Fast Generation of Lexicographic Satisfiable Assignments: Enabling Canonicity in SAT-Based Applications

Ana Petkovska[1]
ana.petkovska@epfl.ch

Alan Mishchenko[2]
alanmi@berkeley.edu

Mathias Soeken[1]
mathias.soeken@epfl.ch

Giovanni De Micheli[1]
giovanni.demicheli@epfl.ch

Robert Brayton[2]
brayton@berkeley.edu

Paolo Ienne[1]
paolo.ienne@epfl.ch

[1]Ecole Polytechnique Fédérale de Lausanne (EPFL)
School of Computer and Communication Sciences, Lausanne, Switzerland
[2]University of California, Berkeley, Department of EECS, Berkeley, USA

ABSTRACT

Lexicographic Boolean satisfiability (LEXSAT) is a variation of the Boolean satisfiability problem (SAT). Given a variable order, LEXSAT finds a satisfying assignment whose integer value under the given variable order is minimum (maximum) among all satisfiable assignments. If the formula has no satisfying assignments, LEXSAT proves it unsatisfiable, as does the traditional SAT. The paper proposes an efficient algorithm for LEXSAT by combining incremental SAT solving with binary search. It also proposes methods that use the lexicographic properties of the assignments to further improve the runtime when generating consecutive satisfying assignments in lexicographic order. The proposed algorithm outperforms the state-of-the-art LEXSAT algorithm—on average, it is 2.4 times faster when generating a single LEXSAT assignment, and it is 6.3 times faster when generating multiple consecutive assignments.

1. INTRODUCTION

Lexicographic satisfiability (LEXSAT) is a decision problem similar to the *satisfiability (SAT) problem*—for a given SAT formula it returns a satisfying assignment, if the problem is *satisfiable (SAT)*, or otherwise it returns *unsatisfiable (UNSAT)*. The only difference is that SAT can return any satisfying assignment, while LEXSAT returns deterministically the one whose integer value under a given variable order is the minimum (or maximum) among all satisfying assignments. The assignments with the minimum and maximum integer value are called *lexicographically smallest* and *lexicographically greatest* assignment, respectively. For simplicity, we assume that LEXSAT always generates the lexicographically smallest assignment, but the same principles apply when generating the lexicographically greatest one.

ICCAD'16 November 07-10, 2016, Austin, TX, USA
© 2016 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-4466-1/16/11.
DOI: http://dx.doi.org/10.1145/2966986.2967040

Example 1. *Assume a 4-input function $f(x_1, x_2, x_3, x_4)$ with the satisfying assignments for the inputs* $\{0001, 0101, 1010, 1011, 1101\}$. *SAT can return any of the given assignments, while LEXSAT always returns either the lexicographically smallest assignment* 0001 *or the lexicographically greatest assignment* 1101*, depending on the user preference.*

Knuth [5] mentions two implementations of an algorithm for generating satisfying assignments in a lexicographic order. The first one [5, Ex. 7.2.2.2-109] calls a SAT solver multiple times. The first call generates a satisfying assignment that is iteratively minimized with the successive SAT calls. On the other hand, the second one [5, Ex. 7.2.2.2-275] implements the same concept by modifying the decision heuristic of the SAT solver to perform decisions on the input variables in a given order, while for the other variables decisions can be performed in any order. However, Knuth [5] does not evaluate the performance of these two algorithms.

Independently, Nadel and Ryvchin [8] propose Knuth's LEXSAT algorithm, which they call `OBV-BS`, in the context of *Satisfiability Modulo Theories (SMT)* solving. They also propose another algorithm integrated in a SAT solver. Their results show, first, that the two proposed algorithms are faster than algorithms based on SMT solvers. Second, they show that the `OBV-BS` algorithm, which uses the SAT solver repeatedly, is slower than the one integrated in the SAT solver but it is more robust—it succeeds to find solutions for difficult instances for which the integrated one exceeds the given time limit. Generalization of Knuth's algorithm is also proposed by Marques-Silva et al. [6].

With this paper, we propose a scalable and fast LEXSAT algorithm that also uses the SAT solver repeatedly. But, instead of starting from a satisfying assignment that is iteratively minimized, we start from a potential assignment that is the lexicographically smallest assignment that might be satisfiable. Then, for each variable, we iteratively either confirm that its assignment is identical to the one in the lexicographically smallest satisfying assignment, or we increase it, if possible. To achieve a good performance, we also propose a version of the algorithm that is based on the concept of binary search. Moreover, we propose methods that use the lexicographic properties of the assignments to further improve the runtime when consecutive satisfying assignments are generated in lexicographic order, which is required in applications such as the canonical SAT-based SOP generation [9]. For all algorithms, we propose to use incremental SAT solving to mimic the alternative implementation that modifies the SAT solver, which leads to a good performance while keeping the SAT solver unmodified for general use. The experimental results show that our algorithm is faster than the first algorithm proposed by Knuth [5] when generating single and multiple consecutive LEXSAT assignments.

Following, Section 2 motivates using LEXSAT for electronic design automation applications. Section 3 gives the background information. In Section 4, we describe two versions of our algorithm and the methods for improving the runtime. Our experimental setup and results are presented in Section 5. In Section 6, we argue that our implementation with repetitive SAT calls is expected to be as efficient as an implementation that modifies the SAT solver. We conclude and present ideas for future work in Section 7.

2. APPLICATIONS OF LEXSAT

Although LEXSAT has emerged recently, it is already shown useful for many *Electronic Design Automation (EDA)* applications. For example, Soeken et al. [11] show that LEXSAT enables heuristic NPN classification of large functions with up to 194 variables. In this case, LEXSAT improves an algorithm that was previously limited to functions with up to 16 variables, for which truth tables could be computed [4].

LEXSAT is also used for fixing cell placement during the physical design stage of an industrial computer-aided design flow [8]. By finding the maximal value of a bit-vector, which encodes that a potential violation is solved, a fixer tool generates a placement that has as few violations as possible while giving preference to fixing high-priority violations that are encoded with the most significant bits of the bit-vector.

Another example is a recent work [9] where LEXSAT enables generation of canonical *Sums Of Products (SOPs)* using a SAT solver because it generates assignments in a deterministic lexicographic order. Moreover, assuming a function $f(x_1, \ldots, x_n)$ if the assignments of the d most left variables x_i, where $1 \leq i \leq d$ for some $d \leq n$, are fixed to some value, LEXSAT would generate an assignment that is lexicographically closest to the value defined when the d most left variables are assigned to the fixed values and the rest of the variables x_j, where $d + 1 \leq j \leq n$ are assigned to 0.

Example 2. *For the function $f(x_1, x_2, x_3, x_4)$ from Example 1, if we fix the most left variable x_1 to 1, then LEXSAT returns the assignment* 1010 *as lexicographically smallest, because it is the satisfying assignment with the smallest integer value after the assignment* 1000.

With this, LEXSAT enables generating canonical simulation vectors used to generate canonical signatures for Boolean functions using a SAT solver. Similarly, applications such as constraint solving [14] and random assignment generation [7] can benefit from LEXSAT because it can derive the closest satisfying assignments for random valuations of inputs.

In general, since LEXSAT generates deterministic assignments, it enables canonicity in SAT-based applications with two important consequences: On the one hand, the result of computation depends only on the Boolean function and the user-specified variable order (and is independent of the SAT solver and the problem representation, in particular, of the CNF generation algorithm). On the other hand, subproblems encountered during SAT solving can be cached in a way similar to how BDD-based applications cache the results of intermediate computations, resulting in runtime reduction. To this end, BDD-based applications maintain a hash table mapping BDD nodes into results of computation for these nodes. Similarly, a SAT-based application can use LEXSAT to compute a canonical representation of Boolean functions (such as the canonical SOP mentioned above). This canonical representation can be used as a hash key in a table of computed results, similarly to how BDD nodes are used as hash keys in BDD-based applications.

Additionally, algorithms for approximate computing [10, 13] can use LEXSAT to compute the worst-case error by finding the lexicographically greatest solution for the difference between the approximate output and a correct reference version for all possible inputs. In formal verification, LEXSAT can analyze bugs that the SAT solver finds when solving verification instances. Suppose, for example, a satisfying assignment is found that indicates a mismatch between the specification and the implementation of a hardware design. LEXSAT can determine the lexicographically closest correct minterms before and after the buggy minterm. The difference between the two correct minterms outlines the region of the input space where the bug is present. When one bug is characterized in this way, a question can be asked: are there other bugs before and after the given one in the lexicographical order? Repeatedly calling LEXSAT allows exploring the input space step by step, and understanding the distribution and the size of buggy regions, which can provide crucial information for debugging.

In summary, an appealing aspect of LEXSAT is that it enables canonicity in SAT-based applications, leading to the same benefits as BDD-based applications reap from the canonicity of BDDs, which are unique for a given function and for a given variable order. Further, there could be practically important applications of LEXSAT in verification, such as "canonical" random simulation based on evenly-distributed input patterns, or bug characterization based on exploration of input space performed by LEXSAT.

3. BACKGROUND INFORMATION

In this section, we give the terminology associated with Boolean functions, and the SAT and LEXSAT problems.

3.1 Boolean Functions

For a variable v, a *positive literal* represents the variable v, while the *negative literal* represents its negation $\bar{v}$. A *cube*, or product, c, is a Boolean product (AND, $\cdot$) of literals, $c = l_1 \cdot \cdots \cdot l_k$. If a variable is not represented by a negative or a positive literal in a cube, then it is represented by a *don't-care* $(-)$, meaning that it can take both values 0 and 1. A cube with n don't-cares covers 2^n minterms. A *minterm* is the smallest cube in which every variable is represented by either a negative or a positive literal. Let $f(X) : B^n \rightarrow \{0, 1, -\}$, $B \in \{0, 1\}$, be an *incompletely specified Boolean function* of n variables $X = \{x_1, \ldots, x_n\}$. The *support set* of f is the subset of variables that determine the output value of the function f. Any Boolean function can be represented as a two-level *sum of products (SOP)*, which is a Boolean sum (OR, $+$) of cubes, $S = c_1 + \cdots + c_m$.

A *canonical representation* is a unique representation for a function under certain conditions. For example, given a Boolean function and a fixed input variable order, a canonical SOP is an SOP independent of the original representation of the function.

3.2 Boolean Satisfiability

A disjunction (OR, $+$) of literals forms a *clause*, $t = l_1 + \cdots + l_k$. A *propositional formula* is a logic expression defined over variables that take values in the set $\{0, 1\}$. To solve a SAT problem, a propositional formula is converted into its *Conjunctive Normal Form (CNF)* as a conjunction (AND, $\cdot$) of clauses, $F = t_1 \cdot \cdots \cdot t_k$. Algorithms such as the Tseitin transformation [12] convert a Boolean function into a set of CNF clauses.

A *satisfiability (SAT) problem* is a decision problem that takes a propositional formula in CNF form and returns that the formula is *satisfiable (SAT)* if there is an assignment of variables from the formula for which the CNF evaluates to 1. Otherwise, the propositional formula is *unsatisfiable (UNSAT)*. A program that solves SAT problems is called a *SAT solver*. SAT solvers provide a *satisfying assignment* when the problem is satisfiable.

Modern SAT solvers can receive as input one or more *assumptions*, which are single-literal clauses that hold only for one specific invocation of the SAT solver. The process of determining the satisfiability of a problem under given assumptions is called *incremental SAT solving*. Some SAT solvers support an internal stack of assumptions, which allows for adding and removing assumptions between consecutive SAT calls via a *push/pop mechanism*. This enables preserving the state of the SAT solver between incremental runs, while incremental runs themselves allow for reusing learned clauses from previous calls of the SAT solving procedure. Thus, both incremental SAT solving with assumptions, and incremental adding/removing of assumptions lead to flexibility and efficiency in SAT-based applications.

Example 3. *For the function $f(x_1, x_2, x_3, x_4)$ from Example 1, if we give the assumption $x_1 = 1$ as input, then the SAT solver returns one of the assignments* 1010, 1011, *or* 1101, *because those assignments are satisfiable considering the given assumption.*

3.3 Lexicographic Boolean Satisfiability

The *lexicographic satisfiability (LEXSAT) problem* is a variation of the SAT problem that takes a propositional formula in CNF form and a given variable order, and returns a satisfying variable assignment whose integer value under the given variable order is minimum (maximum) among all satisfying assignments. If the formula has no satisfying assignments, LEXSAT proves it unsatisfiable.

As described in Section 1, Knuth [5] proposes two solutions for generating a LEXSAT assignment. In this paper, we compare to the first implementation that calls the SAT solver multiple times. Assuming a function $f(x_1, \ldots, x_n)$, with the first call, the algorithm generates an *initial satisfying assignment* $a_1 \ldots a_n$, or terminates if the problem is UNSAT. Then, if the problem is SAT, it minimizes the assignment iteratively. For this, a pointer d is set to 0 before the first iteration, and later points to the next variable that is assigned to 1 and can be flipped to 0 to decrease the assignment. Assignments for the variables x_i for $1 \leq i < d$ are considered to be fixed. Thus, to minimize the assignment, first, d is set to the index of the next variable that is assigned to 1. If $d > n$, then no variable in the assignment can be flipped, and the algorithm returns $a_1 \ldots a_n$. Otherwise, using the assumption mechanism, the SAT solver is called again with the assumptions $x_i = a_i$, for $1 \leq i < d$, and $x_d = 0$. If the problem is SAT, the assignment $a_1 \ldots a_n$ is updated with the newly received assignment; otherwise, the old assignment is kept. Finally, it performs another iteration for minimization to find the next non-fixed 1 to be flipped.

Example 4. *For a function $f(x_1, x_2, x_3, x_4, x_5)$, assume that the assignment* 00101 *is received with the first SAT call. Then, in the first iteration for minimization, the pointer d is set to* 3*, since x_3 is the first variable that can be flipped from* 1 *to* 0*. Next, the SAT solver is called with the assumptions $x_1 = 0$, $x_2 = 0$, $x_3 = 0$. If the problem is UNSAT, the value of x_3 remains* 1*, since there is no SAT assignment that satisfies the given assumptions (i.e., that starts with* 000*); thus, the old assignment is kept and in the second iteration for minimization d is set to* 5*. Otherwise, assuming that the SAT solver returns the assignment* 00010*, it is considered as a potential assignment in the second iteration, so $d = 4$.*

4. GENERATING LEXICOGRAPHIC SAT ASSIGNMENTS

In this section, we first describe a simple and a binary search-based version of our algorithm for generation of LEXSAT assignments. Then, we describe several methods that improve their runtime when generating consecutive LEXSAT assignments.

4.1 Simple Version

Instead of using a SAT solver to find the initial assignment, our algorithm receives as input an initial assignment $a_1 \ldots a_n$ that in this case is smaller or equal to the next LEXSAT assignment. When generating consecutive LEXSAT assignments, this enables the search to start from the last generated LEXSAT assignment. For the first assignment or when generating non-consecutive assignments, for a function $f(x_1, \ldots, x_n)$, the initial assignment is $a_i = 0$ for $1 \leq i \leq n$. Having this initial assignment, our algorithm iteratively verifies if the assignment of each variable can be fixed or it should be increased, and converts the initial assignment into the LEXSAT assignment that is returned as output.

Basic idea. A simple version of our algorithm fixes the assignments of the variables one by one. A pointer d, which is initially set to 1, gives the index of the first non-fixed variable whose assignment should be fixed, while for the previous variables the assignments $x_i = a_i$, for $1 \leq i < d$, are already fixed. To fix the assignments, a SAT solver is called iteratively with the assumptions $x_i = a_i$, for $1 \leq i \leq d$. If the problem is SAT, then there is a satisfying assignment that starts with $a_1 \ldots a_d$ and d is incremented. Otherwise, if there is no SAT assignment that starts with $a_1 \ldots a_d$, the problem is UNSAT. In this case, if $a_d = 0$, we set $a_d = 1$, set $a_i = 0$ for $d < i \leq n$ to keep the assignment the smallest possible for the future iterations, and perform another iteration. But, if the problem is UNSAT when $a_d = 1$, then there is no satisfying assignment both when $a_d = 0$ and $a_d = 1$, and thus the algorithm returns UNSAT. Once $d > n$, the assignments for all variables are fixed and $a_1 \ldots a_n$ is returned as a LEXSAT assignment.

Example 5. *To generate the first LEXSAT assignment for a function $f(x_1, x_2, x_3, x_4, x_5)$, the received initial assignment is* 00000*. Initially, $d = 1$ and the first SAT call assumes $x_1 = 0$. If the problem is SAT, then d is incremented* *to $d = 2$, and in the next iteration the SAT call assumes $x_1 = 0$ and $x_2 = 0$. Otherwise, if the problem is UNSAT, we flip $a_1 = 1$, and iterate with the assumption $x_1 = 1$. This time, if we receive SAT, we increment d, and in the next iteration the SAT call assumes $x_1 = 1$ and $x_2 = 0$. But, if we receive UNSAT again, it means that there is no assignment both with $x_1 = 0$ and $x_1 = 1$, and thus we return UNSAT.*

Improving performance by learning from satisfying assignments. Similarly to the algorithm by Knuth [5] described in Section 3.3, when the SAT solver returns a satisfying assignment, we can learn some variable assignments from it. Thus, we always save the last satisfying assignment, and use it as following. First, same as before, if the first variable assigned to 1 after d is on position $d+t$, where $1 \leq t \leq n-d$, then we can learn and fix to 0 the $t-1$ variables between d and $d+t$. Moreover, in our case, the potential assignment $a_1 \dots a_n$ is the lexicographically smallest assignment that might be satisfiable. Thus, if the potential assignment for a variable x_i is $a_i = 1$, then we cannot flip it to 0 to minimize the assignment as in the algorithm by Knuth. This allows us to learn from the SAT solver and fix all assignments up to the first variable for which the potential assignment and the assignment returned by the SAT solver differ. Assume that the last satisfying assignment returned by the solver is $v_1 \dots v_n$. Instead of incrementing d by 1, we can set it to the index i, such that $a_j = v_j$ for $1 \leq j < i$ and $a_i \neq v_i$. Finally, same as Knuth's algorithm, for a given literal x_d, where $1 < d \leq n$, with $v_d = 1$, if we get UNSAT when assuming $x_d = 0$, we can immediately fix x_d to 1, as this value is confirmed by the last satisfying assignment.

EXAMPLE 6. *For a function $f(x_1, \dots, x_6)$, assume that* 101000 *is received as an initial assignment. When the SAT solver is called with the assumption $x_1 = 1$, it returns a satisfying assignment* 101101*, which is saved as a last satisfying assignment. Besides fixing $x_1 = 1$, from this assignment, we can learn and fix $x_2 = 0$ and $x_3 = 1$, because their initial assignments are confirmed by the last satisfying assignment. The variable x_4 is the most left variable for which the assignments differ and might be flipped to* 0*, so for the next iteration we set $d = 4$ and call the SAT solver with the assumptions $x_1 = 1$, $x_2 = 0$, $x_3 = 1$ and $x_4 = 0$. If the problem is SAT, we fix x_4 to* 0 *and update the last satisfying assignment. But, if the problem is UNSAT, from the last satisfying assignment* 101101*, we already know that the problem is satisfiable when $x_4 = 1$, we can additionally fix $x_5 = 0$, and set $d = 6$ for the next iteration.*

4.2 Binary Search-Based Version

To further enhance the simple version of our algorithm, instead of fixing the assignments of variables one by one, we propose to set the pointer d using binary search. Two additional pointers l and r show the first and last variable with non-fixed assignments, respectively, and initially are set $l = 1$ and $r = n$. Then, d is set to the middle variable in the array of variables bounded by x_l and x_r. This assumes the assignments of the left half of the variables x_i, where $1 \leq i \leq d$, in the first iteration. Later, whenever the SAT solver returns SAT, it confirms that a satisfying assignment that starts with $a_1 \dots a_d$ exists. As shown in Section 4.1, from the returned satisfying assignment we can confirm and fix t additional assignments from the potential assignment, where $0 < t < n - d$. After this step, the assignments for the variables x_i, where $1 \leq i \leq d + t$ are fixed. For the next iteration, we set $l = d + t + 1$ and $r = n$ to assume the assignments for the non-fixed variables in the right half. Otherwise, if the problem is UNSAT, if $a_d = 0$, then we proceed as in the simple version of the algorithm: we set $a_d = 1$, set $a_i = 0$ for $d < i \leq n$ for the future iterations, and perform another iteration; while, if $a_d = 1$, for the next iteration $r = d - 1$ to assume fewer non-fixed variables.

EXAMPLE 7. *To generate the first LEXSAT assignment for a function $f(x_1, x_2, x_3, x_4, x_5, x_6)$, the initial assignment* 000000 *is received as input. Initially, $l = 1$, $r = 6$ and $d = 3$. Thus, the first SAT call would assume $x_1 = 0$, $x_2 = 0$, and $x_3 = 0$. If the problem is SAT and the satisfying assignment* 000010 *is returned, then the assignment $x_4 = 0$ is learned since it is the same in the initial assignment, and the values of the pointers are updated to $l = 5$, $r = 6$ and $d = 5$ for the next iteration. Otherwise, if it is UNSAT, we would first try the assumptions $x_1 = 0$, $x_2 = 0$, and $x_3 = 1$. This time, if we receive SAT we would proceed same as before; while, if we receive UNSAT again, for the next iteration, we would update the values of the pointers to $l = 1$, $r = 2$ and $d = 1$ to assume less variables.*

4.3 Runtime Improvement when Generating Consecutive LEXSAT Assignments

Applications such as the SAT-based generation of canonical SOPs [9] generate consecutive satisfying assignments in lexicographic order. To allow generation of new satisfying assignments, each generated assignment is added to the SAT solver as a *blocking clause*, which is an additional clause that blocks known solutions of the SAT problem.

EXAMPLE 8. *For the function $f(x_1, x_2, x_3, x_4)$ from Example 1, the first LEXSAT call returns the assignment* 0001*. If we add this assignment as a blocking clause to the SAT solver, with the next LEXSAT call the assignment* 0101 *is generated because it is the lexicographically smallest satisfying assignment that is not blocked.*

For these types of algorithms, we present three methods that improve the runtime of the newly proposed algorithm by using the lexicographic properties of the assignments and the fact that, after the first LEXSAT call, the received initial assignment is the last generated LEXSAT assignment.

Fixing leading 1s. When generating consecutive LEXSAT assignments, after some time, assignments that start with one or more consecutive 1s are generated. Generating a lexicographically smallest SAT assignment that starts with one or more consecutive 1s implies that all unblocked satisfying assignments are greater than the generated, and therefore also start with the same number of 1s. Thus, when generating a LEXSAT assignment, assume that $a_i = 1$ for $1 \leq i \leq t$, for some $t \leq n$ (i.e., the received initial assignment starts with t consecutive 1s). Then, we can fix these t assignments for the corresponding variables x_i, and the initial value of l (or of d in the simple version) is set to $t + 1$ to point the first variable that is assigned 0.

EXAMPLE 9. *For a function $f(x_1, x_2, x_3, x_4, x_5)$, assume that the last generated LEXSAT assignment is* 11010 *and it is received as an initial assignment. Since the next LEXSAT assignment has to be greater than the last generated assignment, we know that it also starts with* 11*. Thus, we can skip*

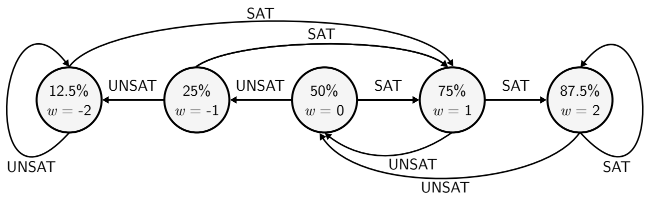

Figure 1: Changing the percentage of assumed variables for the first SAT call of LEXSAT depending on the success of the previous first SAT calls. In this case, at most three iterations of binary search are performed at once.

assuming assignments for x_1 and x_2, and directly fix them to 1. *Initially, l is set to* 3, *r to* 5, *d is computed to be* 4, *and thus, the first SAT call would be with the assumptions* $x_1 = 1$, $x_2 = 1$, $x_3 = 0$, *and* $x_4 = 1$.

Correcting the initial assignment. When generating consecutive LEXSAT assignments, for the first LEXSAT assignment, the initial assignment received as input assigns all variables to 0. Afterwards, for the following LEXSAT assignments, the initial assignment is equal to the last generated LEXSAT assignment. But, the first unblocked assignment is the one whose integer value is one unit greater than the last LEXSAT assignment. Thus, assuming that the last LEXSAT assignment ends with t 1s, for some $t \leq n$, i.e., $a_{n-i} = 1$ for $0 \leq i < t$, we flip the most right 1s by setting $a_{n-i} = 0$ and the first 0 from the right by setting $a_{n-t} = 1$.

Example 10. *For a function $f(x_1, x_2, x_3, x_4, x_5)$, assume that the assignment* 11011 *is generated with the previous LEXSAT call and received as an initial assignment. Since the next lexicographical assignment has to be greater than the last generated, the first possible satisfying assignment is* 11100. *Thus, we flip the* 1*s and the first* 0 *starting from the right to get the potential assignment* 11100.

Profiling the success of the first SAT calls. For the LEXSAT algorithm, we consider satisfiable SAT calls as successful because they confirm the assumed assignments, while unsatisfiable SAT calls are considered unsuccessful. Further, we propose to profile the success of the first SAT call from the LEXSAT algorithm and use this profile to alter the percentage of assumed assignments in the first SAT calls in the subsequent invocation of the LEXSAT algorithm based on binary search. This method does not apply to the simple version of the algorithm.

The binary search-based version always sets the pointer d to point the middle variable of the array of variables bounded by x_l and x_r. Thus, with the first SAT call we always assume the non-fixed assignments for the first 50% of the variables between x_l and x_r. In the next iterations, with every satisfiable SAT call, we increase the number of assumptions and add 50% more of the right subarray. With every unsatisfiable SAT call, we decrease the number of assumptions and the next time we use only 50% of the assignments of the left subarray. Thus, for example, assuming 75% of the assignments in the first SAT call is equivalent to having two consecutive iterations with successful SAT calls.

To profile and alter the percentage of assumed assignments in the first SAT call, we keep a variable w which tells us how many iterations to perform at once and in which direction we should perform them. We iterate $|w|$ times to decrease or increase the percentage when $w < 0$ or $w > 0$, respectively. Initially, $w = 0$, which means that we should assume 50% of the assignments. If the first SAT call is satisfiable, we increase w for 1 when $w \geq 0$ or we set $w = 1$ when $w < 0$. If the first SAT call is unsatisfiable, we decrease w for 1 when $w \leq 0$ or we set $w = 0$ when $w > 0$. Figure 1 shows how the percentage of assumed variables for the first SAT call and the value of w changes with the success of the first SAT calls. In this example, at most three iterations of binary search are performed at once.

Example 11. *For a function $f(x_1, \ldots, x_{10})$, assume that the assignment* 0000110000 *is received as an initial assignment and $w = 0$. Since, $l = 1$, $r = 10$ and $w = 0$, for the first SAT call d is computed as $d = \lfloor (1+10) \cdot 0.5 \rfloor = 5$. Thus, we assume $x_1 = 0$, $x_2 = 0$, $x_3 = 0$, $x_4 = 0$ and $x_5 = 1$. If this call is satisfiable, then w is set to* 1 *for the next LEXSAT assignment. Assume that with the following SAT calls the LEXSAT assignment* 0000110001 *is generated. Then, when generating the next LEXSAT assignment, for the first SAT call, $d = \lfloor (1 + 10) \cdot 0.75 \rfloor = 8$ because $w = 1$, so instead of assuming the initial assignments only for the first five inputs as before, we assume the assignments for the first eight inputs. For the remaining SAT calls of the current LEXSAT assignment, we always use the regular binary search algorithm, which always assumes 50% of the assignments.*

5. EXPERIMENTAL RESULTS

In this section, for convenience we call the algorithm from Knuth [5] `KLEX` (Section 3.3), and the simple and binary search-based versions of our algorithm `SIMPLE` and `BINARY`, respectively (Section 4). We implemented in ABC [2] the three algorithms `KLEX`, `SIMPLE`, and `BINARY`, as well as the methods for improving the runtime from Section 4.3. ABC features an integrated incremental SAT solver derived from an early version of MiniSAT [3]. Also, this SAT solver supports pushing and popping of assumptions.

To evaluate the runtime of the algorithms and the speedup achieved from the additional methods, we use the set of large MCNC benchmarks, as well as a set of logic tables from the instruction decoder unit [1], which we denote with LT-DEC. The names of the LT-DEC benchmarks are given in the form "[N_{PI}].[N_{PO}]", where N_{PI} is the number of primary inputs and N_{PO} is the number of primary outputs. For a given benchmark, each algorithm generates a user specified number of consecutive LEXSAT assignments for each *combinatorial output*, that is each primary output and each latch input. However, to avoid repeatedly calling the procedure for output functions with isomorphic circuit structure, we divide the outputs into equivalence classes. An *equivalence class* contains outputs that implement an identical function expressed over different inputs. Thus, for each benchmark, we actually generate LEXSAT assignments only for the representative of each class.

For a given function and a variable order, the LEXSAT assignments are deterministic and must be generated in the same order when generating consecutive LEXSAT assignments. The correctness of our algorithms is validated by generating assignments with each algorithm, and comparing them to ensure that all algorithms generate the same assignments in the same order. For generating a given number of LEXSAT assignments, the number of SAT calls depends on how often the algorithm calls the SAT solving procedure.

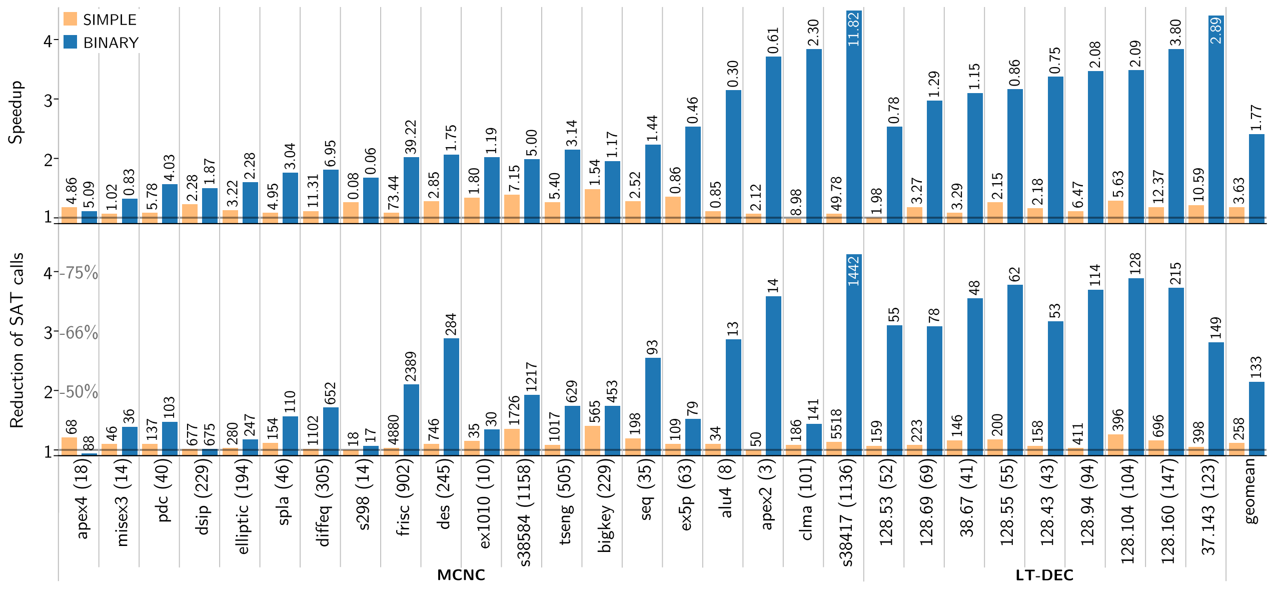

Figure 2: Performance of our algorithms `SIMPLE` and `BINARY` compared to the `KLEX` algorithm when generating a single LEXSAT assignment per combinatorial output. Next to each bar is the actual runtime (in milliseconds) and the number of SAT calls, respectively. Next to the name of the benchmark, we give the number of LEXSAT calls in brackets.

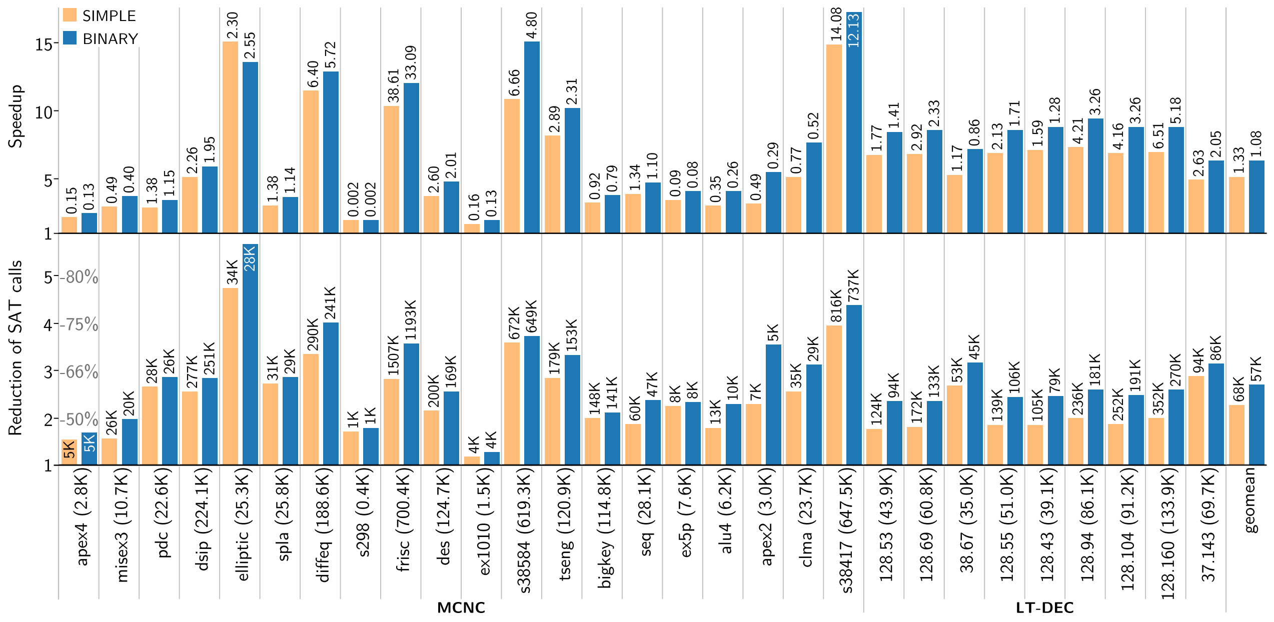

Figure 3: Performance of our algorithms `SIMPLE` and `BINARY` compared to `KLEX` when generating 1000 consecutive LEXSAT assignments per combinatorial output. Next to each bar is the actual runtime (in seconds) and the number of SAT calls (in thousands), respectively. Next to the name of the benchmark, in brackets, is the number of LEXSAT calls (in thousands).

Next we compare the runtime of `KLEX`, and the two versions of our algorithm `SIMPLE` and `BINARY` enhanced with the methods described in Section 4.3. We evaluate the three algorithms for both generation of a single and multiple consecutive LEXSAT assignments. Afterwards, we evaluate the speedup achieved by each of the additional methods.

5.1 Runtime Comparison

Generation of a single LEXSAT assignment. Some LEXSAT-based applications, such as the NPN classification [11], require multiple LEXSAT assignments, but they are not in a consecutive order or they are for different functions. Thus, first, we evaluate the runtime and number of SAT calls required by each algorithm for generating a single LEXSAT assignment. For each benchmark, a single LEXSAT assignment is generated per combinatorial output. Since the algorithms generate these assignments in few milliseconds, to get a precise comparison, we generate each LEXSAT assignment 1000 times, and then divide the total runtime by 1000. As Figure 2 shows, both versions `SIMPLE`

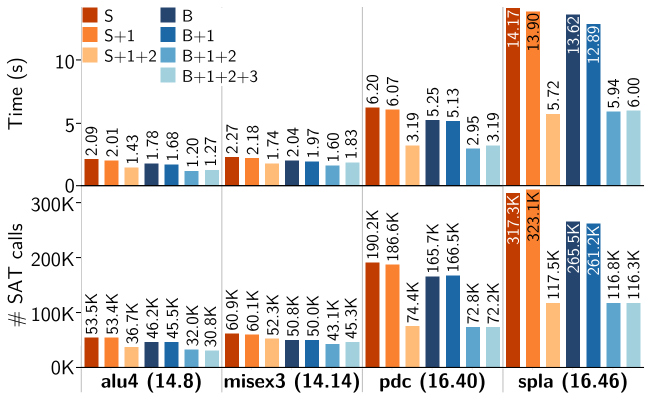

Figure 4: Runtime and number of SAT calls of SIMPLE (S) and BINARY (B) when different methods for improving the runtime are used for 4 benchmarks from the MCNC set. Next to the name of the benchmark, we give the number of combinatorial inputs and outputs, respectively.

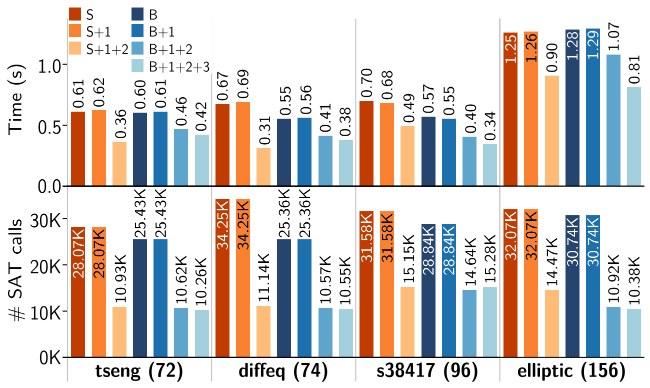

Figure 5: Runtime and number of SAT calls of SIMPLE (S) and BINARY (B) when different methods for runtime improvement are used for one of the outputs from 4 benchmarks from the MCNC set. Next to the name of the benchmark, we give its number of combinatorial inputs.

and BINARY perform better than KLEX for almost all benchmarks. Since the algorithmic steps of SIMPLE are very similar to those of KLEX when generating a single assignment, SIMPLE makes only 9.7% less calls to the SAT solver, and thus is only 14.7% faster than KLEX. On the other hand, assuming more assignments at once with BINARY leads to about 2x less SAT calls and 2x faster runtime than SIMPLE. Finally, BINARY is 2.4x faster than KLEX and makes 2.1x less SAT calls.

Generation of multiple consecutive LEXSAT assignments. On the other hand, some applications, such as the LEXSAT-based generation of canonical SOPs [9], require consecutive LEXSAT assignments. In this case, the methods described in Section 4.3 also contribute to reducing the runtime of SIMPLE and BINARY. For this experiment, we generate at most 1000 consecutive LEXSAT assignments for each combinatorial output. For each output we perform the experiment 5 times, and thus the presented results represent the average over 5 runs. As Figure 3 shows, both SIMPLE and BINARY outperform KLEX—for SIMPLE, we have 2.3x less SAT calls on average, which reduces runtime 5.1x, while for BINARY we have 2.7x less SAT calls on average, which reduces runtime 6.3x. Regarding the two proposed versions of our algorithm, on average, BINARY has 16.1% less SAT calls that contribute to 18.9% better runtime than SIMPLE.

Thus, BINARY has the best performance both when generating a single assignment and when generating multiple consecutive LEXSAT assignments.

5.2 Evaluation of the Methods for Runtime Improvement

Section 4.3 presented the following methods for runtime improvement when generating consecutive assignments.

1. Fixing leading 1s.
2. Correcting the initial assignment.
3. Profiling the success of the first SAT calls.

Since the method for fixing the leading 1s affects the runtime only when generating assignments in which the most significant bits are assigned to 1, we evaluate the methods by generating the complete truth table (i.e., generating all assignments for which the function evaluates to 1) for a subset of the MCNC benchmarks. The selected benchmarks have at most 16 combinatorial inputs, which means that, for each combinatorial output, we can have at most 65536 minterms when the function is 1. Similarly to before, the presented results represent the average over 5 runs. Figure 4 shows the runtime and number of SAT calls for four of the selected benchmarks. First, it shows the results when the algorithms SIMPLE (S) and BINARY (B) are used without the additional methods. We can see that fixing the leading 1s (S+1, B+1) decreases the runtime moderately. Contrarily, if we additionally correct the initial assignment (S+1+2, B+1+2) then the runtime decreases for 32%, on average. Finally, for BINARY, although the method for profiling the success of the first SAT calls in general decreases the number of SAT calls, for functions with small number of inputs it slightly increases the runtime. However, we have observed reduction of runtime for benchmarks with a large number of combinatorial inputs. Figure 5 shows the runtime and number of SAT calls required to generate 1000 minterms for a single output of 4 large MCNC benchmarks. The considered outputs have more than 70 combinatorial inputs. In this case, the method for fixing leading 1s does not affect the number of SAT calls because the most significant bits of all generated assignments are 0s.

Note that in Section 5.1 the results for SIMPLE and BINARY are obtained when all methods are used (i.e., with S+1+2 and B+1+2+3, respectively).

6. ON INTEGRATING THE LEXSAT ALGORITHMS IN A SAT SOLVER

The algorithms presented and evaluated in this paper use repeatedly the SAT solver. Another option is to modify the SAT solver such that it generates LEXSAT assignments. For convenience, we refer to them with OUTSAT and INSAT, respectively. Knuth [5, Ex. 7.2.2.2-275] suggests an INSAT implementation of KLEX. Nadel and Ryvchin [8] show that an INSAT algorithm is faster than an OUTSAT implementation of the KLEX algorithm, but unlike the OUTSAT implementation, it is not scalable for difficult instances. In this section, we discuss the difference in these two implementation options.

Generally, in an `INSAT` implementation, the SAT solver performs decisions on the input variables in the order and with the values given by the LEXSAT algorithm, while for the other variables decisions can be performed in any order. With this solution, to generate LEXSAT assignments for a function, a single SAT solver instance is created and, for each LEXSAT assignment, the procedure for SAT solving is called only once, with a given order for the input variables. Note that the concepts of the algorithms `SIMPLE` and `BINARY` can also be used to determine the order of issuing decisions and the values for the input variables.

On the other hand, in our `OUTSAT` implementations, incremental SAT solving allows generating multiple LEXSAT assignments of a function also by using only a single SAT solver instance. Moreover, for each LEXSAT assignment, the interfaces for pushing and popping assumptions, which we suggest to use, preserve the internal state of the solver between consecutive invocations of the SAT solving procedure. With this, on a higher level, we mimic the solution based on modifying the SAT solver. With such implementation, and by using the algorithm `BINARY`, we expect our `OUTSAT` implementation to be as fast as an `INSAT` implementation, but confirming this experimentally is left for future work.

Moreover, assume a function with n inputs for which the assignments of the first d inputs are already fixed, from some $1 \leq d \leq n$. In the `OUTSAT` implementation, the SAT solving procedure can and do change the order of decisions for the least significant $n - d - 1$ variables whose value is not yet fixed, when running the query to fix the value of the variable $d+1$. However, the `INSAT` implementation always makes the same decisions in the same order, and can not change the order even if that would lead to faster UNSAT calls during LEXSAT solving. Thus, for difficult instances, such as functions with large number of variables when different variable orders affect the efficiency of the SAT solving procedure, as well as when all calls are not satisfiable, an `OUTSAT` implementation is more scalable than an `INSAT` implementation.

7. CONCLUSION

This paper presents a novel variation of the Boolean satisfiability problem, called LEXSAT, which in addition to determining the status of a problem (satisfiable or unsatisfiable), also returns satisfying assignments that are minimum (maximum) considering a given variable order. We demonstrate that LEXSAT allows developing SAT-based algorithms that share desirable properties with BDDs but are less likely to suffer from the scalability problems that beset BDD-based computations in many EDA applications. In particular, LEXSAT can achieve *canonicity* of the computed results: when for the given Boolean function under the given variable order, the result is deterministic and independent of the SAT solver and the CNF generation algorithm.

The paper also proposes a fast binary search-based algorithm for generation of a single LEXSAT assignment, which is 2.4 times faster than the state-of-the-art LEXSAT algorithm. Furthermore, it proposes several improvements to the LEXSAT algorithms for the typical use-model when it is applied iteratively and the resulting satisfying assignments are monotonically increasing. For such use, the proposed algorithm enhanced with the new features is 6.3 times faster than an existing LEXSAT algorithm. Finally, we propose a way of using incremental SAT solving to improve performance without modifying the SAT solver.

We expect that LEXSAT has many potential uses in EDA. Future work on LEXSAT will focus on exploring several promising applications: SAT-based constraint simulation, SAT-based factoring, SAT-based exclusive sum-of-product minimization, etc.

Acknowledgments. This work was partly supported by NSF/NSA grant "Enhanced equivalence checking in cryptoanalytic applications" at University of California, Berkeley, and partly by H2020-ERC-2014-ADG 669354 CyberCare.

8. REFERENCES

[1] The EPFL Combinational Benchmark Suite, "Multi-output PLA benchmarks". `http://lsi.epfl.ch/benchmarks`.

[2] Berkeley Logic Synthesis and Verification Group, Berkeley, Calif. *ABC: A System for Sequential Synthesis and Verification.* `http://www.eecs.berkeley.edu/~alanmi/abc/`.

[3] N. Eén and N. Sörensson. An extensible SAT-solver. In *Proceedings of the International Conference on Theory and Applications of Satisfiability Testing*, volume 2919, pages 502–18. Springer, May 2003.

[4] Z. Huang, L. Wang, Y. Nasikovskiy, and A. Mishchenko. Fast Boolean matching based on NPN classification. In *Proceedings of the 2013 International Conference on Field Programmable Technology*, pages 310–13, Kyoto, Dec. 2013.

[5] D. E. Knuth. *Fascicle 6: Satisfiability*, volume 19 of *The Art of Computer Programming.* Addison-Wesley, Reading, Mass., Dec. 2015.

[6] J. Marques-Silva, J. Argelich, A. Graça, and I. Lynce. Boolean lexicographic optimization: algorithms & applications. *Annals of Mathematics and Artificial Intelligence*, 62(3):317–43, May 2011.

[7] A. Nadel. Generating diverse solutions in SAT. In *Proceedings of the International Conference on Theory and Applications of Satisfiability Testing*, pages 287–301, Ann Arbor, Mich., June 2011.

[8] A. Nadel and V. Ryvchin. Bit-vector optimization. In *Proceedings of the 22nd International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 851–67, Eindhoven, The Netherlands, Apr. 2016.

[9] A. Petkovska, A. Mishchenko, D. Novo, M. Owaida, and P. Ienne. Progressive generation of canonical sum of products using a SAT solver. In *Proceedings of the 25th International Workshop on Logic and Synthesis*, Austin, Tex., June 2016.

[10] M. Soeken, D. Große, A. Chandrasekharan, and R. Drechsler. BDD minimization for approximate computing. In *Proceedings of the 21st Asia and South Pacific Design Automation Conference*, pages 474–79, Macao, Jan. 2016.

[11] M. Soeken, A. Mishchenko, A. Petkovska, B. Sterin, P. Ienne, R. Brayton, and G. De Micheli. Heuristic NPN classification for large functions using AIGs and LEXSAT. In *Proceedings of the International Conference on Theory and Applications of Satisfiability Testing*, Bordeaux, France, July 2016.

[12] G. S. Tseitin. On the complexity of derivation in propositional calculus. In *Automation of Reasoning 2: Classical Papers on Computational Logic 1967-1970*, Symbolic Computation, pages 466–83. Springer, Berlin, 1983.

[13] R. Venkatesan, A. Agarwal, K. Roy, and A. Raghunathan. MACACO: Modeling and analysis of circuits for approximate computing. In *Proceedings of the International Conference on Computer Aided Design*, pages 667–73, San Jose, Calif., Nov. 2011.

[14] J. Yuan, A. Aziz, C. Pixley, and K. Albin. Simplifying Boolean constraint solving for random simulation-vector generation. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 23(3):412–20, Mar. 2004.