AllSATCC: Boosting AllSAT Solving with Efficient Component Analysis

Jiaxin Liang[1] , **Feifei Ma**[2] , **Junping Zhou**[1*] and **Minghao Yin**[1*]
[1]Northeast Normal University, China
[2]Institute of Software, Chinese Academy of Sciences, China
{liangjx348, zhoujp877, ymh}@nenu.edu.cn, maff@ios.ac.cn

Abstract

All Solution SAT (AllSAT) is a variant of Propositional Satisfiability, which aims to find all satisfying assignments for a given formula. AllSAT has significant applications in different domains, such as software testing, data mining, and network verification. In this paper, observing that the lack of component analysis may result in more work for algorithms with non-chronological backtracking, we propose a DPLL-based algorithm for solving AllSAT problem, named AllSATCC, which takes advantage of component analysis to reduce work repetition caused by non-chronological backtracking. The experimental results show that our algorithm outperforms the state-of-the-art algorithms on most instances.

1 Introduction

Propositional Satisfiability (SAT) is the problem of determining whether there exists an assignment that makes a given boolean formula evaluate to *True*. As the first problem proved to be NP-complete, SAT has numerous real-life applications, including hardware design, software verification, intelligence planning, etc. These applications typically rely on the ability of SAT solvers to find one satisfying assignment. Recently, an increasing number of applications require enumerating all the satisfying assignments. This new field opens up a variant problem of SAT, called *All Solutions SAT* (AllSAT for short), which is described as: enumerate all satisfying assignments if a given Conjunctive Normal Form (CNF) formula is satisfiable. Unlike model counting, specific satisfying assignments must be output in AllSAT, which makes it more challenging to develop efficient AllSAT solvers.

The practical applications of AllSAT problem involves in many different areas. For example:

(1) *Software testing*. To test a program, automated software testing requires generating a test suite, i.e., a set of test inputs, and checking the correctness of each output. We can model the specification describing the allowed inputs into a CNF formula; then, all the enumerated satisfying assignments together constitute a test suite [Khurshid *et al.*, 2003].

(2) *Data mining*. Frequent itemset mining is an important part of data mining, which consists in finding all sets of items with high support in a given transaction database. The problem of frequent itemset mining can be encoded into a CNF formula, with the whole set of frequent closed itemsets corresponding to all the satisfying assignments [Dlala *et al.*, 2016].

(3) *Network Verification*. Computing all reachable sets from source to destination can be viewed as the AllSAT problem, which has considerable significance for incremental calculating and fast debugging [Lopes *et al.*, 2013]. Real-time monitoring of bugs is requested when adding a new rule. Only a few changes are required to adapt to the new rule by recording all previous reachable sets.

Despite its practical importance, AllSAT is still an underexplored domain compared with other problems related to SAT. The existing AllSAT solvers are generally classified into three categories: blocking solvers, non-blocking solvers, and BDD (Binary Decision Diagram) solvers. The blocking solvers [McMillan, 2002; Jin *et al.*, 2005; Yu *et al.*, 2014; Zhang *et al.*, 2020a] repeatedly run SAT solvers and keep adding the negation of the resulting satisfying assignments (called blocking clauses) to the formula until the given formula is falsified. Non-blocking solvers [Grumberg *et al.*, 2004; Li *et al.*, 2004; Toda and Soh, 2016] are based on the DPLL (Davis-Putnam-Logemann-Loveland) algorithm, integrating with clause learning strategies and using flipped decision to get rid of the dependence on blocking clauses. BDD solvers [Huang and Darwiche, 2004; Toda and Soh, 2016; Toda and Inoue, 2017] compile the CNF formula into BDD to avoid repeatedly computing subformulas.

In this paper, we propose a DPLL-based algorithm for solving AllSAT, named AllSATCC, which employs non-chronological backtracking and component analysis (including components and caching). Although [Morgado and Marques-Silva, 2005] states that components and component caching are not readily applicable to AllSAT, we realize that the two techniques are beneficial for the DPLL-based AllSAT algorithms integrated with conflict-driven clause learning and non-chronological backtracking [Zhang *et al.*, 2001]. Conflict-driven clause learning has been widely used in mod-

*Corresponding author

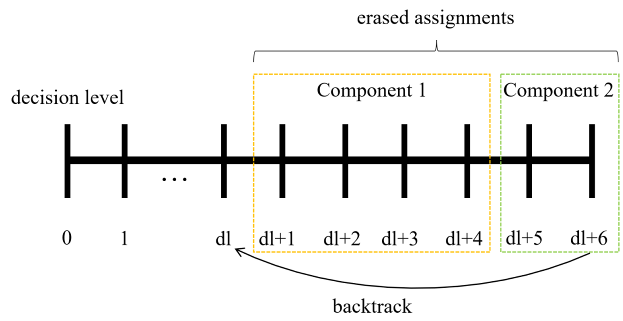

Figure 1: Assignments are erased due to non-chronological backtracking

ern AllSAT solvers. When a clause is learned, the algorithm may non-chronologically backtrack to a higher level of the search tree where the literals in the conflict clauses are located. Non-chronological backtracking has the potential of significantly reducing the amount of search. However, we observe that the non-chronological backtracking may erase the satisfying assignments of the previous search by lacking component analysis, which results in solving some components multiple times and consuming more time (described in Figure 1). Therefore, component analysis, including components and caching, is essential in the DPLL-based AllSAT algorithms. Investing in memory for precise component analysis is, in fact, costly. We employ a component selection strategy in the branching stage, as well as a new component caching strategy, to decrease memory costs. Using these techniques, AllSATCC can quickly determine that the formula is unsatisfiable, saving the invalid memory consumption and calculations of other components. Furthermore, AllSATCC transfers the satisfying assignments from memory to the external storage space to release the memory. In addition, we use methods to generate partial assignments to shorten the satisfying assignments since the performance of an AllSAT solver is greatly influenced by caching a large number of satisfying assignments.

We compared AllSATCC with four other AllSAT solvers, including two blocking-based solvers, BASolver and BC, a non-blocking-based solver, NBC, and a BDD-based solver, BDD. The results suggest that the efficiency of AllSATCC is noteworthy regardless of the number of backbone variables in the formulas. AllSATCC solved 467 of the 501 CNF benchmarks, 414, 429, 215, and 83 more than BASolver, BC, NBC, and BDD, respectively. Moreover, AllSATCC substantially outperformed the other solvers on 401 instances from SATLIB. AllSATCC not only enumerated all satisfying assignments but also solved each instance in less than 100 seconds. Additionally, we evaluated the solvers' potential solving ability on 34 unsolved instances. We set three time limits: 600 seconds, 1200 seconds, and 1800 seconds to check how many satisfying assignments the five solvers could find. The results show that the solving ability of each solver increases with the growing time limits. AllSATCC is still promising because it can find a part of satisfying assignments for all 34 instances in 1800 seconds.

The remainder of our paper is structured as follows. In Section 2, we describe some preliminary definitions. Section 3 provides the algorithm framework and several vital techniques in detail. In Section 4, we analyse experimental results on CNF benchmarks to present the efficiency of AllSATCC. Section 5 concludes this paper and future work.

2 Preliminaries

A *propositional variable* v is a boolean variable. A *literal* is a variable v or its negation $\neg v$. A *clause* is a logical disjunction on a finite set of literals. A *unit clause* is a clause with only one literal. The *Conjunctive Normal Form* (*CNF*) is a standard form of propositional formula defined over a finite set of variables $V = \{v_1, v_2, ..., v_n\}$, which is described as the conjunction ("$\wedge$") of clauses. A formula *F* in CNF can be expressed as an undirected graph, called a *constraint graph*. The graph is constructed by representing each variable with a vertex and connecting an edge between two vertices if the corresponding variables appear in the same clause. To some extent, there is a one-to-one correspondence between a formula and its constraint graph. A *component* of a constraint graph is a maximal connected subgraph such that every pair of vertices in the subgraph has a path. If a constraint graph contains several maximal connected subgraphs, it can be decomposed into several components, each of which is equivalent to a subformula F' of F.

Given a formula *F*, an *assignment* of *F* is a mapping from *V* to $\{1, 0, u\}$, where *u* represents an undefined value. An assignment is called a *full assignment* if each variable in *V* is assigned 1 or 0; otherwise, it is known as a *partial assignment*. A full (or partial) assignment is called a *full* (*or partial*) *satisfying assignment* if the formula F evaluates to 1 under the assignment. In all full satisfying assignments, if the value of a variable remains unchanged, the variable is defined as a *backbone variable*. Note that we sometimes use satisfying assignments to express full (or partial) satisfying assignments without affecting the correctness of understanding in the rest of the paper.

Definition 1 (AllSAT Problem)**.** *The All Solutions Satisfiability Problem (AllSAT for short) is to enumerate all full satisfying assignments for a given propositional formula F.*

In the following, we present an example to illustrate the above definitions.

Example 1. *Suppose a CNF formula* $F = (v_1 \vee v_2) \wedge (\neg v_1 \vee v_2) \wedge v_3$*. The constraint graph indicating F can be decomposed into two disjoint components, whose corresponding subformulas are* $F_1 = (v_1 \vee v_2) \wedge (\neg v_1 \vee v_2)$ *and* $F_2 = v_3$*. There are two satisfying assignments* $\{\neg v_1, v_2, v_3\}$ *and* $\{v_1, v_2, v_3\}$*, where an assignment containing a literal* l *means that* l *evaluates to 1. In the formula,* v_2 *and* v_3 *are both backbone variables because the values of the two variables are unchanged in all the satisfying assignments.*

After specifying the definitions, we present some basic rules in DPLL-based algorithms for AllSAT. For DPLL-based algorithms, finding all satisfying assignments is a matter of branching on a variable and recursively calling DPLL until each branch returns a satisfying assignment or UNSAT. In this process, a branching variable is called a *decision variable*. Each decision variable is associated with a *decision level*, expressed by $dlevel$ ($dlevel \in \{-1, 0, 1, ..., |V|\}$), which denotes the depth of the decision tree whose nodes are associated with decision variables. *Unit propagation* is applied to simplify formulas, which recursively eliminates the clauses containing the literal l and the literal $\neg l$ in the remaining clauses if the unit clause is l.

3 Algorithm for AllSAT Problem Combining Components with Caching

In this section, we present a DPLL-based algorithm for solving AllSAT problem, called AllSATCC, which combines components with caching. We enumerate partial assignments to reduce the number of assignments and make AllSATCC more practical. In the following, we first introduce the general framework of AllSATCC and then describe the key techniques in detail.

3.1 Overview of AllSATCC

Algorithm 1 shows the framework of AllSATCC based on DPLL searching in a sole decision tree. All satisfying assignments are kept in *SAs*. The satisfiability of the formula is indicated by $status \in \{$*UNKNOWN*, *SAT*, *UNSAT*$\}$. The set C is a set storing the candidate components. A flag variable $flag$ is used to control whether the algorithm executes the main loop. A component is represented as c_{clevel}, where $clevel$ specifies the processing order of the components.

AllSATCC begins by performing unit propagation and backbone variables detection repeatedly until there are no unit clauses and backbone variables in the input formula F (line 3). The backbone variables detection method uses EDUCIBone [Zhang *et al.*, 2020b]. If the satisfiability of the formula is determined after preprocessing, $flag$ changes to *FALSE* (line 4). Then AllSATCC dynamically extracts the components (line 6–8), which will be described in the next subsection. AllSATCC iteratively selects a component from C afterwards and removes it from the set. Once deciding a component from C, AllSATCC checks whether the component has been already in cache. If it is cached, it means that duplicate search can be avoided by acquiring its values from the cache. There are three cases. (1) Unsatisfiable state is cached, and $dlevel$ is -1, it means that the entire search space is finished; (2) unsatisfiable state is cached and $dlevel$ is not -1, it means that there are still unflipped decision variables, and AllSATCC non-chronologically backtracks to a proper decision level and flips the decision variable in the component c_{clevel}; (3) satisfiable state is cached, the satisfying assignments are updated in cache (line 11–15). If the selected component is not in cache, AllSATCC iteratively chooses a variable as a decision variable by VSADS strategy [Sang *et al.*, 2005] to descend until a satisfying assignment is found for each extracted component in C or backtracking to -1 $dlevel$. In the iterative procedure, the 1-UIP scheme [Moskewicz *et al.*, 2001] is performed to generate learning clauses and obtain the decision level of the search tree where the literals in the conflict clauses are located. If $dlevel$ is equal to -1, the main loop is terminated. Otherwise, it repeatedly does a

Algorithm 1: AllSATCC

Input: a CNF formula F
Output: specific satisfying assignments *SAs* if F is SAT; UNSAT otherwise

1 *SAs* $\leftarrow \emptyset$; $C \leftarrow \emptyset$;
2 $flag \leftarrow$ *TRUE*; $status \leftarrow$ *UNKNOWN*;
3 $status \leftarrow$ preprocess(F);
4 **if** $status$ = *SAT* or *UNSAT* **then** $flag \leftarrow$ *FALSE*;
5 **while** $flag$ = *TRUE* **do**
  // extract components
6   **if** *extract new components* **then**
7     $C \leftarrow$ new components;
8   **else** update *SAs*;
  // decide next branch
9   **if** $C \neq \emptyset$ **then**
10   decide a component c and remove c from C;
11   **if** c *is in cache* **then**
12   **if** *cached UNSAT* **then**
13   **if** $dlevel = -1$ **then** $flag \leftarrow$ *FALSE*;
14   **else** backtrack to component c_{clevel} at $dlevel$;
15   **else** update *SAs*;
16   **else**
17   decide a variable in component c;
18   **else**
19   update *SAs*;
20   **if** $dlevel = -1$ **then** $flag \leftarrow$ *FALSE*;
21   **else** backtrack to component c_{clevel} at $dlevel$;
  // clause learning
22   **while** *deduce*() = *CONFLICT* **do**
23     $dlevel \leftarrow$ analyze_conflict();
24   **if** $dlevel = -1$ **then** $flag \leftarrow$ *FALSE*;
25   **else** backtrack to component c_{clevel} at $dlevel$;
26 **if** *SAs* $\neq \emptyset$ **then return** *SAs*;
27 **else return** UNSAT;

traceback to the nearest component to flip the decision literal until satisfying assignments of all components are calculated (line 16–25). Finally, the algorithm returns all satisfying assignments of the formula if *SAs* is not empty; otherwise, the formula is proved to be unsatisfiable (line 26–27).

In the algorithm, the key techniques for speeding up the search are component extraction, the strategies for shortening the satisfying assignments, and component caching, which are described in the following subsections.

3.2 Dynamic Component Extraction in Branching

Rymon first presented components for generating prime implicants algorithm, which used the structural information of a problem [Rymon, 1994]. In our algorithm, this technique works by analysing the connectivity structure of a CNF formula and dynamically extracting the components depending on the current partial assignment. The partial assignment and the subformula in the current branch are changed if AllSATCC assigns a variable v or non-chronologically backtracks to flip a decision variable, which may lead to the corresponding component splitting into several components. All the newly extracted components are stored in a candidate set C, sorted in ascending order of the number of variables in each component.

Determining how to select a component for search is important for improving the efficiency of solving AllSAT problem. The component selection strategy in AllSATCC is to choose a component with the smallest number of variables from the current candidate set. The search starts with a component and goes down one of its child components recursively until the satisfiability of the input formula is determined. If the result of this process is UNSAT, which means that the component is unsatisfiable, we can directly draw the conclusion. Otherwise, the algorithm solves the last component completely and then backtracks to the next-to-last component until the entire input formula has been explored.

The component technique plays an important role in AllSATCC. The non-chronological backtracking without the component technique may erase the satisfying assignments of previous satisfied components after generating a learning clause, resulting in some components being solved numerous times. AllSATCC switches to the next component when a satisfying assignment is found for a component (or the component is unsatisfiable). When unsatisfiable components occur, invalid calculation of other components is saved as much as possible. It also enables AllSATCC to rapidly determine the UNSAT state. Moreover, when the formula is hard to solve, it is possible for AllSATCC to enumerate a part of all satisfying assignments. As a result, the AllSAT solver, which combines component method with non-chronological backtracking, has a clear advantage whether the formula is UNSAT or SAT.

3.3 Generating Partial Assignments

The performance of AllSAT solvers is susceptible to the number of satisfying assignments. To hold the satisfying assignments compactly, we enumerate partial assignments instead of full satisfying assignments because a set of full satisfying assignments can be represented as a condensed partial assignment. In the following, we propose our methods for generating partial assignments.

Partial assignments based on decision variables. In a satisfying assignment δ, each of the assigned variables is assigned either by a decision or by an implication due to unit propagation. The assignments of the implied variables are eliminated to shorten the satisfying assignments δ. The unit propagation at each node in a decision tree can be completed in polynomial time, which ensures that the satisfying assignment recovery is quick. Suppose a formula $F = (v_1 \vee v_2) \wedge (\neg v_1 \vee v_2)$, where v_1 is a decision variable. If v_1 is assigned to 1, the implied variable v_2 must be assigned to 1. We generate a partial assignment $\delta' = \{v_1\}$ from $\delta = \{v_1, v_2\}$, which only contains a decision assignment.

Partial assignments omitting irrelevant variables. The values of irrelevant variables have no effect on the satisfiability of the formula. As a result, the irrelevant variables' assignments can be deleted from the full satisfying assignments. For example, consider a formula $F = v_1 \vee v_2 \vee \neg v_3$. When the variable v_1 is assigned to 1, the formula F is satisfiable regardless of the values of v_2 and v_3. So the set of satisfying assignments, $\{\{v_1, v_2, v_3\}, \{v_1, \neg v_2, v_3\}, \{v_1, v_2, \neg v_3\}, \{v_1, \neg v_2, \neg v_3\}\}$, can be expressed by a short partial assignment $\{v_1\}$, which makes the solver more practical.

Partial assignments omitting backbone variables. The pre-process of AllSATCC includes backbone variables detection. The identified backbone variables can be removed from the satisfying assignments. That is because (1) the values of the backbone variables are fixed; (2) the result of the reduced formula remains unchanged, which is simplified by setting the fixed values of the backbone variables; and (3) the recovery process from the satisfying assignments to the partial assignments is easy since the values of backbone variables are stored.

3.4 Component Caching Scheme in AllSATCC

In this subsection, we present how to apply the caching scheme in AllSATCC. Component caching techniques are widely employed in model counting solvers [Sang *et al.*, 2004; Thurley, 2006; Sharma *et al.*, 2019], most of which use the hash caching to record the satisfiable probability or the number of satisfying assignments for a component. However, this caching technique is not applicable to the AllSAT problem since unit propagation in some components with specific structures is not allowed in this caching scheme, preventing us from obtaining the specific assignments. Furthermore, when backtracking happens in this caching scheme, the assignments of some components are erased, which is unfavourable for accelerating the solving process.

To address the problems, our component caching mechanism maintains a stack to keep track of the values of each decision variable, an array A_1 to cache the partial assignments of each component, and another array A_2 to follow up the relationship of components, including the sibling components, child components, and parent components.

When a satisfying assignment of a component is obtained, the relevant values of the decision variables in the stack are cleared, and the array A_1 is updated. Note that if the partial assignment is unsatisfiable, a label recording UNSAT is preserved in the array. Now, no matter when the solver performs backtracking, the erased assignments can be retained in the array.

After capturing all the partial assignments of a component in the array, they are sent to the component's parent component in order to merge the current partial assignments. In addition, when the algorithm backtracks to a component with a smaller $clevel$, the algorithm allows the partial assignments of its sibling components to be transferred to the external storage space, therefore reducing the size of the array A_1.

4 Experimental Results

In this section, we carry out experimental investigations to evaluate our solver, AllSATCC[1], which is implemented in C++ with g++ compiler with version 4.8.1. All experiments

[1]The executable code and benchmarks are available at https://github.com/LyreRabbit/AllSATCC.

Instances	ave_vars	ave_cls	ave_bb	ave_SAs	BASolver	BC	NBC	BDD	AllSATCC
Flat125-301 (100)	375	1403	0.00%	3.49×10^8	0	0	98	99	100
Flat150-360 (101)	450	1680	0.00%	5.72×10^{10}	0	0	72	100	101
Flat175-417 (100)	525	1951	0.00%	2.27×10^{12}	0	0	23	80	100
Flat200-479 (100)	600	2237	0.00%	2.22×10^{13}	0	0	1	52	100
AProve (3)	6615	24867	0.00%	0	3	3	3	3	3
complete (4)	600	27147	44.13%	2.75	2	0	0	0	2
dimacs (17)	655	2856	100.00%	1	17	17	17	17	17
Manthey (25)	11693	52634	71.37%	3.94	18	8	15	15	17
mp1 (12)	13038	202156	74.74%	454.38	7	4	4	3	7
iscas (39)	3078	7395	0.24%	7.32×10^{10}	6	6	19	15	20
Total (501)	1592	9961	8.82%	5.83×10^{12}	53	38	252	384	467

Table 1: Comparison of AllSATCC with BASolver, BC, NBC, and BDD

SAs	BASolver	BC	NBC	BDD	AllSATCC
$[10^4, 10^5)$	0	0	1	1	1
$[10^5, 10^6)$	0	0	7	7	7
$[10^6, 10^7)$	0	0	22	22	22
$[10^7, 10^8)$	0	0	61	61	62
$[10^8, 10^9)$	0	0	61	62	62
$[10^9, 10^{10})$	0	0	42	64	65
$[10^{10}, 10^{11})$	0	0	0	69	78
$[10^{11}, 10^{12})$	0	0	0	26	47
$[10^{12}, 10^{13})$	0	0	0	16	34
$[10^{13}, \infty)$	0	0	0	3	24
Total (401)	0	0	194	331	401

Table 2: Distribution of the Solved Instances over Flat Instances with Respect to Number of Full Satisfying Assignments

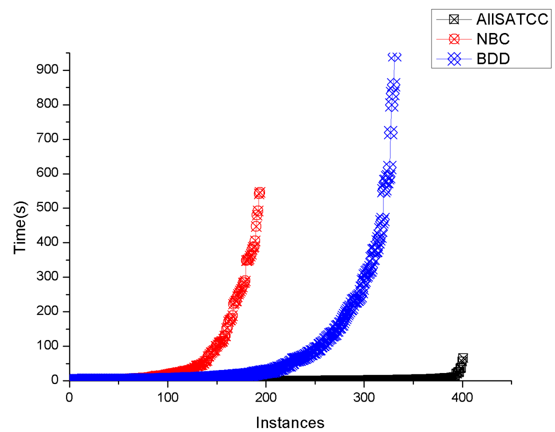

Figure 2: Comparison of the Solving Time over Flat Instances with the Increasing Cutoff Time

are run on Intel Xeon CPU E5-2650 v4 @ 2.20GHz with 128GB RAM under CentOS release 6.10.

4.1 Experiment Settings

Benchmarks. In our experiments, we empirically access the performance of AllSATCC on real-world benchmark instances from SAT competitions from 2011 to 2017, SATLIB, and ISCAS85/89, which have recently been used for testing the solving efficiency of AllSAT solvers in [Zhang *et al.*, 2020a]. In all the benchmark instances, dimacs, AProve, complete, Manthey, and mp1 are downloaded from SAT competitions from 2011 to 2017, lacking Encryption compared with [Zhang *et al.*, 2020a] because of the failure of the link; the set iscas is from ISCAS85/89, which is converted by TG-Pro [Chen and Marques-Silva, 2012]; and our extended set Flat is from SATLIB, which is produced by graph coloring problems.

Baseline solvers. We compare AllSATCC with four solvers: BASolver, BC, NBC, and BDD. BASolver [Zhang *et al.*, 2020a] is the best competitive solver for AllSAT so far, using shorter blocking clauses and backbone variable detection EDUCIBone [Zhang *et al.*, 2020b] to speed up the solving. BC, NBC, and BDD are three state-of-the-art solvers from [Toda and Soh, 2016]. BC is also a blocking-based solver; NBC is a DPLL-based solver using conflict learning and non-chronological backtracking without components; and BDD is a BDD-based solver, which compiles each CNF formula into a BDD so as to simply generate all satisfying assignments. For each instance, BC, NBC and BDD output the full assignments, while BASolver and AllSATCC output the partial assignments and the number of full assignments, which is calculated according to the partial assignment strategies. Note that we obtain the number of full assignments by slightly modifying the source code of BASolver.

4.2 Evaluation

Part 1: Table 1 shows the results of the comparison of AllSATCC with the other four solvers, where the cutoff time for each instances is 1200 seconds. In the table, the first column records the name of each set as well as the number of instances in each set (in brackets). Four sets of these instances are graph coloring problems, denoted as Flat$x$-y, where x and y are the number of vertices and edges, respectively. For each set of instances, we report the number of variables (ave_vars), the number of clauses (ave_cls), the percentage of backbone

SAs	BASolver	BC	NBC	BDD	AllSATCC
$[0, 0]$	14	14	14	18	7
$[1, 10^1)$	0	0	0	1	0
$[10^1, 10^2)$	4	0	0	0	0
$[10^2, 10^3)$	14	8	4	3	0
$[10^3, 10^4)$	2	3	4	2	3
$[10^4, 10^5)$	0	9	2	4	16
$[10^5, 10^6)$	0	0	10	6	8
Total	34	34	34	34	34

Table 3: Distribution of the found satisfying assignments over unsolved instances, where the cutoff time is 600 seconds.

SAs	BASolver	BC	NBC	BDD	AllSATCC
$[0, 0]$	14	14	12	18	3
$[1, 10^1)$	0	0	1	0	0
$[10^1, 10^2)$	0	0	0	0	0
$[10^2, 10^3)$	2	8	0	3	1
$[10^3, 10^4)$	11	3	1	0	1
$[10^4, 10^5)$	5	4	2	1	6
$[10^5, 10^6)$	2	5	6	4	12
$[10^6, 10^7)$	0	0	10	8	8
$[10^7, 10^8)$	0	0	2	0	3
Total	34	34	34	34	34

Table 4: Distribution of the found satisfying assignments over unsolved instances, where the cutoff time is 1200 seconds.

variables (ave_bb, ratio of the number of backbone variables to the number of variables), the number of full satisfying assignments (ave_SAs), and the number of instances that each solver can solve within the cutoff time. AllSATCC outperforms the other four solvers, as seen in Table 1. By comparing with BASolver, we find that the performance of AllSATCC is not affected by ave_bb even though the two solvers both use backbone variable detection. Moreover, the compared results of NBC and AllSATCC show that component analysis is essential in AllSAT algorithms integrating with non-chronological backtracking, since AllSATCC incorporates with component analysis while NBC does not.

Part 2: In this part, we examine the outcomes for Flat instances in further detail, focusing on two aspects: solution ability and solving efficiency. Table 2 illustrates the distribution of the total number of solved instances over Flat instances within time limit, where each interval is divided according to the number of full satisfying assignments. Since the total number of satisfying assignments of all instances in Flat is greater than 10^4, the interval starts from 10^4. As shown in Table 2, AllSATCC achieves the best results, especially when the number of satisfying assignments exceeds 10^{13}, demonstrating that it has higher ability. Moreover, we conduct an experiment to test the efficiency of our algorithm. Figure 2 illustrates how the number of solved instances increases as the time increases over the Flat instances. The instances are sorted by the time required to solve them, and each point in Figure 2 represents the time required to solve

SAs	BASolver	BC	NBC	BDD	AllSATCC
$[0, 0]$	7	7	4	11	0
$[1, 10^1)$	0	0	0	0	2
$[10^1, 10^2)$	1	2	1	4	1
$[10^2, 10^3)$	5	5	6	3	1
$[10^3, 10^4)$	9	6	1	0	2
$[10^4, 10^5)$	5	9	3	1	4
$[10^5, 10^6)$	4	5	4	2	8
$[10^6, 10^7)$	1	0	10	3	8
$[10^7, 10^8)$	2	0	1	8	7
$[10^8, 10^9)$	0	0	4	2	1
Total	34	34	34	34	34

Table 5: Distribution of the found satisfying assignments over unsolved instances, where the cutoff time is 1800 seconds.

each instance. Due to the fact that BAsolver and BC cannot solve these instances within 1200 seconds indicated in Table 2, we only list the results of NBC, BDD, and AllSATCC. As illustrated in the figure, AllSATCC surpasses NBC and BDD by solving each instance in less than 100 seconds.

Part 3: We conduct an evaluation of the potential solving ability of the solvers on all the instances except Flat. To save space, we only display the results of the 34 instances that can't be solved by AllSATCC to check how many full satisfiable assignments the five solvers can find within different cutoff times in Table 3, 4, and 5. From these tables, it is clear that the ability of finding solutions of each solver increases with the increasing time limit. The solving ability of AllSATCC is still promising because it can find satisfying assignments (though not all) of all 34 instances in 1800 seconds, while BASolver, BC, NBC, and BDD can't find even one satisfying assignment for 7, 7, 4, and 11 instances, respectively. Although AllSATCC finds the highest order of magnitude of satisfying assignments for relatively few instances, it is still competitive.

5 Conclusion and Future Work

In this paper, we propose a new AllSAT algorithm called AllSATCC, which incorporates components and caching with non-chronological backtracking. Besides, we present methods to shorten the full satisfying assignments. The results of the experiments show that AllSATCC achieves good performance across a broad range of instances. In the future, we plan to study better component selection strategies to improve the solving efficiency, and enumerate the top-k solutions for hard problems, such as the diversified top-k cliques problem [Zhou *et al.*, 2021].

Acknowledgments

We are grateful to Professor Geguang Pu for providing the source code of BASolver, and to the anonymous reviewers for their numerous comments. This work is supported by the Fundamental Research Funds for the Central Universities (No. 2412019FZ050, 2412019ZD013), NSFC

(No.61976050, 61972384, and 62106040), and the Educational Department of Liaoning Province (No. QL202016).

References

[Chen and Marques-Silva, 2012] Huan Chen and Joao Marques-Silva. TG-Pro: a SAT-based ATPG system. *Journal on Satisfiability, Boolean Modeling and Computation*, 8(1-2):83–88, 2012.

[Dlala *et al.*, 2016] Imen Ouled Dlala, Said Jabbour, Lakhdar Sais, and Boutheina Ben Yaghlane. A comparative study of SAT-based itemsets mining. In *International Conference on Innovative Techniques and Applications of Artificial Intelligence*, pages 37–52. Springer, 2016.

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

[Huang and Darwiche, 2004] Jinbo Huang and Adnan Darwiche. Using DPLL for efficient OBDD construction. In *Proceedings of the Seventh International Conference on Theory and Applications of Satisfiability Testing*, pages 157–172. Springer, 2004.

[Jin *et al.*, 2005] HoonSang Jin, HyoJung Han, and Fabio Somenzi. Efficient conflict analysis for finding all satisfying assignments of a boolean circuit. In *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 287–300. Springer, 2005.

[Khurshid *et al.*, 2003] Sarfraz Khurshid, Darko Marinov, Ilya Shlyakhter, and Daniel Jackson. A case for efficient solution enumeration. In *Proceedings of the Sixth International Conference on Theory and Applications of Satisfiability Testing*, pages 272–286. Springer, 2003.

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

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

[McMillan, 2002] Ken L. McMillan. Applying SAT methods in unbounded symbolic model checking. In *International Conference on Computer Aided Verification*, pages 250–264. Springer, 2002.

[Morgado and Marques-Silva, 2005] António Morgado and Joao Marques-Silva. Good learning and implicit model enumeration. In *Proceedings of the 17th IEEE International Conference on Tools with Artificial Intelligence*, pages 6–136. IEEE, 2005.

[Moskewicz *et al.*, 2001] Matthew W Moskewicz, Conor F Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th annual Design Automation Conference*, pages 530–535, 2001.

[Rymon, 1994] Ron Rymon. An se-tree-based prime implicant generation algorithm. *Annals of Mathematics and Artificial Intelligence*, 11(1):351–365, 1994.

[Sang *et al.*, 2004] Tian Sang, Fahiem Bacchus, Paul Beame, Henry A Kautz, and Toniann Pitassi. Combining component caching and clause learning for effective model counting. In *Proceedings of the Seventh International Conference on Theory and Applications of Satisfiability Testing*, pages 20–28. Springer, 2004.

[Sang *et al.*, 2005] Tian Sang, Paul Beame, and Henry Kautz. Heuristics for fast exact model counting. In *Proceedings of the Eighth International Conference on Theory and Applications of Satisfiability Testing*, pages 226–240. Springer, 2005.

[Sharma *et al.*, 2019] Shubham Sharma, Subhajit Roy, Mate Soos, and Kuldeep S. Meel. Ganak: A scalable probabilistic exact model counter. In *IJCAI*, volume 19, pages 1169–1176, 2019.

[Thurley, 2006] Marc Thurley. SharpSAT–counting models with advanced component caching and implicit BCP. In *Proceedings of the Ninth International Conference on Theory and Applications of Satisfiability Testing*, pages 424–429. Springer, 2006.

[Toda and Inoue, 2017] Takahisa Toda and Takeru Inoue. Exploiting functional dependencies of variables in all solutions SAT solvers. *Journal of Information Processing*, 25:459–468, 2017.

[Toda and Soh, 2016] Takahisa Toda and Takehide Soh. Implementing efficient all solutions sat solvers. *Journal of Experimental Algorithmics*, 21:1–44, November 2016.

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

[Zhang *et al.*, 2001] Lintao Zhang, Conor F Madigan, Matthew H Moskewicz, and Sharad Malik. Efficient conflict driven learning in a boolean satisfiability solver. In *Proceedings of the 2001 IEEE/ACM International Conference on Computer-Aided Design*, pages 279–285. IEEE, 2001.

[Zhang *et al.*, 2020a] Yueling Zhang, Geguang Pu, and Jun Sun. Accelerating All-SAT computation with short blocking clauses. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, pages 6–17, 2020.

[Zhang *et al.*, 2020b] Yueling Zhang, Min Zhang, and Geguang Pu. Optimizing backbone filtering. *Science of Computer Programming*, 187:102374, 2020.

[Zhou *et al.*, 2021] Junping Zhou, Chumin Li, Yupeng Zhou, Mingyang Li, Lili Liang, and Jianan Wang. Solving diversified top-k weight clique search problem. *Science China Information Sciences*, 64:150105, 2021.