

Satometer: How Much Have We Searched?

Fadi A. Aloul
Brian D. Sierawski
Karem A. Sakallah

Technical Report

June 1, 2002



THE UNIVERSITY OF MICHIGAN

Computer Science and Engineering Division
Department of Electrical Engineering and Computer Science
Ann Arbor, Michigan, 48109-2122
USA

Abstract

We introduce Satometer, a tool that can be used to estimate the percentage of the search space actually explored by a backtrack SAT solver. Satometer calculates a normalized minterm count for those portions of the search space identified by conflicts. The computation is carried out using a zero-suppressed binary decision diagram (ZBDD) data structure and can have adjustable accuracy. The data provided by Satometer can help diagnose the performance of SAT solvers and can shed light on the nature of a SAT instance.

1 Introduction

The last few years have seen significant algorithmic advances in, and carefully-crafted implementations of, Boolean Satisfiability (SAT) solvers [3, 16, 18, 24, 27, 29]. This has led to their successful application to a wide range of large-scale electronic design automation (EDA) problem instances consisting of thousands of variables and millions of clauses [4, 13, 19, 22, 25, 26]. Despite these remarkable developments, SAT solvers cannot escape the underlying worst-case exponential complexity of their search space and must sometimes be aborted after a certain time-out limit has been reached. Typically, when a solver aborts it provides very little data about how much progress it had achieved up to that point. Such data can be quite useful. Knowing, for instance, that the solver had managed, after several hours, to explore only 1% of the search space might suggest a very hard problem instance and the need, perhaps, to try a different approach. If, on the other hand, the solver reports exploring more than 99% of the search space without finding a solution, it may be reasonable to assume that the instance has very few satisfying assignments or is possibly unsatisfiable.

Satometer (pronounced like barometer) is an accessory that can be used with any backtrack search SAT solver to report the percentage of search space actually explored by the solver. It requires the solver to emit the set of clauses corresponding to the conflicts encountered during the search. It can be used dynamically, while the SAT solver is running, to indicate progress in the search for a solution. It is more useful, however, as a postprocessor to analyze the result of an aborted or completed search.

The paper is organized as follows. In Section 2 we present an overview of SAT. This is followed by a summary of previous work in Section 3. In Section 4, we introduce our measure of search progress. We then describe, in Section 5, how this measure can be computed using BDDs and ZBDDs. In Section 6 we illustrate the utility of this measure in a variety of experimental scenarios and conclude, in Section 7, with a summary of the paper’s main contributions.

2 SAT Definition

Traditionally, a Boolean formula ϕ is expressed in *product of sums* or *conjunctive normal form* (CNF). Each sum ω , referred to as a *clause*, is a disjunction of *literals*, where a literal l is either a variable x or its negation $\neg x$. A clause ω_i subsumes another clause ω_j if $\omega_i \rightarrow \omega_j$. A literal with no assigned truth value is referred to as a *free literal*. An unresolved clause with a single free literal is known as a *unit clause*. A clause is satisfied if at least one of its literals is set to 1. Consequently, a formula is satisfied if all its clauses are satisfied, and unsatisfied if at least one clause is unsatisfied. The goal is to identify a set of assignments for variables that would satisfy the formula or prove that no such assignment exist and that the formula is unsatisfiable.

Several ways exist to solve this problem. A trivial solution, referred as *exhaustive search*, is to try all 2^n possible assignments, where n is the number of variables in the problem. Clearly, it is impossible to try all assignments. An example is shown in Figure 1(a). Another solution, referred to as *local search* [21], identifies satisfiable instances by randomly selecting variable assignments. This approach is characterized as incomplete, since it can only prove satisfiability, but can’t establish unsatisfiability. Finally, backtrack search algorithms have been proposed. Unlike local search, backtrack search [9] is known to be complete, as it can establish satisfiability and unsatisfiability given enough time. Therefore, we believe backtrack search is the most robust approach for solving SAT problems.

The Davis, Logemann, and Loveland search procedure (DLL) [9], provides the basis for the majority of backtrack search algorithms. The procedure performs a depth-first search in the decision tree over the problem variables. It starts by selecting a decision variable according to a branching heuristic. Implications are identified by Boolean Constraint Propagation (BCP) [16] which itera-

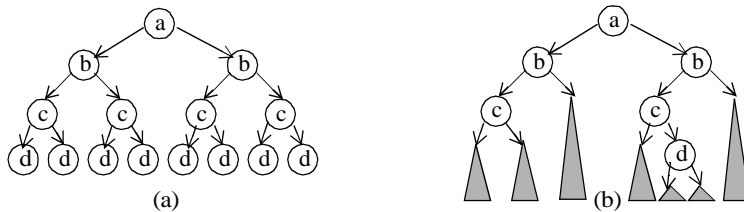


Fig. 1. Examples of (a) Exhaustive search (b) Backtrack search

tively applies the unit clause rule. The unit clause rule searches the clause database for a unit clause and assigns the single free literal to satisfy the clause. If a conflict is detected, in which a partial variable assignment unsatisfies one or more clauses, the procedure backtracks and unassigns the last decision level. The procedure terminates and proves satisfiability when all variables are assigned and no conflicts have been detected or unsatisfiability when no new variable assignment can be made without producing an unsatisfied clause.

We should note that applying BCP during the search process can effectively prune the search space, as the search algorithm is forced to skip areas of the search space that are covered by the opposite assignments of the implications. Thus, conflicts can be detected at an early stage of the search process as shown in Figure 1(b).

Recently, several enhancements to the DLL approach have been proposed. Among the various enhancements, conflict analysis, which was introduced in GRASP [16], is worth mentioning, since it can significantly prune the search space. The procedure is called after each conflict to analyze the causes of the conflict and generate adequate information to prevent the conflict from occurring at different parts of the search space. A conflicting assignment consists of a conjunction of antecedent variable assignments that identify a sufficient condition to lead to a conflict. Negating the conflicting assignment produces a conflict-induced clause that doesn't exist in the clause database. Adding the new clause to the clause database has the advantage of helping the deduction engine identify future implications that are not derivable with the original clause database and avoids regenerating the same conflicting assignment in future parts of the search process. Therefore, almost all backtrack search SAT algorithms today implement conflict analysis in their diagnosis engines.

3 Previous Work

Several researchers have explored various techniques for predicting the cost of solving a SAT problem by estimating the size of the decision tree that will be searched to solve the problem. Knuth [11] laid the foundation for this work by using iterative sampling. The basic idea is to explore a single random path from the root node to a leaf node. The number of successors d_i of each node at level i is recorded. Assuming that all nodes, at the same decision level, share the same number of successors, the number of nodes in the decision tree can be estimated using the formula:

$$1 + d_1 + d_1 d_2 + d_1 d_2 d_3 + \dots \quad (1)$$

The estimate is improved by averaging over a number of iterations.

More recent work by Purdom [20], extended Knuth's algorithm with *partial backtracking*. The idea is to traverse a number of successor nodes m for each visited node as opposed to traversing a single successor node as shown in Knuth's algorithm. Purdom showed that this modification is more effective on deep trees. The proposed algorithm is identical to Knuth's algorithm when $m = 1$ and follows a complete backtracking approach when all successors are traversed. The trade off between the estimate accuracy and the runtime overhead depends on the value of m .

Knuth’s algorithm was also extended by Chen [7]. Chen’s approach is based on *heuristic sampling* of the nodes in the decision tree, in which a cost function is associated with each node. The goal is to estimate the cost of processing the complete set of nodes in the decision tree.

Recently, Kokotov and Shlyakhter [12] described a progress bar that can be integrated into a backtrack SAT solver to estimate the time left to complete the search. The bar is updated based on either *historical* or *predictive* estimates of the size of the decision tree maintained by the SAT solver. Historical estimators are based on averaging the time spent on nodes at the same decision level of the node being examined. The assumption is that particular, structured problems, are likely to share similar subtree sizes between nodes at the same decision level. They propose to further improve the average by weighting the average according to the distance between the nodes. The predictive estimators ignore the subtree sizes of previous nodes and instead focus on estimating the needed runtime by analyzing the clauses in which the variable appears in. The premise is that variables that eliminate more clauses, either by satisfying clauses directly or implying other assignments, are likely to require less time to explore. They reported that the bar is able to predict progress with an accuracy of 80-90% without significantly impacting the solver’s run time. The results, however, are unbounded and no guarantees are given to confine the accuracy.

Unlike the previous approaches, the focus of Satometer is to measure the search space explored by a backtrack search solver and report on the progress of the solver. Satometer doesn’t provide any estimate on the size of the decision tree.

4 Measuring Search Progress

In our approach, we view the search process as a sequence of moves that continually (and systematically) modify a (partial) variable assignment until 1) a satisfying assignment (a solution) is found, 2) the formula is proven to be unsatisfiable (has no solution), or 3) a time-out limit is reached. Along the way, many assignments that are explored will correspond to zeros of the function represented by the formula and will cause the search process to backtrack. Every time such a “conflict” occurs, it identifies a portion of the search space that can be regarded as having been *explored* and found to contain no solutions.

Let A_1, A_2, \dots, A_i denote the assignments that correspond to the first i conflicts. We can measure how much of the search space has been explored by counting the number of minterms* covered by the function $A_1 + A_2 + \dots + A_i$. Normalizing this count by the total size of the space yields the percentage of the space that has been explored up to this point. We will use the notation $\|f\|$ to express the normalized number of minterms of the function f . Thus, $\|a + b\| = 75\%$, $\|a \cdot b\| = 25\%$, and $\|a \oplus b\| = 50\%$.† In the sequel, we will refer to $\|f\|$ as the *size* of f .

This measure can be equivalently computed by considering the *conflict clauses* identified at each conflict. Let C_1, C_2, \dots, C_j denote the conflict clauses identified after the first i conflicts. In general, $j \geq i$ as one or more conflict clauses may be identified at each conflict. The portion of the search space that would have been explored after processing the *ith* conflict can now be computed as $1 - \|C_1 \cdot C_2 \cdot \dots \cdot C_j\|$.

An illustration of these computations is shown in Figure 2 for the 4-variable formula:

$$\begin{aligned} \varphi = & (a + b + c)(a + b + c')(a' + b + c')(a + c + d) \cdot \\ & (a' + c + d)(a' + c + d')(b' + c' + d')(b' + c' + d) \end{aligned} \quad (2)$$

* Complete truth assignment that sets the function to 1.

† Assuming that the number of variables is 2.

| Decisions | Impli- cations | Conflicts | | Explored Space | |
|-----------|-------------------|-----------|------------------|----------------|------|
| | | Y/N | Clause | Minterms | % |
| 1 | a | N | | | |
| 2 | ab | N | | | |
| 3 | abc | d' | $(a' + b' + c')$ | 2 | 12.5 |
| 4 | abc' | d | $(a' + b' + c)$ | 4 | 25 |
| 5 | ab' | $c'd$ | $(a' + b)$ | 8 | 50 |
| 6 | a' | N | | | |
| 7 | $a'b$ | N | | | |
| 8 | $a'bc$ | d' | $(a + b' + c')$ | 10 | 62.5 |
| 9 | $a'bc'$ | d | Solution! | | |

(a) Execution trace of a basic backtrack SAT Solver

| Decisions | Impli- cations | Conflicts | | Explored Space | |
|-----------|-------------------|-----------|-------------|----------------|------|
| | | Y/N | Clause | Minterms | % |
| 1 | a | N | | | |
| 2 | ab | N | | | |
| 3 | abc | d' | $(b' + c')$ | 4 | 25 |
| 4 | ab | $c'd$ | $(a' + b')$ | 6 | 37.5 |
| 5 | a | $b'c'd$ | (a') | 10 | 62.5 |
| | | a' | N | | |
| 6 | b | $a'c'd$ | N | Solution! | |

(b) Execution trace of a conflict-based backtrack SAT Solver

Fig. 2. Execution traces of two different SAT solvers on the formula in (2) illustrating how search progress is measured.

5 Computing Space Coverage

When the conflicting assignments are disjoint (i.e., when $A_k \cdot A_l = 0$ for $k \neq l$), space coverage can be simply calculated by the formula:

$$\|A_1 + A_2 + \dots + A_i\| = \sum_{1 \leq k \leq i} \|A_k\|. \quad (3)$$

Equivalently, if the conflict clauses are disjoint, i.e. if $C_k + C_l = 1$ for $k \neq l$, then space coverage is simply

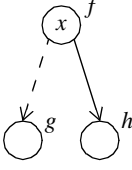


$$1 - \|C_1 \cdot C_2 \cdot \dots \cdot C_j\| = \sum_{1 \leq k \leq j} (1 - \|C_k\|). \quad (4)$$

In other words, if conflicts identify non-overlapping pieces of the search space, then the size of the explored space can be found by simply adding the sizes of the different pieces. In general, this will not be the case except for standard backtrack algorithms that do not employ conflict diagnosis to prune the search space. To compute the size of the explored space in such cases we have no choice but to build some type of symbolic representation for the disjunction of conflict assignments or the conjunction of conflict clauses. We describe below the two representations we examined and show how we used them to measure space coverage. Without loss of generality, we restrict the discussion to building representations for conjunctions of conflict clauses.

5.1 Using BDDs

The conflict clauses can be symbolically “anded” using a reduced ordered binary decision diagram (ROBDD or BDD for short) [5]. BDD semantics allow us to write the function f at a node labeled with variable x using Boole’s expansion:

TABLE I: Semantics of Decision Diagrams

| Diagram Type | | Internal Nodes | | Terminal Nodes | |
|--------------|------------|---|------------------|---|---|
| | |  | |  |  |
| BDD | | $f = x' \cdot g + x \cdot h$ | $f = \mathbf{0}$ | $f = \mathbf{1}$ | |
| ZBDD | Set | $f = g \cup \{x\} \times h$ | $f = \emptyset$ | $f = \{\emptyset\}$ | |
| | CNF | $f = (g) \cdot (x + h)$ | $f = \mathbf{1}$ | $f = \mathbf{0}$ | |
| | DNF | $f = (g) + (x \cdot h)$ | $f = \mathbf{0}$ | $f = \mathbf{1}$ | |

$$f = x' \cdot g + x \cdot h \quad (5)$$

where g and h are the functions associated with the 0- and 1-children of that node (see Table I.) This immediately leads to the following formula for the size of f :

$$\|f\| = \frac{1}{2}(\|g\| + \|h\|) \quad (6)$$

The size of the function represented by a BDD can now be obtained by sweeping the BDD from the terminal nodes towards the top node and applying (6) at each visited node. The sweep is initialized by setting $\|0\| = 0$ and $\|1\| = 1$ for the constant functions of the terminal nodes.

5.2 Using ZBDDs

The problem with the BDD representation, of course, is that it quickly runs out of memory. An alternative that has lower memory requirements is the zero-suppressed BDD (ZBDD) originally proposed by Minato [17] for manipulating large combination sets, including sets of Boolean cubes. A combination set S can be regarded as a *set of sets*, e.g. $\{\{a, b\}, \{c, d, e\}, \{a, d\}, \{b\}\}$. Recently, Chatalic and Simon [6] demonstrated that ZBDDs can be an effective implicit representation of large CNF formulas and showed how they can be used to perform “multi-resolution” to solve some large structured SAT instances. In this scenario, the above example set corresponds to the CNF formula $(a + b)(c + d + e)(a + d)(b)$, i.e. each combination is viewed as an OR term (a clause) and the entire set (a union of combinations) as an AND term. Such an interpretation allows the semantics of Boolean algebra to be layered on top of the semantics of set algebra to obtain further compression of the ZBDD structure. In particular, Chatalic and Simon extended the standard ZBDD set-union operation to a subsumption-free union that automatically removes any clause that is completely subsumed by another clause. In the above example, combination $\{a, b\}$ is subsumed by combination $\{b\}$ yielding the *logically equivalent* set $\{\{c, d, e\}, \{a, d\}, \{b\}\}$. Additional reduction rules based on literal absorption, i.e. $(a)(a' + b + c) = (a)(b + c)$, were subsequently described in [1].

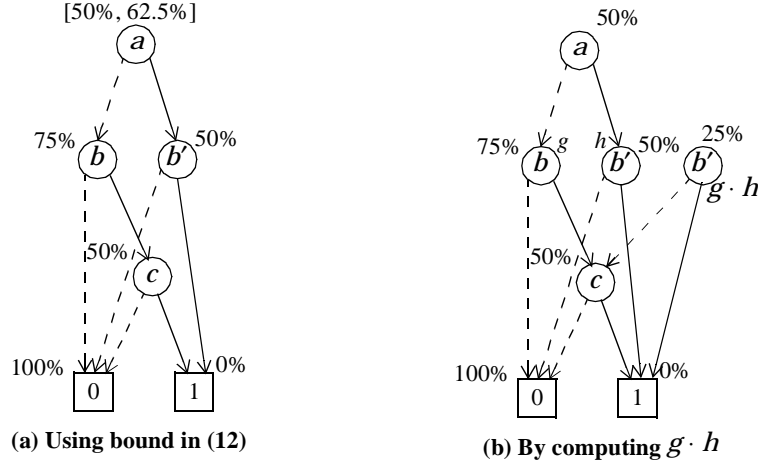


Fig. 3. Computation of $\|(a + b')(b + c)\|$ using (12) and (13).

The semantics of ZBDD nodes were first articulated by Lobbing et al. in [14]. Given a set of atoms $\{a, b, c, \dots\}$, a ZBDD node labeled with atom x represents a combination set f constructed according to the formula:

$$f = g \cup \{x\} \times h \quad (7)$$

where g and h are the combination sets associated with the 0- and 1-children of that node (see Table I.) The terminal 0 and 1 nodes correspond, respectively, to the empty set (set of no combinations) and to the set of consisting of the empty combination. The “product” in (7) is similar to the Cartesian product of two sets and is defined by

$$S \times T = \bigcup_{s \in S, t \in T} \{s \cup t\} \quad (8)$$

For example, given the combination sets $S = \{\{a, b\}, \{b, c\}\}$ and $T = \{\{a, d\}, \{e\}\}$, their product is[‡]

$$S \times T = \{\{a, b, d\}, \{a, b, e\}, \{a, b, c, d\}, \{b, c, e\}\} \quad (9)$$

When used to represent a CNF formula, the formula f associated with a ZBDD node labeled by variable x follows the same template of (7) except that the union of atoms in a combination is viewed as logical OR and the union of the combinations is viewed as logical AND yielding

$$f = (g) \cdot (x + h) \quad (10)$$

where g and h are the formulas associated with the 0- and 1-children of that node (see Table I.) The terminal 0 and 1 nodes, correspond, respectively, to the constant 1 and constant 0 functions.

To represent CNF formulas with ZBDDs, the set of atoms is taken to be the set of literals over which the formula is defined. In addition, the positive and negative literals of each variable are grouped together so that they are adjacent in the total order used in constructing the ZBDD. This

[‡] Note that $S \times T \neq S \cup T$. For this example, $S \cup T = \{\{a, b\}, \{b, c\}, \{a, d\}, \{e\}\}$.

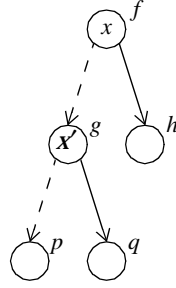


Fig. 4. The special case when g is not vacuous in x .

restriction facilitates, among other things, the identification and automatic removal of tautologies, i.e. combinations that have the form $(x + x' + \dots)$, to further reduce the size of the ZBDD [6].

To determine the size of the function represented by the CNF formula associated with a ZBDD node, we must first re-write (10) as the disjoint sum of two terms:

$$f = (g) \cdot (x + h) = x \cdot g + x' \cdot (g \cdot h) \quad (11)$$

This immediately leads to

$$\|f\| = \frac{1}{2}(\|g\| + \|g \cdot h\|) \quad (12)$$

which, unlike (6) for BDDs, requires that we compute the size of the product of the two child formulas. This is not a problem if one or both of the children is a terminal node, but does pose a serious complication if they are both internal nodes. We propose three solutions:

Exact. One way to resolve this complication is to (recursively) create additional ZBDD nodes for such products until one of the children becomes terminal. This will provide us with the *exact* answer, but may exponentially increase the size of the ZBDD. Some of that increase can be ameliorated with caching and garbage collection. In particular, created nodes can be eliminated as soon as they have been used to tighten the bound of their parent.

Unrestricted bound. An alternative to computing $\|g \cdot h\|$ exactly is to *bound* it. The upper bound is easily established as $\min(\|g\|, \|h\|)$ and occurs when either $g \leq h$ or $h \leq g$. The lower bound can be determined by noting that $\|g \cdot h\| = 1 - \|g' + h'\|$. Thus $\|g \cdot h\|$ is smallest when $\|g' + h'\|$ is largest which occurs when g' and h' are disjoint. This gives a lower bound of $\|g\| + \|h\| - 1$ and yields the interval

$$\|g \cdot h\| \in [\max(0, \|g\| + \|h\| - 1), \min(\|g\|, \|h\|)] \quad (13)$$

where the max in the lower bound insures that the estimate remains non-negative.

An illustration of these computations is given in Figure 3 for the example formula $(a + b') \cdot (b + c)$. The percentages annotating the ZBDD nodes denote the function sizes of their corresponding formulas as computed by (12) and (13). The uncertainty in the size at the top node is resolved, in part b of the figure, by creating a node for the product of its children.

Restricted Bound. Between the two extremes of an *exact* count and a *bound* computed according to (13) we can produce a range of approximations that trade accuracy with speed and memory consumption. Specifically, when a given level of accuracy, say 10%, is exceeded by the bound com-

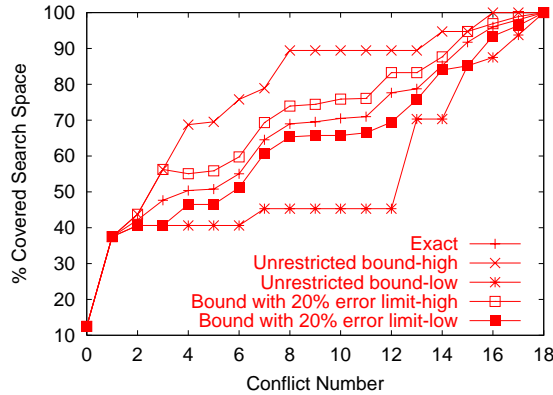


Fig. 5. Applying three modes of the proposed approach to *bf2670-001.cnf* instance

puted from (13), additional ZBDD nodes are created for the product formulas until the desired level of accuracy is achieved.

We must finally note that (12) is correct only when g is vacuous in x . The only situation when this is not true is depicted in Figure 4 where g 's node is labeled by the literal x' .** Substituting $g = (p) \cdot (x' + q)$ in (11) produces the disjoint sum

$$f = x \cdot (p \cdot q) + x' \cdot (p \cdot h) \quad (14)$$

which readily leads to

$$\|f\| = \frac{1}{2}(\|p \cdot q\| + \|p \cdot h\|) \quad (15)$$

Figure 5 illustrates the three possible modes of our approach on the bridging-fault *bf2670-001* instance. Despite setting an error limit of 20%, on average, the restricted bound and the unrestricted bound methods reported results within 7% and 24%, respectively, of the exact answer.

6 Experimental Evaluation

Satometer is implemented in C++ using the CUDD package [23]. It incorporates the ZBDD enhancements described in [1] and [6] for symbolic manipulation of CNF formulas. In this section we demonstrate its utility by applying it in a number of experimental scenarios. We configured it to report the size of the explored search space to within 20% of the exact answer; in many cases it was able to achieve a higher level of accuracy or to even report the exact answer. In the tables to follow, a single number in the explored space columns indicates that an exact answer was reported; ranges are indicated as intervals. All experiments were performed on an AMD Athlon 1.4 GHz machine with 1GB of RAM running the Linux operating system.

6.1 Effect of Preprocessing the CNF Formula

A variety of preprocessing techniques have been proposed to modify a CNF formula before submitting it to a SAT solver. These techniques generally add clauses to the formula in order to increase

**Note that h 's node cannot be labeled by x' as this would create a tautology that is automatically eliminated.

the number of potential implications or perform stylized algebraic simplifications to reduce the number of variables. We used Satometer to study the effectiveness of such techniques. In each case we compare the size of the space explored by a standard DLL algorithm^{††} [9] (i.e. without conflict analysis) on the original as well as on the modified formula. The time-out limit in these experiments was set to 10 seconds; *Satometer's run time was negligible*. The results of these experiments are given in Table II, Table III, and Table IV.

Addition of consensus clauses. In [2] the authors report that augmenting a CNF formula with clauses identified using consensus can reduce search time. To avoid generating an exponential number of clauses, they proposed a truncated iterative consensus procedure that augments the original formula with clauses whose size (number of literals) is limited by a small user-specified constant. They report speedups on the *aim* benchmarks from the DIMACS set [10] when the size of added clauses are limited to 3 or fewer literals.

A sampling of results on some unsatisfiable instances from this suite is shown in Table II. Column 1 lists the name of the benchmark; columns 2 and 3 give the number of variables (V) and clauses (C) in the original formula; column 4 gives the number of consensus clauses that are added to the formula; and columns 5 and 6 indicate the size of explored space reported by Satometer. The data in this table clearly show the effectiveness of these added clauses. For the two smaller instances, the search algorithm was actually able to explore the entire search space, and thus prove the unsatisfiability of the modified formula. In all cases, the addition of these clauses helped the SAT solver explore a significantly larger portion of the search space in the allotted amount of time.

Addition of symmetry-breaking predicates. In [8] the authors propose analyzing a CNF formula 1) to identify its symmetries, and 2) to augment it with clauses that break those symmetries. The intuition here is that the symmetry-breaking clauses act by allowing only one of many equivalent variable assignments to be a potential solution to the formula. If the original formula is satisfiable, the number of solutions may considerably decrease after pre-processing, clearly indicating that the search space was reduced. However, even if the original instance was not satisfiable, “the number of equivalent roads leading nowhere” would be reduced, and a generic SAT solver is likely to conclude much faster that no solution exists.

This intuition is confirmed by the data in Table III (whose layout is identical to that of Table II.) The benchmarks in this experiment are members of the unsatisfiable *hole* suite (which relates to the Pigeonhole principle.) The augmentation of each instance by a small number of symmetry-breaking clauses drastically enhances the ability of the SAT solver to prove unsatisfiability. This trend is clearly accentuated as instance sizes increase.

Algebraic simplification. Another formula preprocessing technique is based on formula simplification rules aimed at reducing the number of variables or clauses in the formula [15]. We studied this approach on some large hard bounded model checking [4] and microprocessor verification [26] instances. Results on a representative sample are given in Table IV.

Unlike the earlier experiments, the performance of the SAT solver on the modified formulas is not significantly better than its performance on the original formulas. The best improvement is in the *barrel7* benchmark and can be attributed to the simplifier’s ability to drastically reduce the number of variables (from 3523 to 800.) The low coverage in this experiment is also an indication of the difficulty of these instances.

^{††}The solver uses a fixed decision heuristic, chronological backtracking, and implements BCP as implemented in Chaff.

TABLE II: Addition of consensus clauses

| Benchmark | Original | | Modified | Explored Space, % | |
|------------------|----------|-----|----------|-------------------|--------------|
| | V | C | Extra C | Original | Modified |
| aim-50-1_6-no-4 | 50 | 80 | 54 | 57.06 | 100 |
| aim-100-1_6-no-3 | 100 | 160 | 73 | 0.015 | 100 |
| aim-200-1_6-no-3 | 200 | 320 | 233 | 0.049 | [97.72, 100] |
| aim-200-2_0-no-1 | 200 | 400 | 191 | 0 | [87.75, 100] |

TABLE III: Addition of symmetry-breaking predicates

| Benchmark | Original | | Modified | Explored Space, % | |
|-----------|----------|-----|----------|-------------------|--------------|
| | V | C | Extra C | Original | Modified |
| hole-7 | 56 | 204 | 14 | 100 | 100 |
| hole-8 | 72 | 297 | 16 | 79.2 | 100 |
| hole-9 | 90 | 415 | 18 | 37.5 | 100 |
| hole-10 | 110 | 561 | 20 | 18.75 | [99.98, 100] |
| hole-11 | 132 | 738 | 22 | 9.39 | [99.96, 100] |
| hole-12 | 156 | 949 | 24 | 4.68 | [99.96, 100] |

TABLE IV: Algebraic simplification

| Benchmark | Original | | Modified | | Explored Space, % | |
|---------------|----------|-------|----------|-------|-------------------|----------|
| | V | C | V | C | Original | Modified |
| longmutl7 | 3319 | 10335 | 2184 | 7635 | 0.280 | 0.341 |
| queinvar20 | 2435 | 20671 | 2343 | 28438 | 50 | 50.1 |
| barrel7 | 3523 | 13765 | 800 | 3447 | 51.02 | 62.46 |
| dlx2_cc_bug08 | 1515 | 12808 | 1486 | 13875 | 0 | 9.38 |

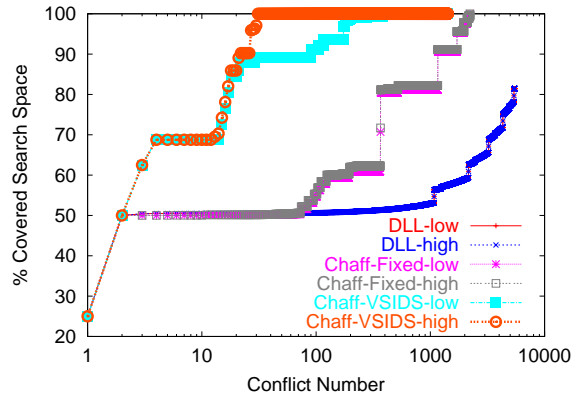
6.2 Analysis of Dynamic Techniques

In this set of experiments, we report on the application of Satometer to various SAT solvers with a variety of parameters. Our experiments involve three different SAT solvers: a simple DLL solver [9], SATIRE [27], and Chaff [18]. The last two solvers represent efficient implementations of the basic DLL solver. Chaff, however, is currently known as the leading DLL-based SAT solver. The goals of this experiment are to determine a) the best of two black-box SAT solvers, in which each solver’s description is hidden, b) the best of a variety of decision heuristics c) the difficulty of CNF instances d) the best of various conflict analysis techniques, and e) an estimate of the number of satisfying assignments in a satisfiable instance.

Black box A vs. black box B experiment. In the following experiment, several SAT solvers are provided. However, the user has no knowledge of the internals of any of the SAT solvers. Given a set of hard instances, the user is required to identify the best solver in the shortest possible time. In general, the user will need to run each SAT solver for a specified time or randomly select a solver and hope that it is the best among all others. Using the proposed method, however, can give an insight to which solver performs best within the specified run time limit. Table V shows several results for various hard instances from bounded model checking [4], microprocessor verification [26], FPGA routing [19], and the DIMACs set [10]. We tested each instance for 10 seconds using the

TABLE V: Percentage of explored search space for various SAT solver and decision heuristics

| Benchmark | | | | Space Explored,% | | |
|------------------------|-----------------|-------|--------|------------------|----------------|----------------|
| Family | Name | V | C | DLL | Chaff-Fixed | Chaff-VSIDS |
| uP Verification | 2dlx_cc | 4524 | 41704 | 0 | [81, 100] | [99.06, 100] |
| | 3pipe | 2392 | 27533 | 0.098 | [47.23, 62.63] | [80.41, 100] |
| | 4pipe | 5096 | 80213 | 0.025 | [69.68, 88.15] | [77.77, 95.46] |
| | 9vliw | 19148 | 179492 | 0 | [28.91, 35.16] | [99.97, 100] |
| DIMACS | par32-1-c | 1315 | 5254 | 0 | [78.64, 89.39] | [82.72, 100] |
| Bounded Model Checking | barrel6 | 2306 | 8931 | 52.77 | [60.94, 63.83] | 100 |
| | barrel7 | 3523 | 13765 | 51.02 | [60.95, 68.79] | [98.34, 100] |
| | barrel9 | 8903 | 36606 | 50.11 | [58.59, 58.84] | [99.94, 100] |
| | longmult6 | 2848 | 8853 | 0.40 | [72.39, 80.43] | [99.93, 100] |
| | longmult8 | 3810 | 11877 | 0.21 | [80.27, 87.78] | [90.48, 100] |
| | queuin18 | 2081 | 17368 | 0 | [96.57, 100] | 100 |
| | queuin20 | 2435 | 20671 | 50 | [92.3, 100] | [97.59, 100] |
| FPGA Routing | alu2_gr_rcs_w7 | 3570 | 73478 | 2.36 | [29.99, 36.55] | [50, 58.75] |
| | k2fix_gr_rcs_w8 | 10056 | 271393 | 1.18 | [0.665, 7.65] | [0.798, 9.03] |
| | k2fix_gr_rcs_w9 | 11313 | 305160 | 0.59 | [0.393, 5.147] | [0.400, 3.34] |
| | vda_gr_rcs_w8 | 5776 | 116522 | 0 | [0.615, 6.65] | [0.819, 9.75] |

Fig. 6. Search space coverage for *Barrel5.cnf*

following three SAT solvers and options: standard DLL solver, Chaff with a fixed decision heuristic, and Chaff with the default cherry.smj heuristic. The results clearly indicate the superiority of the third solver among the other two solvers for almost all benchmarks, due to the significantly high search space coverage achieved in the given time limit. Figure 6 shows a detailed space coverage analysis of the *barrel5* instance for all three solvers.

Comparison of decision heuristics. As shown in Table V, the proposed method can also be used to classify decision heuristics and rate their performance on various SAT instances. We show the results for two decision heuristics: a) static fixed [9]: unresolved variables with minimum index are selected first for decisions; b) dynamic VSIDS [18]: variables that appear in the highest number of clauses are selected first. (Some weight is given to variables appearing in recent conflict-induced clauses). Again, the results show the effectiveness of VSIDS as opposed to the fixed decision heuristic. Nevertheless, the *k2fix_gr_rcs_w9* instance shows a larger upper bound of the explored search

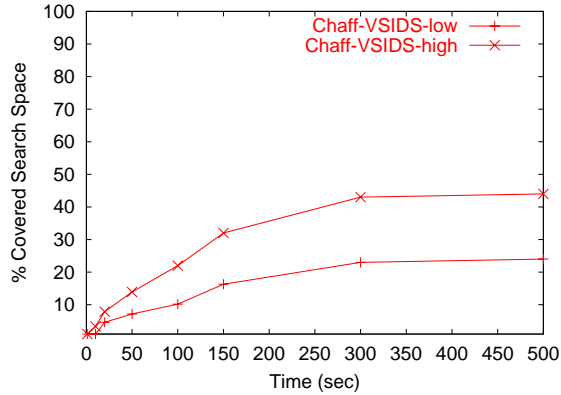


Fig. 7. Search space coverage for *k2fix_gr_rcs_w9.cnf*

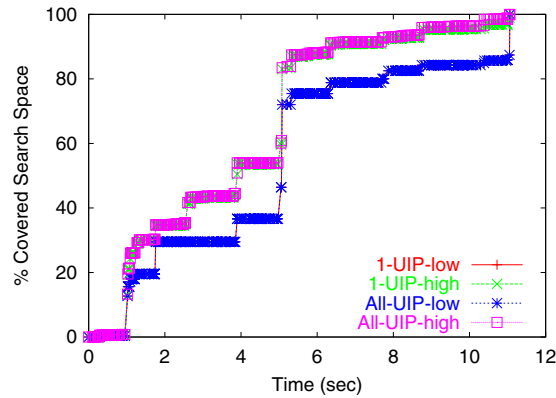


Fig. 8. Search space coverage of the *queueinvar8* instance using the 1 UIP vs. All UIP conflict analysis learning scheme

space using the fixed decision order as opposed to VSIDS. However, since the ranges for both heuristics overlap, it's hard to identify the optimal decision heuristic.

Hard problem prediction. Table V also shows the difficulty of solving the FPGA routing instances as opposed to other hard instances for the given decision heuristics and SAT solvers. Figure 7 shows a detailed space coverage analysis of the *k2fix_gr_rcs_w9* instance after unsuccessfully trying to solve it with Chaff for up to 500 seconds. Perhaps, this method can be used as a metric to rate the difficulty of SAT instances and assist SAT solver developers in improving their SAT tools.

One UIP vs. all UIPs conflict analysis. Recently, [28] analyzed various conflict clause learning schemes. They found that different learning schemes can significantly effect the behavior of SAT solvers. Based on various EDA instances, they were able to prove that the learning scheme based on the first Unique Implication Point (UIP) [16] of the implication graph can be very effective in solving SAT problems in comparison with other schemes such as the “All UIP” approach. In order to further confirm this conclusion, we plotted the growth range, using the “All UIP” and the “1 UIP” approach, for the *queueinvar8* instance from the bounded model checking set. We implemented both approaches in SATIRE. Figure 8 shows the runs using the SATIRE SAT solver. The coverage percentage was measured after each backtrack call. As the plot clearly shows, the addition of all UIPs resulted in a minor benefit and perhaps slowed the search process as additional time is spent to generate all the UIP clauses. This detailed analysis of the internals of the search process provides

TABLE VI: Percentage of explored search space for satisfiable instances with different numbers of satisfying assignments

| Benchmark | Explored Space, % |
|------------------------|-------------------|
| aim-200-1_6-yes1-1.cnf | 99.999 |
| ssa7552-160.cnf | 28.125 |

a better understanding of the problem’s structure and the effectivity of the SAT solver and enhancement being tested.

Number of satisfiable assignments. As mentioned earlier, the search space will never be totally explored in “satisfiable” instances, as SAT solvers typically abort after identifying the first satisfying assignment. However, in some cases, several satisfying assignments, if not all, are needed. An example is to identify all possible primary input assignments for a circuit that would minimize the total gate delay. An insight into the number of possible satisfying assignments can be very helpful. A satisfiable instance in which a satisfying assignment is identified at an early stage of the search process is likely to have many satisfying assignments. In contrast, an instance that identifies a satisfying assignment after exploring almost the complete search space probably has a few satisfying assignments only. In order to test our assumption, we selected two satisfiable instances from the DIMACS set [10], namely the *aim-200-1_6-yes1-1.cnf* and *ssa7552-160.cnf*. The former is known to have a single satisfying assignment only, whereas the latter represents a stuck-at-fault problem with many satisfying assignments. Both instances were solved by Chaff in less than a second. We measured the explored search space after the search was completed for a single satisfying assignment. Table VI shows the results.

As expected, the percentage of the search space explored by the *aim** instance was tremendously larger than the *ssa** instance.

Again, as in the experiments in Section 6.1, the accuracy of our results are significant. Although a user specified error limit of 20% is set, out of the 78 runs, 47, 6, 16, 8, reported results with 100%, >99%, 90%~99%, 80%~90% accuracy.

In terms of run time and memory consumption, constructing the ZBDDs is fast and is usually dependent on the size of the clauses. Furthermore, the high compression power of the ZBDD data structure utilizes less memory than a list data structure. As mentioned in Section 5.2, computing the search space coverage with an unrestricted bound is done by a single traversal of the ZBDD. On the other hand, the restricted bound and the exact count methods are slower, since additional ZBDD nodes are created during the ZBDD traversal. The size of the ZBDD, however, doesn’t grow exponentially since the additional ZBDD nodes are removed as soon as the function sizes of their corresponding formulas are computed.

One way to reduce the run time and memory consumption is to only analyze conflict-induced clauses of size k or less. In general, smaller clauses are more useful in measuring the explored search space and require less ZBDD construction time and fewer ZBDD nodes. This approach, however, can only be used to measure the lower bound of the explored search space. For the instances reported in Table V and Table VI, Satometer was able to compute the search space coverage for almost all instances in less than a second each.

7 Summary and Conclusions

We described Satometer, a tool that measures the percentage of search space explored by a SAT solver. The tool can provide helpful diagnostic information, either during or at the conclusion of a SAT run. We believe that tools such as this are needed to complement the powerful SAT engines

that have been developed in recent years. We plan to identify other metrics that can help characterize a search process (e.g., the maximum number of satisfied clauses encountered at any point during the search), to look for ways to further improve the efficiency of Satometer (e.g., by caching computation results), and to use it to analyze the performance of solvers on hard SAT instances. We are also planning to integrate Satometer into known SAT solvers and use the search space information to improve decision and restart heuristics.

8 Acknowledgments

This work is funded by the DARPA/MARCO Gigascale Silicon Research Center and an Agere Systems/SRC Research fellowship.

9 References

- [1] F. Aloul, M. Mneimneh, and K. Sakallah, "Backtrack Search Using ZBDDs," in *International Workshop on Logic Synthesis (IWLS)*, 293-297, 2001.
- [2] F. Aloul, J. Silva, and K. Sakallah, "An Experimental Study of Satisfiability Search Heuristics," in *Proc. of the Design, Automation and Test in Europe Conference (DATE)*, 745, 2000.
- [3] R. Bayardo Jr. and R. Schrag, "Using CSP look-back techniques to solve real world SAT instances," in *Proc. of the 14th National Conference on Artificial Intelligence (AAAI)*, 203-208, 1997.
- [4] A. Biere, A. Cimatti, E. Clarke, M. Fujita, and Y. Zhu, "Symbolic Model Checking using SAT procedures instead of BDDs," in *Proc. of the Design Automation Conference (DAC)*, 317-320, 1999.
- [5] R. Bryant, "Graph-based algorithms for boolean function manipulation," in *Proc. of the IEEE Transactions on Computers*, 35(8), 677-691, 1986.
- [6] P. Chatalic and L. Simon, "Multi-Resolution on Compressed Sets of Clauses," in *Proc. of the International Conference on Tools with Artificial Intelligence*, 2-10, 2000.
- [7] P. C. Chen, "Heuristic sampling: A method for predicting the performance of tree searching programs," in *Proc. of the SIAM Journal on Computing*, 21(2), 295-315, 1992.
- [8] J. Crawford, M. Ginsberg, E. Luks, and A. Roy, "Symmetry-Breaking Predicates for Search Problems," in *Knowledge Representation: Principles of Knowledge Representation and Reasoning*, 148-159, 1996.
- [9] M. Davis, G. Logemann, and D. Loveland, "A Machine Program for Theorem Proving," in *Proc. of the Communications of the ACM*, 5(7), 394-397, 1962.
- [10] DIMACS Challenge benchmarks in <ftp://Dimacs.rutgers.EDU/pub/challenge/sat/benchmarks/cnf>.
- [11] D. E. Knuth, "Estimating the efficiency of backtrack programs," in *Proc. of the Mathematics of Computation*, 29, 121-136, 1975.
- [12] D. Kokotov, I. Shlyakhter, "Progress Bar for SAT Solvers," unpublished manuscript, <http://sdg.lcs.mit.edu/satsolvers/progressbar.html>, 2000.
- [13] T. Larrabee, "Test Pattern Generation Using Boolean Satisfiability," in *Proc. of the IEEE Transactions on Computer-Aided Design*, 11(1), 4-15, 1992.
- [14] M. Lobbing, O. Schroer, and I. Wegner, "The Theory of Zero-Suppressed BDDs and the Number of Knight's Tours," in *IFIP WG 10.5 Workshop on Applications of the Reed-Muller Expansion in Circuit Design*, 1995.
- [15] J. Marques-Silva, "Algebraic Simplification Techniques for Propositional Satisfiability," in *Proc. of the International Conference on Principles and Practise of Constraint Programming*, 2000.

- [16]J. Marques-Silva and K. Sakallah, "GRASP: A Search Algorithm for Propositional Satisfiability," in *Proc. of the IEEE Transactions on Computers*, 48(5), 506-521, 1999.
- [17]S. Minato, "Zero-Suppressed BDDs for Set Manipulation in Combinatorial Problems," in *Proc. of the Design Automation Conference (DAC)*, 272-277, 1993.
- [18]M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an Efficient SAT Solver," in *Proc. of the Design Automation Conference (DAC)*, 530-535, 2001.
- [19]G. Nam, F. Aloul, K. Sakallah, and R. Rutenbar, "A Comparative Study of Two Boolean Formulations of FPGA Detailed Routing Constraints," in *Proc. of the International Symposium on Physical Design (ISPD)*, 222-227, 2001.
- [20]P. Purdom, "Tree size by partial backtracking," in *Proc. of the SIAM Journal on Computing*, 7(4), 481-491, 1978.
- [21]B. Selman, H. Kautz, and B. Cohen, "Noise strategies for local search," in *Proc. of the Eleventh National Conference on Artificial Intelligence*, 337-343, 1994.
- [22]L. Silva, J. Silva, L. Silveira and K. Sakallah, "Timing Analysis Using Propositional Satisfiability," in *Proc. of the IEEE International Conference on Electronics, Circuits and Systems*, 1998.
- [23]F. Somenzi, CUDD: CU Decision Diagram Package, University of Colorado at Boulder, <ftp://vlsi.colorado.edu/pub/>.
- [24]G. Stalmarck, "System for Determining Propositional Logic Theorems by Applying Values and Rules to Triplets that are Generated from Boolean Formula," *Swedish Patent no. 467,076*, *European Patent no. 0403454*, and *United States Patent no. 5,276,897*, 1994.
- [25]P. R. Stephan, R. K. Brayton and A. L. Sangiovanni-Vincentelli, "Combinational Test Generation Using Satisfiability," in *Proc. of the IEEE Transactions on Computer-Aided Design*, 15(9), 1167-1176, 1996.
- [26]M. Velev and R. Bryant, "Boolean Satisfiability with Transitivity Constraints," in *Proc. of the Conference on Computer-Aided Verification (CAV)*, 86-98, 2000.
- [27]J. Whittemore, J. Kim, and K. Sakallah, "SATIRE: A New Incremental Satisfiability Engine," in *Proc. of the Design Automation Conference (DAC)*, 542-545, 2001.
- [28]L. Zhang, C. Madigan, M. Moskewicz, S. Malik, "Efficient Conflict Driven Learning in a Boolean Satisfiability Solver," in *Proc. of the International Conference on Computer Aided Design (ICCAD)*, 279-285, 2001.
- [29]H. Zhang, "SATO: An Efficient Propositional Prover," in *Proc. of the International Conference on Automated Deduction*, 272-275, 1997.