All-SAT using Minimal Blocking Clauses

Yinlei Yu* Pramod Subramanyan† Nestan Tsiskaridze‡ Sharad Malik†
*Two Sigma Investments LLC †Princeton University ‡University of Iowa

***Abstract*—The All-SAT problem deals with determining all the satisfying assignments that exist for a given propositional logic formula. This problem occurs in verification applications including predicate abstraction and unbounded model checking. A typical All-SAT solver is based on iteratively computing satisfying assignments using a traditional Boolean satisfiability (SAT) solver and adding blocking clauses which are the complement of the total/partial assignments. We argue that such an algorithm is doing more work than needed and introduce new algorithms that are more efficient. Experiments show that these algorithms generate solutions with up to $14\times$ fewer partial assignments and are up to three orders of magnitude faster.**

I. Introduction

This paper considers the All-SAT problem [1], [2] which deals with enumerating all satisfying assignments of a propositional logic formula. The All-SAT problem has many diverse applications such as model checking [1], [3], the computation of *backbones* of a propositional logic formula [4], [5], quantifier elimination [6], logic minimization [7], reachability analysis [2] and predicate abstraction [8].

In this paper, we introduce new algorithms which enumerate all satisfying assignments for a formula in conjunctive normal form (CNF). The number of satisfying assignments for a formula can be exponential in the size of the formula. Therefore, for All-SAT to be practical, we enumerate *partial assignments*[1] or *cubes*. A single partial assignment encompasses a set of *total assignments*. As a result, the set of all satisfying partial assignments may be efficiently enumerable even if the set of all satisfiable total assignments is not. Effectively, this means that an efficient solution to the All-SAT problem is one of converting a formula in CNF into a tractable disjunctive normal form (DNF) representation.

One method for enumerating satisfying partial assignments is as follows. We use a SAT solver to find a satisfying total assignment, "enlarge" this total assignment into a partial assignment, block this partial assignment by adding a *blocking clause* to the formula and repeat this process until the formula becomes unsatisfiable. This template for a solution to the All-SAT problem was introduced by McMillan [1]. We show that such a solution results in a DNF representation with cubes that are pairwise disjoint cubes. This trivially enables counting the number of satisfying solutions, which is a $\#P$-complete problem [9]. Since most problems related to DNF minimization are in Σ_2^P [10] and model counting is $\#P$-complete, it is likely that the All-SAT problem, *i.e.* determining a DNF cover, is simpler than model counting. In this paper, we present two solutions to All-SAT, called the *Non-Disjoint All-SAT algorithm* and the *Non-Disjoint-Dec All-SAT algorithm* that do not produce a disjunction of pairwise disjoint cubes. Our algorithms result in more compact DNF representations and experiments with a diverse set of benchmarks show that these algorithms generate DNF representations with up to $14\times$ fewer partial assignments and are up to three orders of magnitude faster than the All-SAT template described above.

[1]Formal definitions of assignments and cubes are given in §II-A.

II. Preliminaries

A. Notation

Let $\mathbb{B} = \{0, 1\}$ be the Boolean domain. Let X be a set of propositional variables of cardinality n: $X = \{x_1, x_2, \ldots, x_n\}$. The *logical operations* over $\mathbb{B}$: $\wedge$, $\vee$ and $\neg$ are defined as usual. The set of *literals* over X is $L_X = \{x_i, \neg x_i \mid x_i \in X\}$. Given a literal $\ell_i \in L_X$, the variable occurring in ℓ_i is denoted by $var(\ell_i)$. A *cube* of size k over X is a conjunction of k literals $\ell_1 \wedge \ell_2 \wedge \cdots \wedge \ell_k$ where each $\ell_i \in L_X$. In this paper, we consider only cubes such that $var(\ell_i) \neq var(\ell_j)$ if $i \neq j$. A *minterm* is a cube of size n. Recall $n = |X|$. A *clause* of size k over X is a disjunction of k literals $\ell_1 \vee \ell_2 \vee \cdots \vee \ell_k$, where each $\ell_i \in L_X$. We assume that $var(\ell_i) \neq var(\ell_j)$ if $i \neq j$. We define a propositional formula F in conjunctive normal form (CNF) as the conjunction of a finite set of clauses over X: $F = c_1 \wedge c_2 \wedge \cdots \wedge c_m$. A propositional formula G in disjunctive normal form (DNF) is defined as the disjunction of a finite set of cubes over X: $G = q_1 \vee q_2 \vee \cdots \vee q_m$.

An assignment σ is a mapping from $X \mapsto \mathbb{B}$ and σx_i denotes the value of variable x_i under the assignment σ. An assignment is called a total assignment if σ is a total function and partial otherwise. Assignments, both total and partial, correspond to cubes; total assignments correspond to minterms, e.g., $\sigma = \{x_1 \mapsto 1, x_2 \mapsto$

$0, x_3 \mapsto 1\}$ corresponds to $x_1 \wedge \neg x_2 \wedge x_3$. In the sequel, we use minterms and total assignments as well as cubes and partial assignments interchangeably. An assignment σ is said to satisfy a formula $F(x_1, x_2, \ldots, x_n)$ iff F evaluates to 1 under the assignment σ, *i.e.*, $F(\sigma x_1, \sigma x_2, \ldots, \sigma x_n) = 1$. We write this as $\sigma \models F$. Formulas F and G are equivalent, written as $F \Leftrightarrow G$, if for every assignment σ: $\sigma \models F$ iff $\sigma \models G$.

B. Problem Definition

Given a propositional formula F in CNF, we wish to derive an equivalent formula Q as a disjunction of cubes $Q = \vee_{i=1}^{m} q_i$ where each q_i is a cube over X. Note this problem corresponds exactly to converting a conjunction of clauses (CNF) to a disjunction of cubes (DNF).

III. ALL-SAT ALGORITHMS

This section presents our algorithms for All-SAT and proofs of their correctness.

A. Overview of Algorithms

Algorithm 1 is a template for the All-SAT algorithms discussed in this paper. Line 3 uses a SAT solver to compute a satisfying minterm for F. Line 4 calls the unspecified function $compute$ to obtain the partial assignment (cube) q_i from m_i (total assignment). A blocking clause c_i is found and added to F_i preventing these assignments from being enumerated again. This process is repeated until F_i becomes unsatisfiable.

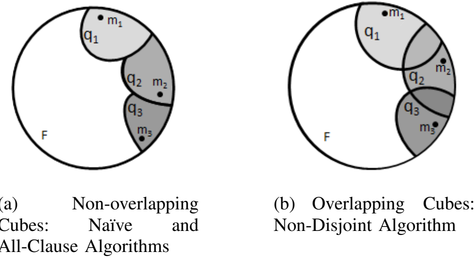
Algorithm 1 Algorithm Template

Input: F in CNF.
Output: Q in DNF, such that $Q \Leftrightarrow F$.

```
1: i := 1, F_1 = F, Q = 0
2: while not unsat(F_i) do
3:     m_i := sat(F_i)                ▷ get a satisfying assignment of F_i
4:     (q_i, c_i) := compute(m_i)     ▷ get cube and blocking clause
5:     Q := Q ∨ q_i, F_{i+1} = F_i ∧ c_i   ▷ update Q, F_i
6:     i := i + 1
7: end while
8: return Q.
```

Algorithm 1 Naïve All-SAT: A very simple but inefficient All-SAT algorithm can be derived by implementing $compute$ as follows. We simply set $q_i := m_i$. The blocking clause c_i is the complement of q_i: $c_i = \neg q_i$. This algorithm exhaustively enumerates each satisfying assignment of F. Since the number of satisfying assignments of a Boolean formula can be exponential in the size of the formula, this algorithm is likely not practical.
Algorithm 2 All-Clause All-SAT: This algorithm improves upon the naïve algorithm by *enlarging* minterm m_i into cube q_i. Cube q_i represents a *set* of satisfying minterms. Fig. 1(a) illustrates this. We define the function $compute$ in line 4 as: $compute(m_i) := minimal(m_i, F_i)$. The function $minimal$ takes as input a minterm m_i and a formula $\mathcal{F}$ in CNF. It returns the enlarged cube q_i along with its corresponding blocking clause c_i. The implementation of $minimal$ is discussed in §III-B.[2] An important observation here is that cubes derived by the All-Clause algorithm are pairwise disjoint.
Theorem (Disjoint Cubes): *Let* $Q = \vee_{i=1}^{k} q_i$ *be the disjunction of cubes produced by Algorithm 2.* $\forall i, j \in \{1 \ldots k\} : (i \neq j) \Rightarrow (q_i \wedge q_j = 0)$. Proof omitted.

Counting the number of satisfying assignments of a formula is $\#P$*-complete [9]. However, this can be computed in polynomial time given a set of pairwise disjoint cubes, as the disjoint cube property ensures no satisfying minterm is counted twice. However, most problems related to computing a minimal DNF cover are in* Σ_2^P *[10], so it is likely that determining a DNF representation is simpler than model counting. This suggests the All-Clause algorithm is doing more work than necessary.*



(a) Non-overlapping Cubes: Naïve and All-Clause Algorithms

(b) Overlapping Cubes: Non-Disjoint Algorithm

Fig. 1. Difference between All-Clause and Non-Disjoint Algorithms.

Algorithm 3 Non-Disjoint All-SAT: Using the above insight, the Non-Disjoint All-SAT Algorithm computes a disjunction of *overlapping* cubes Q with the expectation that Q would be more compact. We define the function $compute$ in line 4 as: $compute(m_i) := minimal(m_i, F)$. Here $minimal$ enlarges the minterm m_i into a cube q_i that *satisfies F but not necessarily* F_i. This is illustrated in Fig. 1(b). Cubes q_1, q_2, q_3 etc. can now overlap with the other cubes.

B. Finding Minimal Blocking Clauses

The $minimal$ function takes as input a minterm m_{sat} and a formula $\mathcal{F}$ and returns the "enlarged" cube q_i and corresponding blocking clause c_i. Some applications of All-SAT require existential quantification over a certain

[2] We note that minimal cubes correspond to *prime implicants* [11].

set of variables while computing all satisfying solutions. This quantification can be performed on the outputs of the $minimal$ function.

We now describe two implementations of $minimal$.

1) Minimal Satisfying Cube: Given the minterm m_{sat}, the goal here is to identify a subset of the literals in m_{sat} which guarantee satisfiability of $\mathcal{F}$. This is the unate covering or hitting set problem. Precisely stated, the problem is to select a subset of columns that cover every row of a matrix $A_{m \times n}$. Here each of the m rows corresponds to a clause in $\mathcal{F}$ while each column corresponds to a literal in the minterm m_{sat}. $A_{ij} = 1$ if the j-th literal in m_{sat} occurs in clause c_i, and 0 otherwise. Column j is said to cover row i iff $A_{ij} = 1$.

We solve the unate covering problem using the well-known greedy heuristic from [11] which is as follows. We first select all essential columns. A column j is essential if there is at least one row i such that $A_{ij} = 1$ and $A_{ik} = 0$ for all $k \neq j$. Essential columns and rows covered by them are eliminated from the matrix. Next we prune dominated rows. Row i is dominated by row j if $\forall k : A_{jk} = 1 \Rightarrow A_{ik} = 1$. Column i is dominated by column j if $\forall k : A_{ki} = 1 \Rightarrow A_{kj} = 1$. Dominated columns are also pruned. These pruning techniques are applied repeatedly until no new rows/columns can be eliminated. At this point, we greedily select a column that covers the most rows, remove this column and the rows covered by it, and repeat the above procedure until all rows are covered. The conjunction of the selected columns (*i.e.*, literals in m_{sat}) yields q_i and its negation is the disjunction c_i, the blocking clause.

2) Decision-Based Minimal Satisfying Cube: Modern SAT solvers based on the DPLL [12] procedure and conflict-driven clause learning [13] work by starting with a set of *decision* variables and computing values of *implied* variables required for satisfiability. Given a formula F and satisfying assignment σ, the values of the decision variables in σ and F itself completely determine values of the implied variables in σ. This fact can be used to reduce the size of the blocking clauses. For example, consider $F = (a \vee \neg b) \wedge (a \vee b \vee c)$. Suppose the solver makes the decision $b = 1$. Then $a = 1$ is implied by F and c is irrelevant to satisfiability. Now $(\neg b)$ can be used as a blocking clause instead of $(\neg b \vee \neg a)$. Specifying $a = 1$ is unnecessary in the blocking clause as there are no satisfying assignments of F with $a = 0$ when $b = 1$.

Given the minterm m_{sat}, define the set of *required literals* $\mathbf{R}$ as follows: (i) $\mathbf{R}$ contains all literals in m_{sat} which are sole literals evaluating to 1 in at least one clause in $\mathcal{F}$ and (ii) $\mathbf{R}$ also contains every literal that participates either directly or indirectly in an implication that sets the value of a literal included due to (i). In the example above, $m_{sat} = a \wedge b \wedge c$. $\mathbf{R} = \{a, b\}$. a is included due to condition (i) and b due to condition (ii).

The set of required literals $\mathbf{R}$ are treated as essential variables in the unate covering problem and a satisfying subset of the remaining literals is computed. Let this subset be $\mathbf{U}$. Note $\mathbf{U}$ contains only decision variables because any implied variable would satisfy condition (i) in the definition of $\mathbf{R}$ above. Let $\mathbf{R}$ be expressed as $\mathbf{R} = \mathbf{R_D} \cup \mathbf{R_I}$, where $\mathbf{R_D}$ contains literals which are *decision* variables in the SAT solver and $\mathbf{R_I}$ corresponds to the implied variables. The cube q_i is the conjunction of the literals in $\mathbf{R_D} \cup \mathbf{R_I} \cup \mathbf{U}$. Let q_b be defined as the conjunction of the literals in $\mathbf{R_D} \cup \mathbf{U}$. Blocking clause c_i is the complement of q_b. In our example, $\mathbf{U}$ is empty because $a = 1$ and $b = 1$ guarantees satisfiability of all clauses. $\mathbf{R_D} = \{b\}$ and $\mathbf{R_I} = \{a\}$. So $q_i = a \wedge b$, $q_b = b$ and $c_i = \neg b$.

C. Proofs

Properties of the Minimal Function: We will prove termination and correctness of our algorithms assuming the properties shown below for the $minimal$ function. Here we assume that $(q_i, c_i) = minimal(m_i, \mathcal{F})$.

(a) $m_i \models q_i$
(b) $q_i \Rightarrow F$
(c) $\forall m : (m \not\models q_i \wedge m \models F) \Rightarrow (m \models c_i)$
(d) $\forall m : (m \models q_i) \Rightarrow m \not\models c_i$; in particular $m_i \not\models c_i$
(e) $q_i \Rightarrow \mathcal{F}$

Property (a) states that each invocation of $minimal$ with the arguments m_i and F_i must return a cube q_i that covers[3] the minterm m_i. Property (b) states that cube q_i must imply F, which means that every assignment that satisfies q_i also satisfies F. Property (c) asserts that if there exists a satisfying assignment m for F that is not covered by q_i, then the corresponding clause c_i *must not* block m. Property (d) states that the blocking clause c_i *must* block m_i itself. Finally, property (e) states that the enlarged cube returned by $minimal$ implies the argument supplied to minimal (which is $\mathcal{F}$).

First, we show that both implementations of minimal introduced in this paper satisfy these properties. Next, we prove termination and correctness of the algorithms assuming these properties.

Minimal Satisfying Cube: It is easy to see that the implementation of minimal satisfying cube from III-B1 satisfies the properties listed above. $m_i \models q_i$, $q_i \Rightarrow \mathcal{F}$ are true by construction. $\mathcal{F}$ is either F_i for Algorithm 2 or F for 3, so $\mathcal{F} \Rightarrow F$ from which $q_i \Rightarrow F$. Properties (c) and (d) are due to the fact that $c_i = \neg q_i$. ❑.

[3]We say that a cube q_i *covers* cube q_j if every literal in q_i is also present in q_j. This means that every satisfying assignment of q_j is also a satisfying assignment of q_i.

TABLE I
EXAMPLE: EXECUTION OF ALL-CLAUSE ALGORITHM FOR $F = (a \vee \neg b) \wedge (a \vee b \vee c)$

i	m_i	q_i	c_i	F_{i+1}
1	$\neg a \wedge \neg b \wedge c$	$\neg b \wedge c$	$b \vee \neg c$	$F \wedge (b \vee \neg c)$
2	$a \wedge \neg b \wedge \neg c$	$a \wedge \neg c$	$\neg a \vee c$	$F \wedge (b \vee \neg c) \wedge (\neg a \vee c)$
3	$a \wedge b \wedge c$	$a \wedge b \wedge c$	$\neg a \vee \neg b \vee \neg c$	$F \wedge (b \vee \neg c) \wedge (\neg a \vee c) \wedge (\neg a \vee \neg b \vee \neg c)$

TABLE II
EXAMPLE: EXECUTION OF NON-DISJOINT ALGORITHM FOR $F = (a \vee \neg b) \wedge (a \vee b \vee c)$

i	m_i	q_i	c_i	F_{i+1}
1	$\neg a \wedge \neg b \wedge c$	$\neg b \wedge c$	$b \vee \neg c$	$F \wedge (b \vee \neg c)$
2	$a \wedge \neg b \wedge \neg c$	a	$\neg a$	$F \wedge (b \vee \neg c) \wedge (\neg a)$

Decision-Based Minimal Satisfying Cube: We now prove the correctness of the minimal algorithm introduced in Section III-B2. First, we claim that properties (a), (b) and (e), *i.e.*, $q_i \Rightarrow \mathcal{F}$, $q_i \Rightarrow F$ and $m_i \models q_i$ follow directly from the construction of q_i. The key property that helps prove the correctness of this "asymmetric" choice of the cube and blocking clause is the claim that satisfying assignments to q_b which also satisfy $\mathcal{F}$ are identical to satisfying assignments of q_i.
Lemma (Identical Satisfying Minterms): $q_b \wedge \mathcal{F} \Leftrightarrow q_i$.
(1) $q_i \Rightarrow q_b \wedge \mathcal{F}$. *Proof.* This follows directly from the definitions of q_b and q_i. Note q_b contains a subset of the literals in q_i.
(2) $q_b \wedge \mathcal{F} \Rightarrow q_i$. *Proof.* Assume there exists a minterm m such that $m \models q_b \wedge \mathcal{F}$. Since $m \models q_b$, a subset of the literals in m are identical to those in $\mathbf{R_D} \cup \mathbf{U}$. Therefore, the only way $m \not\models q_i$ could occur is if one of the literals in $\mathbf{R_I}$ is different between q_i and m. However, for a satisfying assignment of $\mathcal{F}$, the values of the literals in $\mathbf{R_I}$ are fully determined by the values of the literals in $\mathbf{R_D}$. Therefore it is not possible this to occur and the claim must be true. ❑.

Properties (c) and (d) of the *minimal* function are a direct consequence of the above lemma.
Theorem (Termination): *Algorithms 2 and 3 terminate.*
Proof. The algorithm terminates because each successive iteration of the while loop is called on a function which has fewer satisfying assignments. This is guaranteed by property (d) and the construction of F_{i+1} from F_i by adding a clause c_i that blocks at least the minterm m_i.

D. Example: All-Clause and Non-Disjoint Algorithms

Consider $F = (a \vee \neg b) \wedge (a \vee b \vee c)$. The naïve algorithm enumerates all five minterms of F. Table I shows one possible execution of the All-Clause algorithm for the same function. The execution starts by finding the minterm $a = 0, b = 0, c = 1$. This minterm can be expanded to the cube $(\neg b \wedge c)$ but blocking this cube now means that cube $q_2 = a$ cannot be found as a alone does not satisfy the blocking clause $(b \vee \neg c)$. The result is a disjunction of three pairwise disjoint cubes. Table II shows the execution of the Non-Disjoint algorithm for the same function and the same initial minterm. In this case, the ability to select overlapping cubes results in a more compact (and in fact optimal) DNF representation.

IV. EVALUATION

We implemented the algorithms described in this paper by building on MiniSat [14]. We used a subset of the SATLIB [15] and SAT Challenge 2012 (SC'2012)[16] benchmarks. The CPU used was Intel® Xeon® E5645 CPU, execution time limit was one hour and the memory limit was 4GB. The four algorithms we evaluate are: (a) Naïve (b) All-Clause (c) Non-Disjoint using Minimal Satisfying Cubes ("Non-Disjoint"; see §III-B1) and (d) Non-Disjoint using Decision-Based Minimal Satisfying Cubes ("Non-Disjoint-Dec"; see §III-B2).

We evaluate all instances in SATLIB benchmarks except for 3SAT-CBS, FlatGC and MorphedGC. These three have very large sets of instances so we randomly selected 1000, 100 and 100 out of 40000, 1700 and 901 instances respectively. For the SC'2012 benchmarks, we selected all cases that MiniSat found satisfiable within 150 seconds. There are a total of 2621 instances, of which 2506 are satisfiable. Naïve solved 2076 cases, whereas All-Clause, Non-Disjoint and Non-Disjoint-Dec solved 2181, 2238 and 2236 instances respectively.

A. Aggregate Results of Benchmark Groups

Table III shows the aggregate statistics. The column "random?" indicates whether the instances are random. AveVar, and AveCl columns are the average number of variables and clauses respectively. #Inst and #SATInst indicates the number of all instances and satisfiable instances respectively. We show the number of instances solved by the following algorithms: Naïve, All-Clause, "Non-Disjoint", which is the Non-Disjoint algorithm using Minimal Satisfying Cubes (III-B1) and "Non-Disjoint-Dec" which is the Non-Disjoint algorithm using the Decision-Based Minimal Satisfying Cube (III-B2).

In most non-random cases, we found that all four algorithms solve the same set of All-SAT instances and failed on the rest. In benchmarks *beijing, FlatGC* and *dimacs*, All-Clause actually solves slightly fewer cases than Naïve as it takes more time to generate blocking clauses. On the other hand, the new algorithms show

TABLE III
AGGREGATE RESULTS FOR ALL-SAT

GroupName	Random?	AveVar	AveCl	#Inst	#SATInst	Naïve	All-Clause	Non-Disjoint	Non-Disjoint-Dec
FlatGC	No	239	882	100	100	71	64	71	74
MorphedGC	No	500	3,100	100	100	41	39	40	40
planning	No	1,730	2,3261	11	11	8	8	8	8
dimacs	Yes/No	505	6,011	240	138	87	86	92	91
beijing	No	9,119	51,706	16	15	4	3	5	5
bmc	No	30,077	160,366	13	13	0	0	0	0
ais	No	155	2,729	4	4	4	4	4	4
QuasiGroup	No	1,043	58,470	22	10	10	10	10	10
3SAT-CBS	Yes	100	449	1,000	1,000	982	997	1,000	1,000
3SAT-BMS	Yes	100	360	1,000	1,000	868	960	998	994
sc2012*	No	492,723	3,401,426	115	115	10	10	10	10

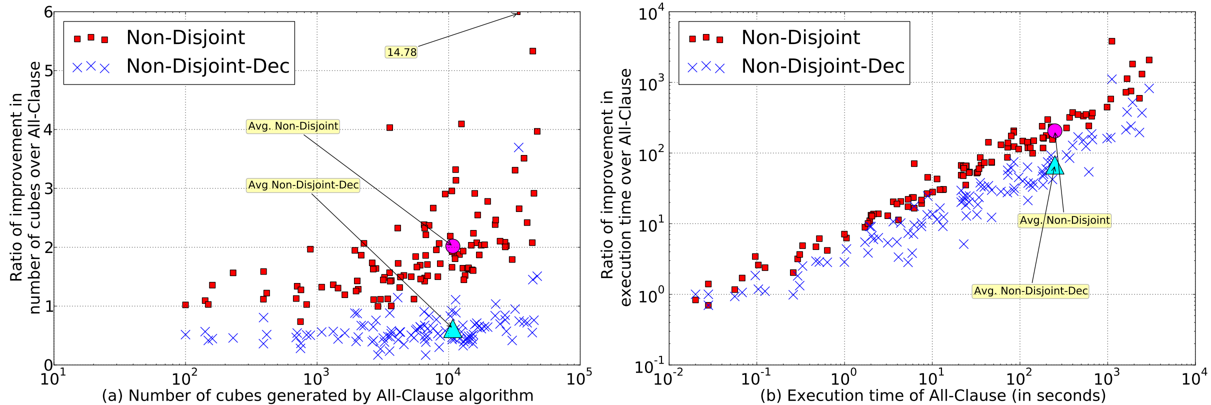


Fig. 2. Comparing All-Clause, Non-Disjoint and the Non-Disjoint-Dec Algorithms.

improvements in random SAT benchmarks like 3SAT-CBS and 3SAT-BMS.

The lack of advantage by Non-Disjoint and All-Clause over Naïve on these non-random benchmarks is mainly due to the properties of the underlying SAT instances. Most non-random instances are circuit-based and converted to CNF using the Tseitin Transformation [17]. To satisfy such instances, we have to select all variables representing internal nodes and outputs. This means the *minimal* blocking clause generated by Non-Disjoint or All-Clause contains all but a few input variables.

B. Performance comparison of algorithnms

In Fig. 2(a), we show the distribution of number of blocking clauses for 108 *hard All-SAT* instances completed by All-Clause, Non-Disjoint and Non-Disjoint-Dec but *not* completed by Naïve . The X-axis is the number of cubes generated by All-Clause. The Y-Axis is the ratio by which the number of cubes improves using Non-Disjoint/Non-Disjoint-Dec. A value >1 means more "compression" and we see the ratio is in fact greater than 1 for most benchmarks when using the "Non-Disjoint" algorithm. The average ratio for the Non-Disjoint algorithm is 2.0, which means that on average the Non-Disjoint algorithm produces a DNF representation with half as many cubes as the All-Clause algorithm. Larger instances see greater benefit. Surprisingly, Non-Disjoint-Dec creates more cubes than both Non-Disjoint and All-Clause. We believe this is due to the inclusion of blocking clauses by the implication-graph backtracking while computing $\mathbf{R}$.

Fig. 2(b) compares the execution times for the same set of instances. The X-axis is the execution time of All-Clause. The Y-axis is the speedup of Non-Disjoint/Non-Disjoint-Dec compared to All-Clause. A value >1 means Non-Disjoint/Non-Disjoint-Dec are faster than All-Clause. We see that Non-Disjoint and Non-Disjoint-Dec are *orders of magnitude faster than All-Clause and the advantage grows for larger instances*. Non-Disjoint is $205\times$ faster than All-Clause on average while Non-Disjoint-Dec is $68\times$ faster. Non-Disjoint is faster due to more effective blocking and fewer cubes in the DNF representation. Non-Disjoint-Dec is faster because it creates much smaller blocking clauses.

V. RELATED WORK

SAT-based techniques for prime implicant computation were investigated by Manquinho, Marques-Silva et

al. [18], [19]. They formulate an integer linear program (ILP) to find the shortest prime implicant of a Boolean function. While this algorithm can be extended to find all prime implicants, enumerating all prime implicants is distinct from and computationally more expensive than the problem tackled in this paper which is to compute an equivalent DNF representation of a Boolean function that consists only of prime implicants. Enumerating all prime implicants is not necessary to derive a complete cover and it is not clear how one would modify these algorithms to derive a cover.

The technique of enumerating satisfiable solutions using *blocking clauses* was first introduced by McMillan [1]. Brauer et al. [6] introduced an elegant formulation for All-SAT based on a "dual-rail" encoding that allows enumeration of the shortest implicants first. Both these techniques result in pairwise disjoint cubes.

Morgado et al. [20] suggested deriving a disjunction of overlapping cubes for the All-SAT problem. However, this was not evaluated in their experiments and they presented results which derived a disjunction of pairwise disjoint cubes. Jin et al. [21] introduced an All-SAT algorithm for Boolean circuits. They introduced algorithms based on circuit-analysis for minimizing the total assignment and generate overlapping cubes in their solution. This requires a circuit representation and is very distinct from our approach. Another circuit-based approach for All-SAT using circuit cofactoring was introduced by Ganai et al. [3]. Neither Morgado et al. [20] nor Jin et al. [21] make the connection between pairwise disjoint cubes and $\#P$-completeness of model counting.

Exploiting the information available in the internal state of the solver while solving the All-SAT problem is also done in [1] and [2]. Grumberg et al. [2] introduced a search based technique for All-SAT. They explore satisfying solutions for each value of a set of *important variables*. Their key advantage is that they do not use blocking clauses and so avoid "mem outs". The solution still results in pairwise disjoint cubes. Our contribution, the Non-Disjoint-Dec algorithm, generates extremely short blocking clauses that do not contain any of the implied variables in the solver. Note that typically the majority of the variables in a satisfying minterm are implied. Short blocking clauses are very beneficial for solver performance as demonstrated by the evaluation.

VI. Conclusion

This paper introduced new algorithms for the All-SAT problem which has many applications in model checking, logic minimization, reachability analysis and predicate abstraction. We first showed that existing solutions to the All-SAT problem are likely doing more work than necessary because they produce a result that consists of pairwise disjoint cubes. We then introduced new algorithms for the All-SAT problem which generate solutions that consist of overlapping partial assignments and a new technique for generating blocking clauses that consist of only the decision variables in a SAT solver. We evaluated these algorithms by experimenting with a diverse set of SAT benchmarks and found that these algorithms generate All-SAT solutions which have up to $14\times$ fewer partial assignments and are up to $1000\times$ faster than traditional solutions to the All-SAT problem.

References

[1] K. L. McMillan, "Applying SAT Methods in Unbounded Symbolic Model Checking," in *CAV'02*, pp. 250–264.

[2] O. Grumberg, A. Schuster, and A. Yadgar, "Memory Efficient All-Solutions SAT Solver and Its Application for Reachability Analysis," in *FMCAD'04*, pp. 275–289.

[3] M. K. Ganai, A. Gupta, and P. Ashar, "Efficient SAT-based Unbounded Symbolic Model Checking using Circuit Cofactoring," in *ICCAD'04*, pp. 510–517.

[4] C. Zhu, G. Weissenbacher, D. Sethi, and S. Malik, "SAT-based Techniques for Determining Backbones for Post-Silicon Fault Localisation," in *HLDVT'11*, pp. 84–91.

[5] J. Marques-Silva, M. Janota, and I. Lynce, "On Computing Backbones of Propositional Theories," in *ECAI'10*, pp. 15–20.

[6] J. Brauer, A. King, and J. Kriener, "Existential Quantification as Incremental SAT," in *CAV '11*, pp. 191–207.

[7] S. Sapra, M. Theobald, and E. Clarke, "SAT-Based Algorithms for Logic Minimization," in *ICCD '03*, pp. 510–517.

[8] S. K. Lahiri, R. E. Bryant, and B. Cook, "A Symbolic Approach to Predicate Abstraction," in *CAV '03*, pp. 141–153.

[9] C. P. Gomes, A. Sabharwal, and B. Selman, *Handbook of Satisfiability*. IOS Press, 2008, ch. 20: Model Counting.

[10] D. Buchfuhrer and C. Umans, "The Complexity of Boolean Formula Minimization," *J. Comput. Syst. Sci.*, vol. 77, no. 1, pp. 142–153, Jan. 2011.

[11] G. D. Hatchel and F. Somenzi, *Logic Synthesis and Verification Algorithms*, 1996, ch. 4: Synthesis of Two-Level Circuits.

[12] M. Davis, G. Logemann, and D. Loveland, "A Machine Program for Theorem-Proving," *Comm. ACM*, vol. 5, no. 7, pp. 394–397, Jul. 1962.

[13] J. Marques-Silva and K. A. Sakallah, "GRASP - A New Search Algorithm for Satisfiability," in *ICCAD'96*, pp. 220–227.

[14] N. Eén and N. Sörensson, "An Extensible SAT-solver," in *SAT'03*, pp. 502–518.

[15] H. H. Hoos and T. Stützle, "SATLIB: An Online Resource for Research on SAT," in *SAT'00*, pp. 283–292.

[16] A. Balint, A. Belov, M. Järvisalo, and C. Sinz, "SAT-Challenge 2012, held in conjunction with SAT 2012," http://baldur.iti.kit.edu/SAT-Challenge-2012/index.html, 2012.

[17] G. S. Tseitin, "On the complexity of derivation in propositional calculus," in *Automation of Reasoning 2: Classical Papers on Computational Logic 1967-1970*, J. Siekmann and G. Wrightson, Eds. Berlin, Heidelberg: Springer, 1983, pp. 466–483.

[18] V. M. Manquinho, P. E. Flores, J. P. Marques-Silva, and A. L. Oliveira, "Prime Implicant Computation using Satisfiability Algorithms," in *ICTAI '97*, 1997, pp. 232–239.

[19] J. P. Marques-Silva, "On Computing Minimum Size Prime Implicants," in *IWLS '97*, 1997.

[20] A. Morgado and J. P. Marques-Silva, "Good Learning and Implicit Model Enumeration," in *ICTAI '05*, 2005, pp. 131–136.

[21] H. Jin, H. Han, and F. Somenzi, "Efficient Conflict Analysis for Finding All Satisfying Assignments of a Boolean Circuit," in *TACAS'05*, pp. 287–300.