AI Communications 31 (2018) 267–280
DOI 10.3233/AIC-180763
IOS Press

Towards backbone computing: A Greedy-Whitening based approach

Yueling Zhang [a,*], Min Zhang [a], Geguang Pu [a], Fu Song [b] and Jianwen Li [a]
[a] *National Research Center of Trustworthy Embedded Software, East China Normal University, China*
[b] *School of Information Science and Technology, ShanghaiTech University, China*

Abstract. Backbone is the set of literals that are true in all formula's models. Computing a part of backbone efficiently could guide the following searching in SAT solving and accelerate the process, which is widely used in fault localization, product configuration, and formula simplification. Specifically, iterative SAT testings among literals are the most time consumer in backbone computing. We propose a Greedy-Whitening based algorithm in order to accelerate backbone computing. On the one hand, we try to reduce the number of SAT testings as many as possible. On the other hand, we make every inventible SAT testing more efficient. The proposed approach consists of three steps. The first step is a Greedy-based algorithm which computes an under-approximation of non-backbone $\overline{\mathsf{BL}}_{\downarrow}(\Phi)$. Backbone computing is accelerated since SAT testings of literals in $\overline{\mathsf{BL}}_{\downarrow}(\Phi)$ are saved. The second step is a Whitening-based algorithm with two heuristic strategies which computes an approximation set of backbone $\widehat{\mathsf{BL}}(\Phi)$. Backbone computing is accelerated since more backbone are found at an early stage of the computing by testing the literals in $\widehat{\mathsf{BL}}(\Phi)$ first, which makes every individual SAT testing more efficient. The exact backbone is computed in the third step which applies iterative backbone testing on the approximations. We implemented our approach in a tool BONE and conducted experiments on instances from Industrial tracks of SAT Competitions between 2002 and 2016. Empirical results show that BONE is more efficient in industrial and crafted formulas.

Keywords: Backbone, satisfiability, approximation, greedy, whitening

1. Introduction

The *backbone* of a satisfiable formula is a set of literals that are true in all formula's models, which plays an important role for understanding the hardness of problems in computation complexity. For satisfiability problem [20], backbone provides a good explanation for the apparent inevitably high cost of heuristic search near the phase boundary.

The identification of backbone has many practical applications [14,17,18,23]. The seminal work of using backbone information in solving SAT formulas is introduced in [5]. Dubois and Dequen designed an efficient DPL-type algorithms for solving hard random k-SAT formulas, more specifically, 3-SAT formulas. They chose backbone variables as branch nodes for the tree developed by a DPL-type procedure. Experiments show that the performance of handling unsatisfiable hard 3-SAT formulas had improved significantly. Backbone improves the performance of the random SAT solver like WalkSAT [22] by making biased moves in a local search [21,25]. In addition, backbone can significantly contributes to the Lin-Kernighan local search algorithms for Travel Salesman Problem [24]. Another recent successful application of backbone is post-silicon fault localisation in integrated circuits [26,27].

However, deciding whether a literal is a backbone literal is co-NP complete [7,10]. Many heuristic approaches were proposed to compute backbone, such as model enumeration, iterative SAT-testing and filtering with modern SAT solvers. Marques-Silva et al. conducted an experimental evaluation by integration existing algorithms with optimisations in a modern SAT solver and showed that backbone computation for large practical formulae is feasible [11,12,15].

In this paper, we propose a novel Greedy-Whitening based approach BONE for computing backbones. Greedy Algorithm is widely used in optimization application. We use Greedy Algorithm to greedily select the best fit variables at each iteration. Whitening Algorithm [4,8,9] was used to find essential nodes that cannot be colored as white without changing the color of their adjacent nodes in a k-coloring problem, based on

[*]Corresponding author. E-mail: yueling671231@163.com.

a given coloring plan. Getting a new coloring plan is intuitively easier with known essential nodes.

There are two types of backbone computing algorithms, upper bound estimations and literals testing. The upper bound estimations tries to find the exact backbone by removing (adding) literals from the set of all literals (an empty literal set) of the given formula. The literals testing tests the satisfiability of the original formula together with the given literals (as assumptions), backbone information is extracted from the SAT solving process.

The insight of our Greedy-Whitening approach has two-folds: 1) we present a fast procedure to compute an under-approximation set of non-backbone based on Greedy Algorithm, which prunes the search space during the computation of backbone; 2) we also computes an approximation of backbone in polynomial time based on Whitening Algorithm, and the elements in this set have high possibility to be backbone literals which help us to compute the exact backbone using SAT solvers.

We implemented our approach in a tool BONE and evaluated this tool with empirical experiments. We tested 784 industrial and crafted formulae with time limits, the formulae are selected from SAT competitions during 2002 to 2016.[1] 9 more formulas are solved by BONE. For selected industrial formulas, BONE solved 34 from satisfiable industrial formulae in 3600 seconds and reduces 21% total solving time comparing to MINIBONES. Especially, for group *mrpp*, BONE reduces 38% solving time.

2. Preliminaries

We fix a finite set $\mathcal{X}$ of *Boolean variables*. A *literal* l is either a Boolean variable $x \in \mathcal{X}$ or its negation $\neg x$. The negation of a literal $\neg x$ is x, i.e., $\neg\neg x = x$. A *clause* ϕ is a disjunction of literals $\bigvee_{i=1}^{n} l_i$, which may be regarded as the set of literals $\{l_i \mid 1 \leqslant i \leqslant n\}$. W.l.o.g., we assume that for every clause ϕ, if $l \in \phi$, then $\neg l \notin \phi$.

A *formula* Φ over $\mathcal{X}$ is a Boolean combination of variables $\mathcal{X}$. We assume that formulae are given in conjunctive normal form (CNF), namely each formula Φ is a conjunction of clauses $\bigwedge_{i=1}^{n} \phi_i$ which may be regarded as a set of clauses $\{\phi_i \mid 1 \leqslant i \leqslant n\}$. Given a formula Φ, let $\mathsf{var}(\Phi)$ (resp. $\mathcal{L}(\Phi)$ and $\mathsf{cls}(\Phi)$) denote the set of variables (resp. literals and clauses) used in Φ. The *size* $|\Phi|$ of Φ is the number of literals of Φ. We use $\|\Phi\|$ to denote $\sum_{\phi\in\Phi} |\phi|$, and $\neg\mathcal{L}(\Phi)$ to denote the set $\{\neg l \mid l \in \mathcal{L}(\Phi)\}$.

Given a formula Φ and a literal $l \in \mathcal{L}(\Phi)$, let $\Phi_l \subseteq \Phi$ be the set of clauses $\{\phi \in \Phi \mid l \in \phi\}$. Given two different variables $x_1, x_2 \in \mathsf{var}(\Phi)$, x_1 and x_2 are adjacent variables iff there exists a clause $\phi \in \Phi$ such that $(x_1 \in \phi) \wedge (x_2 \in \phi)$.

An *assignment* is a function $\lambda : \mathcal{X} \to \{0, 1\}$, where 1 (resp. 0) denotes true (resp. false). Given an assignment λ and a literal l that is x or $\neg x$, let $\lambda[\neg l]$ be the assignment which is equal to λ except for $\lambda[\neg l](x) = \neg\lambda(x)$. Given a set of variables $x = \{x_1, \dots, x_n\}$, let $\lambda[\neg L]$ denote the assignment $\lambda[\neg x_1]...[\neg x_n]$.

An assignment λ *satisfies* a formula Φ, denoted by $\lambda \models \Phi$, iff assigning $\lambda(x)$ to x for $x \in \mathsf{var}(\Phi)$ makes Φ true.

Definition 1 (Backbone)**.** Given a satisfiable formula Φ, a literal l is a *backbone literal* of Φ iff for all assignments λ such that $\lambda \models \Phi$, $\lambda(l) = 1$. The *backbone* $\mathsf{BL}(\Phi)$ of Φ is the set of backbone literals of Φ.

It is known that the backbone $\mathsf{BL}(\Phi)$ for each formula Φ is unique [5]. The backbone of an unsatisfiable formula can be defined as an empty set. Therefore, in this work, we focus on satisfiable formulae. We will use $\overline{\mathsf{BL}}(\Phi)$ to denote the set $\mathcal{L}(\Phi) \setminus \mathsf{BL}(\Phi)$.

Definition 2. Given an assignment λ of a formula Φ, λ is a model of Φ iff $\lambda \models \Phi$.

Theorem 1 ([7,10])**.** *Given a satisfiable formula Φ and a literal l, deciding whether l is a backbone literal is co-NP complete.*

Definition 3 (Satisfied literal)**.** Given a model λ of the formula Φ and a clause $\phi \in \Phi$, for each literal $l \in \phi$, l is a *satisfied literal* of ϕ iff $\lambda(l) = 1$. l is a *unique satisfied literal* of ϕ if there is no satisfied literal l' of $\phi \setminus \{l\}$.

For instance, let us consider the formula $\Phi = \{\neg l_1 \vee \neg l_2, l_1, l_3 \vee l_4\}$, $\mathsf{var}(\Phi) = \{x_1, x_2, x_3, x_4\}$, $\mathcal{L}(\Phi) = \{\neg l_1, l_1, \neg l_2, l_3, l_4\}$, $\Phi_{\neg l_2} = \{\neg l_1 \vee \neg l_2\}$ and $\mathsf{BL}(\Phi) = \{l_1, \neg l_2\}$. Given a model λ such that $\lambda(l_1) = 1, \lambda(\neg l_2) = 1$. The satisfied literal of clause $(\neg l_1 \vee \neg l_2)$ is $\neg l_2$.

3. Overview of our approach

We show the overview of our approach BONE in Fig. 1. Taking a satisfiable formula Φ as an in-

[1] http://www.satcompetition.org/

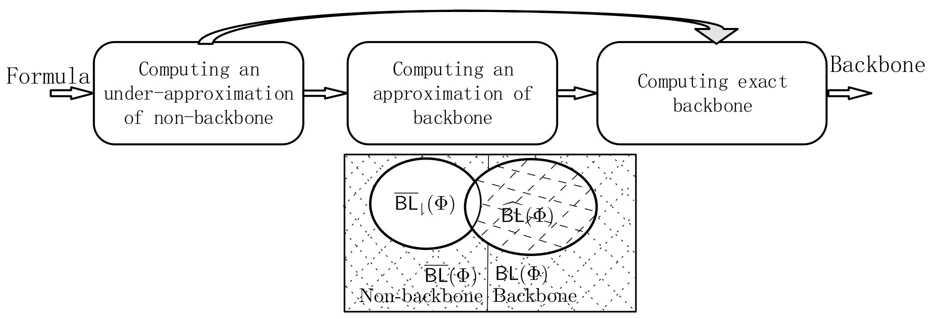

Formula
Computing an under-approximation of non-backbone
Computing an approximation of backbone
Computing exact backbone
Backbone
$\overline{\mathsf{BL}}_{\downarrow}(\Phi)$
$\widehat{\mathsf{BL}}(\Phi)$
$\overline{\mathsf{BL}}(\Phi)$
Non-backbone
$\mathsf{BL}(\Phi)$
Backbone

Fig. 1. Overview of our approach.

put, BONE first computes a subset of non-backbone $\overline{\mathsf{BL}}_{\downarrow}(\Phi) \subseteq \overline{\mathsf{BL}}(\Phi)$. Then, BONE computes an intermediate set of backbone $\widehat{\mathsf{BL}}(\Phi)$ based on the set $\overline{\mathsf{BL}}_{\downarrow}(\Phi)$, where each literal $l \in \widehat{\mathsf{BL}}(\Phi)$ has a high possibility to be a backbone literal of Φ. Finally, BONE removes non-backbone literals from $\widehat{\mathsf{BL}}(\Phi)$ and adds backbone literals into $\widehat{\mathsf{BL}}(\Phi)$ to compute the exact backbone of Φ.

As shown in Fig. 1, $\overline{\mathsf{BL}}_{\downarrow}(\Phi)$ only contains a part of non-backbone literals. Most of the literals in $\widehat{\mathsf{BL}}(\Phi)$ are backbone, only a small part of them is non-backbone. With more known backbone literals, SAT testings are accelerated.

Computing an under-approximation of non-backbone. Given a satisfiable formula Φ, we first compute a model λ of Φ by calling a SAT solver. From the model λ, we compute a base under-approximation of non-backbone. Later, we apply a Greedy-based algorithm to add more non-backbone literals into the base set, which results in $\overline{\mathsf{BL}}_{\downarrow}(\Phi)$.

The algorithm iteratively computes new models until no new model can be found. We choose to change the literal that satisfies the least clauses at each iteration, as the number of clauses effected by this literal is the least one, provided a higher possibility to find a new model. K models will be generated after the k iterations. New non-backbone literals are found from each new model. It's worth noticing that for every literal that has different assignment in all models, we can put them into the under-approximation of non-backbone literals.

Computing an approximation of backbone. At this step, we apply a Whitening-based algorithm to compute an approximation $\widehat{\mathsf{BL}}(\Phi)$. Whitening Algorithm was used to compute *essential nodes* that cannot be colored as white without changing the color of its adjacent nodes in a graph coloring problem.

We consider essential nodes as 'possible' backbone literals in backbone computing. To increase the proportion of backbone literals found by Whitening Algorithm, we use two heuristic strategies to refine Whitening Algorithm. First, we check whether the generated assignment is a model to eliminate some of the non-backbone literals returned by Whitening-based Algorithm. Moreover, we use assumptions features of MINISAT [12] to find some accurate backbone literals. If the conflict size returned by the assumptions features is one, then the negation of the conflict literal is a backbone literal. If the formula is satisfiable under the given assumptions, then we get a new model. By comparing the new model with the old one, we are also able to find non-backbone literals. With the refinement of heuristic strategies, Whitening-based Algorithm is able to return a set of literals that are highly likely to be backbone.

Computing exact backbone. Based on $\widehat{\mathsf{BL}}(\Phi)$, we test whether a literal $l \in \widehat{\mathsf{BL}}(\Phi)$ is a backbone one by one. A naïve but efficient approach [13] is to test the satisfiability of $\Phi \wedge l$, and $\Phi \wedge \neg l$. It's widely used in backbone computing approaches because of its efficiency, therefore, we implemented an iterative SAT testing approach to compute exact backbone based on the idea of this naïve approach. Iterative SAT testing algorithm uses SAT solvers to test whether a literal is a backbone literal or not. For instance, if $\Phi \wedge \neg l$ is unsatisfiable but Φ is satisfiable, then l must be a backbone literal of Φ.

We use the model $\lambda \models \Phi$ and $\widehat{\mathsf{BL}}(\Phi)$, $\overline{\mathsf{BL}}_{\downarrow}(\Phi)$ computed in the previous steps as input. We first iteratively select one literal l from $\widehat{\mathsf{BL}}(\Phi) \setminus \overline{\mathsf{BL}}_{\downarrow}(\Phi)$ such that $\lambda \models \neg l$ and test l by checking the satisfiability of $\Phi \wedge l$ (dashed area in Fig. 1). If l is a backbone, we will add l into Φ as a clause. Adding known backbone literals into Φ as clauses potentially speedups the later SAT testing [5,12,19]. If a new model is generated, we are able to find some non-backbone literals by comparing the difference between the newly generated model and the original one. Then, we do the same testing for literals from $\mathcal{L}(\Phi) \setminus (\overline{\mathsf{BL}}_{\downarrow}(\Phi) \cup \widehat{\mathsf{BL}}(\Phi))$ (dotted area in

Fig. 1). After this step, the exact backbone and non-backbone are found.

Comparing to the approach of directly test all literals in $\mathcal{L}(\Phi)$, there are two contributions of BONE. One is that we reduce the number of SAT calls using the known non-backbone literals $\widehat{\mathsf{BL}}(\Phi)$. The another one is that we first check literals that have high probability to be backbone literals so that backbone literals can be found as early as possible.

4. Greedy-based computing $\overline{\mathsf{BL}}_{\downarrow}(\Phi)$

In this section, we propose an algorithm to compute the under approximation of non-backbone, namely Greedy-based Algorithm. As mentioned in Section 3, Greedy-based Algorithm is a straight forward algorithm which is able, for a given model, to compute parts of non-backbone in quadratic time. It's able to reduce the number of total SAT calls since literals in $\overline{\mathsf{BL}}_{\downarrow}(\Phi)$ don't need SAT testings. Experiments show that Greedy-based Algorithm reduces 5% solving time.

4.1. Computing $\overline{BL}_{\downarrow}(\Phi)$ using one model

Computing $\overline{\mathsf{BL}}_{\downarrow}(\Phi)$ using only one model is equivalent to rotatable literals mentioned in [12]. Given a formula Φ, we first compute a model λ using SAT solver. We then compute the set of non-backbone literals using λ, referred as $L(\Phi, \lambda)$. To compute $L(\Phi, \lambda)$, we first find clauses that have at least two satisfied literals and put them into a set, refereed as $\Phi_{\models_2}$. We then scan all literals in $\mathcal{L}(\Phi)$, and skip literal l and $\neg l$ if l satisfies a clause $\phi \notin \Phi_{\models_2}$. The rest literals are in $L(\Phi, \lambda)$.

The only line in this naïve Algorithm is computing the non-backbone literals set of a given model λ.

Theorem 2. $L(\Phi, \lambda) \subseteq \overline{BL}(\Phi)$.

Proof. Suppose a literal $l \in L(\Phi, \lambda)$. According to the Naïve Algorithm, $\lambda(l) = 1$, and there must exists a literal l' in any clause $\phi \in \Phi_l$, such that $(l' \in \phi) \wedge l('\neq l) \wedge (\lambda(l') = 1)$. Therefore, there must exist a model $\lambda' = \lambda[\neg l]$. It concludes that l is a non-backbone literal of Φ, i.e., $l \in \overline{\mathsf{BL}}(\Phi)$. Given a formula Φ and a model $\lambda \models \Phi$, suppose a literal $l \in L(\Phi, \lambda)$, then for every clause $\phi \in \Phi_l$, there must exists a literal l_2 such that $l \neq l_2$ and $\lambda(l) = 1, \lambda(l_2) = 1$. l_2 is another satisfied literal of ϕ. Therefore, there must exists an assignment $\lambda[\neg l] \models \Phi$, since all clauses contains l will be continue satisfied by another literal in that clause. □

4.2. Computing $\overline{BL}_{\downarrow}(\Phi)$ using Greedy-based algorithm

To save more SAT testings, we propose the Greedy-based algorithm shown in Algorithm 2 to get more non-backbone literals. Comparing to Algorithm 1, Greedy-based algorithms is able to compute more non-backbone literals using more models generated from the original model λ. The rotatable literals in [12] is a subset of the result in Greedy-based Algorithm.

We use *HS* to denote the heuristic strategy of greedy algorithm. In the default Minisat 2.2 setting, the literals

Algorithm 1: Naïve Algorithm: Computing non-backbone literals using λ

```
Input  : a satisfiable formula $\Phi$ and a model $\lambda$ of $\Phi$
Output: a set of literals $\overline{\mathsf{BL}}_{\downarrow}(\Phi)$
1  $\Phi_{\models_2} := \emptyset$;
2  foreach $\phi \in \Phi$ do
3      if $\exists l_1, l_2 \in \phi \wedge l_1 \neq l_2$ then
4          if $\lambda(l_1) = 1 \wedge \lambda(l_2) = 1$ then
5              $\Phi_{\models_2} := \Phi_{\models_2} \cup \{\phi\}$;
6  $L(\Phi, \lambda) := \mathcal{L}(\Phi)$;
7  foreach $l \in L(\Phi, \lambda)$ do
8      foreach $\phi$ s.t. $l \in \phi$ do
9          if $\phi \notin \Phi_{\models_2}$ then
10             $L(\Phi, \lambda) := L(\Phi, \lambda) \setminus \{l\}$;
11             break;
12 return $L(\Phi, \lambda)$;
```

Algorithm 2: Greedy-based algorithm

```
Input  : a satisfiable formula $\Phi$ and a model $\lambda$ of $\Phi$
Output: a set of literals $\overline{\mathsf{BL}}_{\downarrow}(\Phi)$
1  $\overline{\mathsf{BL}}_{\downarrow}(\Phi) := L(\Phi, \lambda)$;
2  $C := \mathsf{HS}(\Phi)$;
3  $i := 0$;
4  while $i < |C|$ do
5      $i := i + 1, l := C[i]$;
6      if $\lambda[\neg l] \models \Phi$ then
7          $\overline{\mathsf{BL}}_{\downarrow}(\Phi) := \overline{\mathsf{BL}}_{\downarrow}(\Phi) \cup L(\Phi, \lambda[\neg l])$;
8          $\overline{\mathsf{BL}}_{\downarrow}(\Phi) := \overline{\mathsf{BL}}_{\downarrow}(\Phi) \cup \{\forall l | l \in \lambda, l' \in \lambda\}$;
9          $\lambda := \lambda[\neg l]$;
10 return $\overline{\mathsf{BL}}_{\downarrow}(\Phi)$;
```

are weighted by the number of appearance in clauses initially, revision are made during the solving process according to the number of appearance in conflicts. In Greedy-based Algorithm, we want to find literals that affect the least of the given formula Φ, therefore, the reverse of weighted literals in Minisat 2.2 solver is the correct order of HS that we wished. C is an order set of literals, resulted from the heuristic strategy HS. When changing the assignment of a literal l, the less clauses l satisfies, the less clauses are affected, the higher possibility that a new model is found.

Given a formula Φ and a model λ, we construct the ordered set of literals C according to HS at Line 2. From Line 4, we start to change the assignment of the literal in C one by one. At Line 6, for each selected literal l, we construct a new assignment $\lambda[\neg l]$ from the model λ and check whether $\lambda[\neg l]$ satisfies Φ in polynomial time. If $\lambda[\neg l]$ satisfies Φ, then we add the set of non-backbone literals $L(\Phi, \lambda[\neg l])$ into $\overline{\mathsf{BL}}_{\lfloor}(\Phi)$, and assign $\lambda[\neg l]$ to λ which will be severed as the model of Φ at the next step. With λ changing at every iteration, we are able to obtain more various assignments and find more non-backbone literals. If λ keeps the same at each iteration, we can only obtain the assignments $\lambda[\neg l], l \in \mathcal{L}(\Phi)$, which is not as various as the assignments by changing model λ at every iteration.

Theorem 3. $\overline{BL}_{\lfloor}(\Phi) \subseteq \overline{BL}(\Phi)$.

Proof. Suppose a literal $l \in \overline{\mathsf{BL}}_{\lfloor}(\Phi)$, if l is added to $\overline{\mathsf{BL}}_{\lfloor}(\Phi)$ at Line 1, then $l \in L(\Phi, \lambda)$, $l \in \overline{\mathsf{BL}}(\Phi)$ (Theorem 2). Otherwise, l is added to $\overline{\mathsf{BL}}_{\lfloor}(\Phi)$ at Line 6, we know that there must exists a model λ', $l \in L(\Phi, \lambda')$ (Line 6). Since $L(\Phi, \lambda') \subseteq \overline{\mathsf{BL}}(\Phi)$ (Theorem 2), $l \in \overline{\mathsf{BL}}(\Phi)$. It concludes that $\overline{\mathsf{BL}}_{\lfloor}(\Phi) \subseteq \overline{\mathsf{BL}}(\Phi)$. $\square$

Theorem 4. *The complexity of Greedy-based Algorithm from Line 2 to Line 10 is $O(m \times n)$, where m is the size of $\mathcal{L}(\Phi)$ and n is the size of clauses in Φ.*

Proof. Given a model $\lambda \models \Phi$ as the input of Greedy-based Algorithm. Started from Line 1, we scan all m clauses to count number of satisfied literals, and scan all n literals to compute $L(\Phi, \lambda)$. The complexity of Line 1 is $O(m \times n)$. With the information collected from Line 1, Line 2 will be finished in $O(m^2)$ time since it needs to sort a set of literals decreasingly according to HS. Line 4 scans all literals, the complexity is $O(m)$. With the information from Line 1, Line 6 will be finished in $O(1)$ time. The complexity of the loop started from 4 is $O(m)$. The total complexity of Greedy-based Algorithm is $O(m \times n)$. $\square$

Greedy-based Algorithm saved 5% solving time in total. With non-backbone literals recognized ahead, SAT testing numbers are reduced. Backbone computing is expediting by the save of SAT testings.

5. Whitening-based computing $\widehat{\mathsf{BL}}(\Phi)$

In this section, we propose an algorithm to compute approximate set of backbone literals $\widehat{\mathsf{BL}}(\Phi)$, namely Whiten-based Algorithm. As mentioned in Section 3, finding more backbone at the earlier stage of the computing provides benefits for the later backbone computing. Experiments show that the proportions of backbone in $\widehat{\mathsf{BL}}(\Phi)$ are generally higher than that in the original formula. Moreover, 20% solving time is saved by Whiten-based Algorithm.

5.1. *Whitening algorithm*

In [9], the authors proposed an Whitening algorithm that computed the essential nodes in a coloring problem named Whitening Algorithm. We propose a Whitening-based Algorithm for 'possible' backbone (essential) literals computing based on the original one. We use W_c to denote the clauses that have at least two satisfied literals to a given model. We use W_v to represent a set of variables, every variable x in W_v only satisfies clauses in W_c. W_c and W_v are updated concurrently.

We first compute a set of clauses that have at least two satisfied literals under the current model, named W_c at Line 1. We find variables that only satisfied clauses in W_c, and put them into a set of variables, named W_v at Line 2. We then start to iteratively extend W_c and W_v from Line 3. For every variable $v \in W_v$, if a clause contains $\neg v$, it will be added to W_c at Line 4. After that, we compute W_v again with the extended W_c. We repeat the procedure until no clause are added to W_c. At last, the complement of W_v is the set of essential variables. It's important that no elements are taken away from W_c, the algorithm will terminate since the number of clauses is finite.

Given a model λ, the complexity of Whitening Algorithm is polynomial since it doesn't need SAT testing. For any variable $x \in W_v$, all clauses in $\Phi_{\neg x}$ will have at least two satisfied literals in the assignment $\lambda[\neg x]$, one is $\neg x$, the other is one of the satisfied literals of ϕ under λ. In this way, Whitening Algorithm extended W_c without SAT testing.

Limited by the only model given to Whitening Algorithm, only a part of backbone variables are in W_v. Different models will have different W_v. For example, given a formula

$$(\neg a \vee b \vee \neg c) \wedge (\neg a \vee \neg b \vee c) \\ \wedge (\neg a \vee b \vee d) \wedge (\neg c \vee d) \wedge (a \vee d)$$

and a model λ such that $\lambda(a) = 1 \wedge \lambda(b) = 1 \wedge \lambda(c) = 1 \wedge \lambda(d) = 1$. At Line 4, W_c is initialized with $\{(\neg a \vee b \vee d), (a \vee d)\}$. W_v is initialized with $\{a\}$. The result of Whitening Algorithm is empty set. It indicates that non of the variable is backbone. Actually, d is a backbone literal, since there does not exist a model λ that $\lambda(\neg d) = 1$.

Given a model λ, the assignment change of a given variable v will generate a new assignment $\lambda[\neg v]$. As we record the assignments generated during the compute step by step, we found that a is a non-backbone variable, because $\lambda[\neg a]$ is another model. b, c are non-backbone variables because $\lambda[\neg a, \neg b]$ and $\lambda[\neg a, \neg c]$ are models of the given formula. However, neither $\lambda[\neg a, \neg b, \neg d]$ nor $\lambda[\neg a, \neg c, \neg d]$ is a model of the given formula.

We consider that an iteration of a repeat loop ended, if the last line of code in the loop body has been executed. In Algorithm 3, we consider that an iteration of the repeat loop ended when Line 5 was executed. A new iteration of the repeat loop started after that if the terminating condition of the loop hasn't been satisfied by then.

The detail iteration of Algorithm 3 using the given example are as follows: At Line 1, $(\neg a \vee b \vee d)$ and $(a \vee d)$ were added to W_c. $W_c = \{(\neg a \vee b \vee d), (a \vee d)\}$. At Line 2, only a is added to W_v. Variable d was not added to W_v since variable d appeared in $(\neg c \vee d)$, which is not in W_c. $W_v = \{a\}$.

In the first iteration of the repeat loop started from Line 3, we added $(\neg a \vee b \vee \neg c)$ and $(\neg a \vee \neg b \vee c)$ to W_c at Line 4. $W_c = \{(\neg a \vee b \vee d), (a \vee d), (\neg a \vee b \vee \neg c), (\neg a \vee \neg b \vee c)\}$. We added b, c to W_v, since all clauses that literal b and literal c appeared were in W_c. $W_v = \{a, b, c\}$. The repeat loop continued at Line 6 since both W_c and W_v were updated.

In the second iteration, $(\neg c \vee d)$ was added to W_c at Line 4, since variable c appears in $(\neg c \vee d)$. $W_c = \{(\neg a \vee b \vee d), (a \vee d), (\neg a \vee b \vee \neg c), (\neg a \vee \neg b \vee c), (a \vee d)\}$. At Line 5, we added variable d into W_v since all clauses that literal d appeared were in W_v. $W_v = \{a, b, c, d\}$. Since both W_v and W_c were updated, the repeat loop continued at Line 6.

In the third iteration, no clause was added to W_c at Line 4, since all clauses were already in W_c. No variable was added to W_v at Line 5, since all variables were already in W_v. The repeat loop exited at Line 6 because neither W_c nor W_v updated. The program returned an empty set at Line 7.

Algorithm 3: Whitening-based algorithm

```
  Input  : a formula $\Phi$ and a model $\lambda$ of $\Phi$
  Output: white clauses $W_c$ and white variables $W_v$
1 $W_c := \{\phi \in \Phi \mid \exists l_1, l_2 \in \phi : \lambda \models l_1 \wedge l_2\}$;
2 $W_v := \{x \in \mathsf{var}(\Phi) \mid \lambda \models x \wedge x \notin \mathcal{L}(\Phi \setminus W_c)$ or
  $\lambda \models \neg x \wedge \neg x \notin \mathcal{L}(\Phi \setminus W_c)\}$;
3 repeat
4     $W_c := W_c \cup \{\phi \in \Phi \mid \mathsf{var}(\phi) \cap W_v \neq \emptyset\}$;
5     $W_v := W_v \cup \{x \in \mathsf{var}(\Phi) \mid \lambda \models x \wedge x \notin$
      $\mathcal{L}(\Phi \setminus W_c)$ or $\lambda \models \neg x \wedge \neg x \notin \mathcal{L}(\Phi \setminus W_c)\}$
6 until No Update of $W_c$ and $W_v$;
7 return $\mathsf{var}(\Phi) \setminus W_v$;
```

5.2. *Assignments checking based Whitening algorithm*

To avoid missing backbone literals like d, we propose a Whitening-check-based Algorithm (WCB for short) to compute the approximation of backbone literals with satisfiability check of each generated assignment. $\mathsf{Pre}(x)$ is used to record the literals that changed its assignment at each iteration. With the changed literals, we are able to generate a new assignment at each iteration. We choose a variable $x \in W_v$ at each iteration, and update W_c by adding $\Phi_{\neg x}$, W_v is extended accordingly. For every new variables x' in W_v, we test the satisfiability of assignment $\lambda[\neg(\mathsf{Pre}(x) \cup \{x, x'\})]$. x' maintains in W_v if a new model is found. Otherwise, x' removes from W_v. Algorithm stops when there is no new clause added to W_c.

We compute W_c and W_v at the first two lines, and extend W_c and W_v at Line 4. At Line 10, we test whether the generated assignment is a model of the given formula Φ. At each iteration, we change the assignment of $x \in W_v$, it results in adding x' to W_v. If the assignment $\lambda[\neg(\mathsf{Pre}(x) \cup \{x, x'\})]$ passes the satisfiability check at Line 10, x' remains in W_v, and $\mathsf{Pre}(x')$ is $\mathsf{Pre}(x) \cup \{x\}$. Otherwise, x' is removed from W_v.

The WCB Algorithm finds some missing backbone literals in Whitening Algorithm. Since the test of satisfiability at Line 10 is in polynomial time, the time complexity remains the same.

Theorem 5. $\forall x \in W_v$, $x \in \overline{BL}_{\downarrow}(\Phi)$.

Proof. Given a formula Φ and a model $\lambda \models \Phi$, suppose a variable $x' \in W_v$. If x' is added to W_v at Line 2, then there must exists a literal $l = x$ or $\neg l = x$, for every clause ϕ that contains l, there must exists another literal l', such that $\lambda(l') = 1$. Therefore, there must exists another model $\lambda' = \lambda[\neg l]$ of Φ, x is a non-backbone literal of Φ. If x' is added to W_v at Line 11, there must exists a variable x, where a new model $\lambda[\neg\mathsf{Pre}(x) \cup \{x, x'\}]$ of Φ is generated at Line 10. Therefore, there exists two different models λ_1 and λ_2 of Φ, such that $\lambda_1(x') = 0$ and $\lambda_2(x') = 1$. x' is a non-backbone literal of Φ. $\square$

We detail the computation of Algorithm 4 on the same example we used above. The formula in the example is

$$(\neg a \vee b \vee \neg c) \wedge (\neg a \vee \neg b \vee c) \\ \wedge (\neg a \vee b \vee d) \wedge (\neg c \vee d) \wedge (a \vee d)$$

W_c was initialized as $\{(\neg a \vee b \vee d), (a \vee d)\}$ at Line 1, W_v was initialized as $\{a\}$ at Line 2. Pre(a) was initialized as empty set at Line 3. In the first iteration of the repeat loop started from Line 4, x was assigned with a at Line 5, all clauses that contains $\neg a$ were added to W_c, $W_c = \{(\neg a \vee b \vee d), (a \vee d), (\neg a \vee b \vee \neg c), (\neg a \vee \neg b \vee c)\}$.

In the first iteration of foreach loop started from Line 5, and the first iteration of the foreach loop started from Line 8, b was assigned to x', since all clauses that contains b were in W_c, we went to the true branch of Line 9. At Line 10, we tested whether $\lambda(\neg a, \neg b) = (\neg a \wedge \neg b \wedge c \wedge d)$ is a model of the given formula, b was added to W_v at Line 11, since $\lambda(\neg a, \neg b)$ is a model of the given formula. Pre(b) was assigned with $\{a\}$ at Line 12.

In the second iteration of the foreach loop started from Line 8, c was assigned to x', since $(\neg a \vee \neg b \vee c)$ was in W_c, which is the only clause containing c, we tested whether $\lambda(\neg a, \neg c) = (\neg a \wedge b \wedge \neg c \wedge d)$ is a model. $\lambda(\neg a, \neg c)$ is a model and we added c to W_v at Line 11, Pre(c) was assigned with a at Line 12.

At the third iteration of the foreach loop started from Line 8, d was assigned to x', since $(\neg c \vee d)$ was not in W_c, the iteration ended at Line 12, the foreach loop stared from Line 8, and the foreach loop started from Line 5 ended.

In the second iteration of the repeat loop, x was assigned to a and b in the first two iterations of the foreach loop started from Line 6, these iterations ended without changing W_c or W_v. x was then assigned to c in the third iteration of the foreach loop started from Line 6, $(\neg c \vee d)$ was added to W_c after Line 7. In the first iteration of the foreach loop started from Line 8, d was assigned to x', since $\lambda(\neg c, \neg d)$ is not a model of the given formula, W_v remains the same, at the foreach loop started from Line 8, and Line 5 ended.

In the third iteration of the repeat loop, x was assigned to a, b, and c in each iterations, W_c and W_v remains the same after all the iterations. There is no update of W_c or W_v, the repeat loop ended. The program returns $\{d\}$ at Line 14.

Algorithm 4: WCB Algorithm with Assignment Satisfiability Checking

```
Input  : a satisfiable formula $\Phi$ and a model $\lambda$ of $\Phi$
Output: a set of literals $\overline{\mathsf{BL}}_{\downarrow}(\Phi)$
1  $W_c := \{\phi \in \Phi \mid \exists l_1, l_2 \in \phi : \lambda \models l_1 \wedge l_2\}$;
2  $W_v := \{x \in \mathsf{var}(\Phi) \mid \lambda(x) = 1 \forall \phi \in \Phi_x : \phi \in W_c\}$;
3  $\forall x \in W_v, \mathsf{Pre}(x) = \emptyset$;
4  repeat
5      foreach $x \in W_v$ do
6          foreach $\phi \in \Phi_{\neg x}$ do
7              $W_c := W_c \cup \phi$;
8          foreach $x' \notin W_v \wedge x' \in \Phi_{\neg x}$ do
9              if $\Phi_{x'} \subseteq W_c$ then
10                 if $\lambda[\neg(Pre(x) \cup \{x, x'\})] \models \Phi$ then
11                     $W_v := W_v \cup x'$;
12                     $\mathsf{Pre}(x') := \mathsf{Pre}(x) \cup x$;
13 until No update of $W_v$;
14 return $\mathsf{var}(\Phi) \setminus W_v$;
```

5.3. Computing parts of exact backbone using assumptions

Although WCB Algorithm is able to compute an approximation set of backbone, it still needs at least one SAT testing to determine whether a literal is backbone, which may need a long solving time. Inspired by [12], we use assumptions in MINISAT as a heuristic strategy to accelerate SAT testing, named Whitening-assumptions-based Algorithm, WAB for short.

Experiments show that given the same formula Φ, SAT testings with assumptions are generally faster then the ones without assumptions. It's because that assumptions help to make the initial decisions of a SAT solving, given an assumption γ with k literals in it, it reduces 2^k states of the searching spaces. SAT testing with a longer assumptions will return faster.

A formula Φ is satisfiable with assumption γ indicates that there exists a model $\lambda \models \Phi$, such that $\forall l \in \gamma, \lambda(l) = 1$. We use $\mathrm{SAT}(\Phi, \gamma)$ to denote a SAT testing of Φ with the assumption of γ. We use (b, λ, r) to denote the result of $\mathrm{SAT}(\Phi, \gamma)$. If b is assigned to 1, a model is returned in λ, otherwise, a reason of unsatisfiable is returned in r. For every literal $l \in \widehat{\mathsf{BL}}(\Phi)$, $\neg l$ is selected to γ. If there is exactly one reason l_b returned from $\mathrm{SAT}(\Phi, \gamma)$, it indicates that $\neg l_b$ must be a backbone of Φ.

Given a model λ and a satisfiable formula Φ. We initialize γ with $\widehat{\mathsf{BL}}(\Phi)$ at Line 2. We add $\neg l$ to γ for every $l \in \widehat{\mathsf{BL}}(\Phi)$ to block the known model λ. At Line 4, we compute the result of $\mathrm{SAT}(\Phi, \gamma)$. All literals in γ will be removed from $\widehat{\mathsf{BL}}(\Phi)$ at Line 11 if b is assigned to 1. Otherwise, all literals in r are removed from γ at Line 9. Once γ is empty, the iteration stops. If there is exactly one literal l_b in r, $\neg l_b$ must be backbone and added to $\mathsf{BL}_{\lfloor}(\Phi)$ at Line 8.

Theorem 6. *Given a satisfiable formula Φ and a set of assumptions γ, $\neg l_b \in \mathsf{BL}(\Phi)$ if $l_b \in \gamma$ and l_b is the only reason that Φ is not satisfiable under the assumption of γ.*

Proof. Given a satisfiable formula Φ and a set of assumptions γ, suppose literal l_b is the only reason returned r from $\mathrm{SAT}(\Phi, \gamma)$, i.e., $\forall l \in r, l = l_b$. It means that there doesn't exist a model $\lambda \models \Phi$, such that $\lambda(l_b) = 1$. Therefore, $\neg l_b$ is a backbone literal of Φ, i.e., $\neg l \in \mathsf{BL}(\Phi)$. $\square$

WAB Algorithm (Algorithm 5) is able to compute the approximation of backbones of the given formula. When the length of unsatisfiable reason is 1, WAB saved solving time when determining if a variable is a backbone. Experiments show that, SAT testings with assumptions are generally finished within 1 second, while original SAT testing may take more than 1 minute.

Compared with the original Whitening Algorithm, the accuracy of Whitening-based Algorithm improved by two heuristic strategies (WCB and WAB) has increased. Missing backbone are added to the approximation through WCB and accurate backbone are confirmed with WAB. With a higher accuracy, backbone computing are guided better with $\widehat{\mathsf{BL}}(\Phi)$.

Algorithm 5: WAB Algorithm for computing under-approximation of backbone $\mathsf{BL}_{\lfloor}(\Phi)$

```
Input  : a satisfiable formula $\Phi$ and $\widehat{\mathsf{BL}}(\Phi)$
Output: under-approximation of backbone
         literals $\mathsf{BL}_{\lfloor}(\Phi)$
1  $\mathsf{BL}_{\lfloor}(\Phi) := \emptyset$;
2  $\gamma := \{l \in \mathcal{L}(\Phi) \mid \neg l \in \widehat{\mathsf{BL}}(\Phi)\}$;
3  repeat
4      $(b, \lambda', r) := \mathrm{SAT}(\Phi, \gamma)$;
5      if $b == 0$ then
6          if $|r| == 1$ then
7              $l_b := r_0$;
8              $\mathsf{BL}_{\lfloor}(\Phi) := \mathsf{BL}_{\lfloor}(\Phi) \cup \{\neg l_b\}$;
9          $\gamma = \gamma \setminus r$;
10     if $b == 1$ then
11         $\widehat{\mathsf{BL}}(\Phi) := \widehat{\mathsf{BL}}(\Phi) \setminus \gamma$;
12         $\overline{\mathsf{BL}}_{\lfloor}(\Phi) := \overline{\mathsf{BL}}_{\lfloor}(\Phi) \cup \gamma$;
13         break;
14 until $\gamma == \emptyset$;
15 return $\mathsf{BL}_{\lfloor}(\Phi)$;
```

In BONE, we first compute the under-approximation of non-backbone literals $\overline{\mathsf{BL}}_{\lfloor}(\Phi)$, using Greedy-based Algorithm. Then we compute an over-approximation $\widehat{\mathsf{BL}}(\Phi)$ of backbone literals using WCB, we use WAB to find exact backbone and non-backbone literals in the $\widehat{\mathsf{BL}}(\Phi)$. The scheme graph is shown in Fig. 2.

6. Experimental study

To study the performance of BONE, we compare it to state-of-the-art tool MINIBONES [12] and analysis total solving time for different groups of formulae. Different algorithms are implemented in MINIBONES. We choose the best algorithms of MINIBONES recommended by the author in [12], namely, *CoreBased* algorithm with a chunk size of 100.

We implemented BONE in C++ interfacing MINISAT 2.2 [6].The experiments were conducted on a cluster of IBM iDataPlex 2.83 GHz, each industrial formula was running with a memory limit of 4 GB. Each random formula was running with a memory limit of 256 MB.

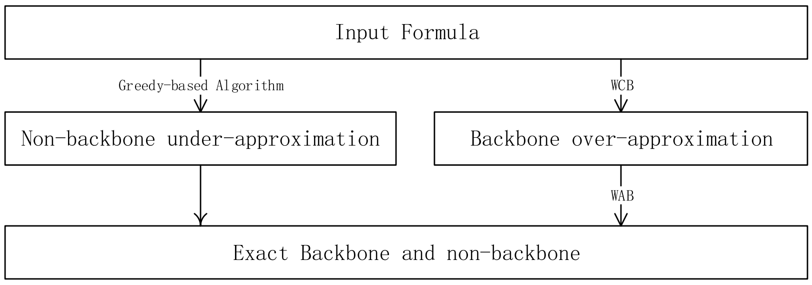

Input Formula
Greedy-based Algorithm
WCB
Non-backbone under-approximation
Backbone over-approximation
WAB
Exact Backbone and non-backbone

Fig. 2. Scheme Graph of BONE.

In the experiments, we separate formulae into groups based on different applications. We study the performance of BONE in 3 different groups, under different time limits. In 3600 seconds, results show that BONE saved 21% solving time in total and performs the best in *manthey* group, saving 34% of solving time. In 16000 seconds, results show that BONE solved 2 more formulae (49 formulae) than MINIBONES does.

Experiments show that both BONE and MINIBONES are good at solving industrial formulae which have clear partitions of variables. In the experiments of random formulae, results indicate that BONE is good at solving formula with more variables that have over 10 adjacent variables.

6.1. Benchmark setup

We select 784 crafted and industrial formulas from SAT competitions between 2002 and 2016. 291 formulas can't find a model using Minisat 2.2 in 3600 seconds. 196 formulas are unsatisfiable formulas, which do not have any backbone.

To study the performance difference between MINIBONES and BONE under different formula group, we choose three industrial groups: *mrpp*, *manthey* and *dimacs*. *mrpp* comes from multi robot path planning problems, *manthey* comes from finding gray codes problems, these codes might attack encryptions, and *dimacs* comes from graph problems. These groups are neither nor too hard. Both BONE and MINIBONES need more than 5 seconds to solve the formulas in these groups. In every group, 2/3 formulas can be solved by Minisat 2.2 in 3600 seconds. Both BONE and MINIBONES use Minisat 2.2 as SAT solving, solving by Minisat 2.2 is the basis of finishing finding backbone literals.

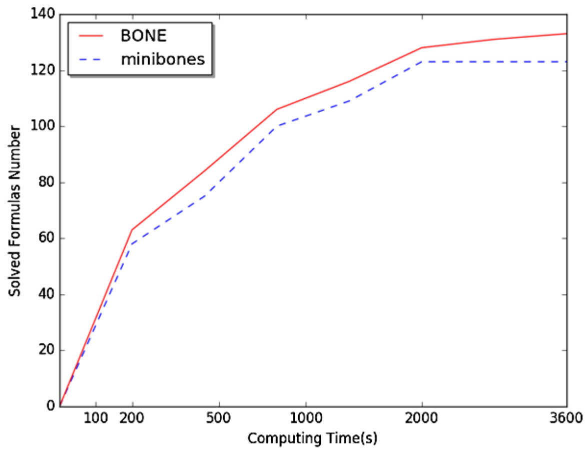


Fig. 3. Numbers of Solving Formulas of MINIBONES and BONE.

6.2. Means of presentation

We use **st** to denote the solving time and **sc** to represent SAT testings number. Comparisons of solving time among individual formula are shown as plots figures. In the figures, each line represents a tool performance of the formulae. The x-axis stands for the individual formula and the y-axis stands for the solving time of the corresponding formula.

6.3. Experimental results on industrial formulae

BONE solved 132 formulas in 3600 seconds, while MINIBONES solved 123 formulas. All formulas solved by MINIBONES are also solved by BONE. BONE solved 9 more formulas than MINIBONES. Figure 3 shows the number of solved formulas of MINIBONES and BONE. The x-axis is solving time (seconds), the y-axis is the number of formulas that MINIBONES and BONE can solved under this time limitation. The red line stands for the solving formulas number of BONE and the blue dotted line stands for the solving formulas number of MINIBONES. As we can observe, MINI-

Table 1

Total Solving Time and SAT Testing Number of MINIBONES and BONE

Tool	**st** (s)	**sc**
BONE	35779	1860897
MINIBONES	38497	3246679

Table 2

Solving Time Comparison on Industrial Formulae

Benchmark	**st** of BONE(s)	**st** of MINIBONES (s)	**st** Difference
mrpp	6112	6900	**11%**
manthey	4845	7363	**34%**
dimacs	1339	1369	**2%**
total	12296	15632	**21%**

Table 3

Number of SAT Testings Comparison on Industrial Formulae

Benchmark	**sc** of BONE(s)	**sc** of MINIBONES (s)	**sc** Difference
mrpp	28261	28839	**3%**
manthey	49356	72943	**33%**
dimacs	2018	2042	**1%**
total	79635	103224	**23%**

BONES solved less formulas than BONE under a longer time limitation, it indicates that BONE has a better scalability than MINIBONES when the formulas requires a longer time to solve.

Consider the formulas that are solved by both MINIBONES and BONE, the total SAT testing number and solving time are listed in Table 1. BONE saved 7% solving time than MINIBONES. BONE saved 43% SAT testing numbers than MINIBONES. It indicates that BONE saved solving time by saving SAT testing.

To study the performance difference in different problems, we analyze the result of formulas in *mrpp*, *manthey*, *dimacs* groups. Among the 72 industrial formulae, both BONE and MINIBONES are able to solve 34 of them in 3600 seconds (1 hour). If more solving time is provided, BONE solved 49 formulae in 16000 seconds (almost 5 hours) while MINIBONES solved 47 formulae. The details of solving time and SAT tests number comparison of each group are shown in Table 2, and Table 3 (with 1 hour time limits). The **st** of Difference is the one minus division between **st** of BONE and **st** of MINIBONES, indicating how much solving time has been saved by BONE.

Solving time and SAT testings number of these three groups are listed in Table 2, and Table 3. We can observe that BONE performs the best on *manthey* group, saved 34% solving time, with 33% saving on SAT testings. In total, BONE saved 21% solving time and 23% SAT testings. BONE saved 11% solving time and 3% SAT testings in *mrpp* group, while saving 2% time and 1% SAT testings in *dimacs* group. It proves the observation in [12] from experiment concepts that less SAT testing will lead to a faster solving.

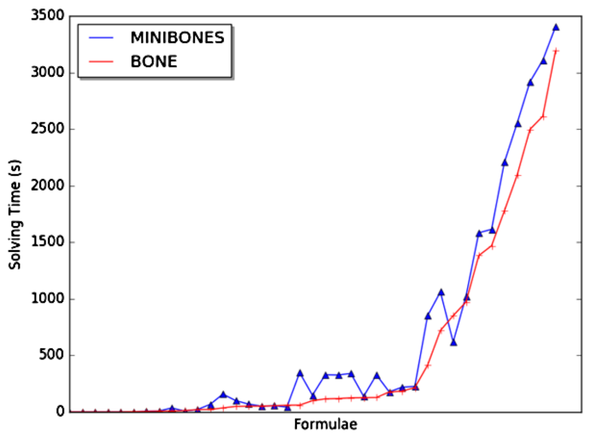


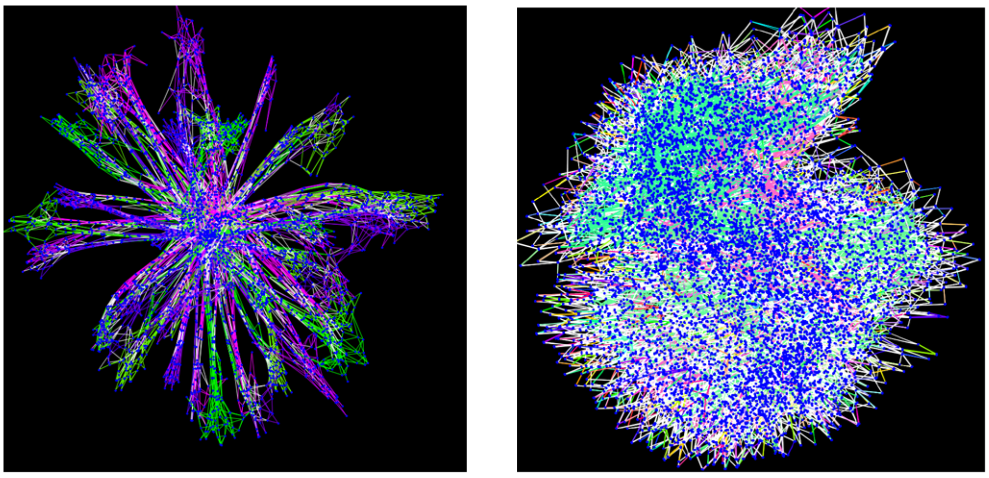
Fig. 4. Solving Time Comparison on Industrial Formulae.

Figure 4 shows the solving time of all 34 industrial formulae. The x-axis represented the id of the formula, the y-axis represented the solving time (seconds). The blue line shows the solving time of MINIBONES, each small triangle represents the solving time of the corresponding formula. While the red line represented the solving time of BONE, each cross represents the solving time of the corresponding formula using BONE. As we can observe, there are only 3 formulae that MINIBONES outperforms BONE.

When we take a look at the formulae in *manthey* group that BONE performs the best, we find that they all have a star-like adjacent structure as shown in Fig. 5(a). It means that the variables is separate into different partitions, different variables from different partitions only appears in a few clauses at the same time. Each branch in the graph stands for a partition.

Figure 5(b) is the adjacent structure of formulae that can't be solved in 16000 seconds using both tools. It's obvious that there is no clear partition among the variables.

For industrial formulae, BONE performs better than MINIBONES in general. We also found that the performance of BONE and MINIBONES are related to the adjacent structure of the given formula. A formula with clear partitions of variables tend to performs better on both BONE and MINIBONES.



(a) Formulae with clear variables partition (b) Formulae with a tightly co-related variables

Fig. 5. Figure (a) shows the star-like adjacent structure, indicates that variables separate into individual partitions, both BONE and MINIBONES are able to solve these formulae in 3600 seconds, BONE saved 34% solving time in total. Figure (b) shows the adjacent structure with a core and no branches, indicates that variables are tightly co-related in the formula. These formulae are more complex than the formula in 5(a), BONE solved 2 more formulae than MINIBONES among these formulae, within 16000 seconds.

6.4. *Result analysis*

Iterative SAT testings, which will hold the assignment of a literal l using assumptions, take the major part of solving time in BONE. Different solving trees will be generated when choosing different l. The SAT testing will be faster if a SAT solving tree is able to find conflicts earlier. Conflicts will be found earlier if the chosen literal l is a backbone literal and mainly appears in a small part of the clauses in the formula, due to the reducing of searching clauses. Therefore, given a formula Φ in *manthey* group, iterative SAT testing of a backbone literal $l \in \mathsf{BL}(\Phi)$ will be faster since Φ has a clear partitions of variables and each backbone literal mainly appears in a subset of $\mathsf{cls}(\Phi)$. In this way, computing $\widehat{\mathsf{BL}}(\Phi)$ in BONE performs better on formulae in *manthey* group. There is no big difference of $\overline{\mathsf{BL}}_{\lfloor}(\Phi)$ performance among groups since the computing time that $\overline{\mathsf{BL}}_{\lfloor}(\Phi)$ is only related to the numbers of clauses and variables of the formula.

Intuitively, given a formula Φ a model $\lambda \models \Phi$, a literal l, $\lambda(l) = 1$, l can be determined using a unique SAT testing $\neg l \wedge \Phi$. In our Greedy-based, WAB Algorithm, and MINIBONES Algorithm, a non-backbone (backbone) literal l could be determined without using $\neg l \wedge \Phi$. We calculate the number of literals that determined as non-backbone (backbone) literals without using unique SAT testing in BONE and MINIBONES. The number comparison of these literals in BONE and MINIBONES are listed in Table 4. We use **pl** to present these literals. We say that a literal is pruned by BONE or MINIBONES, if no unique SAT testing are needed in determining the literal. The difference row presents how much more literals that BONE has pruned than MINIBONES.

Table 4

Number of Literals Determined without Unique SAT Testing in MINIBONES and BONE

Benchmark	Average Literals Number	**pl** of MINIBONES	**pl** of BONE	**pl** Difference
mrpp	6163	858	782	**9%**
manthey	2457	1739	1638	**6%**
dimacs	5087	638	276	**57%**
total	13707	3235	2696	**17%**

7. Related work

There are basically two types of backbone computing algorithms, model enumeration and literals testing. Model enumeration tries to find every model of the given formula, it uses SAT solvers as oracles. Zhu et al. proposed an iterative SAT testing based algorithm [26,27] and applied to fault localisation, experimental results demonstrate that the use of backbone reduced the size of searching space of it, such as sliding windows and trace buffer. In this algorithm, an upper bound of backbone estimation is refined by enumerating the models of the given formula. BONE computes the non-backbone subset at the first step, which is the complement of upper bound of backbone estimation.

Literals testing decide whether a set of literals are backbone based on the procedure of SAT solving. Kaiser and Kühlin proposed algorithms for computing backbones [13] using SAT solvers. It tries to reduce SAT testings by reusing the previous SAT testing results. Since it is based on SAT techniques, it is still applicable when BDDs reach their space limits. It have been applied to automotive product data verification. BONE is able to generate more models without SAT testing than this algorithm. Climer et al. proposed a graph-based approach to discover backbones which approximates lower and upper bounds to compute backbones [3]. It is able to consistently find two to three more times backbone than previous work CDT [2]. It's an algorithm using geometry techniques, which is similar to analysing the procedure of SAT solving.

Marques-Silva et al. investigated algorithms for computing backbones emphasizing the integration of existing algorithms which include model enumeration, iterative SAT-testing and filtering with modern SAT solvers, as well as optimisations. They conducted an experimental evaluation of existing techniques and showed that backbone computation for large practical formulae is feasible, with over 70,000 variables and 250,000 more clauses. [11,12,15]. The *CoreBased* Algorithm with a chunk size of 100 is selected and recommend by the author, which is also the configuration of MINIBONES in our experiments.

BONE is a combination of model enumeration and literals testing. Greedy-based Algorithm enumerates a set of models based on a given model λ, the literals that are not always assigned to 1 are added to $\overline{\mathsf{BL}}_{\downarrow}(\Phi)$. Whitening-based Algorithm computes a set of 'possible' literals $\widehat{\mathsf{BL}}(\Phi)$ by analyzing the solving procedure with assumptions, which consists of literals. Through analyzing the unsatisfiable reason of a SAT testing, we are able to find a part of accurate backbone and a set of literals that are more possible to be backbone. Experiments demonstrate that the proportions of backbone in $\widehat{\mathsf{BL}}(\Phi)$ are generally higher than that in the original formula. Moreover, comparing to MINIBONES, with the best configuration selected by the author of it, 15% solving time are saved by testing the literals in $\widehat{\mathsf{BL}}(\Phi)$ first.

For applications, Dubois and Dequen proposed a heuristic search based on backbone information of hard 3-SAT formulae. It's able to solve unsatisfiable formula with more than 700 variables, which yields DPL-type algorithms with a significant performance improvement over the best previous algorithms [5].

Another work similar to Greedy-based Algorithm in BONE is model rotation, introduced in [1,16]. Model rotation is used to compute MUS of an unsatisfiable formula. New transition clause were found by rotating the literals in the previous transition clause. All MUS must contain transition clause, experiments show that model rotation techniques found more than a half transition clauses in MUS computing, saved 75% computing time without increasing the length of MUS.

8. Conclusion and future work

In this paper, we proposed a backbone computing approach named BONE, using Greedy-Whitening based algorithm. First, we computed an under-approximation of non-backbone $\overline{\mathsf{BL}}_{\downarrow}(\Phi)$ using Greedy-based Algorithm which is able, for a given model, to compute a part of non-backbone in quadratic time. Literals in $\overline{\mathsf{BL}}_{\downarrow}(\Phi)$ didn't need an iterative SAT testing which can save of solving time. Next, we computed an approximation of backbone $\widehat{\mathsf{BL}}(\Phi)$ using Whitening-based Algorithm. The elements in $\widehat{\mathsf{BL}}(\Phi)$ would be backbone with high possibility. We iteratively extended the set of clauses W_c and variables W_v accordingly. $\widehat{\mathsf{BL}}(\Phi)$ was the complement of W_v. Experiments showed that the proportions of backbone in $\widehat{\mathsf{BL}}(\Phi)$ were higher than that in the original formula. Finding more backbone earlier will expedite backbone computing. It's because that more known backbone can prune more states in SAT solving. At last, we iteratively confirmed backbone from previous approximations.

We compared BONE with state-of-art tool MINIBONES, on both industrial formulas and crafted formulas. Experimental results for industrial formulae demonstrated that both BONE and MINIBONES were able to solve 34 formulae from 72 formulae in 3600 seconds. Among the three industrial groups, BONE saved 21% solving time in total than MINIBONES does. BONE performed the best in *manthey* group. For every formula in *manthey*, BONE needed less solving time than MINIBONES does. BONE solved 49 formulae while MINIBONES solved 47 formula when the time limit was 16000 seconds. In general, BONE performs better on formulae with clear variables partitions.

There were two major strategies used in Greedy-Whitening Algorithm, experiments showed that they performed differently on different benchmarks when applied independently, it opens a possibility for portfolio approach. How to decide which strategy to use on a given benchmark is the most important part portfolio approach.

In the future, we will try to apply the whitening algorithm to other existing backbone computing approaches. Whitening algorithm is an approximation algorithm which can find a more exact upper bound of backbone literals, it can be applied as the preprocessing method for other backbone computing algorithms. Another future work is to explore the use of satisfying partial assignment in backbone computing. A satisfying partial assignment, no clause in the formula will be assigned to false under the satisfying partial assignment. Backbone computing efficiency may be improved since the computing of satisfying partial assignment is in polynomial time, while the compute of a model is a NP-hard problem.

Acknowledgements

This work is supported the by the program of Theoretical and Practical Research about Linear Time Logic (LTL) Satisfiability (61572197) from National Natural Science Foundation of China (NSFC).

References

[1] A. Belov and J. Marques-Silva, Accelerating MUS extraction with recursive model rotation, in: *Formal Methods in Computer-Aided Design (FMCAD)*, 2011.

[2] G. Carpaneto, M. Dell'Amico and P. Toth, Exact solution of large-scale, asymmetric traveling salesman problems, *ACM Transactions on Mathematical Software (TOMS)* (1995).

[3] 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.

[4] J. Culberson and I.P. Gent, Frozen development in graph coloring, *Theoretical Computer Science* (2001).

[5] 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)*, 2001.

[6] N. Eén and N. Sförensson, An extensible SAT-solver, in: *International Conference on Theory and Applications of Satisfiability Testing*, 2003.

[7] M.R. Garey and D.S. Johnson, *A Guide to the Theory of NP-Completeness*, WH Freemann, New York, 1979.

[8] I.P. Gent and T. Walsh, The SAT phase transition, in: *Proceedings of European Association for Artificial Intelligence (ECAI)*, 1994.

[9] P. Giorgio, On local equilibrium equations for clustering states, Technique Report cs.cc/0212047, 2003.

[10] M. Janota, SAT solving in interactive configuration, 2010.

[11] 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.

[12] M. Janota, I. Lynce and J. Marques-Silva, Algorithms for computing backbones of propositional formulae, *AI Communications* (2015).

[13] A. Kaiser and W. Kühlin, Detecting inadmissible and necessary variables in large propositional formulae, University of Siena, 2001.

[14] P. Kilby, J. Slaney and T. Walsh, The backbone of the travelling salesperson, in: *Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence (IJCAI-05)*, 2005.

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

[16] J. Marques-Silva and I. Lynce, On improving MUS extraction algorithms, in: *International Conference on Theory and Applications of Satisfiability Testing*, 2011.

[17] M.E.B. Menaı, A two-phase backbone-based search heuristic for partial MAX-SAT – An initial investigation, in: *International Conference on Industrial, Engineering and Other Applications of Applied Intelligent Systems*, 2005.

[18] M.E.B. Menaı, A backbone-based co-evolutionary heuristic for partial MAX-SAT, in: *International Conference on Artificial Evolution (Evolution Artificielle)*, 2005.

[19] C. Mencıa, A. Previti and J. Marques-Silva, Literal-based MCS extraction, in: *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence (IJCAI-15)*, 2015.

[20] R. Monasson, R. Zecchina, S. Kirkpatrick, B. Selman and L. Troyansky, Determining computational complexity from characteristic phase transitions, *Nature* (1999).

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

[22] B. Selman, H. Kautz and B. Cohen, Local search strategies for satisfiability testing, in: *Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge*, 1993.

[23] T. Walsh and J. Slaney, Backbones in optimization and approximation, in: *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence (IJCAI-01)*, 2001.

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

[25] W. Zhang, A. Rangan and M. Looks, Backbone guided local search for maximum satisfiability, in: *Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence (IJCAI-03)*, 2003.

[26] 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.

[27] 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.