Optimizing Backbone Filtering

Yueling Zhang* Jianwen Li* Min Zhang*‡§✉ Geguang Pu*✉ Fu Song†

‡MOE International Joint Lab of Trustworthy Software §International Research Center of Trustworthy Software

*Computer Science and Software Engineering Institute, East China Normal University

†School of Information Science and Technology, ShanghaiTech University

***Abstract*—Backbone is the common part of each solution in a given propositional formula, which is a key to improving the performance of SAT solving and SAT-based applications, such as model checking and program analysis. In this paper, we propose an optimized approach that combines implication-driven (IDF), conflict-driven (CDF), and unique-driven (UDF) heuristics to improve backbone computing. IDF uses the particular binary structure of the form $a \leftrightarrow b \wedge c$ to find more backbone literals. CDF comes from the observation that for a clause $\neg a \vee b$, if a is a backbone literal, then b is also a backbone literal. Besides CDF, we are also able to detect new non-backbone literals by UDF. A literal l is not a backbone literal, if there is no clause $\phi \in \Phi$ that is only satisfied by l. We implemented our approach in a tool named DUCIBone with the above optimizations (IDF+CDF+UDF), and conducted experiments on formulas used in previous work and SAT competitions (2015, 2016). Results demonstrate that DUCIBone solved 4% (507 formulas) more formulas than minibones (minibones-RLD, 490 formulas) does under its best configuration. Among 486 formulas solved by all tools (DUCIBone, minibones-RLD, minibones-cb100), DUCIBone reduced 7% (35131 seconds) than minibones (37454 seconds). Experiments indicate that the advantage of DUCIBone is more obvious when the formulas are harder.**

I. Introduction

The *backbone* of a satisfiable formula is a set of literals that are always *true* in all models of the formula, which plays an important role in finding heuristic strategies of NP-hard problems [1]. The identification of backbone improves the performance of random SAT solvers [2], [3], [4], Lin-Kernighan local search algorithms for Travel Salesman Problem [5] and the post-silicon fault localization in integrated circuits [6], [7]. It also improved the performance of chip verification [8], graph coloring problems [9] and artificial intelligence strategies generation [10].

However, computing backbone is a co-NP-hard problem. There are mainly three kinds of backbone computing frameworks [11]. The naive one is the so-called implicant (assignment) enumeration, which enumerates all models of a satisfiable formula. To avoid finding duplicate implicants, it inserts previous implicants as new constraints into the formula. The second one is called iterative SAT testing, which is often equipped with optimizations of backbone filtering. Filtering optimizations are used to reduce the number of SAT testings. These optimizations aim to detect non-backbone literals without invoking the expensive SAT testings, and reduce the total overhead of the backbone computation. The third approach is named core-based algorithm, which is motivated by the solving procedure of modern SAT solvers, i.e., constructing an unsatisfiable formula by adding assumptions to the original formula. In this way, SAT solvers are able to return the reason for unsatisfiability of the input formula, which is a subset of the given assumptions. If the reason contains only one element (literal), then a backbone literal is found (negation of the reason).

Although filtering is cheap comparing to SAT testings, [11] shows that in practice equipping filtering optimizations in the backbone computation slows down the overall performance. They claimed that the core-based algorithm without backbone filtering optimizations outperforms other algorithms, and formulas containing large amount of binary clauses are not suitable for filtering optimizations. However, formulas with more than a half binary clauses appear frequently in practice, such as circuit checking, silicon chips testing, and microprocessors verification. Better solutions for computing the backbone of such formulas are still highly in demand.

In this paper, we aim to improve the performance of iterative SAT testing on these formulas, and propose three kinds of filtering optimizations. The first one is called *implication-driven* filtering. The idea comes from the structural features of formulas with large amount of binary clauses. Such clauses are often obtained from formulas with the form of $a \leftrightarrow b \wedge c$. It is not hard to see that a is a backbone literal if and only if b and c are both backbone literals. Based on this observation, if a is a backbone literal, then b and c are both backbone literals. Symmetrically, if $\neg b$ or $\neg c$ is a backbone literal, then $\neg a$ is a backbone literal as well. Given a CNF formula, the implication-based filtering identifies such binary clauses in the formula, and then applies the above idea to detect additional backbone literals without SAT testing.

The second heuristics is called *conflict-driven* filtering, which is based on the fact that, for the clauses of the form $\neg a \vee b$, if a is a backbone literal, then b must be a backbone literal. The last heuristics is named *unique-driven* filtering. Since a backbone literal a has the property that there must be a clause only satisfied by a, literals that do not meet such property can be ruled out directly without using any additional SAT testing.

978-1-5386-1925-4/17/$31.00

We implemented our approach in a tool DUCIBone and evaluated with empirical experiments. We compared DUCIBone with the state-of-the-art tool minibones [11]. There are many configurations in minibones corresponding to different algorithms, we choose two configurations, minibones-RLD, and minibones-cb100. minibones-cb100 using core-based algorithms is the best configuration suggested by Janota et.al. [11]. minibones-RLD is also a backbone filtering technique.

DUCIBone was able to solve 507 formulas, 4% more than minibones-RLD (490 formulas), and 5% more than minibones-cb100 (486 formulas). Moreover, DUCIBone solved all the formulas that were solved either by minibones-RLD or minibones-cb100. Considering the formulas that were solved by all three tools, DUCIBone reduced 22% time than minibones-RLD, and 7% time than minibones-cb100. Moreover, DUCIBone performs better on harder formulas. The advantage of DUCIBone is more obvious for formulae whose time of satisfiability checking precedes 300 seconds.

The rest of the paper is organized as follows. Section II introduces notations and Section III introduces the iterative-testing-based backbone computing framework we follow in this paper. Section IV introduces the overview of our backbone computing framework. Section V presents the heuristics for backbone filtering, and Section VI shows the experimental results. Finally Section VII and Section VIII discusses and concludes the paper, respectively.

II. Preliminaries

Let $\mathcal{X}$ be a finite set of *Boolean variables*. A *literal* l is either a Boolean variable $x \in \mathcal{X}$ or its negation $\neg x$. A *clause* ϕ is a disjunction of literals $\bigvee_{l_i \in \phi} l_i$, which is also represented as a set of literals $\{l_i \mid 1 \leq i \leq n\}$. We say a set L of literal *satisfies* a clause ϕ, denoted as $L \models \phi$, iff $\phi \cap L \neq \emptyset$. Given a set C of clauses, $L \models C$ iff $L \models \phi$ for each clause ϕ in C. W.o.l.g., we assume that every clause ϕ is consistent, i.e., l and $\neg l$ cannot be in ϕ at the same time.

A *Boolean formula* Φ over $\mathcal{X}$ is a Boolean combination of variables in $\mathcal{X}$. We assume that formulas are given in conjunctive normal form (CNF), i.e., each formula Φ is a conjunction of clauses $\bigwedge_{l_i \in \phi} \phi_i$. Analogously, the formula Φ can also be represented as a set of clauses $\{\phi_i \mid 1 \leq i \leq n\}$. We denote by Φ_l the set of clauses that contains l, i.e., $\phi \in \Phi_l$ iff $l \in \phi$.

An *assignment* v of a Boolean formula Φ is a mapping $\mathcal{X} \rightarrow \{0, 1, -1\}$, where 1 (resp. 0) denotes true (resp. false) and -1 means the *unspecified value*. v is a *full assignment* if it is a mapping from $\mathcal{X}$ to $\{0, 1\}$, otherwise it is a *partial assignment*. We say that a full assignment v *implies* a partial assignment v' if for each variable $x \in X$, $v'(x) \neq -1$ implies $v'(x) = v(x)$. v is a *model* of Φ, denoted by $\Phi(v) \equiv true$, if v is a full assignment and the evaluation of Φ under v is true. Generally, v is an *implicant* of Φ if v is an assignment of Φ and $\Phi(v)$ is true. In the rest of the paper, we consider the assignment, model and implicant to be sets of literals in the following way: a variable $x \in v$ if $v(x) = 1$, and $\neg x \in v$ if $v(x) = 0$.

A Boolean formula Φ is *satisfiable* if there is a model of Φ, otherwise Φ is *unsatisfiable*. We now give the formal definition of *backbone*.

Definition 1 (Backbone)**.** *Given a Boolean formula Φ, the backbone of Φ, denoted by $\mathsf{BL}(\Phi)$, is a maximal set of literals such that $\mathsf{BL}(\Phi) \subseteq v$ for each model v of Φ. Every literal in $\mathsf{BL}(\Phi)$ is called a* backbone literal*, and every literal not in $\mathsf{BL}(\Phi)$ is a* non-backbone literal*. We will use $\overline{\mathsf{BL}}(\Phi)$ to denote the set of non-backbone literals.*

It is known that the backbone for each satisfiable formula Φ is unique [11]. The backbone of an unsatisfiable formula can be defined as an empty set. Therefore, in this work, we focus on satisfiable formulas.

Theorem 1. *[1][12] Given a satisfiable formula Φ and a literal l, deciding whether l is a backbone literal is co-NP-complete.*

Let us consider the formula $\Phi = \{\neg x_1 \vee \neg x_2, x_1, x_3 \vee x_4\}$, we have $\mathsf{BL}(\Phi) = \{x_1, \neg x_2\}$.

We introduce the concept of equilibrium literals which will be used later.

Definition 2 (Equilibrium Literals)**.** *Given a formula Φ, for each pair (l, l') of literals, l' is an equilibrium literal of l iff for all models v of Φ, $l \in v \Longrightarrow l' \in v$. Let E_l denote the set of equilibrium literals of l.*

III. Computing Backbone via Iterative Testing

Our approach DUCIBone proposed in this paper is an optimized algorithm based on the iterative-testing backbone computation. The idea comes from the simple property of backbone literals as below.

Property 1. *l is a backbone literal of the Boolean formula Φ, iff Φ is satisfiable and $\Phi \cup \{\neg l\}$ is unsatisfiable.*

From the definition of backbone, backbone literals are contained in every model of Φ. We can first compute a model via a SAT testing, and then iteratively check the literals in the model one by one. This is the basic idea of iterative-testing backbone computation [11]. The algorithm is shown in Algorithm 1.

We suppose there is a SAT testing oracle (i.e., $SAT(\Phi)$) which takes the formula Φ as input, and returns $\langle false, \emptyset \rangle$ if Φ is unsatisfiable; otherwise returns $\langle true, v \rangle$ in which v is a model of Φ. By fixing a model v of Φ, iteratively check the satisfiability of $\Phi \cup \{\neg l\}$ for each $l \in v$. If the result is false (unsatisfiable), l is a backbone literal and is removed from v. Otherwise, a new model v' of Φ is obtained, then v is reduced to $v \cap v'$. The algorithm terminates until v becomes empty.

In Algorithm 1, the process $filtering$ is used to reduce the size of v as far as possible by deleting non-backbone literals. Previous works on such filtering techniques mainly include: 1) Implication-Extraction: extracting an implicant v'

Algorithm 1: Computing backbone via iterative testing.

Input : A formula Φ
Output: The backbone of Φ, $\mathsf{BL}(\Phi)$

```
1  ⟨ret, v⟩ := SAT(Φ);
2  if ret = false then
3      return ∅;
4  v := filtering(v);
5  Y := ∅;
6  foreach l ∈ v do
7      ⟨ret', v'⟩ := SAT(Φ ∪ {¬l});
8      if ret' = false then
9          Y := Y ∪ {l};
10         v := v\{l};
11         Φ := Φ ∪ {l};
12     else
13         v' := filtering(v');
14         v := v ∩ v';
15 return Y;
```

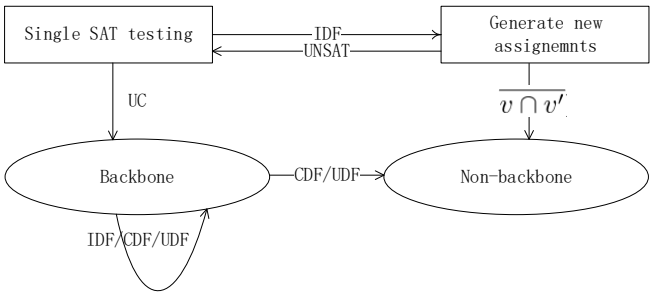

Fig. 1: The General Framework of DUCIBone

from the model v as small as possible, or 2) Rotatable-Literal Deletion (short for RLD): deleting the so-called *rotatable literals* in the model v. It is in principle guaranteed that the second optimization subsumes the first one. For more details we refer to [11]. A surprising conclusion from [11] is that, backbone filtering is not helpful in practice, namely, Algorithm 1 performs better without the filtering optimization. In this paper, we present three further optimizations for backbone filtering based on the results from RLD.

IV. Overview of DUCIBone

In this section, we introduce the overview of our approach DUCIBone, which is able to fill the performance gap between Filtering Optimizations and the core-based techniques in backbone computing.

Figure 1 illustrates the framework of our backbone computing approach. Given a satisfiable formula Φ, and a model v, we first use a SAT testing to check the satisfiability of $\Phi \wedge l$, for every literal l in Φ. If $\Phi \wedge l$ is unsatisfiable, then l is the unique unsatisfiable reason (UC) since Φ is satisfiable. Therefore, l is a backbone literal. We compute a set of backbone literals E_l based on l using IDF. If both $\Phi \wedge l$ and $\Phi \wedge \neg l$ are satisfiable, then l is a non-backbone literal. We try to generate a new model by fixing the assignments of l and every literal in E_l to false, the fixed literals are called assumptions. If a new model v' is generated, literals that are assigned to different values in v and v' are non-backbone literals (i.e., $\overline{v \cap v'}$). If no model is generated, we choose another literal l' in Φ which has not been decided as backbone or non-backbone literal, and compute the satisfiability of $\Phi \wedge l'$ using a SAT testing again. During the computation, DUCIBone uses the Conflict-driven Filtering (CDF) and Unique-driven Filtering (UDF) optimizations to find new backbone and non-backbone literals. The algorithm terminates when $\mathsf{BL}(\Phi)$ is found.

V. Heuristics for Backbone Filtering

In this section, we first present the three optimizations for backbone filtering, then introduce the explicit implementation of the improved backbone computing algorithm equipped with the optimizations.

Implication-Driven Filtering (IDF)

The implication-driven filtering (IDF) is designed for accelerating backbone computing for the particular binary structure of the form $a \leftrightarrow b \wedge c$. In this formula, if a is a backbone literal, then both b and c are backbone literals. Symmetrically, if $\neg b$ or $\neg c$ is a backbone literal, then $\neg a$ is a backbone literal.

Taking the formula $(a \leftrightarrow b \wedge c) \wedge (c \leftrightarrow d \wedge e)$ as an example. From Definition 2, we know that $E_c = \{d, e\}$ and $E_a = \{b, c, d, e\}$. The following lemma guarantees that if the literal l is a backbone literal, all elements (literals) in E_l are backbone literals as well.

Lemma 1. *For every literal in Φ, if $l \in \mathsf{BL}(\Phi)$, then all literals $l' \in E_l$ are also backbone literals.*

Proof. If $l \in \mathsf{BL}(\Phi)$, then for each model $v \models \Phi$, $l \in v$. Thus for each literal $l' \in E_l$, we get that $l' \in v$. The result immediately follows. □

However, it is non-trivial to compute E_l for a give literal l. We design a heuristic algorithm to find equilibrium literals. We consider the form of $l \Leftrightarrow a \wedge b$, whose CNF form is

$$(a \vee \neg l) \wedge (b \vee \neg l) \wedge (\neg a \vee \neg b \vee l).$$

We will compute the equilibrium literals of l.

Algorithm 2 presents how to find equilibrium literals of a given literal l. The loop from Line 2 to Line 6 traverses on every clause $\phi \in \Phi$. If the literal l appears in current ϕ, the algorithm tries to find equilibrium literals by finding the clauses that containing the negation of literal $\neg l$. Let $\phi := l \vee \neg l_1 \vee \neg l_2$, the algorithm tries to find clauses $\neg l \vee l_1$ and $\neg l \vee l_2$ in the given formula.

After obtaining the set of pairs $\langle l, E \rangle$, IDF first checks whether l in each pair is a backbone literal. If l is a backbone literal, then all elements in E are identified as backbone literals without any additional SAT testings.

Algorithm 2: Finding Equilibrium Literals

Input : A formula Φ, a literal l
Output: A subset of Equilibrium Literals E_l

$E := \emptyset$;
foreach $\phi \in \Phi$ **do**
  if $l \in \phi$, *and* $|\phi| = 3$ **then**
    Let $\phi := l \vee \neg l_1 \vee \neg l_2$;
    if $\exists \phi_1, \phi_2 \in \Phi$, *s.t.* $\phi_1 := l_1 \vee \neg l$, *and* $\phi_2 := l_2 \vee \neg l$ **then**
      $E := E \cup \{l_1, l_2\}$;
return E;

Conflict-Driven Filtering (CDF)

The conflict-driven filtering (CDF) optimization comes from the observation that, for a clause ϕ of the form $\neg a \vee b$, if a is a backbone literal, then b is also a backbone literal. The correctness of this heuristics is guaranteed by the following theorem.

Theorem 2. *Given a CNF formula Φ and a clause $\phi := \neg a \vee b$ of Φ, if a is a backbone literal of Φ, then b is also a backbone literal of Φ.*

Proof. Since a is a backbone literal of Φ, for every model v of Φ, $\neg a \notin v$. Consider now an arbitrary model v of Φ here. Since $\neg a \notin v$, $b \in v$ must holds, otherwise the clause ϕ cannot be satisfied by v, which is a contradiction with the assumption that v is the model of Φ. As a result, we prove that b is a backbone literal of Φ as well. □

Algorithm 3: The implementation of CDF.

Input : A formula Φ, a backbone literal l
Output: A subset of new backbone literals

$B := \emptyset$;
foreach $\phi \in \Phi$ **do**
  if $\phi = \neg l \wedge l'$ **then**
    $B := B \cup \{l'\}$;
return B;

Although CDF is applicable only for clauses with two literals, there are many such cases in practice (recall the benchmarks from hardware model checking and etc). The algorithm of CDF is shown in Algorithm 3.

Unique-Driven Filtering (UDF)

In addition to finding new backbone literals via CDF, we are also able to find new non-backbone literals by using the unique-driven filtering optimization (UDF). According to the definition of backbone, we can get the following theorem.

Theorem 3. *Given a satisfiable formula Φ, and a backbone literal $l \in \mathsf{BL}(\Phi)$, there exists at least one clause $\phi \in \Phi$, such that l is the unique satisfying literal of ϕ.*

Proof. Suppose there exists a model $v \models \Phi$:

- If $l \notin v$, then l is a non-backbone literal by the definition of backbone (Definition 1), which is a contradiction.
- If $l \in v$, suppose there is no clause $\phi \in \Phi$, such that l is the unique satisfying literal of ϕ, let $v' = v \setminus \{l\} \cup \{\neg l\}$, then $v' \models \Phi$, thus l is a non-backbone literal by the definition of backbone (Definition 1), which is a contradiction.

Therefore, for a backbone literal l of a formula Φ, there exists at least one clause $\phi \in \Phi$ such that l is the unique satisfying literal of ϕ.

□

Given a literal l and a subset of backbone literals in a satisfiable formula Φ, for every clause $\phi \in \Phi_l$ that contains l, if there always exists a backbone literal in ϕ, then l is a non-backbone literal. Since l violates the condition in Theorem 3, that l is not the unique satisfying literal of any clause $\phi \in \Phi$.

Backbone Computing with Optimized Filtering

With the above optimizations (IDF+CDF+UDF), we develop an algorithm to compute backbone literals. We iteratively check if a literal l is a backbone literal by a SAT testing. The selection of later literals in SAT testings are guided by the information obtained from the previous literals. During the computation, more backbone and non-backbone literals are extracted without invoking SAT testing.

Algorithm 4: Computing backbone using improved Filtering Optimizations

Input : A formula Φ, a model v
Output: Backbone literals of Φ

$L := v$;
foreach $l \in L$ **do**
  $L := L \setminus \{l\}$;
  if $\forall \phi \in \Phi_l$, $\mathsf{BL}(\Phi) \cap \phi \neq \emptyset$ **then**
    continue;
  $(ret, v') := SAT(\Phi \wedge \neg l)$;
  if $ret = true$ **then**
    $L := L \setminus \{c \in L | v(c) \neq v'(c)\}$;
    $v := v'$;
    $(ret, v) := SAT(\bigwedge_{a \in E_l} \neg a \wedge \Phi)$;
    if $ret = true$ **then**
      $L := L \setminus \{c \in L | v(c) \neq v'(c)\}$;
  if $ret = false$ **then**
    $\mathsf{BL}_l := E_l \cup \mathsf{BL}_l$;
    $L := L \setminus \{E_l\}$;
    foreach $\phi \in \Phi_{\neg l}, \phi = l' \vee \neg l$ **do**
      $\mathsf{BL}_l := \{E'_l\} \cup \mathsf{BL}_l$;
      $L := L \setminus \{E'_l\}$;
return BL_l;

Algorithm 4 computes backbone literals using our improved filtering optimizations. At Line 8, it enumerates the different literals between two different models. If the values of literals are different in different models, these literals must be non-backbone literals. At Line 10, we construct an assignment by assigning the value of all literals in E_l to false. Suppose that there are k literals in E_l, then we add k unit clauses to Φ, one unit clause $\neg a$ for each literal $a \in E_l$. At Line 4, we use UDF to find more non-backbone literals without a new SAT testing. At Line 14, we find more backbone literals using IDF, and we find more backbone literals using CDF at Line 16.

VI. Experimental Study

DUCIBone is implemented with IDF, CDF, and UDF optimizations, interfacing Minisat 2.2 as SAT solver. In this section, we conduct an experimental study on 1276 formulas to check performance of DUCIBone. The performance of DUCIBone is evaluated in two dimensions: the performance of DUCIBone on formulae with large amount of binary clauses, and the scalability of DUCIBone on industrial formulas.

We compared DUCIBone with the state-of-the-art tool minibones [11], which implemented several backbone computing algorithms. We consider the core-based algorithm of minibones, denoted by minibones-cb100, which outperforms others pointed out in [11], as well as the iterative testing-based algorithm of minibones with RLD filtering, denoted by minibones-RLD.

The experiments were conducted on a cluster of IBM iDataPlex 2.83 GHz, with a memory limit of 4GB, and a time limit of 3600 seconds. There is no paralleling or portfolio in our experiments of all tools. Neither DUCIBone nor minibones uses the incremental feature of Minisat.

DUCIBone was able to solve 507 formulas, while minibones-RLD solved 490 formulas and minibones-cb100 solved 486 formulas. DUCIBone solved 4% more formulas than minibones-RLD, and 5% more formulas than minibones-cb100. DUCIBone is able to solve all the formulas solved either by minibones-RLD or minibones-cb100. On the formulas solved by all the three tools, DUCIBone reduced 22% computing time than minibones-RLD, and 7% computing time than minibones-cb100.

Benchmark Setup

In our experiments, we considered 1276 formulas consisting of all the available formulas of [11] (779 formulas), and all the formulas from the industrial tracks of 2015 and 2016 SAT competitions (497 formulas). We notice that all 779 formulas from [11] are satisfiable.

To evaluate the performance of DUCIBone on formulas with large amount of binary clauses, we considered 435 formulas from [11] which consist of a large amount of binary clauses. Moveover, minibones-RLD needs more time on these formulas than minibones-cb100. Therefore, these formulas are more suitable to evaluate the performance of DUCIBone on formulas with large amount of binary clauses. These formulas are divided into 5 groups according to their names, that are the first 5 groups of Table I .

Since it does not make sense to compute backbone of unsatisfiable formulas, we have to remove all the unsatisfiable ones from the formulas taken from the industrial tracks of 2015 and 2016 SAT competitions. According to the results of SAT competitions, among 497 formulas, there are 256 satisfiable formulas that are solved, in which 168 formulas are from SAT competition 2015, and 88 formulas are from SAT competition 2016. Moreover, these formulae were divided into different groups according to the origin of them, e.g., planning problems. We classified all the 256 formulas into 46 groups based on their names. From 46 groups, we considered 5 groups according to the following criterion:

- There exists at least one formula Φ in a group such that the $\mathsf{BL}(\Phi)$ can be successfully computed by at least one tool.
- There are no less than 10 formulas in this group.

The selected 5 groups showing as the last 5 groups in Table I, consist of 122 formulas that were created from several well known applications of SAT, including formal verification [13], planning [14], [15], [16], and cryptanalysis [17]. The *2dlx* group consists of formulas from formal verification, the *mrpp* group consists of formulas from multi-robot path planning, the *aprove* group consists of formulas from term-rewriting, the *vmpc* group consists of formulas from the problem of stream chip verification in cryptanalysis, and the *manthey* group consists of formulas from the problem of finding gray codes which might attack encryptions. These formulas are used to evaluate the performance on industrial formulas.

In Table I, the first column is the name of the group, the second column is the number of formulas in the group. The average of the numbers of variables, clauses, and binary clauses of each group are listed in column 3, 4, and 5 respectively.

There are two features of the formulas that minibones-RLD performed poorly: (1) large formulas with massive clauses, and (2) formulas with more than half of binary clauses. In Table I, the groups *9vliw*, *bin_**, *2dlx* and *1394** have these features.

Performance Comparison

In Table II, the numbers of formulas solved by DUCIBone, minibones-RLD and minibones-cb100 in different groups are given. The first two columns are names and numbers of formulas in each benchmark. The numbers of the formulas solved by DUCIBone are presented in column 3, the number of the formulas solved by minibones-RLD (resp. minibones-cb100) and the division of between DUCIBone and minibones-RLD (resp. minibones-cb100) are presented in column 4 (resp .6) and 5 (resp. 7), respectively.

All three tools have similar performance on simple formulas, namely, *2dlx* and *1394**. DUCIBone performed better on formulas of the groups *manthey* and *vmpc*. DUCIBone solved

Name	No. of formulas	Average of the No. of variables	Average of the No. of clauses	Average of the No. of binary clauses
2dlx	100	20731	63673	42448
1394*	100	19587	56178	37452
9vliw	100	220717	652779	435186
bin_*	135	21820	626473	417649
aprove	18	26744	68845	28130
mrpp	26	6163	46460	0
vmpc	13	780	134273	53709
manthey	44	2457	10308	2835
dimacs	21	5087	19935	1030
total	557	324086	1052451	600790

TABLE I: Benchmark

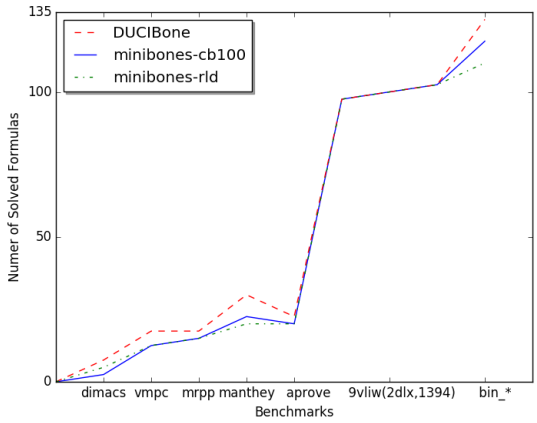

Fig. 2: Number of Solved Formulas on Different Benchmarks

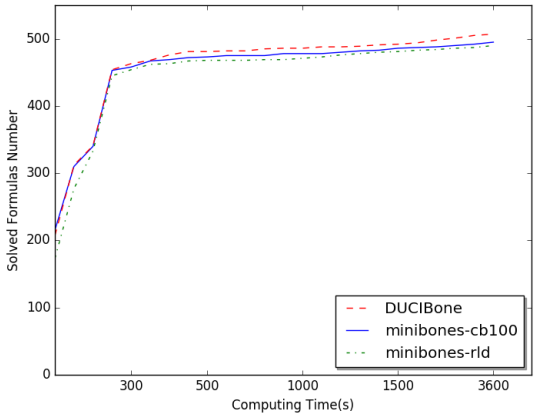

Fig. 3: Numbers of solved formulas within different computing times

24 formulas of *manthey*, which is 30% more than minibones-RLD (i.e., 17 formulas). DUCIBone solved all 13 formulas of *vmpc*, while minibones-cb100 only solved 9 formulas. Figure 2 shows the numbers of solved formulas in different benchmarks.

Figure 3 presents the number of solved formulas within different computing time. All three tools were able to solve 100 formulas within 60 seconds. DUCIBone solved 10 more formulas than minibones-RLD and minibones-cb100 around 300 seconds, and solved 17 more formulas by 3600 seconds. These experiments demonstrate that the advantage of DUCIBone is more obvious when the formulas are harder.

Table III shows the computing time of different groups by DUCIBone, minibones-RLD, and minibones-cb100. For a fair comparison, we only show the result of the formulas that are solved by all three tools simultaneously. The first column presents groups as usual, the second column gives the number of formulas of each group that are solved by all three tools. The following three columns show the computing time of DUCIBone, minibones-RLD, and the division value between them. The last two columns show the computing time of minibones-cb100, and the division value between the time used in DUCIBone and minibones-cb100.

To avoid turbulence, we repeatedly computed the same formula for three times, and use the average computing time of the three computations. In general, DUCIBone used less

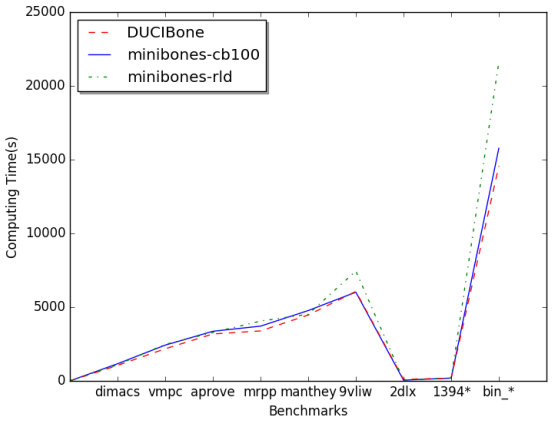

Fig. 4: Total computing time of each group

time than the other two tools, saving 7% and 22% time than minibones-cb100 and minibones-RLD, respectively.

All three tools performed steadily on the formulas of *2dlx*, and *1394**, as these formulas are relatively simple which costed less than 1 second to compute. minibones-RLD performed poorly on the formulas of *9vliw*, and *bin_**, which contains large amounts of binary clauses, preventing minibones-RLD

Name	No. of formulas	DUCIBone	minibones-RLD	$\frac{\text{minibones-RLD}}{\text{DUCIBone}}$	minibones-cb100	$\frac{\text{minibones-cb100}}{\text{DUCIBone}}$
2dlx	100	100	100	100%	100	100%
1394*	100	100	100	100%	100	100%
9vliw	100	99	99	100%	99	100%
bin_*	135	132	128	96.9%	127	96.2%
aprove	18	18	18	100%	18	100%
mrpp	26	15	13	86.7%	13	86.7%
vmpc	13	13	10	76.9%	9	69.2%
manthey	44	24	17	70.8%	17	70.8%
dimacs	21	6	5	83.3%	3	50%
total	557	507	490	96.6%	486	95.5%

TABLE II: Number of solved formulas

Name	No. of formulas	DUCIBone	minibones-RLD	$\frac{\text{DUCIBone}}{\text{minibones-RLD}}$	minibones-cb100	$\frac{\text{DUCIBone}}{\text{minibones-cb100}}$
2dlx	100	50	50	100%	50	100%
1394*	100	188	196	95.9%	190	98.9%
9vliw	99	6064	7449	81.2%	6014	100.8%
bin_*	127	14570	21611	67.4%	15771	92.3%
aprove	18	3180	3282	96.9%	3364	94.5%
mrpp	13	3388	4057	83.5%	3712	91.2%
vmpc	9	2178	2475	88%	2424	89.9%
manthey	17	4464	4519	98.8%	4767	93.6%
dimacs	3	1049	1097	95.6%	1162	90.2%
total	486	35131	44736	78.5%	37454	93.7%

TABLE III: Computing time of of each group

Name	Average Variables Number	DUCIBone	minibones-RLD	$\frac{\text{minibones-RLD}}{\text{DUCIBone}}$	minibones-cb100	$\frac{\text{minibones-cb100}}{\text{DUCIBone}}$
aprove	26744	16119	15903	93.9%	15093	93.9%
mrpp	6163	858	744	86.7%	782	91.1%
vmpc	780	625	523	83.6%	516	82.6%
manthey	2457	1739	1638	94.1%	1638	94.1%
dimacs	5087	638	489	76.7%	276	43.2%
total	41231	19979	18487	92.5%	18305	91.6%

TABLE IV: Number of literals determined by SAT testing only once

from finding non-backbone literals efficiently. DUCIBone overcomes this problem by detecting backbone literals instead of non-backbone literals. Therefore, binary clauses will not be the obstacle of DUCIBone any more. In contrary, DUCIBone detects more backbone literals in the early stage of computing by the IDF optimization from binary clauses.

Figure 4 shows the computing time of the three tools, from which we can easily observe that: DUCIBone costed less time than minibones-RLD and minibones-cb100, except for *9vliw* group, in which minibones-cb100 and DUCIBone performed similarly. Concluded from Figure 4, DUCIBone is at least a comparable and complementary tool of minibones-cb100.

To understand the diverse performance among three tools, we study the underlying mechanisms of them. minibones-RLD is designed to reduce the number of SAT testings by detecting more non-backbone literals. minibones-cb100 is able to find backbone literals faster with the help of unsatisfiable reasons returned by each SAT solver testing. DUCIBone is able to detect non-backbone literals and backbone literals using IDF, CDF, and UDF. The key of the three tools is to avoid SAT testing by detecting backbone or non-backbone literals. Since minibones-cb100 works different from minibones-RLD and DUCIBone, instead of comparing the number of SAT testings, we compare the numbers of literals that are determined (as backbone or non-backbone) by the initial SAT testing in Table IV.

For a fair comparison, we only considered the formulas that are solved by all the three tools within 300 seconds, that are groups: *bin_**, *mrpp*, *vmpc*, *dimacs* and *aprove*. The first two columns represent the name and the number of average variables in each group as usual, the following 3 columns represent the average of the number of literals that are pruned by DUCIBone and minibones-RLD, and the division between them. The last 2 columns represent the average number of literals pruned by minibones-cb100, and division between the numbers of pruned literals by DUCIBone and minibones-cb100.

As we can observe, DUCIBone determines more literals using the initial SAT testing. On average, by using the first SAT testing, DUCIBone determines 48.5% literals, minibones-RLD determines 44.8% literals, and minibones-cb100 determines 44.3% literals. DUCIBone identified 8% more literals than minibones-RLD, and 9% more literals than minibones-cb100. minibones-cb100 worked differently from minibones-RLD and DUCIBone, it identifies

backbone literals by SAT testings with assumptions. The more the assumptions are, the less the time needs. That's why minibones-cb100 computes faster than minibones-RLD while pruning less literals. We conclude that detecting more backbone literals with the first SAT testing is helpful for backbone computing.

VII. Related Work

Backbone computing has been studied in several works. Kaiser and Kühlin proposed three model-enumeration based algorithms for computing backbone literals [18] using SAT testings. Dubois and Dequen proposed the heuristics for computing backbone literals for hard 3-SAT formulas which yields DPLL-type algorithms with a significant performance improvement over the best previous algorithms [19]. Climer et al. proposed a graph-based approach to discover backbone literals which relies on over-approximation and under-approximation [20]. Zhu et al proposed an iterative SAT testing based algorithm [6], [7] which is more efficient than previous model enumeration algorithms.

Marques-Silva et al. investigated previous algorithms for computing backbone literals mentioned above, including model enumeration, iterative SAT-testing and optimizations with modern SAT solvers [21], [22], [11]. They proposed a tool minibones with several configurations, and claimed that the best configuration of minibones is minibones-cb100. Another configuration of minibones is minibones-RLD, which fails to improve the performance of iterative SAT testing on formulas that contains large numbers of binary clauses. minibones-RLD only worked well on formulas with little binary clause.

VIII. Conclusion

We presented implication-driven, conflict-driven, and unique-driven heuristics in the optimization of backbone computing, focusing on the clauses which can't be improved by minibones-RLD. Experimental results showed that DUCIBone performs better on hard formulas, and fill the performance gap of binary clauses between minibones-cb100 and minibones-RLD. In general, DUCIBone (507 formulas) solved 17 more formulas than minibones-RLD (490 formulas), and 21 formulas more than minibones-cb100 (486 formulas). On the formulas that are solved by all three tools, DUCIBone (35131 seconds) reduced 22% computing time than minibones-RLD (44736 seconds), and 7% computing time than minibones-cb100 (37454 seconds).

Acknowledgment

Yueling Zhang is partially supported by the NSFC Projects (Nos. 61572197 and 61632005). Jianwen Li is partially supported by the Shanghai Collaborative Innovation Center of Trustworthy Software for Internet of Things Project (No. ZF1213). Min Zhang is partially supported by the NSFC Project (No. 61672012). Geguang Pu is partially supported by the MOST NKTSP Project (No. 2015BAG19B02) and the STCSM Project (No. 16DZ1100600). Fu Song is partially supported by the NSFC Projects (Nos. 61402179 and 61532019).

References

[1] P. Kilby, J. Slaney, S. Thiébaux, T. Walsh *et al.*, "Backbones and backdoors in satisfiability," in *Proceedings of the Twentieth National Conference on Artificial Intelligence (AAAI-05)*, vol. 5, 2005, pp. 1368–1373.

[2] B. Selman, H. Kautz, B. Cohen *et al.*, "Local search strategies for satisfiability testing," *Cliques, coloring, and satisfiability: Second DIMACS implementation challenge*, vol. 26, pp. 521–532, 1993.

[3] W. Zhang, A. Rangan, and M. Looks, "Backbone guided local search for maximum satisfiability," in *Proceedings of the Nighteenth International Joint Conference on Artificial Intelligence (IJCAI-03)*. Citeseer, 2003, pp. 1179–1186.

[4] A. Montanari, F. Ricci-Tersenghi, and G. Semerjian, "Solving constraint satisfaction problems through belief propagation-guided decimation," *arXiv preprint arXiv:0709.1667*, 2007.

[5] W. Zhang and M. Looks, "A novel local search algorithm for the traveling salesman problem that exploits backbones," in *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI-05)*, 2005, pp. 343–350.

[6] C. S. Zhu, G. Weissenbacher, D. Sethi, and S. Malik, "SAT-based techniques for determining backbones for post-silicon fault localisation," in *IEEE International High Level Design Validation and Test Workshop*, 2011, pp. 84–91.

[7] C. S. Zhu, G. Weissenbacher, and S. Malik, "Post-silicon fault localisation using maximum satisfiability and backbones," in *International Conference on Formal Methods in Computer-Aided Design*, 2011, pp. 63–66.

[8] M. N. Velev, "Formal verification of vliw microprocessors with speculative execution," in *Computer Aided Verification, International Conference, CAV 2000, Chicago, Il, Usa, July 15-19, 2000, Proceedings*, 2000, pp. 296–311.

[9] J. Culberson and I. Gent, "Frozen development in graph coloring," *Theoretical computer science*, vol. 265, no. 1, pp. 227–264, 2001.

[10] J. Berg and M. Jarvisalo, "Cost-optimal constrained correlation clustering via weighted partial maximum satisfiability," *Artificial Intelligence*, 2015.

[11] M. Janota, nês Lynce, and J. Marques-Silva, "Algorithms for computing backbones of propositional formulae," *AI Communications*, vol. 28, no. 2, pp. 161–177, 2015.

[12] M. Janota, "SAT solving in interactive configuration," Ph.D. dissertation, University College Dublin, November 2010.

[13] K. L. McMillan, "Applying SAT methods in unbounded symbolic model checking," in *International Conference on Computer Aided Verification*. Springer, 2002, pp. 250–264.

[14] P. Surynek, "Application of propositional satisfiability to special cases of cooperative path-planning," vol. 2, pp. 507–512, 2012.

[15] J. Rintanen, K. Heljanko, and I. Niemela, "Planning as satisfiability: parallel plans and algorithms for plan search," *Artificial Intelligence*, vol. 170, no. 12, pp. 1031–1080, 2006.

[16] J. Rintanen, "Planning as satisfiability: Heuristics," *Artificial Intelligence*, vol. 193, no. 6, p. 4586, 2012.

[17] M. Alsed and P. Godoy, "Two linear distinguishing attacks on vmpc and rc4a and weakness of rc4 family of stream ciphers," in *International Conference on FAST Software Encryption*, 2005, pp. 342–358.

[18] A. Kaiser and W. Küchlin, "Detecting inadmissible and necessary variables in large propositional formulae," in *University of Siena*. Citeseer, 2001.

[19] O. Dubois and G. Dequen, "A backbone-search heuristic for efficient solving of hard 3-sat formulae," in *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence (IJCAI-01)*, vol. 1, 2001, pp. 248–253.

[20] S. Climer and W. Zhang, "Searching for backbones and fat: A limit-crossing approach with applications," in *Proceedings of the Eighteenth National Conference on Artificial Intelligence (AAAI-02) /Proceedings of the Fourteenth Innovative Applications of Artificial Intelligence Conference on Artificial Intelligence (IAAI-02)*, 2002, pp. 707–712.

[21] J. Marques-Silva, M. Janota, and I. Lynce, "On computing backbones of propositional theories," in *Proceedings of European Association for Artificial Intelligence (ECAI)*, vol. 215, 2010, pp. 15–20.

[22] M. Janota, I. Lynce, and J. Marques-Silva, "Experimental analysis of backbone computation algorithms," in *International Workshop on Experimental Evaluation of Algorithms for solving problems with combinatorial explosion (RCRA)*, 2012, pp. 15–20.