AllSATCC: Boosting AllSAT Solving with Efficient Component Analysis

Jiaxin Liang\textsuperscript{1}, Feifei Ma\textsuperscript{2}, Junping Zhou\textsuperscript{1*} and Minghao Yin\textsuperscript{1*}

\textsuperscript{1}Northeast Normal University, China
\textsuperscript{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 \textit{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) \textit{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 \textit{et al.}, 2003].

(2) \textit{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 \textit{et al.}, 2016].

(3) \textit{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 \textit{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 under-explored 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 \textit{et al.}, 2005; Yu \textit{et al.}, 2014; Zhang \textit{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 \textit{et al.}, 2004; Li \textit{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 \textit{et al.}, 2001]. Conflict-driven clause learning has been widely used in mod-
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 ("\&") 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, \text{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 \( F = (v_1 \lor v_2) \land (\neg v_1 \lor v_3) \lor v_4 \). The constraint graph indicating \( F \) can be decomposed into two disjoint components, whose corresponding subformulas are \( F_1 = (v_1 \lor v_2) \land (\neg v_1 \lor v_2) \) 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
Algorithm 1: AllSATCC

| Input: | a CNF formula $F$ |
| Output: | specific satisfying assignments $SAs$ if $F$ is SAT; UNSAT otherwise |
| $SAs \leftarrow \emptyset$; | $C \leftarrow \emptyset$; |
| $flag \leftarrow TRUE$; | $status \leftarrow UNKNOWN$; |
| $status \leftarrow preprocess(F)$; | $if\ status = SAT\ or\ UNSAT\ then\ flag \leftarrow FALSE$; |
| $while\ flag = TRUE$ do | $//\ extract\ components$
| $if\ extract\ new\ components\ then$ |
| $C \leftarrow new\ components$; | $//\ decide\ next\ branch$
| $else\ update\ SAs$; |
| $if\ C \neq \emptyset$ then | $if\ C\ is\ in\ cache$ |
| | $if\ cached\ UNSAT\ then$
| | $if\ dlevel = -1\ then\ flag \leftarrow FALSE$; |
| | $else\ backtrack\ to\ component\ c_{dlevel}\ at\ dlevel$; |
| | $else\ update\ SAs$; |
| | $else\ decide\ a\ variable\ in\ component\ c$; |
| | $else$ |
| | $if dlevel = -1\ then\ flag \leftarrow FALSE$; |
| | $else\ backtrack\ to\ component\ c_{dlevel}\ at\ dlevel$; |
| $while\ deduce() = CONFLICT\ do$ |
| $dlevel \leftarrow analyze\ conflict();$ |
| $if\ dlevel = -1\ then\ flag \leftarrow FALSE$; |
| $else\ backtrack\ to\ component\ c_{dlevel}\ at\ dlevel$; |
| $if\ SAs \neq \emptyset\ then\ return\ SAs$; |
| $else\ return\ UNSAT$; |

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 $c_{dlevel}$, 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 $c_{dlevel}$; (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 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 cor-
responding 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 im-
portant for improving the efficiency of solving AllSAT prob-
lem. 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 com-
ponent 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 con-
clusion. 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 All-
SATCC. 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 satis-
fying assignment is found for a component (or the compo-
nent 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 num-
er of satisfying assignments. To hold the satisfying assign-
ments compactly, we enumerate partial assignments instead of
full satisfying assignments because a set of full satisfying
assignments can be represented as a condensed partial assign-
ment. In the following, we propose our methods for generating
partial assignments.

Partial assignments based on decision variables. In a satis-
fying assignment $\delta$, each of the assigned variables is as-
signed 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 $\delta$. The unit
propagation at each node in a decision tree can be com-
pleted in polynomial time, which ensures that the satisfy-
ing assignment recovery is quick. Suppose a formula
$F = (v_1 \lor v_2) \land (\neg v_1 \lor v_2)$, 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 satis-
fiability of the formula. As a result, the irrelevant
variables’ assignments can be deleted from the full satis-
fying assignments. For example, consider a formula
$F = v_1 \lor v_2 \lor \neg v_3$. When the variable $v_1$ is assigned
to 1, the formula $F$ is satisfiable regardless of the val-
ues 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 detec-
tion. 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 re-
covery 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 prob-
lem since unit propagation in some components with spe-
cific structures is not allowed in this caching scheme, pre-
venting us from obtaining the specific assignments. Further-
more, when backtracking happens in this caching scheme, the
assignments of some components are erased, which is un-
favourable for accelerating the solving process.

To address the problems, our component caching mecha-
nism maintains a stack to keep track of the values of each de-
cision variable, an array $A_1$ to cache the partial assignments
of each component, and another array $A_2$ to follow up the re-
lation 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 pre-
served in the array. Now, no matter when the solver performs
backtracking, the erased assignments can be regained in the
array.

After capturing all the partial assignments of a component
in the array, they are sent to the component’s parent com-
ponent in order to merge the current partial assignments. In
addition, when the algorithm backtracks to a component with
a smaller level, the algorithm allows the partial assignments
of its sibling components to be transferred to the external stor-
age 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\textsuperscript{1}, which is implemented in
C++ with g++ compiler with version 4.8.1. All experiments
\textsuperscript{1}The executable code and benchmarks are available at
https://github.com/LyreRabbit/AllSATCC.
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 instance 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

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

Table 1: Comparison of AllSATCC with BASolver, BC, NBC, and BDD
SAs | BASolver | BC | NBC | BDD | AllSATCC
---|---|---|---|---|---
[0, 0] | 14 | 14 | 14 | 18 | 7
[1, 10^1] | 0 | 0 | 0 | 1 | 0
[10^2, 10^3] | 4 | 0 | 0 | 0 | 0
[10^6, 10^9] | 14 | 8 | 4 | 3 | 0
[10^6, 10^9] | 2 | 3 | 4 | 2 | 3
[10^6, 10^9] | 0 | 9 | 2 | 4 | 16
[10^6, 10^9] | 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
[10^2, 10^3] | 2 | 8 | 0 | 3 | 1
[10^6, 10^9] | 11 | 3 | 1 | 0 | 1
[10^6, 10^9] | 5 | 4 | 2 | 1 | 6
[10^6, 10^9] | 2 | 5 | 6 | 4 | 12
[10^6, 10^9] | 0 | 0 | 10 | 8 | 8
[10^6, 10^9] | 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.

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 satisfying 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


