Accelerating All-SAT computation with short blocking clauses

Yueling ZHANG
Singapore Management University, ylzhang@smu.edu.sg

Geguang PU
East China Normal University

Jun SUN
Singapore Management University, junsun@smu.edu.sg

Citation

ZHANG, Yueling; PU, Geguang; and SUN, Jun. Accelerating All-SAT computation with short blocking clauses. (2020). *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE): Virtual, September 21-25: Proceedings*. 6-17.
Available at: https://ink.library.smu.edu.sg/sis_research/5944

2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)

DOI: 10.1145/3324884.3416569

Accelerating All-SAT Computation with Short Blocking Clauses

Yueling Zhang*
ylzhang@smu.edu.sg
Singapore Management University
East China Normal University

Geguang Pu*
ggpu@sei.ecnu.edu.cn
East China Normal University
Shanghai Trusted Industrial Control Platform Co. Ltd.

Jun Sun
junsun@smu.edu.sg
Singapore Management University

ABSTRACT

The All-SAT (All-SATisfiable) problem focuses on finding all satisfiable assignments of a given propositional formula, whose applications include model checking, automata construction, and logic minimization. A typical ALL-SAT solver is normally based on iteratively computing satisfiable assignments of the given formula. In this work, we introduce BASolver, a backbone-based All-SAT solver for propositional formulas. Compared to the existing approaches, BASolver generates shorter blocking clauses by removing backbone variables from the partial assignments and the blocking clauses. We compare BASolver with 4 existing ALL-SAT solvers, namely MBlocking, BC, BDD, and NBC. Experimental results indicate that although finding all the backbone variables consumes additional computing time, BASolver is still more efficient than the existing solvers because of the shorter blocking clauses and the backbone variables used in it.

With the 608 formulas, BASolver solves the largest amount of formulas (86), which is 22%, 36%, 68%, 86% more formulas than MBlocking, BC, NBC, and BDD respectively. For the formulas that are both solved by BASolver and the other solvers, BASolver uses 88.4% less computing time on average than the other solvers. For the 215 formulas which first 1000 satisfiable assignments are found by at least one of the solvers, BASolver uses 180% less computing time on average than the other solvers.

KEYWORDS

satisfiability, all-sat, blocking clauses

ACM Reference Format:
Yueling Zhang, Geguang Pu, and Jun Sun. 2020. Accelerating All-SAT Computation with Short Blocking Clauses. In *35th IEEE/ACM International Conference on Automated Software Engineering (ASE '20), September 21–25, 2020, Virtual Event, Australia.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3324884.3416569

Geguang Pu* and Yueling Zhang* are the Corresponding Authors

1 INTRODUCTION

The satisfiability problem is a typical NP-complete problem. The problem is to decide whether a given propositional formula in Conjunctive Normal Form (CNF) is satisfiable or not. The satisfiability problem is ubiquitous in computer science and has attracted the attention of many researchers because of its significant importance. An SAT (SATisfiability) solver is a software toolkit that is able to find a satisfiable assignment for a given satisfiable formula or an unsatisfiable proof for a given unsatisfiable formula. Although the complexity of SAT solving is NP-complete, modern SAT solvers are still able to find a satisfiable assignment efficiently. With the development of modern SAT solvers, propositional satisfiability theory and satisfiable modulo theory have been widely applied to several fields in computer science, including model checking [3, 4], program analysis [2, 6, 22], network verification [28, 30, 42], quantifier elimination [5] and predicate abstraction [23].

Some of the applications only need the SAT solver to find one satisfiable assignment of the given formula or prove that the formula is unsatisfiable, while other applications may require the SAT solver to find multiple or even all the satisfiable assignments of the given formula.

For example, in the field of network verification, the SAT solvers are asked to find all the satisfiable assignments. Since most modern networks are complex and contain multiple devices such as routers, bridges, and firewalls. Device diversity together with erroneous manual entry often lead to bugs that cause large downtime in which valid packets are dropped, or invalid packets are let through. Therefore, an ALL-SAT solver that computes all reachable packets from a source to a destination is needed in network verification [20, 28, 35, 42]. Lopes et al. [29] use the ALL-SAT solving technique to compute all the reachable packets for a network. They generalize the reachability verification tool to find all the states in the reachable set. The ALL-SAT solving technique is also applied to the predicate abstraction [7, 23], which is a technique to automatically extracting finite-state abstractions in systems with potentially infinite state space. For a formula and a set of predicates in a theory, Lahiri et al. [24] use the ALL-SAT solver to enumerate all the models of the formula over the predicates in the given theory and find the most precise approximation model of the system. Another application of the ALL-SAT technique is in the field of image computation [13, 16, 25, 37] and symbolic model checking [31]. Gupta [17] et al. compute all the solutions below intermediate points in the SAT decision tree.

In this work, we focus on the applications of SAT theory that require computing all the satisfiable assignments (ALL-SAT) of a given formula, such as unbounded hardware model checking [27], logic minimization [36], temporal logic planning [26], and knowledge compilation [8]. Within these applications, the number of variables in the SAT formulas is usually large and the clauses are usually complex. Therefore, the trivial method that blocks every satisfiable assignment after finding it is inefficient.

There are mainly three methods to find all the satisfiable assignments of a given formula, i.e., the blocking based strategy [31], the backtracking based strategy [16] and the BDD (Binary Decision Diagram) based strategy [40]. The key motivation of the blocking based strategy is to generate less and shorter blocking clauses to block more satisfiable assignments. Once the blocking clauses falsified the given formula, then all of the satisfiable assignments are expressed by the negation of the blocking clauses. The backtracking based strategy backtracks to the previous decision level inside a DPLL (Davis-Putnam-Logemann-Loveland) tree. Choosing a good backtrack decision level is the key motivation of this strategy. The BDD based strategy complied with the CNF formula into a Binary Decision Diagram using the knowledge complication techniques, and all the satisfiable assignments are generated by traversing all the possible paths from the root node to the sink node in the diagram.

When dealing with the formulas from these applications, the existing blocking based strategies tend to generate large blocking clauses that hinder the SAT solver from quickly finding a satisfiable assignment. Furthermore, the choices of backtrack decision levels are hard because the number of clauses in the formula is large. The BDD itself tends to be too complex to traverse efficiently when the number of clauses and variables is large. Therefore, existing ALL-SAT solvers with these three strategies usually fail to find all the satisfiable assignments of the formula generated from these applications. Due to the lack of proper ALL-SAT solver, the application of SAT theory in these fields is limited.

In this work, we propose an ALL-SAT solver BASolver following the blocking based strategy. But the generation of blocking clauses in BASolver is different from the existing approaches. BASolver first computes all the backbone variables and parts of the satisfiable assignments of the given formula. At least one satisfiable assignment is generated during the computing of the backbone variables. Then BASolver uses the backbone variables to generate the partial assignments and the blocking clauses from the known satisfiable assignments. All backbone variables are removed from the partial assignments. A clause ϕ in the formula is covered by a variable x in the clause if x is assigned to $True$ in the given satisfiable assignment. BASolver uses a greedy strategy to generate the shortest partial assignment such that each clause in the given formula is either covered by a backbone variable or a variable in the partial assignment. Finally, the blocking clause is generated with the negation of the partial assignment.

Comparing to the existing work, shorter blocking clauses are obtained from the partial assignments in BASolver. There are two advantages of removing backbone variables from the partial assignments and the blocking clauses. Firstly, the complexity of generating a partial assignment from the full satisfiable assignment is $O(n \times m)$, where n is the number of variables in the formula and m is the number of clauses in the formula. By directly removing backbone variables from the partial assignment, the variables that need to be computed in the partial assignment generation is reduced. When the number of clauses is large, the reduction is significant. Secondly, with shorter blocking clauses, the SAT solver is able to return a new satisfiable assignment quicker, especially when the number of clauses is large. The experimental results show that with the help of backbone variables, the SAT solving in BASolver is much faster, and less computing time is needed to generate a given amount of satisfiable assignments. The results also show that when all the backbone variables are added as unit clauses to the formula, BASolver is still more efficient thanks to the shorter blocking clauses used in it.

BASolver uses MiniSAT v2.1.1 [11] as the underlying SAT solver. We compared BASolver with 4 off-the-shelf ALL-SAT solvers, including two blocking based tools, MBlocking [41] and BC, one backtracking based tool, NBC, and one BDD based tool, BDD [39]. Among all the ALL-SAT solvers, BASolver solves (finds every assignment of the given formula) the largest number of formulas (86 out of 608) within 10 hours and 64G memory. While 70, 63, 51 and 46 formulas are solved by MBlocking, BC, NBC and BDD, respectively. For the formulas that are solved by both BASolver and one of the comparing solvers, BASolver also uses 88.4% less computing time than that used in the other solvers.

BASolver uses 24% less total computing time and 345% less blocking time in finding the first 1000 satisfiable partial assignments of a formula whose backbone variables are added as unit clauses. For the 69 formulas whose variables are all backbone variables, BASolver solves the most formulas with the least computing time comparing to each of the solvers. More details of the experiments are shown in Section 4.

The main contributions of our work are as follows:

- We propose an algorithm to compute all the satisfiable assignments based on the backbone variables of the propositional formula. It is the first application of the backbone variables in the ALL-SAT solving problem.
- We propose a blocking based ALL-SAT solver BASolver to compute all the satisfiable assignments of the given formula. Comparing to the existing work, BASolver is more efficient due to the use of the backbone variables, the use of shorter partial assignments, and the use of shorter blocking clauses. Experiments show that BASolver uses less computing time among all the blocking based, backtracking based, and BDD based ALL-SAT solvers.

The remainder of the paper is organized as follows. We describe notations and preliminaries in Section 2. The algorithms of BASolver are discussed in Section 3, experimental results are shown in Section 4, and the related work is discussed in Section 5. We conclude the paper in Section 6.

2 BACKGROUND

In this section, we present the necessary background on the satisfiability problems and explain how backbone variables are identified. We also discuss the straightforward blocking based methods to compute all the satisfiable assignments and the backbone variables of the given satisfiable formula.

2.1 The SAT and the ALL-SAT Problems

In this work, the propositional satisfiability formulas are described in the Conjunction Normal Form (CNF), and the clauses are described in the Disjunction Normal Form (DNF). Let **X** be a finite set of *Boolean variables*. A variable $x \in \mathbf{X}$ has two literals, x and $\neg x$, and the variable x is called the corresponding variable of the literals. A *clause* ϕ is a disjunction of the literals $\bigvee_{l_i \in \phi} l_i$, which is also represented as a set of literals $\{l_i \mid 1 \leq i \leq n\}$. A literal is denoted as $l \in \phi$ if ϕ is consists of the literal l. A variable x is in a clause ϕ if the literal x or the literal $\neg x$ in the clause ϕ. A *formula* Φ is a conjunction of the clauses $\bigwedge_{l_i \in \phi} \phi_i$. The formula Φ can also be represented as a set of clauses $\{\phi_i \mid 1 \leq i \leq n\}$. For clarity, we use ϕ to denote a clause and Φ to denote a formula.

For every variable $x \in \Phi$, Φ_x is the set of clauses that contain either x or $\neg x$, i.e., $\phi \in \Phi_x$ if and only if $x \in \phi$ or $\neg x \in \phi$, and Φ_x^p is the set of clauses that contain x, Φ_x^n is the set of clauses that contain $\neg x$.

Given a set of Boolean variables, an *assignment* v is a mapping $v : \mathbf{X} \rightarrow \{0, 1, -1\}$. The value of a Boolean variable x in an assignment v is $v(x)$. If $v(x) = 1$, then $v(\neg x) = 0$. If $v(x) = 1$, then $\neg v(x) = 0$, i.e., $v(x) = \neg\neg v(x)$. If $v(x) = 1$, $v(\neg x) = 0$, if $v(x) = 0$, $v(\neg x) = 1$, i.e., $v(x) = \neg v(\neg x)$. If the value of x in v is -1, then the value of $\neg x$ in v is also -1, i.e., $v(\neg x) = -1$ if $v(x) = -1$. We also use the conjunction of the literals whose value are 1 in an assignment v to denote the v, for example, an assignment $v(a) = 1$ and $v(b) = 0$ is also written as $v = a \wedge \neg b$, where a and b are literals of the formula.

The value of ϕ under the assignment of v is 1 if and only if there exists a literal $l \in \phi$ and $v(l) = 1$, denoted as $v(\phi) = 1$ if and only if $\exists l \in \phi, v(l) = 1$. For a clause ϕ and an assignment v, we say that $v \models \phi$ if and only if $v(\phi) = 1$. The value of Φ under the assignment of v is 1 if and only if the value every clause $\phi \in \Phi$ is 1, denoted as $v(\Phi) = 1$ if and only if $\forall \phi \in \Phi, v \models \phi$, also written as $v \models \Phi$ if and only if $\forall \phi \in \Phi, v \models \phi$.

An assignment v is a *full assignment* of Φ if and only if there does not exist a variable $x \in \Phi$ such that $v(x) = -1$. Otherwise, x is a partial assignment. For a partial assignment v_p, a full assignment v_f implies v_p if for every variable $x \in \Phi$, such that $v_p(x) \neq -1$, $v_f(x) = v_p(x)$, denoted as $v_f \rightarrow v_p$. If a full assignment implies a partial assignment, then the full assignment can be expressed and replaced by the partial assignment in the set of all the assignments.

An assignment λ is a satisfiable assignment (solution) of Φ if and only if $\lambda \models \Phi$. The ALL-SAT problem is to compute all the satisfiable assignments of a given formula. Since the size of the satisfiable assignments could be exponential, we use partial assignments to express multiple full satisfiable assignments at the same time. For a partial assignment v_p, if every full assignment v_f that implies v_p is a full satisfiable assignment, then v_p is called a satisfiable partial assignment and can be used to express the set of all the assignments that imply v_p, i.e., v_p express the set of full assignments $\{v_f | v_f \rightarrow v_p, v_f \models \Phi\}$.

Modern SAT solvers usually use the DPLL algorithm [10][9] to find a satisfiable assignment. There are two kinds of variables during the solving, the decision variables, and the implication variables. The solver assigns 0 or 1 to the decision variables and checks if there exist any variables that need to be assigned to 1 after the current decision. If there exist such variables, these variables are called implication variables, and 1 is assigned to them. During the DPLL process, a conflict happens if there exists a clause that the value of every literal in the clause is 0. The solver backtracks to the previous decision level whenever a conflict occurs. If a conflict occurs at the root level, that means the formula is unsatisfiable and the current conflict is a proof of it.

SAT solvers also support the use of the assumptions. An assumption is a set of literals that must be assigned to 1 in the satisfiable assignment computed by the SAT solver. For a satisfiable formula Φ, $\Phi \wedge assu$ might be unsatisfiable. In this case, the SAT solver returns a subset of *assu* indicating the reasons that Φ is not satisfiable under the assumption of *assu*.

A straightforward blocking based method to compute all the satisfiable assignments is to iteratively compute satisfiable assignments. To avoid finding the same satisfiable assignment, the negations of the already known solutions is added to the formula. These negations are called blocking clauses. Since a solution is a conjunction of unit literals, the negation of it is a disjunction of unit literals (a clause), i.e., $c_\lambda = \bigvee_l \lambda(l) = 0$. The formula Φ' is updated with Φ and the blocking clause c, i.e., $\Phi' = \Phi \wedge c$, if Φ is unsatisfiable, then all the satisfiable assignments are found. Otherwise a new satisfiable assignment λ' is found by the SAT solver Φ' is updated with $\Phi' \wedge c'_\lambda$ until Φ' is unsatisfiable. If a assignment $\lambda \models \Phi'$ is a satisfiable of Φ' then λ is also a satisfiable assignment of Φ, i.e., $\lambda \models \Phi$ if $\lambda \models \Phi'$. In this way, all the satisfiable assignments of Φ are found by the SAT solver with the help of Φ'.

We show an example to illustrate the above definitions. For a propositional satisfiable formula $\Phi = (a \vee b \vee c) \wedge (a \vee \neg b \vee c) \wedge (a \vee b \vee \neg c) \wedge (a \vee \neg b \vee \neg c)$. The set of clauses that contain the variable a is $\{a \vee b \vee c, a \vee \neg b \vee c, a \vee b \vee \neg c, a \vee \neg b \vee \neg c\}$, the set of clauses that contain the literal $\neg a$ is $\emptyset$. An assignment λ such that $\lambda(a) = 1$, $\lambda(b) = 1$ and $\lambda(c) = 1$ is a satisfiable assignment of λ. A partial assignment v_p such that $v_p(a) = 1$ is implied by λ, and v_p is a satisfiable partial assignment since all the full assignments that imply v_p is a satisfiable assignment of λ.

In an SAT solver with the given formula Φ, the variables a, b, and c may be assigned to 1 as decision variables respectively. In an ALL-SAT solver, suppose the first satisfiable assignment of Φ is λ such that $\lambda(a) = \lambda(b) = \lambda(c) = 1$, then the blocking clause of λ is $\neg a \vee \neg b \vee \neg c$ and $\Phi' = (a \vee b \vee c) \wedge (a \vee \neg b \vee c) \wedge (a \vee b \vee \neg c) \wedge (a \vee \neg b \vee \neg c) \wedge (\neg a \vee \neg b \vee \neg c)$. A new satisfiable assignment λ' such that $\lambda'(a) = 1$, $\lambda'(b = 1)$ and $\lambda'(c) = 0$ is found by the SAT solver and Φ' is updated with $\Phi' = \Phi' \wedge (\neg a \vee \neg b \vee c)$. Eventually, Φ' will be updated to $\Phi' = \Phi \wedge (\neg a \vee b \vee c) \wedge (\neg a \vee \neg b \vee c) \wedge (\neg a \vee b \vee \neg c) \wedge (\neg a \vee \neg b \vee \neg c)$, and it is unsatisfiable. All the four satisfiable assignments are $\lambda_1 = a \wedge b \wedge c$, $\lambda_2 = a \wedge \neg b \wedge c$, $\lambda_3 = a \wedge b \wedge \neg c$ and $\lambda_4 = a \wedge \neg b \wedge \neg c$.

2.2 The Backbone Variables

For a given satisfiable propositional formula Φ, a variable $x \in \Phi$ is a backbone variable if the value of x in every satisfiable assignment of Φ remains the same. For an unsatisfiable propositional formula, there does not exist a backbone variable as there does not exist a satisfiable assignment for the formula. A formal definition of the backbone variable is as follows.

Definition 2.1 (The Backbone Variables). Given a satisfiable formula Φ, a variable x is a backbone variable of Φ if and only if $\Phi \wedge x$ is unsatisfiable or $\Phi \wedge \neg x$ is unsatisfiable.

For a given formula Φ, a variable $x \in \Phi$ is either a backbone variable or a non-backbone variable. For a non-backbone variable

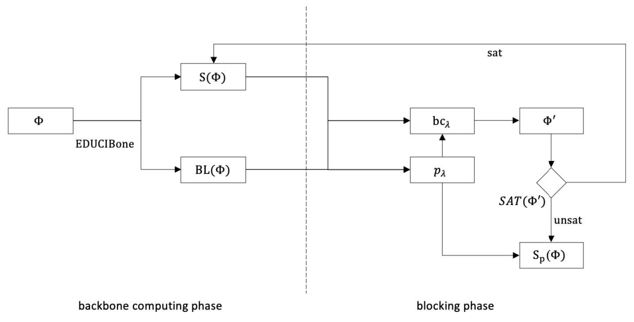

Figure 1: Workflow of BASolver

x of the given formula Φ, there must exist two different satisfiable assignments of Φ such that $\lambda_1 \models \Phi, \lambda_2 \models \Phi, \lambda_1(x) \neq \lambda_2(x)$.

A straightforward algorithm to compute all the backbone variables of a given formula Φ is to iteratively check the satisfiability of $\Phi \wedge x$ and $\Phi \wedge \neg x$ for every variable $x \in \Phi$. If either of the formulas is unsatisfiable, then the variable x is a backbone variable.

For example, given a formula $\Phi = (a \vee b \vee c) \wedge (a \vee \neg b \vee c) \wedge (a \vee b \vee \neg c) \wedge (a \vee \neg b \vee \neg c)$, the variable a is a backbone variable since $\Phi \wedge \neg a$ is unsatisfiable. And both the variable b and c are non-backbone variables, since $\Phi \wedge b \wedge c$ and $\Phi \wedge \neg b \wedge \neg c$ are satisfiable.

3 ALL-SAT SOLVER USING THE BACKBONE VARIABLES

The workflow of BASolver is shown in Figure 1. There are mainly two phases in BASolver, the backbone computing phase, and the blocking phase. In the backbone computing phase, BASolver computes all the backbone variables of the given formula Φ with the EDUCIBone tool [43]. There are two parts in the output of EDUCIBone, a complete set of backbone variables and a set of satisfiable assignments. Notice that it only generates parts of the solutions, but BASolver still benefits from these solutions as it does not need to generate again. All the backbone variables of the given formula Φ are in $BL(\Phi)$ and parts of the satisfiable assignments are in $S(\Phi)$. The partial assignments of $S(\Phi)$ are generated and stored in $S_p(\Phi)$ in the blocking phase.

In the blocking phase, BASolver takes the backbone variables, the given formula and the given satisfiable assignments as the inputs and generates the partial assignment p_λ and the blocking clause bc_λ for each given satisfiable assignment $\lambda \in S_p(\Phi)$. The blocking clause bc_λ is the disjunction of the literals assigned to 0 in the partial assignment p_λ, i.e., $bc_\lambda = \bigvee_{p_\lambda(l)=0} l$. Then the formula Φ' is updated with $\Phi \wedge bc_\lambda$ and the partial assignment p_λ is added to the set of the partial assignments $S_p(\Phi)$. If the new formula Φ' is unsatisfiable, BASolver has found all the satisfiable assignments of the given formula Φ. Otherwise, a new satisfiable assignment is added to $S(\Phi)$. Then, BASolver computes the partial assignment $p_{\lambda'}$ and the blocking clause $bc_{\lambda'}$ for the next satisfiable assignment in $S_p(\Phi)$, and updates Φ' with $\Phi' \wedge bc_{\lambda'}$. The partial assignment $p_{\lambda'}$ is added to $S_p(\Phi)$. The blocking phase only terminates when Φ' is unsatisfiable. Since the number of satisfiable assignments of the formula Φ is finite, and $\Phi \bigwedge_{\forall \lambda \models \Phi} bc_\lambda$ is unsatisfiable, Φ' will eventually become unsatisfiable in the blocking phase. When BASolver terminates, all satisfiable assignments of Φ are expressed by the negation of the blocking clauses in $S(\Phi)$.

3.1 Computing the Backbone Variables

We briefly introduce the algorithm we use in BASolver to compute all the backbone variables. BASolver uses the EDUCIBone [43] to compute the backbone variables, and the algorithm is shown in Algorithm 1. The algorithm first initializes 5 arrays. The set w is the set of the literals that have not been handled in the algorithm, the set t is the set of the literals that need individual SAT solving. The set bl is the set of the currently known backbone variables and the set nbl is the set of the currently known non-backbone variables. S is the set of the satisfiable assignments computed in Algorithm 1.

At first, the algorithm computes a satisfiable assignment of the given formula Φ using $SAT(\Phi)$, if Φ is satisfiable, ret is $True$ and the satisfiable assignment is λ, otherwise c is the unsatisfiable reason (if exits). If Φ is unsatisfiable, there is no backbone variable. If Φ is satisfiable, every literal l of Φ which has been assigned to 0 is added to w, and λ is added to S. Then the algorithm iteratively handles the literals in w. The variables in it are separated into groups, *chunk* is the size of the group. Separating is very helpful in formulas with large amount of variables. The *chunk* amount is 100 according to the most efficient configuration in [43]. We also try different *chunk* values in the experiments, but the performance difference is small. For a group of literals *assu*, if $\Phi \wedge assu$ is satisfiable, then the corresponding variable of every literal in *assu* is added to *nbl*, and the new assignment is added to S. Otherwise, if the size of the unsatisfiable reason c is 1, then the corresponding variable of the literal in c is added to bl and removed from *assu*. If the size of the unsatisfiable reason c is greater than 1, then all the literals in c are added to t and removed from *assu*. The algorithm then iteratively check the satisfiability of $\Phi \wedge assu$ until it is satisfiable. Since the size of *assu* is finite, *assu* will either become empty or $\Phi \wedge assu$ becomes satisfiable. A new chunk of literals will be assigned to *assu* when the computing of the current chunk is finished. When the w set is empty, the algorithm starts to deal with the literal in the t set. For each literal $l \in t$, if $\Phi \wedge l$ is satisfiable, then the corresponding variable of l is a backbone variable, otherwise it is a non-backbone variable.

3.2 Backbone Variables based Blocking Clauses Computing

We then introduce the computing of the partial assignment and the blocking clause (the blocking phase) based on a given satisfiable assignment λ of the given formula Φ and all the backbone variables of Φ.

For a satisfiable formula Φ and a backbone variable x, if the value of x in every satisfiable assignment of Φ is 1, then the satisfiable assignments of Φ and $\Phi \wedge x$ are the same. It means that if a assignment $\lambda \models \Phi$, then $\lambda \models \Phi \wedge x$, and if $\lambda \not\models \Phi$, then $\lambda \not\models \Phi \wedge x$. Therefore, for a formula Φ and backbone variables of Φ in $BL(\Phi)$, the result of ALL-SAT solver will not be changed by replacing Φ with $\Phi \bigwedge_{l \in BL(\Phi), \lambda(l)=1} l$. After replacing the formula Φ, we can remove the backbone variables from all the satisfiable partial assignments since the value of the backbone variables is already fixed by the new

ALGORITHM 1: Computing the Backbone Variables

```
Function backbone $\Phi$
    $w$ = []; $t$ = []; $nbl$ = []; $bl$ = []; $S$ = [];
    (ret, $\lambda$, $c$) = SAT($\Phi$);
    if !$ret$ then return $\emptyset$;
    if $ret$ then
        $S.add(\lambda)$;
        foreach $l \in \Phi, \lambda(l) = 0$ do $w$.add($l$);
        while $w.size() \neq 0$ do
            $assu$ = $w$[0:chunk];
            $w$.remove($assu$);
            (ret, $\lambda$, $c$) = SAT($\Phi \wedge assu$);
            while !$ret$ do
                if $c$.size() == 1
            end
            then $bl$.add($c$);
            else $t$.add($c$);
            $assu$ = $assu$.remove($c$);
            (ret, $\lambda$, $c$) = SAT($\Phi \wedge assu$);
            $nbl$.add($assu$);
            $S.add(\lambda)$;
        end
        foreach $l \in t$ do
            (ret,$\lambda$,$c$) = SAT($\Phi \wedge l$);
            if $ret$ then
                $nbl$.add($l$);
                $S.add(\lambda)$;
            end
            else $bl$.add($l$);
        end
    end
    return $bl$, $S$;
end
```

formula. For a satisfiable partial assignment p, p contains all the backbone variables of Φ, although p' is no longer a satisfiable partial assignment after removing the backbone variables, we can easily recover p from p' as all the backbone variables of Φ are already known. As we discussed before, shorter partial assignments will lead to shorter blocking clauses and accelerate the ALL-SAT solving procedure. Based on the above observation, Algorithm 2 computes the shorter partial assignment and the blocking clause. The idea of the algorithm is to compute the satisfiable partial assignment and removes all the backbone variables from it.

With a given formula Φ, a satisfiable assignment $\lambda \models \Phi$ and all the backbone variables $BL(\Phi)$ of Φ. Algorithm 2 first traverses the non-backbone variables in λ, if the variable x is an implication variable, then x is added to the array of the partial assignment p. Based on the DPLL procedure in the MiniSAT Solver, if a variable x is an implication variable, then there must exist at least a clause ϕ of Φ such that either x or $\neg x$ has to be assigned to 1 to make ϕ satisfiable. Therefore, the implication variable x must exist in the satisfiable partial assignment of λ. If the variable x is not an implication variable, and the value of x in λ is 1, the algorithm iteratively checks every clause ϕ such that $l \in \phi$ to see if l is the only literal in ϕ which has been assigned to 1. If so, l is also in the partial assignment. The algorithm checks the clauses in Φ_l^n if $\lambda(l) = 0$. For every literal added to p, the negation of the literal is added to b.

After checking all the non-backbone variables, the partial assignment p and the blocking clause c are generated. The time complexity of Algorithm 2 is $O(m \times n \times k)$, where m, n, k are the number of clauses, variables and satisfiable assignments of Φ. To reduce the computing in Algorithm 2, we add a small optimization in the BASolver tool. We compare the difference between two satisfiable assignment, and only the variables with different values are computed.

ALGORITHM 2: Computing the Partial Assignment and the Blocking Clause

```
Function blocking $\Phi$, $\lambda$, $BL(\Phi)$
    $b$ = []; $p$ = [];
    foreach $x \in \lambda, x \notin BL(\Phi)\ \lambda(x) = 1$ or $\lambda(\neg x) = 1$ do
        if $x$ is an implication variable then
            $p$.add($x$);
            if $\lambda(x) = 0$ then $b = b \vee x$;
            else $b = b \vee \neg x$;
        end
        else
            if $\lambda(x) = 0$ then
                foreach $\phi \in \Phi_x^p$ do
                    if $\nexists l \in \phi, \lambda(l) = 1$ then
                        $p$.add($x$);
                        $b = b \vee x$;
                        break;
                    end
                end
            end
            if $\lambda(x) = 1$ then
                foreach $\phi \in \Phi_x^n$ do
                    if $\nexists l \in \phi, \lambda(l) = 1$ then
                        $p$.add($x$);
                        $b = b \vee \neg x$;
                        break;
                    end
                end
            end
        end
    end
    return $p, b$;
end
```

There are two advantages of removing the backbone variables from the partial assignments. Firstly, the following SAT solving is easier for the SAT solver since the size of the clauses has been significantly reduced. Moreover, even the time complexity of the computing the partial assignment and the blocking clause is in polynomial time, but it still could be time and memory consuming when the formulas and the number of the satisfiable assignments are large. By removing the backbone variables from the partial assignments, we could save a large amount of computing time and memory resources during the blocking phase. Experiments also indicate that for a special group of the formulas, which only have one satisfiable assignment, finding all the backbone variables is generally faster than directly finding all the satisfiable assignments using the existing ALL-SAT tools.

ALGORITHM 3: Computing all the Satisfiable Assignments

```
$(BL(\Phi), S)$ = backbone($\Phi$);
$S_p$ = [];
$\Phi' = \Phi$;
foreach $\lambda \in S$ do
    $(p, b)$ = blocking($\Phi, \lambda, BL(\Phi)$);
    $Phi' = \Phi' \wedge b$;
    (ret, $\lambda$, $c$) = $SAT(\Phi;)$;
    if ret then
        $S.add(\lambda')$;
        $S_p$.add($p$);
    end
    else return $S_p$, $BL(\Phi)$;
end
```

After computing the partial assignments and the blocking clause of the a given satisfiable assignment, we update the formula Φ to $\Phi \wedge bc$ and continue finding the next satisfiable assignment of it. The algorithm of BASolver is shown in Algorithm 3 with the function of $backbone(\Phi)$ and $blocking(\Phi, \lambda, BL(\Phi))$ introduced in Algorithm 1 and Algorithm 2.

For a given formula Φ, BASolver first computes the backbone variables $BL(\Phi)$ of Φ, the satisfiable assignments generated during the computing is in S. Then BASolver computes the partial assignment p and the blocking clause b for every satisfiable assignment λ in S. If $\Phi' \wedge b$ is satisfiable, new satisfiable assignment is added to S. The partial assignment of every satisfiable assignment in S is added to S_p. If $\Phi' \wedge b$ is unsatisfiable, BASolver returns S_p and $BL(\Phi)$. The satisfiable partial assignments are generated by adding all the backbone variables back to each partial assignment in S_p.

4 EVALUATION

We implemented BASolver based on the EDUCIBone tool, following a blocking based strategy. We use the Minisat 2.2 as the SAT solver. BASolver is implemented in C++ with approximately 1000 lines of code for the main algorithm. The experiments are conducted on a cluster of Linux systems. The runtime limitation is 10 hours and the memory limit is 64GB. Only one formula is running on a node of the cluster at one time, and no parallel techniques are used during the experiments. BASolver uses the *solve*() interface of the MiniSAT solver to solve formulas and the *addClause*() interface to add new blocking clauses to the formula. It did not use other incremental features of the MiniSAT solver. We evaluate the efficiency of BASolver through multiple experiments, answering the following research questions:

- **RQ1**: How effective is BASolver comparing to other ALL-SAT solvers?
- **RQ2:** Is the shorter blocking clause really useful in the ALL-SAT computing?
- **RQ3:** Is the percentage of the backbone variables in the formulas related to the performance of BASolver? Is it cost-effective to compute all the backbone variables first?

4.1 Experimental Setup

• *Benchmarks and Evaluation Metrics.* In a survey by Toda and Soh [39], three sources benchmarks are used. The SAT competitions of 2014, the SATLIB benchmark and the ISCAS85 libraries. We extend the first source with benchmarks of SAT competitions from 2011 to 2017. We found that the link of the third source is no longer accessible, and the formulas in the SATLIB libraries are either too simple (all of the solvers finished within 1 second) or too hard (none of the solver finishes). Therefore, we only uses the extended first source as our benchmark. There are 1816 formulas in total, 552 out of which are known as unsatisfiable, 608 out of which are satisfiable and the satisfiability of the rest remains unknown. The final benchmark is consists of the 608 satisfiable formulas. Among the 608 formulas, EDUCIBone is able to compute all the backbone variables of 214 formulas. For the rest of the formulas, computing one satisfiable assignment always spends most of the time limitation, and finishing ALL-SAT computing using any solver is unlikely possible for them.

For an SAT solver, such as MiniSAT, we say a formula is solved by the SAT solver if it finds at least one satisfiable assignment of the formula. For the ALL-SAT solvers, such as BASolver, we say a formula is completely solved by the ALL-SAT solver if it finds every satisfiable assignment of the formula, if the ALL-SAT solver only finds parts of the satisfiable formula, we say it partially solved the formula. We use the number of solved formulas and the average computing time as metrics to evaluate the performance of the ALL-SAT solvers. For the formulas that ALL-SAT solver are not able to find all the satisfiable assignments, we use the average computing time of fining the first 1000 satisfiable assignments as the metrics.

•*Baseline Approaches.* To conduct a complete and objective comparison between BASolver and the existing tools, we use 4 different ALL-SAT solvers in the experiments. Among the public ALL-SAT solvers with the available toolkit, these 4 performs the best.

The first ALL-SAT solver is MBlocking, it also uses the partial assignments to express the full satisfiable assignments. The authors propose three different strategies in MBlocking, All-Clauses, Non-disjointing, and hybrid. We use the configuration of the hybrid strategy since the authors claim that it performs the best. The other three tools are all from a survey in the year 2016. In the survey, Toda and Soh [39] conclude the existing ALL-SAT solving algorithms and implemented them to ALL-SAT solvers. There are different configurations used in these solvers, we use the default configurations of these solvers. The second ALL-SAT solver is BC, though is also uses the blocking based strategy. The last two ALL-SAT solvers are NBC and BDD. NBC uses a backtracking based strategy to find all the satisfiable assignments and BDD uses the BDD-based strategy. All the last three ALL-SAT solver (BC, NBC

Formulas	AveVAR	AveCL	AveFST	#AveSATInst	BASolver	MBlocking	BC	NBC	BDD
dimacs(17)	680	17079	508.14	1	17	16	14	1	15
AProve(3)	8565	28931	7.32	78	3	2	2	2	0
complete(4)	600	27140	0.02	1	4	3	4	4	0
Encryption(17)	8616	82773	44.75	1	17	12	16	15	7
Manthey(25)	5179	23692	780	2.58	25	22	14	19	15
mp1(12)	16301	185948	630.41	305.75	12	10	9	6	8
Others(8)	11593	44557	325.19	1131	8	5	4	4	1
Total(86)	7323	393119	327.97	227.79	86	70	63	51	46

Table 1: Overall Performance of All-SAT Computing Tools

and BDD) changes the implementation of the MiniSAT solver while BASolver and MBlocking use the MiniSAT solver as a black box.

Due to the strategies used in NBC and BDD, they do not use the partial assignments to express the full satisfiable assignments, and due to the implementation of BC, though it uses a blocking based strategy, it also does not use the partial assignments to express the full satisfiable assignments. The difference between the number of the full satisfiable assignments and the number of partial assignments that express them could be large. Therefore, the comparison on the formulas which are not completely solved by any of the 5 ALL-SAT solvers are only conducted between BASolver and MBlocking.

4.2 Results

• *RQ1: How effective is* BASolver *comparing to other ALL-SAT solvers?* We show the formulas in groups that are at least solved by one of the ALL-SAT solvers. The first column of Table 1 shows the name of the groups. From the second to the fifth columns shows the basic information of the formulas, including the average number of the variables (AveVAR), the average number of the clauses (AveCL), the average computing time for the first satisfiable assignment (AveFST) and the average number of satisfiable formulas (#AveSATInst). The average number of satisfiable formulas in the dimacs, complete, and Encryption group is 1, indicating that all the variables in these formulas are backbone variables. The average number of satisfiable formulas in the Manthey group is also small, indicating that most of the variables are backbone variables in these formulas.

From the sixth to the tenth columns in Table 1 shows the number of formulas in each group solved by BASolver, MBlocking, BC, NBC, and BDD respectively. Figure 2 shows the comparison of the formula numbers solved by the ALL-SAT solvers in different groups. There are 8 groups of bars, indicating the groups of the formulas. There are 5 bars in each of the groups, indicating the number of solved formulas of the ALL-SAT solvers. The y-aixs shows the number of the solved formula. The first bar in each group is always the highest, it means that BASolver solves the most formulas in all the groups. In total, BASolver solves 86 formulas, which is 22% more than MBlocking solves, 36% more than BC solves, 68% more than NBC solves, and 86% more than BDD solves. The results show that BASolver solves more formulas than all the other 4 tools in a given time limit.

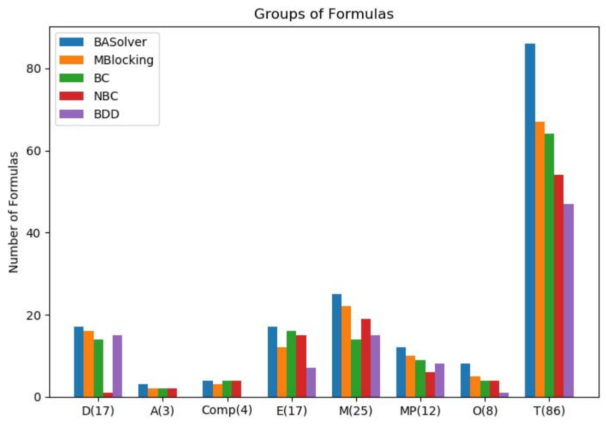

Figure 2: Number of Solved Formulas of the ALL-SAT solvers

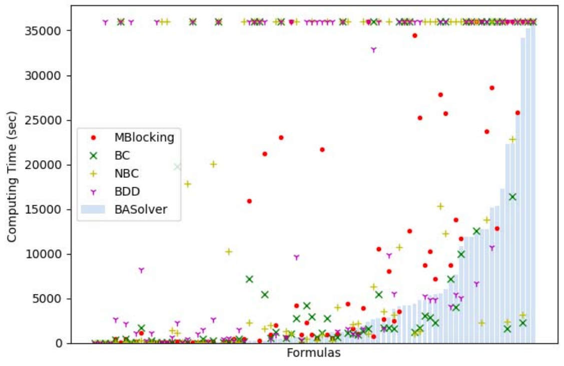

Figure 3: Comparison of Computing Time among the 5 Solvers

Figure 3 shows the comparison of computing time time among the 5 ALL-SAT solvers. Each bar shows the computing time of BASolver to solve the formula. The computing time of MBlocking, BC, NBC, and BDD are shown with the dot, cross, plus, and triangle,

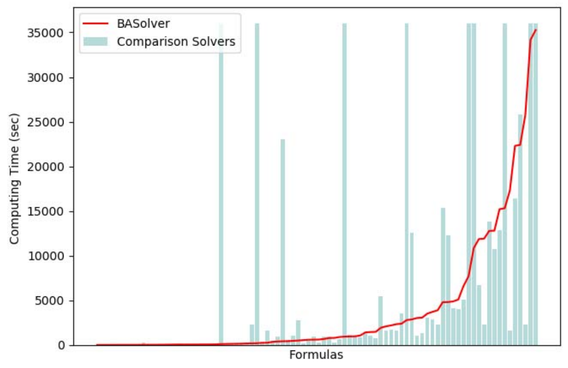

Figure 4: Comparison of the Computing Time between BASolver **and the Most Efficient Solver**

respectively. For most of the formulas, BASolver uses less computing time than the other 4 solvers since most of the shapes are not drawn on the bars. The average computing time of BASolver among the 86 formulas is 3882 seconds. For the 70 formulas that are both solved by BASolver and MBlocking, BASolver needs 209% less computing time than MBlocking does. For the 63 formulas that are both solved by BASolver and BC, BASolver needs 2% less computing time. For the 51 formulas that are both solved by BASolver and NBC, BASolver needs 36% less computing time, and for the 46 formulas that are both solved by BASolver and BDD, BASolver needs 107% less computing time.

Figure 4 shows the comparison of the computing time between BASolver and the 4 comparison solvers. The minimal computing time for a formula among the 4 comparison solvers is used to compare with BASolver. The red line denotes the computing time of BASolver for each formula and the bars denote the comparison computing time of the formulas. It shows that the performance between each side is similar, indicates a good versatility of BASolver.

The BASolver solves more formulas than the comparison solvers with the given time limitation. For the formulas that are both solved by BASolver and each of the comparison solver, BASolver uses less computing time. Moreover, BASolver also shows good versatility by a similar performance comparing to the most effective solver, which is different for every different formula.

Answer to RQ1 :
BASolver *solves more formulas within the given computing limitation. And less computing time is used in* BASolver *for solving the same formulas.*

• *RQ2: Is the shorter blocking clause really useful in the ALL-SAT computing?* Since the use of backbone information in the ALL-SAT computing only has two consequences, shorter blocking clauses and additional unit clauses with backbone variables. To answer RQ2, we add the backbone variables as unit clauses to the original formulas. We then compare the SAT solving time, the blocking time, and the total computing time of finding the first 1000 satisfiable partial assignments of each formula with the different solvers. The solving time indicates the computing time consumed by the MiniSAT solver, and the blocking time represents the computing time consumed by the generation of the partial assignments and the blocking clauses. Since it is difficult to separate the blocking and the solving process in BC, NBC , and BDD, we only compare BASolver with MBlocking in this experiment.

There are 214 formulas in which all the backbone variables are found by EDUCIBone. Within 10 hours, and 64 GB memory limit, BASolver finishes computing the first 1000 satisfiable partial assignments of 105 formulas and MBlocking finishes computing the first 1000 satisfiable partial assignments of 94 formulas. All the 94 formulas finished by MBlocking are also finished by BASolver. Among them, for 76 of the formulas, BASolver uses less blocking time, and for 64 of the formulas, MBlocking uses less solving time. In total, the average computing time for the 94 formulas in BASolver is 472 seconds, which is 24% less than MBlocking (615 seconds). The average blocking time of BASolver (138 seconds) is 345% time less than that used in MBlocking (615 seconds). The average solving time of BASolver is 333 seconds, which is similar to that in MBlocking (339 seconds). Therefore, the more efficient blocking process mainly contributes to the efficiency of BASolver. Since all the variables are added as unit clauses to the given formula, the shorter blocking clauses is the main reason that BASolver uses less blocking time than MBlocking does.

Figure 5 shows the comparison of the blocking time, solving time, and the total computing time used in finding the first 1000 satisfiable partial assignments for a formula. The red bars show the blocking time used in BASolver, the blue bars show the solving time used in BASolver, the green bars show the blocking time used in MBlocking, and the yellow bars show the solving time used in MBlocking. For most the formulas in the plot, the total computing time used in BASolver is less than that used in MBlocking. Also, in both of the solvers, the blocking process consumes more computing time than the solving process, due to a large number of variables and clauses in the formulas.

In this experiment, BASolver uses 345% less blocking time in finding the first 1000 satisfiable partial assignments in each one of the formulas. The shorter blocking clauses are useful in ALL-SAT solving as they are the main difference between BASolver and MBlocking within the experiment.

Answer to RQ2 :
The shorter blocking clauses lead to a less blocking time, which is useful and efficient in the ALL-SAT solving.

• *RQ3: Is the percentage of the backbone variables in the formulas related to the performance of* BASolver*? Is it cost-effective to compute all the backbone variables first?* For all the 5 solvers in the experiments, more formulas are solved when there are more backbone variables in the formulas. Nearly 75% of the formulas solved by the 5 solvers have 100% percentage of backbone variables. Figure 6 shows the computing comparison of the 5 solvers on the formulas with a 100% percentage of backbone variables. The light blue bars are the computing time of BASolver, and the light red bars are the least computing time among the 4 comparing solvers for the same formula. Notice that the lighter (red) bars starts from

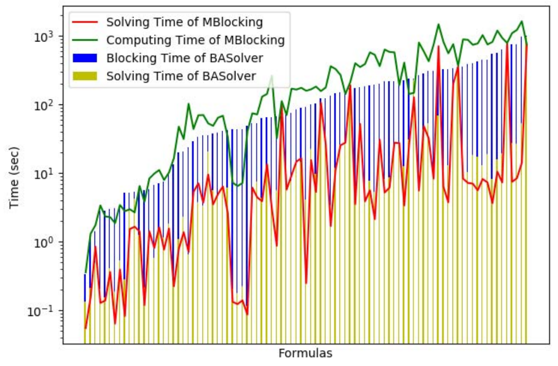

Figure 5: Comparison of the Blocking and Solving Time between BASolver **and** MBlocking

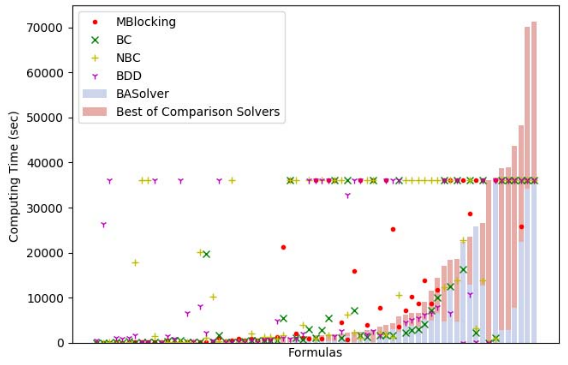

Figure 6: Comparison of the Computing Time for the Formulas with 100% backbone variables

the top of the darker (blue) bars, which means that if that if BASolver does not use the least computing time, the circles, dots, or crosses might below the lighter bars. The dots, crosses, pluses, and triangles represents the computing time of MBlocking, BC, NBC, and BDD for the same formula, respectively. BASolver uses the least total computing time among the 5 solvers. For almost half of the formulas, BASolver needs the least computing time among all the 5 solvers. It indicates that if only one solver is applied to these formulas, BASolver is the most efficient one.

We choose the 64 formulas that both BASolver and MBlocking finds the first 1000 satisfiable partial assignments to study the relation between the performance of BASolver and the percentage of the backbone variables. The percentage of the backbone variable in these formulas ranges from 0.001% to 99.7%, the average percentage is 25.6%.

Figure 7 shows the comparison of the blocking, solving and computing time between BASolver and MBlocking among the 64 formulas. The red plot shows the blocking time of MBlocking, and

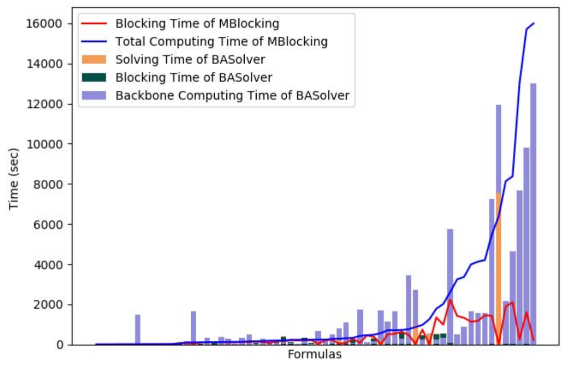

Figure 7: Comparison of the Computing Time for the Formulas with Different Percentages of Backbone Variables

the blue plot shows the total computing time of MBlocking. From bottom to tap, the yellow bars show the blocking time in BASolver, the green bars show the solving time in BASolver, and the purple bars show the backbone computing time in BASolver. For most of the formulas, the blocking time used in BASolver is less than that in MBlocking. The average blocking time in BASolver is 200% times less than that in MBlocking. But due to the computing of the backbone variables, for nearly 20 formulas, BASolver needs more computing time to find the first 1000 satisfiable partial assignments. For most of the formulas, less computing time is needed for BASolver.

Figure 8 shows the comparison of the average computing time between BASolver and MBlocking grouped by the percentage of the backbone variables. MBlocking uses less computing time when the percentage of backbone variables is 50%, 80%, and 90%. This is because the 64 formulas are relatively more difficult to the MiniSAT solver, more computing time are consumed when finding all the backbone variables. When there are not enough satisfiable assignments in the formulas with more backbone variables, the time-consuming in the backbone computing phase becomes more and more serious.

The general performance of BASolver comparing to the other solvers is not affected by the percentage of the backbone variables. But since there are less satisfiable assignments that exist when the percentages of backbone variables are high, more formulas are solved by all of the 5 solvers with more backbone variables in the formulas. The experiments show that even though computing all the backbone variables require additional computing time, but BASolver uses less computing time for each of the partial assignments in most formulas. For the formulas that all the variables are backbone variables, the average computing time used in finding the only satisfiable assignment in BASolver is less than that in each of the comparing solvers. Therefore, computing all the backbone variables in ALL-SAT solving is cost-effective. If there is only one satisfiable assignment, BASolver is able to find it quickly after computing all the backbone variables for most of the formulas. If there are amount of the satisfiable assignments is large, the average

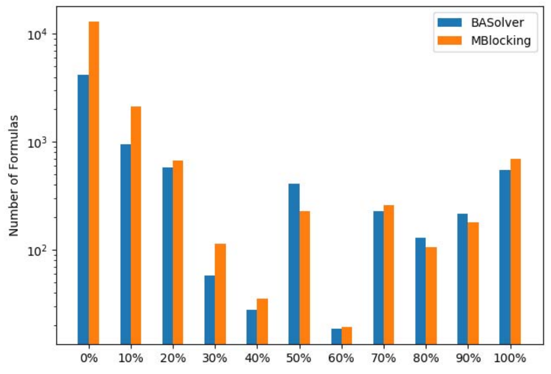

Figure 8: Computing Time of BASolver **and** MBlocking **for Formulas with Different Percentage of Backbone Variables**

computing time of each satisfiable assignment in BASolver is less after computing all the backbone variables.

Answer to RQ3 :
All the 5 solvers solves more formulas when there are more backbone variables in the formulas, the percentage of the backbone variables does not influence the general performance of BASolver *differently. Computing all the backbone variables of the formulas is cost-effective for the formulas, especially for the formulas with 100% percentage of backbone variables.*

4.3 Limitations and Threats to Validity

Firstly, BASolver uses the MiniSAT solver [11] as a black box to solve the propositional formulas, and use the unsatisfiable reasons returned by the MiniSAT solver. There are SAT solvers that are faster than MiniSAT and the reasons returned by MiniSAT may not be most appropriate for ALL-SAT solving. Also, since BC, NBC and BDD change the implementation inside the MiniSAT solver, we can not compare with them in the second and third experiments. Secondly, different blocking clauses inside the same ALL-SAT solver may also influence the performance of the ALL-SAT solver. In the second and third experiments, we did not consider the order of the satisfiable partial assignment. It is possible that some blocking clauses may accelerate the following SAT solving significantly. Moreover, we only use the formulas from the industrial tracks of SAT competitions. These formulas are generally more changeable for the ALL-SAT solvers as more solving time is required. Although there are 608 formulas, the number of formulas that solved by at least one of the solver is still less than 100, and most of them have less than 100 satisfiable assignments. We would like to choose some formulas with more than 1000 satisfiable assignments and the average difficulty for the SAT solver. We hope that at least one of the solvers will solve these formulas completely, and then we can compare the scalability among the solvers. Lastly, since only satisfiable formulas have backbone variables, BASolver is not applicable to over-constrained unsatisfiable formula. Although normally the input of ALL-SAT Solvers are satisfiable formulas, it is also promising to study on the over-constrained formulas with the techniques in BASolver.

5 RELATED WORK

BASolver is a blocking based ALL-SAT solver and the greedy strategy used in it to generate the partial assignments is from approaches proposed by Morgado and Marques-Silva [33] and Yu et al. [41] .

Jin and Somenz [21] proposed a blocking based ALL-SAT solver. It finds all the satisfiable assignments for a generic Boolean circuit using the blocking based framework. It can be directly used in manipulations of the logic circuits such as complementing and flattening. Gebser [14] et al. applied similar techniques to the enumeration of Answer Set Programming.

MBlocking [41] is another blocking based ALL-SAT solver that uses a greedy strategy named Minimal Blocking to generate the minimal set of blocking clauses. For a solution of the given formula, MBlocking either uses the set of dominant variables based on the clauses coverage or the set of decision variables and their corresponding reason variables based on the search tree to generate the blocking clauses. However, backbone variables may occur in the minimal set of blocking clauses and actually can be removed. The short blocking clause used in BASolver is more efficient than that used in MBlocking, and the reduced length of the blocking clauses is a key reason that BASolver computes more formulas with less computing time than MBlocking does.

The main difference between blocking based and non-blocking based solvers is the use of blocking clauses. In order to avoid finding the already known solutions of the formula, blocking based tools generate the blocking clauses of each known solution and add the blocking clauses back to the solver. In the non-blocking based All-SAT solvers [1, 25], backtrack techniques in the search tree are used to find more solutions to the given formula. Once a solution is found, the non-blocking based tools choose a decision level and backtrack the search tree to that level. A Different decision is made at that level and a new search path is generated based on the new decision.

Jabbour [19] et al. proposed a non-blocking ALL-SAT solver. After finding the first satisfiable assignment, it starts to run with the restart configuration. When a conflict happens, the solver backtracks to the previous decision level and propagates with the conflict reasons. Gebser [15] et al. proposes a new conflict-driven learning algorithm that uses an elaborated backtracking scheme. The scheme records all the decision variables after finding some projected solutions. In this way, the scheme can be maintained in the polynomial space and only a linear number of solution-excluding constraints are used. Grumberg [16] et al. introduced the notion of sub-levels and presented a sub-level based first UIP scheme that was compatible with the non-blocking approaches. Gebser [14] et al. presents alternative conflict resolution by means of non-chronological backtracking with a backtrack level limitation.

NBC [39] is a non-blocking based tool that backtracks the search tree to find every solution of the given formula. There are 4 different backtrack strategies in NBC, and by using these strategies in different orders, there are 8 different strategies in total in NBC. Each strategy performs differently on the formulas, therefore, the choice of strategies is a challenge for NBC. Comparing to NBC, BASolver only uses one strategy which is the short blocking clauses obtained by backbone variables to find every solution of the formula.

Formula caching technique associated with concise graph representations of propositional theories is also applied in the ALL-SAT solving field, including FBDD (Free Binary Decision Diagrams), OBDD (Ordered Binary Decision Diagrams) and d-DNNF (deterministic Decomposable Negation Normal Form). Such a representation is able to be efficiently constructed while executing an exhaustive DPLL search. A connection to All-SAT was mentioned in Huang and Darwiche [18]. An application to All-SAT solving itself was more explicitly mentioned in Toda and Tsuda [40] and a compilation-based All-SAT solver has been released.

Besides formula caching, the dualization of the Boolean functions could also be applied to the ALL-SAT solving. Given a DNF (Disjunction Normal Form) formula of a Boolean function, the ALL-SAT problems for the function transfer to the problem of computing the complete DNF formula for the dual function [12, 34, 38].

Another related topic is #SAT counting, which counts the total number of all the satisfiable assignments. Some of the approaches in the #SAT field use similar blocking techniques that are also used in ALL-SAT solvers, but the state-of-the-art #SAT approach is based on the universal hashing technique [32] which can not be applied to the ALL-SAT solvers.

6 CONCLUSION

We propose an All-SAT solver BASolver which uses backbone information to get shorter blocking clauses and achieve higher efficiency. Comparing to other All-SAT solvers, BASolver removes backbone variables from the blocking clauses. With shorter blocking clauses, the complexity of the following SAT solving decreases, the number of SAT solving needed decreases and the efficiency of All-SAT computing improves. Experiments show that BASolver is an efficient ALL-SAT solver, the shorter blocking clauses used in it leads to a reduction of the computing time. Although computing all the backbone variables requires additional computing time, but it is cost-effective since the average computing time of each satisfiable assignment is reduced in BASolver. Also, all the solvers in the experiments solve more formulas when the percentage of the backbone variables is higher, but the percentage does not affect BASolver differently.

ACKNOWLEDGMENTS

This research was partially supported by an industrial grant in collaboration with HUAWEI Technologies Co., LTD. Geguang Pu was partially supported by the NSFC Project No. 61632005 and the NSFC Project. No. 61532019.

REFERENCES

[1] Roberto J Bayardo Jr and Joseph Daniel Pehoushek. 2000. Counting models using connected components. In *AAAI/IAAI*. 157–162.

[2] Dirk Beyer and M Erkan Keremoglu. 2011. CPACHECKER: a tool for configurable software verification. In *International Conference on Computer Aided Verification*. 184–190.

[3] Armin Biere, Alessandro Cimatti, Edmund M Clarke, Ofer Strichman, and Yunshan Zhu. 2003. Bounded Model Checking. *Advances in Computers* 58 (2003), 117–148.

[4] Aaron R Bradley. 2012. Understanding IC3. In *International Conference on Theory and Applications of Satisfiability Testing*. 1–14.

[5] Jörg Brauer, Andy King, and Jael Kriener. 2011. Existential quantification as incremental SAT. In *International Conference on Computer Aided Verification*. Springer, 191–207.

[6] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *International Conference on Operating Systems Design and Implementation*. 209–224.

[7] Edmund Clarke, Daniel Kroening, Natasha Sharygina, and Karen Yorav. 2004. Predicate abstraction of ANSI-C programs using SAT. *Formal Methods in System Design* 25, 2-3 (2004), 105–127.

[8] Adnan Darwiche and Pierre Marquis. 2002. A knowledge compilation map. *Journal of Artificial Intelligence Research* 17 (2002), 229–264.

[9] Martin Davis, George Logemann, and Donald Loveland. 1962. A Machine Program for Theorem-Proving. *Commun. ACM* 5, 7 (July 1962), 394–397.

[10] Martin Davis and Hilary Putnam. 1960. A Computing Procedure for Quantification Theory. *J. ACM* 7, 3 (July 1960), 201–215.

[11] Niklas Eén and Niklas Sörensson. 2003. An extensible SAT-solver. In *International conference on theory and applications of satisfiability testing*. Springer, 502–518.

[12] Andrew Gainer-Dewar and Paola Vera-Licona. 2017. The minimal hitting set generation problem: algorithms and computation. *SIAM Journal on Discrete Mathematics* 31, 1 (2017), 63–100.

[13] Malay K Ganai, Aarti Gupta, and Pranav Ashar. 2004. Efficient SAT-based unbounded symbolic model checking using circuit cofactoring. In *IEEE/ACM International Conference on Computer Aided Design, 2004. ICCAD-2004*. IEEE, 510–517.

[14] Martin Gebser, Benjamin Kaufmann, André Neumann, and Torsten Schaub. 2007. Conflict-driven answer set enumeration. In *International Conference on Logic Programming and Nonmonotonic Reasoning*. Springer, 136–148.

[15] Martin Gebser, Benjamin Kaufmann, and Torsten Schaub. 2009. Solution enumeration for projected Boolean search problems. In *International Conference on AI and OR Techniques in Constriant Programming for Combinatorial Optimization Problems*. Springer, 71–86.

[16] Orna Grumberg, Assaf Schuster, and Avi Yadgar. 2004. Memory efficient all-solutions SAT solver and its application for reachability analysis. In *International Conference on Formal Methods in Computer-Aided Design*. Springer, 275–289.

[17] Aarti Gupta, Zijiang Yang, Pranav Ashar, and Anubhav Gupta. 2000. SAT-based image computation with application in reachability analysis. In *International Conference on Formal Methods in Computer-Aided Design*. Springer, 391–408.

[18] Jinbo Huang and Adnan Darwiche. 2007. The language of search. *Journal of Artificial Intelligence Research* 29 (2007), 191–219.

[19] Said Jabbour, Jerry Lonlac, Lakhdar Sais, and Yakoub Salhi. 2014. Extending modern SAT solvers for models enumeration. In *Proceedings of the 2014 IEEE 15th International Conference on Information Reuse and Integration (IEEE IRI 2014)*. IEEE, 803–810.

[20] Karthick Jayaraman, Nikolaj Bjørner, Geoff Outhred, and Charlie Kaufman. 2014. Automated analysis and debugging of network connectivity policies. *Microsoft Research* (2014), 1–11.

[21] HoonSang Jin and Fabio Somenzi. 2005. Prime clauses for fast enumeration of satisfying assignments to boolean circuits. In *Proceedings. 42nd Design Automation Conference, 2005*. IEEE, 750–753.

[22] Daniel Kroening and Michael Tautschnig. 2014. CBMC - C Bounded Model Checker. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. 389–391.

[23] Shuvendu K Lahiri, Randal E Bryant, and Byron Cook. 2003. A symbolic approach to predicate abstraction. In *International Conference on Computer Aided Verification*. Springer, 141–153.

[24] Shuvendu K Lahiri, Robert Nieuwenhuis, and Albert Oliveras. 2006. SMT techniques for fast predicate abstraction. In *International Conference on Computer Aided Verification*. Springer, 424–437.

[25] Bin Li, Michael S Hsiao, and Shuo Sheng. 2004. A novel SAT all-solutions solver for efficient preimage computation. In *Proceedings Design, Automation and Test in Europe Conference and Exhibition*, Vol. 1. IEEE, 272–277.

[26] Jianwen Li, Kristin Y. Rozier, Geguang Pu, Yueling Zhang, and Moshe Y. Vardi. 2019. SAT-Based Explicit LTLf Satisfiability Checking. In *The Thirty-Third AAAI Conference on Artificial Intelligence*. 2946–2953.

[27] Jianwen Li, Shufang Zhu, Yueling Zhang, Geguang Pu, and Moshe Y. Vardi. 2017. Safety model checking with complementary approximations. In *2017 IEEE/ACM International Conference on Computer-Aided Design*. 95–100.

[28] Nuno P Lopes, Nikolaj Bjørner, Patrice Godefroid, Karthick Jayaraman, and George Varghese. 2015. Checking beliefs in dynamic networks. In *12th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 15)*. 499–512.

[29] Nuno P Lopes, Nikolaj Bjørner, Patrice Godefroid, and George Varghese. 2013. Network verification in the light of program verification. *MSR, Rep* (2013).

[30] Rupak Majumdar, Sai Deep Tetali, and Zilong Wang. 2014. Kuai: A model checker for software-defined networks. In *2014 Formal Methods in Computer-Aided Design (FMCAD)*. IEEE, 163–170.

[31] Ken L McMillan. 2002. Applying SAT methods in unbounded symbolic model checking. In *International Conference on Computer Aided Verification*. Springer,

250–264.

[32] Kuldeep S Meel, Moshe Y Vardi, Supratik Chakraborty, Daniel J Fremont, Sanjit A Seshia, Dror Fried, Alexander Ivrii, and Sharad Malik. 2016. Constrained sampling and counting: Universal hashing meets SAT solving. In *Workshops at the thirtieth AAAI conference on artificial intelligence.*

[33] António Morgado and Joao Marques-Silva. 2005. Good learning and implicit model enumeration. In *17th IEEE International Conference on Tools with Artificial Intelligence*. IEEE, 6–pp.

[34] Keisuke Murakami and Takeaki Uno. 2013. Efficient algorithms for dualizing large-scale hypergraphs. In *2013 Proceedings of the Fifteenth Workshop on Algorithm Engineering and Experiments (ALENEX)*. SIAM, 1–13.

[35] Junaid Qadir and Osman Hasan. 2014. Applying formal methods to networking: theory, techniques, and applications. *IEEE Communications Surveys & Tutorials* 17, 1 (2014), 256–291.

[36] Samir Sapra, Michael Theobald, and Edmund Clarke. 2003. SAT-based algorithms for logic minimization. In *Proceedings 21st International Conference on Computer Design*. IEEE, 510–517.

[37] Shuo Sheng and Michael Hsiao. 2003. Efficient preimage computation using a novel success-driven atpg. In *2003 Design, Automation and Test in Europe Conference and Exhibition*. IEEE, 822–827.

[38] Takahisa Toda. 2013. Hypergraph transversal computation with binary decision diagrams. In *International Symposium on Experimental Algorithms*. Springer, 91–102.

[39] Takahisa Toda and Takehide Soh. 2016. Implementing efficient all solutions SAT solvers. *Journal of Experimental Algorithmics (JEA)* 21 (2016), 1–12.

[40] Takahisa Toda and Koji Tsuda. 2015. BDD construction for all solutions SAT and efficient caching mechanism. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*. 1880–1886.

[41] Yinlei Yu, Pramod Subramanyan, Nestan Tsiskaridze, and Sharad Malik. 2014. All-SAT using minimal blocking clauses. In *2014 27th International Conference on VLSI Design and 2014 13th International Conference on Embedded Systems*. IEEE, 86–91.

[42] Shuyuan Zhang, Sharad Malik, and Rick McGeer. 2012. Verification of computer switching networks: An overview. In *International Symposium on Automated Technology for Verification and Analysis*. Springer, 1–16.

[43] Yueling Zhang, Min Zhang, and Geguang Pu. 2020. Optimizing backbone filtering. *Sci. Comput. Program.* 187 (2020), 102374.