Maxterm Covering for Satisfiability

Liangze Yin, Fei He, *Member*, *IEEE*,
William N.N. Hung, *Sr. Member*, *IEEE*,
Xiaoyu Song, *Sr. Member*, *IEEE*,
and Ming Gu

Abstract—This paper presents a novel efficient satisfiability (SAT) algorithm based on maxterm covering. The satisfiability of a clause set is determined in terms of the number of relative maxterms of the empty clause with respect to the clause set. If the number of relative maxterms is zero, it is unsatisfiable, otherwise satisfiable. A set of synergic heuristic strategies are presented and elaborated. We conduct a number of experiments on 3-SAT and k-SAT problems at the phase transition region, which have been cited as the hardest group of SAT problems. Our experimental results on public benchmarks attest to the fact that, by incorporating our proposed heuristic strategies, our enhanced algorithm runs several orders of magnitude faster than the extension rule algorithm, and it also runs faster than zChaff and MiniSAT for most of k-SAT ($k \geq 3$) instances.

Index Terms—Verification, satisfiability, maxterm covering, heuristics.

1 Introduction

BOOLEAN satisfiability (SAT) is to find if there is a true interpretation for a Boolean formula. The complexity of the problem grows exponentially with increasing number of variables. The 3-SAT problem is a well-known NP-complete problem [1]. Many real-world problems can be transformed into SAT problems and many of these problem instances can be effectively solved via satisfiability, such as testing [2], formal verification [3], [4], synthesis [5], nanofabric cell mapping [6], various routing problems [7], [8], [9], [10], [11], etc. Hence, the research for fast and efficient SAT solvers is an important and useful topic.

Satisfiability has been widely studied for decades. Many well-known and efficient SAT solvers have been published [12], [13], [14], [15]. There are two main kinds of SAT solvers: complete and incomplete solvers. Incomplete SAT solvers can be faster than complete SAT solvers for certain class of problems, but there is no guarantee of finding a solution. Incomplete SAT solvers can use local search or evolutionary techniques, such as WalkSAT [16] and GSAT [17]. Complete solvers may not be as fast as incomplete solvers for certain class of problems, but as long as the problem is satisfiable, complete solvers are guaranteed to find a solution for sufficient memory and runtime. A complete SAT solver can be used to prove that a problem is unsatisfiable. Most complete SAT solvers are developed based on the work of DP [18] and DLL [19], also called DPLL. Many well-known SAT solvers [13], [14], [15] are based on DPLL. Some successful improvements on DPLL include: conflict-driven clause learning, nonchronological backtracking, Boolean constraint propagation with "two-watched-literals", adaptive branching, and random restarts.

- *L. Yin, F. He, and M. Gu are with the School of Software, Tsinghua University, Room 1-118, FIT Building, Beijing 100084, China. E-mail: yinliangze@163.com, {hefei, guming}@tsinghua.edu.cn.*
- *W.N.N. Hung is with the Verification Group, Synopsys, Inc., 700 East Middlefield Road, Mountain View, CA 94043. E-mail: william_hung@alumni.utexas.net.*
- *X. Song is with the Department of ECE, Portland State University, PO Box 751, Portland, OR 97207-0751. E-mail: song@ece.pdx.edu.*

Manuscript received 25 Mar. 2010; revised 10 Sept. 2010; accepted 27 Nov. 2010; published online 9 Dec. 2010.
Recommended for acceptance by S. Shukla.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-2010-03-0206.
Digital Object Identifier no. 10.1109/TC.2010.270.

DPLL and most of the recent algorithms are based on the exploration of the variable space. In this paper, we propose an algorithm in terms of maxterm covering of the clause set. A portion of our idea is similar to the Extension Rule (ER)-based algorithm, which was introduced in [20] for theorem proving purposes. In that paper, they compute the satisfiability of a clause set by extending every clause of the set into maxterms using extension rule, and determine if the set contains 2^n clauses. To handle space complexity, they introduced an inclusion-exclusion principle to circumvent the problem. In our algorithm, we find the satisfiability of a clause set by computing the relative maxterms covered by the empty clause with respect to clause set T, and determine if the relative maxterm covering is $\emptyset$ or not.

We propose a set of optimization strategies for our approach. The experiments indicate that, after improvement by the optimization strategies, our algorithm runs several orders of magnitude faster than the ER Algorithm for all the problems, and 3 to 50 times better than zChaff [13], one to two times better than MiniSAT [15] for most of the k-SAT ($k \geq 3$) instances.

2 Theory

Most SAT solvers focus on solving Boolean formula in the conjunctive normal form (CNF). Suppose we are given a Boolean formula f over a set of Boolean variables. The CNF of f is simply a conjunction of clauses, where each clause is a disjunction of literals, and each literal is either a Boolean variable or the negation of a Boolean variable.

Given a CNF $(p_1 \vee \cdots \vee p_m) \wedge \cdots \wedge (q_1 \vee \cdots \vee q_n)$, we can represent it as a clause set $T = \{p_1 \vee \cdots \vee p_m, \ldots, q_1 \vee \cdots \vee q_n\}$. The satisfiability of a Boolean formula is equivalent to the satisfiability of a CNF. In what follows, we consider the satisfiability of a Boolean formula as the satisfiability of a clause set, where each clause in that set is a disjunction of literals.

A maxterm is a disjunction of literals, with exactly one literal for each variable in the formula. A principal conjunctive normal form (PCNF) is a CNF in which every clause (disjunction) is a maxterm. Given any Boolean formula, there is one and only one PCNF that is equivalent to it.

Theorem 1. *A Boolean formula is unsatisfiable if and only if its PCNF has 2^n maxterms, where n is the number of variables in the formula.*

Proof. Given a PCNF f with n variables which corresponds to the Boolean formula, each maxterm corresponds to a distinct row in the truth table where f evaluates to FALSE. There are 2^n rows in the truth table of f. Hence f is unsatisfiable if and only if the number of maxterms in f is equal to 2^n. □

Based on theorem 1, we can determine the satisfiability of any Boolean formula by computing the number of maxterms in its corresponding PCNF, and checking if this number is 2^n.

Definition 1. *The maxterms covered by a clause set $T = \{C_1, C_2, \ldots, C_m\}$ are the maxterms contained in the PCNF of T, denoted as $MC(T)$.*

In particular, if there is only one clause C in the clause set T, we denote the maxterm cover as $MC(C)$.

Given a clause C, suppose its PCNF is $(p_1 \vee \cdots \vee p_n) \wedge \cdots \wedge (q_1 \vee \cdots \vee q_n)$, then $MC(C) = \{p_1 \vee \cdots \vee p_n, \cdots, q_1 \vee \cdots \vee q_n\}$. Given a clause set $T = \{C_1, C_2, \ldots, C_m\}$, then

$$MC(T) = \bigcup_{i=1}^{m} MC(C_i).$$

Based on Theorem 1 and Definition 1, we have:

Corollary 1. *A clause set T is unsatisfiable if and only if the number of elements in $MC(T)$ is equal to 2^n.*

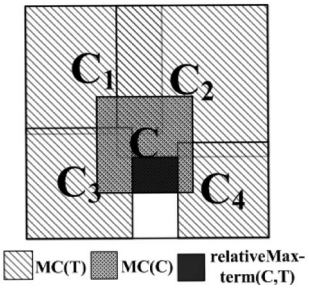

Fig. 1. Definition of $relativeMaxterm(C,T)$.

Theorem 1 and Corollary 1 can also be seen in [20], but to determine if the number of elements in $MC(T)$ is equal to 2^n, they tried to extend every clause into maxterms or use the inclusion-exclusion rule, while we compute the relative maxterms covered by empty clause with repect to clause set T as follows:

Definition 2. *The relative maxterms covered by a clause C with respect to a clause set T are the maxterms covered by C but not by T, denoted as* $relativeMaxterm(C,T) = MC(C) \setminus MC(T)$.

An example is shown in Fig. 1 for the clause set $T = \{C_1, C_2, C_3, C_4\}$.

Given a set of variables, we denote the set of all possible maxterms as the universal set. Note the maxterms covered by the empty clause ϵ is equal to the universal set. Based on corollary 1, determining the unsatisfiability of a clause set T is equivalent to deciding $MC(T) = MC(\epsilon)$. On the other hand, determining the satisfiability of a clause set T is equivalent to finding if there is any relative maxterm covered by the empty clause with respect to T, i.e.,

$$relativeMaxterm(\epsilon, T) \neq \emptyset.$$

This is the main idea of this paper. An example is shown in Fig. 2 where the clause set $T = \{C_1, C_2, C_3, C_4, C_5\}$ is satisfiable, as $relativeMaxterm(\epsilon, T) \neq \emptyset$.

The above analysis shows that the key point to solve the satisfiability problem is to find the relative maxterms covered by a clause C (note C can be an empty clause) with respect to a clause set T, i.e., $relativeMaxterm(C,T)$. We can obtain it using the following formula:

$$\begin{aligned} relativeMaxterm(C,T) &= MC(C) \setminus MC(T) \\ &= MC(C) \setminus \bigcup_{i=1}^{m} MC(C_i) \\ &= MC(C) \setminus MC(C_1) \setminus \cdots \setminus MC(C_m). \end{aligned}$$

Now we face the following question: given clauses C_1 and C_2, how to compute $MC(C_1) \setminus MC(C_2)$? We consider two lemmas.

Lemma 1. *If the literals in clause C_1 form a subset of the literals in clause C_2, i.e., $Literals(C_1) \subseteq Literals(C_2)$, where $Literals(C)$ gives the set of literals in clause C, then the maxterms covered by C_2 is a subset of the maxterms covered by C_1, i.e., $MC(C_2) \subseteq MC(C_1)$.*

The proof of Lemma 1 is trivial. For example, given a satisfiability problem with variables p, q, r, s, and clause $C_1 = p \vee \neg q$, $C_2 = p \vee \neg q \vee r$, then $Literals(C_1) = \{p, \neg q\}$, $Literals(C_2) = \{p, \neg q, r\}$, since $Literals(C_1) \subseteq Literals(C_2)$, so $MC(C_2) \subseteq MC(C_1)$.

Definition 3. *Two clauses are mutually independent if and only if they contain a complementary pair of literals.*

For example, clauses C_1 and C_2 are mutually independent if C_1 contains literal p and C_2 contains the complement literal $\neg p$.

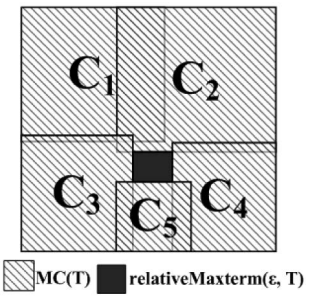

Fig. 2. A satisfiable example.

Lemma 2. *If clauses C_1 and C_2 are mutually independent, there is no intersection between the sets of maxterms covered by C_1 and C_2. i.e., $MC(C_1) \cap MC(C_2) = \emptyset$.*

Proof. If there is a complementary pair of literals p in C_1 and $\neg p$ in C_2, all maxterms covered by C_1 must contain p, and all maxterms covered by C_2 must contain $\neg p$. Hence $MC(C_1) \cap MC(C_2) = \emptyset$. □

Theorem 2. *Given two clauses that are not mutually independent, $C = q_1 \vee q_2 \vee \cdots \vee q_n$ and $C_1 = p_1 \vee p_2 \vee \cdots \vee p_m$, then $MC(C) \setminus MC(C_1) = MC(R)$, where*

$$R = \{C \vee \neg p_1, C \vee p_1 \vee \neg p_2, \ldots, C \vee p_1 \vee \cdots \vee \neg p_m\}. \quad (1)$$

Proof. To prove $MC(R) = MC(C) \setminus MC(C_1)$, we must prove the following three formulas: $MC(R) \subseteq MC(C)$, $MC(R) \cap MC(C_1) = \emptyset$ and $MC(C_1) \cup MC(R) = MC(C_1) \cup MC(C)$.

1. We show $MC(R) \subset MC(C)$. In (1), we see that, for any clause C' in R, $Literals(C) \subseteq Literals(C')$, Using Lemma 1, we know $MC(C') \subseteq MC(C)$. Hence, $MC(R) \subseteq MC(C)$.
2. We show $MC(R) \cap MC(C_1) = \emptyset$ here. In (1), we see that, for any clause C' in R, it has a complementary pair of literals with clause C_1. According to definition 3, C' is mutually independent from C_1. Using Lemma 2, we know $MC(C') \cap MC(C_1) = \emptyset$. Hence, $MC(R) \cap MC(C_1) = \emptyset$.
3. To show that $MC(C_1) \cup MC(R) = MC(C_1) \cup MC(C)$, we use the following steps:

$$\begin{aligned} C &= C \vee (\neg p_1 \wedge p_1) \\ &= (C \vee \neg p_1) \wedge (C \vee p_1) \\ &= (C \vee \neg p_1) \wedge (C \vee p_1 \vee (\neg p_2 \wedge p_2)) \\ &\vdots \\ &= R \wedge C_f, \end{aligned}$$

while $C_f = C \vee p_1 \vee p_2 \vee \cdots \vee p_m$. Hence,

$$MC(C_1) \cup MC(C) = MC(C_1) \cup MC(R) \cup MC(C_f).$$

Since the literals in C_1 are all contained in C_f, we can use Lemma 1 to obtain: $MC(C_f) \subseteq MC(C_1)$. Hence,

$$MC(C_1) \cup MC(R) \cup MC(C_f) = MC(C_1) \cup MC(R).$$

Therefore, $MC(C_1) \cup MC(C) = MC(C_1) \cup MC(R)$. □

Let R be the result of taking operation $MC(C) \setminus MC(C_1)$. Note R may be a clause set. Now we need to compute the maxterms covered by R but not by $T' = T \setminus C_1$. We use the following approach:

$$MC(R) \setminus MC(T') = \bigcup_{i=1}^{m} MC(C_i) \setminus MC(T') = \bigcup_{i=1}^{m} (MC(C_i) \setminus MC(T')).$$

At this point, we have established the core steps of our approach. Given a clause set T, we can compute $relativeMaxterm(\epsilon, T)$ using the following formula:

$$\begin{aligned} & relativeMaxterm(\epsilon, T) \\ &= MC(\epsilon) \setminus MC(T) \\ &= MC(\epsilon) \setminus MC(C_1) \setminus MC(T') \\ &= MC(R) \setminus MC(T') \\ &= \bigcup_{i=1}^{m} (MC(C_i) \setminus MC(T')) \\ &= \bigcup_{i=1}^{m} (relativeMaxterm(C_i, T')). \end{aligned} \tag{2}$$

In (2), $T = T' \cup C_1$, $MC(R) = MC(\epsilon) \setminus MC(C_1)$, and $R = \bigcup_{i=1}^{m} C_i$. From the formula, we can see that it is a recursive process, and our satisfiability algorithm is outlined in Algorithm 1. If $relMaxtermComp(\epsilon, T)$ returns 1, $relativeMaxterm(C, T) \neq \emptyset$, clause set T is satisfiable, otherwise it is unsatisfiable.

Algorithm 1. Basic Satisfiability Algorithm

```
1: procedure RELMAXTERMCOMP(clause C, clause set T)
2:     if (T == ∅) return 1; // SAT
3:     select a clause C1 from T;
4:     if (C and C1 are mutually independent) R = C
5:     else R = MC(C) \ MC(C1) // by Theorem 2
6:     T' = T \ C1;
7:     for each Ci ∈ R
8:         if (relMaxtermComp(Ci, T') == 1)
9:             return 1; // SAT
10:    return 0;
11: end procedure
```

3 Optimization Strategies

The critical step of our algorithm is to compute $relativeMaxterm(C, T)$. Our method compares clause C with one of the clauses of set T, and removes intersecting maxterms of the compared clause from the maxterms of C. However, the result R of removing maxterms of one clause from maxterms of another clause, must be represented by multiple clauses (using Theorem 2). Then for every clause C_i in R, we must compute $relativeMaxterm(C_i, T')$, respectively, where T' denotes the set of surplus clauses. If we keep doing this, the computation time of relative maxterms would increase exponentially. Hence, we have to find some optimization strategies for simplification, thus improving the effectiveness of the approach. We devise eight heuristic strategies as follows:

3.1 Strategy 1: Remove Independent Clauses

In computing $relativeMaxterm(C, T)$, such that $T = \{C_1, C_2, \ldots, C_m\}$, we have to compare C with one clause in set T, and remove maxterms from C that also appear in the compared clause, and then compute $relativeMaxterm(C_i, T')$ for every clause C_i in R. However, some clauses in the original clause set $T = \{C_1, C_2, \ldots, C_m\}$ may be mutually independent from clause C, so they have no intersecting maxterms with C. Therefore, we should first remove these clauses from the set T. Otherwise, if we leave them in T', and compare each clause in R (or the clauses produced by them) with these independent clauses, then it would waste tremendous amount of time.

3.2 Strategy 2: Look for Dominant Clauses

Definition 4. *Clause C_1 is the dominant clause of clause C_2 if and only if the literals of C_1 is a subset of the literals of C_2, i.e., $Literals(C_1) \subseteq Literals(C_2)$.*

For example, if we have two clauses, $C_1 = p \vee \neg q$ and $C_2 = p \vee \neg q \vee r$, then $Literals(C_1) = \{p, \neg q\}$, $Literals(C_2) = \{p, \neg q, r\}$. We have $Literals(C_1) \subseteq Literals(C_2)$. Hence, C_1 is the dominant clause of C_2.

Theorem 3. *If clause set $T = \{C_1, C_2, \ldots, C_m\}$ contains C_i which is a dominant clause of C, then $relativeMaxterm(C, T) = \emptyset$.*

Proof. Since clause C_i is a dominant clause of clause C, using Lemma 1, we have $MC(C) \subseteq MC(C_i)$. However, $MC(C_i) \subseteq MC(T)$. Therefore, $MC(C) \subseteq MC(T)$. Based on the definition of relative maxterms, we have $relativeMaxterm(C, T) = \emptyset$. □

With Theorem 3, before computing $relativeMaxterm(C, T)$, we can scan each clause in T. If we find a dominant clause of C from the scanned clauses, we can directly deduce that $relativeMaxterm(C, T) = \emptyset$.

As an example, suppose we have $C = p \vee q \vee r$, $T = \{q \vee s \vee t, \neg p \vee r \vee s, r \vee t \vee u, \neg r \vee s \vee v, p \vee q\}$, and we want to compute $relativeMaxterm(C, T)$. Since the set T contains clause $p \vee q$ which is a dominant clause of C, we can directly deduce that $relativeMaxterm(C, T) = \emptyset$. With Theorem 3, we may avoid a lot of clause comparisons, thus accelerating the solver.

3.3 Strategy 3: Look for Subdominant Clauses

Definition 5. *Clause C_1 is a subdominant clause of clause C_2 if and only if clause C_1 has one and only one literal that is missing from clause C_2, i.e., $Literals(C_1) \setminus Literals(C_2) = \{p_x\}$, where p_x is a literal.*

For example, if we have two clauses, $C_1 = p_1 \vee \neg p_2 \vee p_3$ and $C_2 = p_1 \vee \neg p_2 \vee p_4 \vee \neg p_5$, then $Literals(C_1) \setminus Literals(C_2) = \{p_3\}$. Hence, C_1 is a subdominant of C_2.

Theorem 4. *If clause C_i is a subdominant of clause C, then $MC(C) \setminus MC(C_i)$ can be expressed by one single clause $R = \{C \vee \neg p_x\}$ where p_x is the missing literal of C_i from C.*

Proof. Without loss of generality, suppose $C_i = p_1 \vee \cdots \vee p_m \vee p_x$, $C = p_1 \vee \cdots \vee p_m \vee q_1 \vee \cdots \vee q_n$, where C_i has only one literal p_x that is missing from C. With Theorem 2, $MC(C) \setminus MC(C_i) = MC(R)$, where $R = \{C \vee \neg p_1, C \vee p_1 \vee \neg p_2, \ldots, C \vee p_1 \vee \cdots \vee \neg p_m, C \vee p_1 \vee \cdots \vee p_m \vee \neg p_x\}$. Since clause C contains literals $p_1, \ldots, p_m$, the first m items in the above expression for R must be TRUE. As a result, it can be simplified to $R = \{C \vee \neg p_x\}$. □

Theorem 4 is very important because the resulting R contains only one clause, and with the new literal, we can use Strategy 1, 2, and 3 again. In our approach, we apply Strategy 1, 2, and 3 recursively, until they cannot be applied anymore. In the process, if Strategy 2 is applicable, we will have $relativeMaxterm(C, T) = \emptyset$ immediately. Even if Strategy 2 is not applicable, the number of clauses in the clause set T would be substantially reduced in general.

By using strategies 1 to 3, we can compute $relativeMaxterm(C, T)$ by the algorithm in Algorithm 2, where R is the expression that contains the remaining maxterms by removing maxterms from C that intersects with clause C_1 in T. If $relativeMaxterm(C, T) = \emptyset$, then the algorithm returns 0, otherwise it returns 1.

Algorithm 2. Computing $relativeMaxterm(C, T)$ with strategies 1 to 3

```
1: procedure RELMAXTERMCOMP(clause C, clause set T)
2:     // Prune clauses from T using strategies 1 to 3
3:     do {
4:         for each clause C_k ∈ T{
5:             if (Strategy 1 is applicable to C_k)
6:                 T = T \ {C_k}
7:             else if (Strategy 2 is applicable to C_k)
8:                 return 0;
9:         }
10:        for each clause C_k ∈ T{
11:            if (Strategy 3 is applicable to C_k){
12:                // the result R has only one clause
13:                C = MC(C) \ MC(C_k)
14:                T = T \ {C_k}; break;
15:            }
16:        }
17:    }while (Strategy 3 is applicable to some C_k);
18:    if (T == ∅) return 1; // SAT
19:    select a clause C_1 from T;
20:    R = MC(C) \ MC(C_1) // by Theorem 2
21:    T' = T \ C_1;
22:    for each C_i ∈ R
23:        if (relMaxtermComp(C_i, T') == 1)
24:            return 1; // SAT
25:    return 0;
26: end procedure
```

3.4 Strategy 4: Clause Ordering

Strategies 1 to 3 take advantage of common variables between clauses. Strategy 1 harness their mutual independence, i.e., same variable, opposite polarity. Strategies 2 and 3 use common literals, i.e., same variable, same polarity. If we can put the same variables together, we can better employ the above three strategies. This idea would require us to order the clauses based on their common variables.

Our ordering approach is as follows: first, we count the number of occurrences of each variable in all clauses, and store the statistical result in the array of var_num. Then we compute the weight W_C of each clause C. If the variables in C are $p_1, p_2, \ldots, p_n$, the weight of C is

$$W_C = \sum_{i=1}^{n} var\_num[p_i].$$

The clause ordering algorithm is outlined as follows:

1. Scan all clauses, gather the number of occurrences of each variable in all clauses. Store the result in the array of var_num.
2. Compute the weight W_{C_i} of each clause C_i.
3. Sort the clauses based on their weights W_{C_i}.

Overall, there is no change to the clauses in set T, so there is no change to the maxterms covered by T. Only the sequence (order) of clauses was rearranged. Consequently, the execution sequence (of $relativeMaxterm$ computation) is affected. We cannot guarantee optimal clause ordering here, but it is the best clause ordering method that we have found so far. The experimental result indicates that clause ordering speeds up the process by several orders of magnitude for problems with many clauses.

3.5 Strategy 5: Remove Single Polarity Variables

The above four strategies exploit common variables between clauses. But if none of the clauses in the set shares any common variable, or if the number of independent clauses is very small, it would be very difficult to utilize the above four strategies. To avoid this scenario, we need to ensure that each variable will occur with both positive and negative polarities in T. Our strategy is based on the following theorem:

Theorem 5. *Given a clause set T with n clauses, where variable p occurs in only one polarity. We can split T into two partitions: $\{C_1, \ldots, C_k\}$ and $\{C_{k+1}, \ldots, C_n\}$, where $\{C_1, \ldots, C_k\}$ are the clauses that contain variable p, and $\{C_{k+1}, \ldots, C_n\}$ are the clauses that do not contain variable p. The satisfiability of T is equivalent to the satisfiability of $\{C_{k+1}, \ldots, C_n\}$.*

Proof. Without loss of generality, assume that variable p occurs in the problem clause set with positive polarity only.

1. If $\{C_{k+1}, \ldots, C_n\}$ is satisfiable, we have ϕ as a set of variable assignments (a solution) that satisfies $\{C_{k+1}, \ldots, C_n\}$. We can add another assignment $p = 1$ to ϕ, then $\{C_1, C_2, \ldots, C_n\}$ can also be satisfied.
2. If $\{C_{k+1}, \ldots, C_n\}$ is unsatisfiable, $\{C_1, C_2, \ldots, C_n\}$ will also be unsatisfiable regardless of the value assignment to p.

Therefore, the above theorem is valid for both satisfiable and unsatisfiable cases. □

Based on Theorem 5, if any variable p occurs with only one polarity, we can directly remove all clauses that contain such variable. With this strategy, we can first remove all clauses that contain variables with single polarity occurrences. Thus the probability of clauses with variables of complementary polarities is much higher. Other approaches similar to this strategy were also used in DPLL [19] and in [21]. Although single polarity variables rarely occurred, its runtime cost is nearly zero, so it is anyway a good idea to adopt the strategy for some special cases.

3.6 Strategy 6: Compare Two Clauses Simultaneously

Strategy 1 to 3 can trim down the number of clauses in T substantially. Shall we then resort to Theorem 2 to process the remaining clauses in T? In fact, there are more chances for further optimization. If there are two mutually independent clauses in T, we can compare our clause C simultaneously with the two clauses.

Suppose we want to remove the maxterms from C that intersects with clauses C_1 and C_2, where C_1 and C_2 are mutually independent, we can obtain the result R as shown in (3).

$$\begin{cases} C_1 = t_{12} \vee p_1 \vee \cdots \vee p_m \\ C_2 = \neg t_{12} \vee q_1 \vee \cdots \vee q_n \end{cases} \Rightarrow R = \begin{cases} C \vee t_{12} \vee \neg p_1 \\ \vdots \\ C \vee t_{12} \vee p_1 \vee \cdots \vee \neg p_m \\ C \vee \neg t_{12} \vee \neg q_1 \\ \vdots \\ C \vee \neg t_{12} \vee q_1 \vee \cdots \vee \neg q_n \end{cases}. \quad (3)$$

It is easy to deduce (3). First, we apply Theorem 2 to compute R from C and C_1. Then we apply the method again for each clause in R with C_2, and we obtain the new R in (3). This scenario is important because removing maxterms from C that are intersecting with maxterms from C_1 and C_2 will attain a clause set R with at most $m + n$ clauses, which is much smaller than the general case with $(m + 1)*(n + 1)$ clauses in R. Therefore, it will substantially reduce the complexity of subsequent comparisons.

3.7 Strategy 7: Choosing Mutually Independent Clauses C_1, C_2

In Strategy 6, we compare clause C with a pair of mutually independent clauses C_1, C_2 simultaneously. In practice, there could be many pairs of mutually independent clauses. Which one should we select? In fact, Strategy 4 (clause ordering) also considers this question. The first pair of mutually independent clauses that we can find using the clause ordering of Strategy 4 would suffice. However, to reduce the number of subsequent comparisons, we want to minimize the number of clauses in the remaining clause set R after this comparison. In Strategy 6, the expression for R will have less clauses if there are more common variables shared by C_1, C_2 with C. Accordingly, we want to choose a pair of mutually independent clauses that has the smallest number of variables that are different from C.

For each clause in C_1 and C_2, there are at least two literals that are not included in C. Otherwise, if any clause has only 0 or 1 variable that is not included in C, we can apply Strategy 2 or 3, and eliminate this clause before Strategy 6. Hence C_1, C_2 in total has at least four variables that are not included in C (here we consider the common variable of C_1 and C_2 as different variables). Our algorithm for choosing the clause pair is shown in Algorithm 3.

Algorithm 3. Choosing Mutually Independent clauses C_1, C_2

```
1: procedure INDEPPAIRSELECT(clause C, clause set T)
2:     int min = ∞, diffs, i1, i2;
3:     for (i = 1 to ||T|| − 1){
4:       for (j = i + 1 to ||T||){
5:           if (clauses i, j are mutually independent){
6:               diffs = # vars in clauses i, j missing in C;
7:               if (min > diffs){
8:                   min = diffs, i1 = i, i2 = j;
9:                   if (min == 4) return ⟨i1, i2⟩;
10:              }
11:          }
12:      }
13:    }
14: end procedure
```

3.8 Strategy 8: Splitting the Universal Set into Multiple Partitions

Our basic idea is to determine whether the number of relative maxterms covered by the empty clause with respect to clause set T is 0 or not. In practice, we can pick k variables to split the universal set into 2^k partitions, and compute their relative maxterms with respect to clause set T, respectively. If there is any relative maxterm found in any partitions, the problem is satisfiable. Otherwise, the problem is unsatisfiable.

This strategy has two advantages. First, for both satisfiable and unsatisfiable cases, we can choose k variables with the most occurrences, and use Strategy 1, 2, and 3 to delete many clauses at once which accelerates the process. Second, for satisfiable cases, if we can first search optimally the partition that is most likely to find the satisfiable solution, then we can solve the problem without searching much of the remaining partitions. In Fig. 3, we split the universal set into $8*8 = 64$ partitions. (The actual search space is n-dimensional, where n is the number of variables. For the ease of presentation, we depict them all as 2D in the figure.) We must separately find the relative maxterms of each of these 64 partitions with respect to clause set T. If we can prioritize the search to first work on the dark region in the figure (the region with solutions), then we can quickly find the solution before searching other partitions.

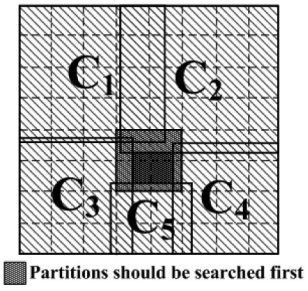

Fig. 3. Split the universal set into 2^k partitions, and prioritize the search to first work on the green region in the figure (the region with solutions).

Suppose the number of occurrences for any variable p_i in the clause set T is $var\_num[p_i]$, we can pick the k variables with the most occurrences, i.e., the variables with the largest $var\_num[p_i]$. Hence, we can eliminate more clauses from T at the very beginning, and we have more opportunities to find clauses that are applicable to Strategies 2 and 3. Thus, the efficiency is greatly improved.

After picking k variables, we search these 2^k partitions in the following manner:

1. Sort the k variables based on their occurrences in the problem from high to low as $p_1, p_2, \ldots, p_k$.
2. For each variable p_i, if the number of its occurrences with positive polarity is greater than those with negative polarity, we pick $p_i = 0$. Otherwise, pick $p_i = 1$. Then we procure a k-bit Boolean vector: $p_1 p_2 \ldots p_k$.
3. Search the 2^k partitions sequentially, where the ith partition is: $(p_1 p_2 \ldots p_k) \oplus i$, for $i = 0, 1, 2, \ldots, 2^k - 1$.

4 Experimental Results

In this section, we list the experimental results of our algorithm with different optimization strategies, and compare our SAT solver implementation with zChaff 2007, MiniSAT 2.0, and the ER solver. All of our experiments are conducted on a computer with Intel Core2 Duo CPU E7200 2.53 GHz with 2 GB memory.

4.1 Experiments on Optimization Strategies

It has been reported that some of the hardest SAT problems are the 3-SAT problems at phase transition region with density 4.3 (the ratio of the number of clauses to the number of variables) [22], [23], [24]. Accordingly, we use the benchmarks from SATLIB [25] in our experiments. These benchmarks provide 3-SAT problems at the phase transition region for variable count at $50, 75, 100, \ldots, 250$. In our experiments, we used the problems with 50, 175, 200, 225, and 250 variables, each with a group of 100 instances that are satisfiable, and another group of 100 instances that are unsatisfiable.

Table 1 shows the average runtime (in seconds) over the 100 problems in our results. The problem groups with name "uf" are all satisfiable, "uuf" are all unsatisfiable, the number following "uf" or "uuf" represents the number of variables in the corresponding group. The Basic Algorithm is the algorithm shown in Algorithm 1; version A is the algorithm with strategies 1, 2, and 3, i.e., algorithm shown in Algorithm 2; version B is the algorithm with strategies 1, 2, 3, and 4; and the algorithm with all strategies is denoted by version C. Since the Basic Algorithm easily blows up, we also randomly generated 100 SAT and 100 UNSAT instances for 30 variables (uf30 and uuf30) for better comparison. From Table 1, we can see that the capability of Basic Algorithm is limited, it can only handle 3-SAT problems for 50 variables, but when enhanced by Strategies 1, 2, and 3, it can handle problems with 200 variables. Strategy 4 also brings several orders of magnitude speedup to the algorithm, with it the algorithm can handle

TABLE 1
Experimental Results with and without Optimization Strategies

Problem Group	# problems	Satisfiability	Basic Algorithm	version A	version B	version C
uf30	100	SAT	0.015	0	0	0
uf50	100	SAT	16.19	0	0	0
uf175	100	SAT	–	1.237	0.055	0.028
uf200	100	SAT	–	5.582	0.153	0.089
uf225	100	SAT	–	40.42	0.539	0.273
uf250	100	SAT	–	–	2.197	1.049
uuf30	100	UNSAT	0.052	0	0	0
uuf50	100	UNSAT	61.38	0	0	0
uuf175	100	UNSAT	–	2.788	0.119	0.075
uuf200	100	UNSAT	–	16.44	0.393	0.236
uuf225	100	UNSAT	–	114.6	1.604	0.932
uuf250	100	UNSAT	–	–	7.159	3.771

TABLE 2
SATLIB 3-SAT Benchmark Results Compared with zChaff and MiniSAT

Problem Group	# problems	Satisfiability	zChaff time	MiniSAT time	MC time	zChaff/MiniSAT/MC
uf200	100	SAT	0.332	0.112	0.089	19 / 29 / 52
uf225	100	SAT	1.017	0.316	0.273	33 / 32 / 35
uf250	100	SAT	8.296	0.948	1.049	28 / 37 / 35
uuf200	100	UNSAT	1.571	0.279	0.236	0 / 18 / 82
uuf225	100	UNSAT	7.162	0.894	0.932	0 / 45 / 55
uuf250	100	UNSAT	55.17	2.922	3.771	0 / 80 / 20

TABLE 3
Random k-SAT Experimental Results Compared with zChaff and MiniSAT

Problem Group	Satisfiability	# problems	zChaff time	MiniSAT time	MC time	zChaff/MiniSAT/MC
4-SAT	SAT	37	76.637	3.837	1.466	6 / 6 / 25
	UNSAT	63	197.63	9.410	3.794	0 / 0 / 63
5-SAT	SAT	67	26.104	2.555	1.362	8 / 22 / 37
	UNSAT	33	97.144	7.202	4.460	0 / 1 / 32
6-SAT	SAT	54	2.072	0.777	0.482	16 / 14 / 24
	UNSAT	46	8.887	1.913	1.774	0 / 9 / 37

problems with 250 variables. Although other strategies did not play such a great role in the algorithm as strategies 1, 2, 3, and 4, they can also accelerate the algorithm by a factor of around two.

4.2 Maxterm Covering versus zChaff and MiniSAT

In Table 2, we compare our algorithm with zChaff and MiniSAT on 3-SAT problems for both SAT and UNSAT instances with 200, 225, and 250 variables. All these problems come from SATLIB. In this table, our Maxterm Covering algorithm is abbreviated as "MC." The columns of "zChaff time," "MiniSAT time," and "MC time" give the average runtime(s) of zChaff, MiniSAT, and Maxterm Covering, respectively, and "zChaff/MiniSAT/MC" lists the number of problems within the group that zChaff, MiniSAT, or MC win over other two solvers, respectively. For example, 19/29/52 means in the group of 100 problems, there are 19 instances zChaff runs the fastest, 29 instances MiniSAT runs the fastest and 52 instances Maxterm Covering runs the fastest.

In Table 3, we compare our algorithm with zChaff and MiniSAT on 100 randomly generated 4-SAT, 5-SAT, and 6-SAT instances, respectively. As presented in [26], 4-SAT, 5-SAT, and 6-SAT instances at the phase transition region are instances in which M/N are about 10, 21, and 44, respectively, where M and N denote the number of clauses and variables, respectively, in the corresponding instance. So to generate instances for 4-SAT, 5-SAT, and 6-SAT at the phase transition region, the numbers of variables for these three kinds of SAT problems are limited to be 100, 60, and 40, respectively, and the numbers of clauses are limited to be 1,000, 1,260, and 1,760 respectively.

From Tables 2 and 3, we can observe that the Maxterm Covering algorithm is 3-50 times faster than zChaff for both satisfiable and unsatisfiable problems. In fact, for all unsatisfiable instances, Maxterm Covering wins over zChaff. For satisfiable instances, as the search process may find a satisfying assignment by sheer luck, no SAT solver can win over another for all instances, but our algorithm is still faster than zChaff for most of the instances.

When compared to MiniSAT, from Table 2, we can see that the performances of these two solvers are similar with 3-SAT problems. Among 600 instances in Table 2, there are 241 problems MiniSAT runs fastest, and 279 problems Maxterm Covering runs fastest. While in Table 3, among the 300 random generated k-SAT instances, we can observe that, our solver ran 1.1-3 times faster than MiniSAT on average; and within each problem group, our solver won MiniSAT for more problems (especially for unsatisfiability cases).

Both zChaff and MiniSAT are popular SAT solvers based on DPLL and depth-first search of variables. The main difference between our algorithm and the other two algorithms is that our algorithm is based on clauses while zChaff and MiniSAT are based on variables. In our algorithm, we continually select clauses, and decompose the problem according to the selected clause, while in DPLL it is purely according to the selected variable. This is the fundamental difference between our approach and existing SAT algorithms. Tables 2 and 3 demonstrate the efficiency of our approach.

TABLE 4
Experimental Results Compared with Extension Rule

Problem Group	# problems	Satisfiability	CF	ER1 time	MC time	MC/ER1
(20, 40, 3)	100	SAT	0.228	41.256	0	100 / 0
(20, 120, 6)	100	SAT	0.653	23.812	0	100 / 0
(20, 200, 7)	100	SAT	0.768	0.17	0	100 / 0
(25, 400, 8)	100	SAT	0.775	1.375	0	100 / 0
(30, 600, 9)	100	SAT	0.791	5.125	0	100 / 0

4.3 Maxterm Covering versus Extension Rule

Maxterm Covering and Extension Rule both use Theorem 1 to determine the satisfiability of a clause set, i.e., a clause set is unsatisfiable if and only if the number of maxterms it contained is 2^n. But to decide if the number of maxterms it contained is 2^n, the ER algorithm tries to extend every clause in the set into maxterms, or use the inclusion-exclusion rule, while in our algorithm we compute the relative maxterms that are covered by the universal set but not covered by the clause set T. The original ER algorithm was first presented in [20]. Subsequently, an improved version ER1 was presented in [21]. In this paper, we implemented the ER1 algorithm and compared it with our algorithm. For the k-SAT ($k \geq 3$) problems shown in Tables 1, 2, and 3, ER1 cannot solve any of them within the time limit (600 seconds). The reason is due to the extension rule algorithm has very strong requirement on the value of CF (complementary factor) for a SAT instance. The extension rule algorithm needs $\mathrm{CF} \geq 0.5$ [20], which means more than half of the clause pairs have complementary literals, where in most of 3-SAT problems (for example, those from SATLIB), the values of CFs are less than 0.1.

To make a comparable experiment with the extension rule algorithm, we randomly generated some SAT instances with small size (20 to 30 variables) as the benchmarks used in [20], [21]. The experimental results are shown in Table 4. In the "Problem Group" column, the triple of values denote the number of variables, the number of clauses, and the length of each clause, respectively. The "CF" column gives the average value of CF among all problems in the group. From Table 4, it is obvious that our solver runs several orders of magnitude faster than extension rule for all instances, no matter what the value of CF is. Another fact to be noted is that, when the value of CF grows, our algorithm can still quickly find the solution with Strategy 1. We believe our solver runs much faster than the extension rule solver is due to its optimization strategies. Unlike the ER algorithm, our algorithm computes the relative maxterms to decide the satisfiability.

5 Conclusion

In this paper, we propose a SAT solver based on maxterm covering. Our algorithm computes the relative maxterms covered by the empty clause with respect to the clause set T, and determines the satisfiability of the problem based on whether the number of relative maxterms is greater than 0. We propose a set of optimization strategies to enhance our approach. The experimental results show our algorithm with the eight optimization strategies runs faster than zChaff and MiniSAT for most of k-SAT ($k \geq 3$) problems, and is several orders of magnitude faster than the extension rule algorithm for almost all the problems.

Acknowledgments

This work was supported in part by the Chinese National 973 Plan under grant No. 2010CB328003, the National Science Foundation of China under grants No. 60903030 and 90718039. F. He is the corresponding author.

References

[1] S.A. Cook, "The Complexity of Theorem-Proving Procedures," *Proc. Third ACM Symp. Theory of Computing*, pp. 151-158, 1971.

[2] T. Larrabee, "Test Pattern Generation Using Boolean Satisfiability," *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, vol. 11, no. 1, pp. 4-15, Jan. 1992.

[3] A. Biere, A. Cimatti, E.M. Clarke, M. Fujita, and Y. Zhu, "Symbolic Model Checking Using SAT Procedures Instead of BDDs," *Proc. Design Automation Conf.*, pp. 317-320, 1999.

[4] W.N.N. Hung and N. Narasimhan, "Reference Model Based RTL Verification: An Integrated Approach," *Proc. IEEE Int'l High Level Design Validation and Test Workshop (HLDVT)*, pp. 9-13, Nov. 2004.

[5] W.N.N. Hung, X. Song, G. Yang, J. Yang, and M. Perkowski, "Optimal Synthesis of Multiple Output Boolean Functions Using a Set of Quantum Gates by Symbolic Reachability Analysis," *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, vol. 25, no. 9, pp. 1652-1663, Sept. 2006.

[6] W.N.N. Hung, C. Gao, X. Song, and D. Hammerstrom, "Defect Tolerant CMOL Cell Assignment via Satisfiability," *IEEE Sensors J.*, vol. 8, no. 6, pp. 823-830, June 2008.

[7] R.G. Wood and R.A. Rutenbar, "FPGA Routing and Routability Estimation via Boolean Satisfiability," *Proc. Int'l Symp. Field-Programmable Gate Arrays*, pp. 119-125, 1997.

[8] X. Song, W.N.N. Hung, A. Mishchenko, M. Chrzanowska-Jeske, A. Kennings, and A. Coppola, "Board-Level Multiterminal Net Assignment for the Partial Cross-Bar Architecture," *IEEE Trans. Very Large Scale Integration Systems*, vol. 11, no. 3, pp. 511-514, June 2003.

[9] W.N.N. Hung, X. Song, T. Kam, L. Cheng, and G. Yang, "Routability Checking for Three-Dimensional Architectures," *IEEE Trans. Very Large Scale Integration Systems*, vol. 12, no. 12, pp. 1398-1401, Dec. 2004.

[10] W.N.N. Hung, X. Song, E.M. Aboulhamid, A. Kennings, and A. Coppola, "Segmented Channel Routability via Satisfiability," *ACM Trans. Design Automation of Electronic Systems*, vol. 9, no. 4, pp. 517-528, 2004.

[11] F. He, W.N.N. Hung, X. Song, M. Gu, and J. Sun, "A Satisfiability Formulation for FPGA Routing with Pin Rearrangements," *Int'l J. Electronics*, vol. 94, no. 9, pp. 857-868, Sept. 2007.

[12] J.P. Marques-Silva and K.A. Sakallah, "GRASP: A New Search Algorithm for Satisfiability," *Proc. IEEE/ACM Int'l Conf. Computer-Aided Design (ICCAD '96)*, pp. 220-227, Nov. 1996.

[13] L. Zhang, C. Madigan, M. Moskewicz, and S. Malik, "Efficient Conflict Driven Learning in a Boolean Satisfiability Solver," *Proc. IEEE/ACM Int'l Conf. Computer-Aided Design (ICCAD)*, Nov. 2001.

[14] E. Goldberg and Y. Novikov, "BerkMin: A Fast and Robust SAT Solver," *Proc. Design Automation and Test in Europe Conf. Exhibition*, pp. 142-149, 2002.

[15] N. Eén and N. Sörensson, "An Extensible Sat-Solver," *Proc. Sixth Int'l Conf. Theory and Applications of Satisability Testing*, pp. 502-518, 2003.

[16] B. Selman, H.A. Kautz, and B. Cohen, "Noise Strategies for Improving Local Search," *Proc. 12th Nat'l Conf. Artificial Intelligence (AAAI '94)*, pp. 337-343, 1994.

[17] B. Selman, H. Levesque, and D. Mitchell, "A New Method for Solving Hard Satisfiability Problems," *Proc. 10th Nat'l Conf. Artificial Intelligence (AAAI)*, pp. 440-446, 1992.

[18] M. Davis and H. Putnam, "A Computing Procedure for Quantification Theory," *J. ACM*, vol. 7, pp. 201-215, 1960.

[19] M. Davis, G. Logemann, and D. Loveland, "A Machine Program for Theorem Proving," *Comm. ACM*, vol. 5, pp. 394-397, July 1962.

[20] L. Hai, S. Jigui, and Z. Yimin, "Theorem Proving Based on the Extension Rule," *J. Automated Reasoning*, vol. 31, no. 1, pp. 11-21, 2003.

[21] X. Wu, J. Sun, S. Lv, and M. Yin, "Propositional Extension Rule with Reduction," *Int'l J. Computer Science and Network Security*, vol. 6, no. 1A, pp. 190-197, Jan. 2006.

[22] P. Cheeseman, B. Kanefsky, and W.M. Taylor, "Where the REALLY Hard Problems Are," *Proc. 12th Int'l Joint Conf. Artificial Intelligence (IJCAI '91)*, pp. 331-337, 1991.

[23] J.M. Crawford and L.D. Auton, "Experimental Results on the Crossover Point in Satisfiability Problems," *Proc. 11th Nat'l Conf. Artificial Intelligence (AAAI '93)*, pp. 21-27, 1993.

[24] C. Coarfa, D.D. Demopoulos, A.S.M. Aguirre, D. Subramanian, and M.Y. Vardi, "Random 3-Sat: The Plot Thickens," *Proc. CP '02: Sixth Int'l Conf. Principles and Practice of Constraint Programming*, pp. 143-159, 2000.

[25] SATLIB—Benchmark Problems, http://www.cs.ubc.ca/hoos/SATLIB/benchm.html, 2001.

[26] S. Kirkpatrick and B. Selman, "Critical Behavior in the Satisfiability of Random Boolean Expressions," *Science*, vol. 264, pp. 1297-1301, 1994.