2005－HPC－103（16）
2005／8／4

論理式の充足可能性問題における変数の依存関係に基づく効率的な変数決定順序

鴨志田　良和[†]　田浦　健次朗[†]　近山　隆[†]

論理式の充足可能性問題 (SAT 問題) を解くには，論理式の中の変数をある順番で選んで真理値を代入して充足されなければ別の組み合わせを選び，可能なすべての場合をチェックするアルゴリズムが広く用いられている．このようなアルゴリズムでは，どのような順序で変数値を決定するかが効率を大きく左右する．我々の提案する変数の順序づけの方法は，変数間の依存関係を計算し，他の変数に依存する変数よりも，他の変数に依存しない独立な変数の値をより早く決定する．この方法は，SAT 問題の応用分野のひとつであるモデル検証における，状態遷移モデルの構造的な性質を捉えており，8 ビットの乗算回路を用いた実験では，高速な SAT solver である zChaff を 200 倍近く高速化することができた．

A Variable Dependency-Based Decision Strategy for Boolean Satisfiability Problems

Yoshikazu Kamoshida,[†] Kenjiro Taura[†]
and Takashi Chikayama[†]

Many decision procedures for Boolean Satisfiability problems solve a instance by searching assignments to its variables. They repeatedly select a free (unassigned) variable based on some heuristics, and assign a truth value to it. Such algorithms are highly dependent on the order of selecting variables. In this work, we propose a method for a variable ordering reflecting dependencies among variables, which gives priority to independent variables, rather than dependent variables of which values can be indirectly determined by assigning truth values to other variables. Our method calculates the order of variables statically, and can be given as a hint to existing state-of-the-art SAT solvers easily. Our method captures structured properties of instances which are derived from state transition models, and experimental results show that the fast SAT solver "zChaff" which incorporates our decision strategy runs about 200 times faster than the original version on the instance derived from 8-bit integer multiplier.

1. Introduction

The Boolean Satisfiability (SAT) problem consists of determining a assignment to variables which satisfies a given boolean function, or determining that no such assignment exists. SAT is not just a theoretically interesting problem as one of NP-complete problems, but has many practical application domains such as designing logical circuits and verifying softwares. In recent years we have seen significant growth and success in research on search-based SAT solvers. Non-chronological backtracking proposed by Silva et al.[7] greatly improved efficiency of the SAT solvers for structured instances, which are derived from real-world applications. Then optimized solvers such as SATO[8], zChaff[6] and BerkMin[5] are proposed, which can solve some of larger SAT instances generated from industrial applications with tens of thousands of variables.

The order of selecting variables to be assigned next is important factor for solving SAT problems efficiently. Different branching heuristics may produce drastically different sized search trees for the same basic search algorithm. We may use dependencies existing between variables in SAT to decide variables. However, because they are very complex in general, it is not possible to use them easily. For this reason many SAT solvers decide the order of variables with relatively simple criteria such as frequency of appearance in the boolean formulas.

In this work, we define more limited, stronger dependencies among variables, and make it possible to easily distinguish whether a variable in a formula is likely dependent to other variables or independent. We apply this technique to decision algorithm

† 東京大学
University of Tokyo

for SAT solvers. Dependencies described later capture properties of state transition models, of which all state are determined if nondeterministic initial states are determined. Therefore, our method for ordering variables is well suitable for structured instances derived from application domains to which can be applied such state transition models, for example, software model checking.

This paper is organized as follows: In section 2, the definition of *CNF* and the basic algorithm to solve SAT problems are introduced. Then section 3 describes our method to order variables based on dependencies of variables. Section 4 presents our preliminary experiments examine the effectiveness of our method. And section 5 closes the paper with some remarks for future work.

2. SAT Problem

2.1 Problem Specification

In this paper, a boolean function f for a SAT problem is given in conjunctive normal form(*CNF*), for example,

$$(a \vee b \vee \neg c) \wedge (b \vee c) \wedge (a \vee \neg b \vee c) \wedge (\neg a \vee \neg b \vee \neg c) \quad (1)$$

This form consists of the logical AND of one or more *clauses* (e.g. "$(a \vee b \vee \neg c)$"), which consist of the logical OR of one or more *literals*. The literal comprises the fundamental logical unit in problems, being an instance of a variable or its negation. All Boolean functions can be described in the *CNF* format. For f to be satisfied, each clause must be satisfied.

To assign 1 (or TRUE) to x is to delete all clauses which contain the literal x and to remove the literal $\neg x$ from the clauses which contain the literal $\neg x$. To assign 0 (or FALSE) to $\neg x$ is to delete all clauses which contain the literal $\neg x$ and to remove the literal x from the clauses which contain the literal x. To assert the literal l is to assign the truth value to the variable to make l be TRUE. On removing the literal from a clause, the size of the clause (number of literals in the clause) may become zero. Such a clause is called a "empty clause." The formula of which one of the clauses is empty is unsatisfiable because we cannot satisfy the empty clause. Similarly, a clause of size one is referred to as a "unit clause."

```
while ( true ) {
  if ( ! decide () ) { // if no free vars
    return SATISFIABLE;
  }
  while ( deduce () == CONFLICT ) {
    blevel = analyze_conflicts ();
    if ( blevel == 0 ) {
      return UNSATISFIABLE;
    } else {
      back_track ( blevel );
    }
  }
}
```

Fig. 1 Pseudo code of Backtrack Search for SAT solver

Each satisfiable formula has at least one model, i.e. a way of deciding the truth value of every variable which let a given *CNF* evaluate to TRUE.

2.2 SAT solvers

SAT solvers are programs which implement the algorithms to solve SAT problems. Although the above example (1) is very small, the formulas arising from the real-world applications may indeed contain thousands of variables, and millions of clauses. Most of SAT solvers employ one of two main strategies to solve problems. Namely, SAT solvers are divided into two types: complete solvers and stochastic solvers. In this paper, we consider mainly on backtrack search-based complete solvers, which can correctly decide unsatisfiability of the formula.

2.3 Backtrack Search

Backtrack search algorithm for SAT problems was originally proposed by Davis et al.[4] in 1960's, and is the basis of most of the existing complete SAT solvers. This algorithm is described as the pseudo-code fragment (augmented for non-chronological backtracking[7]) in **Figure 1**.

The function `decide` chooses a free variable, which is not assigned a truth value, based on various heuristics. Each decision has a decision level associated with it.

The function `deduce` simplifies a formula propagating the effect of the assignment to the variables. After making a decision, some clauses may become unit clauses. The literal in a unit clause must be

asserted to be TRUE to avoid an immediate contradiction. This assignment may results other clauses to be unit, so this rule have to be applied repeatedly until no unit clause exists or conflict is encountered. All variables assigned as a consequence of implications of a certain decision will have the same decision level as the decision variable.

If conflict is encountered (some clauses become empty clause), then the `analyze_conflicts` function is called to analyze the reason for the conflict and to resolve it. It also obtains some knowledge from the current conflict (this is referred to as conflict directed learning), and returns a backtracking level to resolve this conflict. The returned backtracking level indicates the wrong branch decision made previously and `back_track` will undo the bad branches to resolve the conflict. A zero backtracking level means that a conflict exists even without any branching. This indicates that the problem is unsatisfiable.

3. Variable Dependency-based Decision Ordering

Currently, state-of-the-art SAT solvers select a variable and its value to be assigned with some heuristics. Most solvers select variables in dynamic way which allow to change the variable to be selected according to states of present solvers.

On the contrary, our method computes an order of variables statically based on dependencies among variables. This is not intended to be invoked every time a decision is made, but to be invoked as preprocessing of a SAT instance. The result of computation is passed to a SAT solver. The SAT solver is modified to receive a list of ordered variables and decides a variable according to the order in the list while any free variables exist in the list. We do not include variables into the list which have no dependencies we can detect. Therefore, if all variables in the list are assigned, the SAT solver decides a variable in the original way to the solver.

Our algorithm finds independent variables, which can be assumed to be dependent on no other variables (the definition of dependency is described in 3.1). It is important to decide independent variables rather than dependent variables, of which values are implied by other assignment. This is because we can reduce a number of times to decide variables by deciding independent variables.

The variable ordering reflecting dependencies among variables is done by following step:

(1) Extract dependencies among variables from a formula

(2) Resolve multiple and mutual dependencies

(3) Find variables which can be assumed to be dependent on no other variables(independent variables)

(4) Sort the list of independent variables from the variable that is depended by more variables

It is not necessary to decide the order of all dependent variables because their values are implied if values of independent variables are decided.

In later in this section, we describe the definition of dependencies and the method to obtain independent variables in detail.

3.1 Dependency among Variables

In a SAT instance, each variable in a clause has a dependency that its value is decided by a unit clause rule if the values of the other variables are decided to specific values. However it is very difficult to use naively this fundamental notion of dependency because each dependency is complexly intertwined. Therefore, we introduce stronger definition of dependencies which is simple to apply, and decide the order of variables based on these dependencies.

Our definition of dependency is inspired by state transition model. If its initial state is given, the following state is decided one after another. Similarly, we find a variable x and a set of variables $a_1, ..., a_m$ such that the value of x is determined at any assignment to $a_1, ..., a_m$.

Here, we define a dependency among variables as follows:

Definition 1 (Dependency among Variables) A variable x is dependent on variables $a_1, ..., a_m$ if the value of x is always implied when any assignments to each of $a_1, ..., a_m$ are made. Later in this paper, we write $[\ x \leftarrow a_1, ..., a_m\ ]$ to represent such a dependency.

Let g be a set of clauses which include a variable x and do not include any variables other than $x, a_1, ..., a_m$, and h be a formula made from g by removing all occurrence of x. The relation $[\ x \leftarrow a_1, ..., a_m\ ]$ is equivalent to unsatisfiability

$$\begin{array}{l}(\ a\ \vee\ \neg c)\\(\ b\ \vee\ \neg c)\\(\neg a\ \vee\ \neg b\ \vee\ c)\end{array}\quad\Longrightarrow\quad[\ c\ \leftarrow\ a,\ b\ ]$$

Fig. 2 Example of the dependency among variables

$$\begin{array}{l}(\ a\ \vee\ b\ \vee\ c)\\(\ a\ \vee\ \neg b\ \vee\ \neg c)\\(\neg a\ \vee\ b\ \vee \neg c)\\(\neg a\ \vee \neg b\ \vee\ c)\end{array}\quad\Longrightarrow\quad\begin{array}{l}[\ a\ \leftarrow\ b,\ c\ ]\\[\ b\ \leftarrow\ c,\ a\ ]\\[\ c\ \leftarrow\ a,\ b\ ]\end{array}$$

Fig. 3 Example of symmetric dependencies

of h. The variables included in h are $a_1, ..., a_m$. If h is unsatisfiable, any combinations of assignments to $a_1, ..., a_m$ produce at least one empty clause. In other words, at least one unit clause is produced from g under any combination of assignments to $a_1, ..., a_m$, and then x is implied to either positive or negative value.

For example, if we remove c from three clauses in **Figure 2**, we get "$(\ a\ )(\ b\ )(\neg a\ \vee\ \neg b)$." This formula is obviously unsatisfiable, therefore $[\ c\ \leftarrow\ a,\ b\ ]$ holds.

3.2 Mutual Dependency

Variables can depend on each other under our definition of variable dependencies. For example, two clauses $(\ a\ \vee\ \neg b), (\neg a\ \vee\ b)$ express equivalence relation. If one of two values in the equivalence relation is decided, the other value is decided. In addition, three clauses in **Figure 3** also represents symmetric dependencies among three variable a, b, c. Whenever either two values of a, b, c are decided, implication for the value of the remainder is made. In general, two variables x, y have a mutual dependency if both $[\ x\ \leftarrow\ y,\ a_1\ , ...,\ a_m\ ]$ and $[\ y\ \leftarrow\ x,\ b_1\ , ..., b_n\ ]$ hold. In addition, we say that these two rules are mutually dependent. On the other hand, a dependency rule which does not have any mutually dependent rule is referred to as a unique rule. To arrange variables by analyzing dependencies statically it is also necessary to give a order to each mutual dependent variable by some means. To solve this problem, we introduce two heuristics: eliminating redundant dependencies and frequency-based ordering.

3.2.1 Eliminating Redundant Dependencies

If a variable is dependent under more than one

$$\begin{array}{ll}&[\ a\ \nleftarrow\ x\ ]\\&[\ b\ \nleftarrow\ x\ ]\\(1)&[\ x\ \leftarrow\ a,\ b\ ]\\(2)&[\ x\ \leftarrow\ y\ ]\\(3)&[\ y\ \leftarrow\ x\ ]\end{array}\quad\Longrightarrow\quad\begin{array}{ll}&[\ a\ \nleftarrow\ x\ ]\\&[\ b\ \nleftarrow\ x\ ]\\(1)&[\ x\ \leftarrow\ a,\ b\ ]\\(3)&[\ y\ \leftarrow\ x\ ]\end{array}$$

Fig. 4 Elimination of Redundant Dependencies

dependency rules, we say that redundant dependencies exist. For example, when two dependencies, $[\ x\ \leftarrow\ a,\ b\ ,\ c\ ]$ and $[\ x\ \leftarrow\ y,\ z\ ]$ exist, it is necessary that either a, b, and c or y and z are assigned to determine the value of x. So, these two dependency rules are redundant.

Our first heuristic rule for simplifying mutual dependencies among variables is elimination of redundant dependencies. When one of the rules which are mutually redundant is a unique rule, we remove the other redundant rules expecting some mutual dependencies among variables are resolved. The elimination process may make some rules be unique, so this process should be applied repeatedly.

In **Figure 4**, rules $(1), (2)$ are redundant, rules $(2), (3)$ are mutually dependent, and rule (1) is a unique rule. In this case, we eliminate (2) to simplify dependencies.

3.2.2 Frequency-based Variable Ordering

In the case that mutual dependencies remain after elimination of redundant rules, the second rule for simplifying is applied. This rule is a heuristic ordering based on the power to decide values of other variables. We count the frequency of each variable that appears on the being depended side (the right-hand side of dependency rules). Variables appearing at higher frequencies are assumed to have higher power to decide value of other variables and we delete the dependency rules for which these variables depend (appear on the left-hand side).

3.3 Ordering Independent Variables

After analyzing dependencies among variables, each variable is in either of the following state:

(1) It appears only on the being depended side

(2) It is dependent under a dependency rule

(3) It does not appear in any dependency rules

The variables in the first state are assumed to be independent. We obtain the list of the independent variables by sorting them by the power to decide

```
int main () {
  unsigned char a, b;
  unsigned int result = 0, i;
  a = nondet_uchar ();
  b = nondet_uchar ();
  for ( i = 0; i < 8; ++ i ) {
    if ( ( b >> i ) & 1 ) {
      result += ( a << i );
    }
  }
  assert ( result == a * b );
  return 0;
}
```

Fig. 5 Source Code of 8-bit Integer Multiplier in C

described above.

4. Experiments

To examine the effectiveness of our method, we used a *CNF* formula generated from a multiplier of 8-bit integer. It was generated using cbmc, ANSI-C Bounded Model Checker[3]. This tool can generate boolean formulas in *CNF* from ANSI-C programs. The C program for input is shown in **Figure 5**. Cbmc allows to model user-input by means of non-deterministic choice functions. In this source code, `nondet_uchar` returns a nondeterministically chosen value of type `unsigned char`. Cbmc will evaluate all traces arising from all possible choices. Cbmc rewrites the input code before translate it into a boolean formula. First, cbmc unwinds all loops, and then rename variables on each assignment. Arithmetic operations are converted to equivalent bit vector equations. The 8-bit integer multiplier program is transformed into a *CNF* formula with 2,392 variables and 7,758 clauses.

All experiments were done on an Opteron 2.4GHz, running Linux 2.6.11 with 1G byte of physical memory. We used zChaff[6] version 2004.11.15 for the original SAT solver to be modified. The code for dependency analysis is a perl script of about 500 lines. This script finds all dependencies represented by clauses of which size were not over 4. We tried to solve a formula of the 8-bit multiplier described above and 58 structured instances randomly selected from the benchmark instances of sat-2002/2003 competition[1] of which solving times erre from 20 seconds up to 300 seconds with the original SAT solver. We can analyze shuffled instances in benchmarks because our dependency analysis algorithm is not dependent on variable IDs.

Table 1 shows the maximum decision level and the solving time of the original SAT solver, the time for analyzing dependencies among variables, the maximum decision level and the solving time of the SAT solver with the list of independent variables, and number of independent/dependent variables for each instance. "Speedup" equals to the solving time of the original SAT solver divided by the sum of the analyzing time and the solving time. "Timeout" at speedup column means that the solving time for the instance exceeded the timeout limit, 300 seconds.

22 of 59 instances could be extracted any dependency rules. We omitted instances from the table which could not be extracted dependency rules from at all because running times of such instances are the same as the original SAT solver.

The maximum decision levels of the solver with our variable ordering were smaller than those of the original solver in 14 of 22 instances. This indicates our variable ordering can effectively reduce the number of variables assigned independently at the same time. On the contrary, the number of independent variables were much larger than the maximum decision level. This is because some variables were mistakenly assumed to be independent for the limitation in size of examined clauses. Especially in "w10_60," the maximum decision level increased to three times that of the original solver as a result.

Improvements in speed were observed in 10 of 22 instances. It is remarkable that the solver ran 200 times faster in 8-bit integer multiplier.

5. Conclusion and Future Work

Our variable ordering method for solving SAT reflects dependencies among variables. Dependencies among variables for this method are based on the substitution relation on state transition models. We extract dependencies by statically analyzing given *CNF*s. This method yields better running

Instance Name	Variables	Clauses	S/U	Original Max DL	Original Solve Time	Analyze Time	Max DL	Solve Time	Indep. Variables	Dep. Variables	Speedup
8-bit multiplier	2392	7758	U	16	3886.5	1.5	16	18.7	63	2329	192.4
ferry12	4222	32148	S	437	142.2	3.5	449	0.1	2388	1786	40.1
ferry12u	4133	31464	S	426	27.9	3.3	457	0.0	2338	1748	8.3
lisa19_3_a	1201	6522	S	144	261.5	3.0	25	86.1	420	780	2.9
lisa20_0_a	1201	6522	S	135	80.3	3.0	26	21.4	420	780	3.3
lisa20_3_a	1201	6522	S	111	144.5	2.9	25	53.0	420	780	2.6
ezfact48_2	1729	10952	U	55	25.1	6.8	33	96.6	118	1611	0.2
ezfact48_3	1729	10952	U	116	122.7	6.8	34	66.7	118	1611	1.7
ezfact48_5	1729	10952	U	151	177.3	6.8	30	300.0	113	1616	timeout
ezfact48_6	1729	10952	U	104	244.2	6.8	43	220.6	117	1612	1.1
pyhala-braun-sat-30-4-01	5428	17782	S	260	69.9	3.9	42	24.5	285	5143	2.5
pyhala-braun-sat-30-4-03	5428	17782	S	274	20.1	4.0	34	24.2	297	5131	0.7
pyhala-braun-sat-35-4-01	7383	24247	S	307	38.0	5.4	48	4.0	381	7002	4.0
pyhala-braun-sat-35-4-02	7383	24247	S	312	29.8	5.4	55	34.9	408	6975	0.7
pyhala-braun-sat-35-4-04	7383	24247	S	309	22.9	5.4	57	230.4	381	7002	0.1
qg3-9	729	28215	U	93	39.5	4.7	93	39.5	108	108	0.9
qg6-12	1728	90324	U	241	29.0	15.4	241	29.1	12	396	0.7
Urquhart-s3-b2	44	398	U	27	24.9	0.06	29	300.0	13	6	timeout
Urquhart-s3-b10	43	340	U	27	87.4	0.07	27	300.0	16	8	timeout
w10_60	26611	83415	U	149	27.4	11.7	477	300.0	2037	22492	timeout
rand_net40-30-5	2400	7121	U	131	34.1	1.1	39	300.0	40	2360	timeout
rand_net40-30-10	2400	7121	U	111	26.5	1.1	40	300.0	40	2360	timeout

Table 1 Statistics for 8-bit multiplier and selected benchmark instances

times for some class of instances.

It is possible that we could not extract sufficient dependency rules for the instances which made no performance gain with our method. To extract more meaningful dependencies, we can examine larger clauses and exploit weaker dependency rules.

Independent variables obtained as a result of analyzing dependencies among variables tend to represent nondeterministic initial states of a state transition model. When we solve SAT instances derived from arithmetic circuit, some of independent variables may be originally expressed as a single word. The word-level model checking[2] verifies models by processing set of bits as a group with augmented BDD data structures. It may be interesting to consider if word-aware model checking techniques can be applied to implementation of SAT solvers as well as BDD manipulation.

参 考 文 献

1) Berre, D. L. and Simon, L.: The essentials of the SAT 2003 competition, *Proceedings of the Sixth International Conference on Theory and Applications of Satisfiability Testing (SAT2003) Lecture Notes in Computer Science 2919*, pp.452–467 (2003).
2) Clarke, E. M., Khaira, M. and Zhao, X.: Word level model checking–avoiding the Pentium FDIV error, *DAC '96: Proceedings of the 33rd annual conference on Design automation*, New York, NY, USA, ACM Press, pp.645–648 (1996).
3) Clarke, E. and Kroening, D.: Hardware Verification using ANSI-C Programs as a Reference, *Proceedings of ASP-DAC 2003*, IEEE Computer Society Press, pp.308–311 (2003).
4) Davis, M., Logemann, G. and Loveland, D.: A machine program for theorem-proving, *Communications of the ACM*, Vol.5, No.7, pp.394–397 (1962).
5) Goldberg, E. and Novikov, Y.: BerkMin: A Fast and Robust Sat-Solver, *DATE '02: Proceedings of the conference on Design, automation and test in Europe*, Washington, DC, USA, IEEE Computer Society, p.142 (2002).
6) Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L. and Malik, S.: Chaff: Engineering an Efficient SAT Solver, *Proceedings of the 38th Design Automation Conference* (2001).
7) Silva, J. and Sakallah, K.: GRASP – A New Search Algorithm for Satisfiability, *Proceedings of the International Conference on Computer-Aided Design* (1996).
8) Zhang, H.: SATO: an efficient propositional prover, *Proceedings of the International Conference on Automated Deduction (CADE'97), volume 1249 of LNAI*, pp.272–275 (1997).