viewed as model genera-

	Stork		MACE	
	time(s)	mem(K)	time(s)	mem(K)
qg5.8	0.06	462	0.31	601
.9	0.27	506	0.53	607
.10	2.82	555	0.86	1202
.11	27.44	2075	1.57	1504
qg6.7	0.03	350	0.15	303
.8	0.03	367	0.24	308
.9	0.04	386	0.40	607
.10	0.69	408	0.72	909
.11	8.94	1018	2.80	1211
.12	162.82	11884	43.28	1809
qg7.8	0.04	420	0.23	308
.9	0.13	453	0.37	607
.10	6.86	783	0.64	909
.11	out of	memory	1.91	1211

Table 5: Performance on Quasigroup Existence Problems

	splits in		NCP		
	MACE	Stork	splits	time	mem
qg5.8	6	41	4	0.08	475
.9	14	292	19	0.20	524
.10	37	3471	75	0.63	580
.11	75	39343	14	0.20	641
qg6.7	2	9	5	0.03	359
.8	5	15	11	0.03	380
.9	12	44	39	0.07	404
.10	58	1359	1212	1.26	432
.11	537	15789	13113	14.94	1049
.12	7306	273120	224938	272.03	10165
qg7.8	0	22	7	0.07	433
.9	2	181	40	0.13	471
.10	39	13186	1589	3.31	514
.11	291	-	38676	83.51	2027

Table 6: Performance on Quasigroup Existence Problems

tion. This problem is particularly interesting to us because of its structure. The main advantage of the direct generator comes into play when the input contains functional operators and expression trees are deep. In McCarthy's formulation[4] the only operators are relations, and the input statements are already in the flat form. That means that every expression tree is of depth 1. We would like to see if the direct approach is applicable in these conditions. Table 3 describes four configurations we tried by listing the squares removed from an eight-by-eight board. The third configuration is the original problem. Execution times are presented in Table 4. Stork solves the problems, and performs consistently 5 to 7 times slower than MACE. It appears, however, that SEM does not do well in this kind of situation. Its performance improves, as it did for configuration 2, when more squares are removed from the board. That is, SEM's performance depends on the number of constraints.

NEGATIVE CONSTRAINT PROPAGATION HEURISTIC

We attempted to use Stork to solve quasigroup existence problems introduced in [1] and addressed in [5, 6, 9]. Our program performed surprisingly poorly. The investigation revealed that the following difference between two approaches is responsible. A conversion-based model generator derives the same amount of new information from positive and negative functional units, i.e. expressions of the form $f(x,y) = z$ and $f(x,y) \neq z$. A direct model generator uses the positive units to reduce expression trees, but records the information from the negative units only in the function table. As a result, a direct generator reduces the search space more slowly than a conversion-based one. That means that a direct generator performed more splits. Table 5 illustrates the effect the problem has on execution time and memory usage. Table 6 illustrates the differences in the number of splits. The leftmost column in these tables is the problem name as referenced in [5] and [6], followed by the domain size.

The Heuristic

This discovery prompted the implementation of the mechanism, referred to as negative constraint propagation heuristic(NCP), to record units of the form $f(x,y) \neq z$ in the expression trees in order to derive as much new information from such a unit as possible.

Without NCP heuristic, information is propagated only up the expression tree. The object of NCP is achieved by propagating the information down, as well as up, the expression tree. Consider the following example. Suppose the search space contains a clause of one positive literal, namely $f(f(0,1),2) = f(0,f(1,2))$ and that the assignment $f(2,2) = 1$ has already been performed. Suppose further that the next assignment to be done is $f(0,1) = 2$. When we perform this assignment in the given literal, it is reduced to $1 = f(0,f(l,2))$. Without NCP, this is as much information as can be derived from this tree. Since $f(1,2)$ has not been assigned a value yet, we essentially have the $1 = f(0,x)$. This is where NCP comes into play. We have just stated that $f(0,1) = 2$. Therefore, the value of x in the expression above can-

not possibly be 1. That is we have a new unit, namely $f(1,2) \neq 1$. This unit is recorded in the function table and also in the expression trees whenever we have $f(1,x) = 1$ or $f(y,2) = 1$. We know now that $x \neq 2$ and $y \neq 1$.

To accomplish the tasks described above we create additional tables for each symbol in the input. In the original symbol tables, we are trying to determine the value of the function given its arguments. In the new tables we are looking for one of the arguments of the function, given all other arguments and the value. Every position in the new tables contains a list of possible and impossible values and points to the occurrences of the corresponding combination of a function, arguments and a value in the expression trees.

Application of the Heuristic

The NCP heuristic reduces the number of splits, but it is still not as efficient as a conversion-based generator. The heuristic always causes fewer splits, but to have any effect on execution time, the reduction must be at least by an order of magnitude. A twenty-five percent reduction merely compensates for the time and memory spent executing the algorithm. As Table 6 illustrates, dramatic results can be achieved with the heuristic, as for problem qg5. In our opinion, the structure of the original problem, in which the expression trees are highly unbalanced, is at least somewhat responsible for such improved performance. Input for problem qg6 is formed of very regular trees, so it is not surprising that the heuristic does not give better results. (SEM has a very difficult time with this problem as well, taking 166 seconds to solve the problem for the domain of size 11 and almost three hours for the domain of size 12. In general, the authors of SEM successfully address the problem of negative constraints by implementing the least number heuristic[9], designed to reduce the number of isomorphic partial models examined by the program.)

CONCLUSION

Two methods of finite first-order model generation are available. The conversion-based method is more universal, but has memory and time constraints. The direct method is applicable to a fewer number of problems, but can achieve, with appropriate heuristics, dramatic results.

ACKNOWLEDGMENTS

This work was supported by the Department of Computer Science of the University of Missouri-Rolla and by the Office of Scientific Computing, U.S. Department of Energy, under contract W-31-109-Eng-38.

References

[1] F. E. Bennett and L. Zhu. Conjugate-orthogonal latin squares and related structures. In Jeffrey H. Dinitz and Douglas R. Stinson, editors, *Contemporary Design Theory: A Collection of Surveys*, pages 41–96. John Wiley & Sons, Inc., 1992.

[2] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the Association for Computing Machinery*, 7(3):201–215, 1960.

[3] Jun Gu. Local search for satisfiablity (SAT) problem. *IEEE Transactions on Systems, Man, and Cybernetics*, 23(4):1108–1129, 1993.

[4] John McCarthy. A tough nut for proof procedures. Stanford Artificial Intelligence Project, July 1964.

[5] William W. McCune. A Davis-Putnam program and its application to finite first-order model search: Quasigroup existence problems. Argonne National Laboratory, September 1994.

[6] 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.

[7] J. Stanley. FINDER version 3.0 notes and guide. Centre for Information Science Research, Australian National University, 1993.

[8] Hantao Zhang and Mark E. Stickel. Implementing the Davis-Putnam algorithm by tries. University of Iowa, 1994.

[9] Jian Zhang and Hantao Zhang. SEM: a system for enumerating models. In *Fourteenth International Joint Conference on Artificial Intelligence*, pages 298–303, 1995.

[10] Jian Zhang and Hantao Zhang. SEM user guide. Department of Computer Science, University of Iowa, 1995.

[11] Jian Zhang and Hantao Zhang. Combining local search and backtracking techniques for constraint satisfaction. *Forthcoming*, 1996.

[12] Jian Zhang and Hantao Zhang. The model generator SEM (system description). *Forthcoming*, 1996.