Algorithm for Computing Backbones of Propositional Formulae Based on Solved Backbone Information

Zipeng Wang, Yiping Bao, Junhao Xing, Xinyu Chen and Sihan Liu
College of Computer Science and Technology, Jilin University, Changchun 130012

***Abstract*—The problem of propositional satisfiability (SAT) has found a number of applications in both theoretical and practical computer science. However, in many applications, knowing a formulae's satisfiability alone is not enough to solve problems. Often, some other characteristics of formulae need to be known. In 1997, the definition of backbone occur – a set of variables which are true in any assignment of SAT, which starts the research of solution structure. Gradually, backbone has been recognized having great effect in rapidly solving intelligent planning and automated reasoning problems. First of all, this article reviews three existing algorithms for computing backbone, finding that no algorithm uses solved backbone as information. Based on this idea, we bring up a new algorithm for computing backbone of propositional formulae using solved backbone information. This article utilizes latest SAT competition data on original fastest algorithm (one test per time of 30 chuck) and the new algorithm. According to the experiment results, the new algorithm can have 10% advance on rate at the set of data which contain 80% or more chuck so it has practical application value.**

Keywords- propositional satisfiability; backbone

I. INTRODUCTION

In 1997, Parkes bring up the definition of backbone for the first time: A set of variables which are true in any assignment of SAT problem, which starts the research of solution structure[1].

In 2010, based on iterating a literal per time, J. Marques-Silva proposed the computing backbone algorithm based on the iteration to improve the computational efficiency of backbones[2]. In 2011, based on the method of negating a literal per time then conjuncting it with the original formula to compute the backbone, C. Zhu proposed anti-literal algorithm, further boosting the iterative backbone algorithm[3]. In 2015, based on the idea of block and the core, Mikoláš Janota proposed the backbone algorithm with block and the core, improving the computation rate of backbone[4].

Backbone has a number of applications in both theoretical and practical computer science, especially in random 3-SAT problem, optimization problem, and the maximum satisfactory problem (maxSAT), playing an important role in the backdoor analysis and probability information transfer algorithm. For example, in 2002, Telelis and Stamatopoulons designed a heuristic algorithm based on backbone sampling to solve maxSAT problems[5]. In 2005, Zou Peng proposed an ant colony algorithm guided by approximate backbone, pointing out that the approximate backbone of problem instance can largely reduce the search space without degrading the performance of searching[6]. In the same year, Kilby, who studied the backbone of traveling salesman problem, pointed out that utilizing the information of the backbone can rapidly guide the search of the optimization problems. The algorithm can effectively improve the backbone scale and extend the research range of backbone computational complexity[7]. In 2014, Anton belov proposed utilizing the backbone to verify the redundant clause, thus accelerating the computation speed of the smallest equivalence sub-formula[8].

This paper provides the following contributions: 1) we propose an algorithm which utilizes the solved backbone information to accelerate the computation of new backbones; 2) we present a comparison strategy of the algorithm which easy to understand then apply this article's algorithm and the previous two fastest algorithms to the SAT competition data, finding that our algorithm is 10% slower at the data with 80% backbone component content or lower and is 10% faster at the data with 80% backbone component content or higher. The experiments also verify the previous experiment result: the practical SAT problems mostly have large amount of backbones and the backbone component content can reach to 90%.

This paper is organized as below: the second section introduces some marks that definite the basic properties of the backbone. The third section refers to three newest algorithms for backbone computing. The forth section puts forward a new proposition based on solved backbone information. The fifth section brings up a new backbone algorithm based on the proposition of the forth section. It also lists the experimental results of the new algorithm running on a large number of practical data. These data are derived from the representative practical application and the recent SAT competition. The last section summarizes the whole article.

II. PRELIMINARIES

First, we introduce some definitions: use X to represent a boolean variable. A literal l is a boolean variable or its negation. A clause is a disjunction of some words. Conjunctive normal form (CNF) is the conjunction of some clauses. All the formulas processed in this article are the conjunction normal form, so the formula mentioned in this article is automatically considered as a conjunctive normal form. At any time, a clause can be regarded as a collection of literals, and the formula can be regarded as a set of clauses. For a literal l, we use $\neg l$ to donote its negation, i.e. $\neg(x) = \neg x$, $\neg(\neg x) = x$. For a literal l, we use var(l) to represent its corresponding variable, var(x) = x, var($\neg x$) = x. Analogously, for a clause c, var(c) = {var(l) | $l \in c$}. And for a formula φ, var(φ) = $\cup_{\omega\in\varphi}$ var(ω).

© 2016. The authors - Published by Atlantis Press

A. Assignments, Solutions and Implicants

The assignment v is complete iff any $x \in X$ satisfies $v(x) \in \{0,1\}$. Otherwise, v is a partial assignment. For a literal l, if there is $l = x$, then $v(l) = v(x)$, and if $l = \neg x$, then $v(l) = 1 - v(x)$. Assignment rules are also applicable to the clause and the formula, their values are defined as $v(c) = \max_{l \in c} v(l)$ and $v(\varphi) = \min_{\omega \in \varphi} v(\omega)$. If there is an assignment which satisfies $v(\varphi) = 1$, then the formula φ is considered to be satisfied, or the formula is unsatisfiable.

An assignment satisfies a formula φ iff the formula evaluates to 1 under the assignment, i.e. $v(\varphi) = 1$. Similarly, we also say that an assignment satisfies a literal (clause). We consider that v satisfies a series of literals l if it meets all literals $l \in L$. If an assignment meet the formula φ and it is a complete assignment, the assignment is a solution to the formula, i.e. $v(x) \in \{0, 1\}$ for any $x \in \text{var}(\varphi)$.

Definition 1(implicant). An implicant v of a formula φ is a collection of literals. It does not contain the negation of its elements and any assignment u which satisfied v meets the formula φ.

If there is no other implicant v' of the formula φ included in v, the implicant v is a minimal implicant of the formula.

We can observe that the result of any implicant v of φ intersecting any clause of φ is not empty. Therefore, any collection which includes v is another implicant of φ.

B. Definition and Properties of Backbone

An extensively used definition of backbone is said as follows[9].

Definition 2 (backbone literal) Assuming φ is a satisfiable formula, a literal l is a backbone literal of φ iff for any solution u, $u(l) = 1$.

Definition 3 (backbone) For a satisfiable formula φ, the set of all its backbone literals is its backbone[9].

Example 1: Supposing there is a formula $\varphi = \{\neg x \vee \neg y, x\}$, x and $\neg y$ is the backbone of φ.

Notice that any satisfiable formula φ has a unique backbone and any backbone literal must be the element of that backbone. Therefore, "l is a backbone literal of φ" and "l is included in the backbone of φ" will be equivalent. The backbone computation of a formula is called the backbone problem. It is worth noting that this article only pays attention to the backbone of satisfiable formula. But there may be a backbone definition of unsatisfiable formula.

Lemma 1 (backbone literal and implicants). Let φ be a satisfable formula, l a literal, and v an implicant of φ. If l is a backbone literal of φ, then l must belong to v. Conversely, if l does not belong to v, then l is not a backbone literal.

Lemma 2 (backbone and implicant cover) Φ is a satisfiable formula and l is a literal. Assuming that a collection of implicant can make any solution of formula meets at least one of these implicants. The literal l is a backbone literal if and only if l appears in all the implicants.

Lemma 2 gives us the idea of computing backbone, which keeps calculating the new implicants of the formula Φ until these implicants cover the entire formula, and finally computes the intersection of all the implicants.

Lemma 3. Let φ be a satisfiable formula and $x \in \text{var}(\varphi)$. Construct two formula $\varphi_P = \varphi \cup \{x\}$ and $\varphi_N = \varphi \cup \{\neg x\}$:

1) If φ_P is satisfiable and φ_N is unsatisfiable, then the literal x is a backbone literal of φ.

2) If φ_P is unsatisfiable and φ_N is satisfiable, then the literal $\neg x$ is a backbone literal of φ.

3) If φ_P and φ_N are both satisfiable, then neither the literal x nor $\neg x$ is a backbone literal of φ.

III. Recent Popular Algorithm

This part introduces a series of algorithms for computing backbone. First, we introduce an algorithm based on implicant enumeration (based on proposition 2). Then we introduce two test per time algorithm based on proposition 3. At last, we introduce one test per time algorithm, which is the improvement of algorithm 2 and the fastest algorithm for computing backbones based on proposition 3.

In order to present algorithm easier, we assume using SAT solver as SAT(Φ), and return a pair of number(out_c, v). The first variable out_c has two possible values: true or false, mapping to satisfiability or disatisfiability of the formula. If Φ is satisfiable, then out_c = true. The second variable v is an implicant of Φ.

The attention is that the SAT solver returns a result of SAT problems and not an implicant of SAT problems. Apparently, any implicant maps to an assignment. This article considers a result of SAT solver as an implicant so we can simplize the presentation and make the algorithm more widespread.

A. Algorithm 1: Enumeration Algorithm[10]

Based on proposition 2, backbone can be computed by intersecting all the implicants of the formula, so we can bring up an algorithm based on implicant enumeration.

At first, Θ represents the upper bound of backbone (all the literal), and after filtering, it saves all the backbone. This algorithm enumerates implicant until it unable to find new implicant. To avoid finding same implicant, this algorithm has a restrict clause w_B. An implicant's restrict clause is defined as $V_{l \in v} l$. In order to prevent implicant v being recomputed, we conjunct restrict clause with original formula. Every time finding an implicant v, we use v to intersect with upper bound Θ to decrease the upper bound. Because we have different implicants in every computing process, we can at least delete one possible result from upper bound each time, so the whole process is limited.

We can find algorithm 1 has an upper bound Θ of backbone, so it can have enlightening effect to the other algorithm.

This algorithm is unsatisfactory because the enumeration's violence property and it works slow in all the test data with different backbone component content.

B. *Algorithm2: Two Test Per Time Algorithm[11]*

The method of implicant enumeration has a clear drawback: the number of implicant has exponent relation with the number of variable in the worst situation, so the process of computing needs lots of time.

The idea of algorithm 2: in every computing process we pick up a variable x and conjunct it with formula Φ as a new formula Φ_1, and conjunct $\neg x$ with formula Φ as another new formula Φ_2.

According to proposition 3, if Φ_1 is unsatisfiable, then $\neg x$ is backbone, if Φ_2 is unsatisfiable then x is backbone, and if Φ_1, Φ_2 are both satisfiable then x and $\neg x$ are not backbone.

Because of the analysis above, we can enumerate every literal in formula and test them to judge whether a literal is backbone so that we can filter out the backbone literal. The number of SAT solver function call is nearly twice of the number of variable, so algorithm 2 is two times faster than algorithm 1.

The effect of algorithm 2 is excellent in 10% backbone data, in the higher backbone component content, however, is not satisfactory. Because we want to solve the problems which has higher backbone component content, so we still need to improve the algorithm.

C. *Algorithm 3: One Test Per Time Algorithm[12]*

We can easily find that if we can get a smaller upper bound instead of all literals, we will solve the problem by filtering one literal per time efficiently. Because all the backbones have to be in every implicant of the formula, one implicant can be considered as the backbone's upper bound. We can have algorithm 3: one test per time, according to the solution above. This algorithm has an upper bound of backbone Θ to save the literals remaining to be tested and has a lower bound Ω to save the literals which are confirmed to be backbones. Θ is initialized as an implicant which SAT solver returns. The lower bound is initialized as the empty set. So this algorithm at least has $|\mathrm{var}(\Phi)|$ backbone solver calls. In every turns in computation process, we judge whether a literal l is a backbone by the call $\mathrm{SAT}(\Phi \cup \{\neg l\})$, if l is backbone, then this formula is unsatisfiable. We add l into the backbone's lower bound Ω. On the opposite, this formula is satisfiable and returns a new implicant v. We intersect original upper bound Θ with v so that we can decrease the backbone's upper bound. We filter at least one literal from the upper bound and delete it every turn. So this algorithm is limited.

Algorithm 3 guarantees the number of loops is at most $|\mathrm{var}(\Phi)|$, so the number of SAT solver calls are at most $|\mathrm{var}(\Phi)|+1$.

This algorithm has great effect in all the data with different backbone component content, and is one of the fastest algorithms now. This article tends to bring up a more efficient algorithm based on algorithm 3.

IV. New Propositions

According to the algorithms above, no algorithm uses the information of solved backbones, but we can find that the backbones that are about to be computed and the solved algorithm has some relations. And the number of backbones is nearly 90% of all the variables in the practical data. How to fully using this kind of information need to be thought.

Proposition 1: Assume existing a literal l which is confirmed as backbone. If existing another literal b in every sentences having l, then the negative of b is backbone.

Proof: if the negative of b is not backbone, then there must existing an assignment of formula φ, in which the assignment of b is true. All the sentences having b is true contributes to all the sentences having l is true. So a can be true or false, which is conflict with the definition that a is backbone. The proposition above is correct.

Through great amount of data testing, we find the optimization situation mentioned in proposition 1 appears less frequently in low backbone component content backbone data than in higher backbone component content backbone data. So it has good effect to SAT problems which has higher backbone component content.

V. Algorithm for Computing Backbones of Propositional Formulae Based on Solved Backbone Information

According to the proposition 1 we can get a more efficient algorithm based on solved backbone information, and we predict that this algorithm has little effect to lower backbone component content data, great effect to higher backbone component content data.

A. *Initialize*

We assume existing a sentence set V_1 mapping to every literal l in the formula. V_1 saves every sentence which contains literal l.

First we initialize, scanning every sentence Γ in the formula, adding sentence Γ into the sentence sets which maps to all the literals in the sentence, in order to make ergodic easier.

B. *Filter backbone*

According to the algorithm 3, first we have a SAT solver call to get an implicant as the upper bound Θ of backbone, and set the lower bound Ω the empty set. Then we start to use algorithm 3 to filter backbone. Every time we compute a backbone literal l, we intersect all the sentences in the sentence set V_1 which maps to the backbone literal l. If we can get another literal except backbone literal l, according to the proposition 1, $\neg l$ must be the backbone. We use intersect operation to delete $\neg l$ from the upper bound and conjunct $\neg l$ with the original sentence to make sure l is true in every assignment in order to decrease the computation amount. We use proposition 1 to test until we cannot find another new backbone. At this time we return back to the backbone filtering part until we get all the backbones.

Algorithm 4.
Input: Satisfiable formula Φ, with variables x
Output: Backbone of Φ, Ω
1. t=Φ
2. while t≠Ø do
3. Γ= pick a clause from t

```
4.    while Γ≠Ø do
5.       d←pick a literal from Γ
6.       Vd←Vd∨Γ
7.       Γ←Γ\d
8.    t←t\Γ
9.  (outc,v)←SAT(Φ)
10. Ω←Ø
11. Θ←v
12. while Θ≠Ø do
13.    Γ←pick a literal from Θ
14.    (outc,v)←SAT(Φ ∪{¬ l})
15.    if outc=false then
16.       Ω←Ω ∪Γ
17.       Θ←Θ\Γ
18.       Φ←Φ ∪{l}
19.       e←Γ
20.       while e≠Ø do
21.          p←pick a literal from e
22.          e←e\p
23.          h←∩c∈Vp c
24.          h←(h\p) ∩ Θ
25.          h'←{cl | l∈h}
26.          if |h'|>0 then
27.             Ω←Ω∪h'
28.             Θ←Θ\h'
29.             Φ←Φ∪{{l}|l∈h'}
30.          e←e∪h'
31.    else
32.       Θ←Θ ∩ v
33. return Ω
```

According to the idea in this article, we predict that the situation in proposition 1 can frequently occur because of large amount of backbones in practical data. Using 100 variables 400 sentences standard data as example, after initializing every literal maps to probably only 2 sentences, and the time for ergodicing the sentence sets mapping to the literals after initializing can be dismissed compared to the time for the SAT solver call, so we think this algorithm is effective.

Our algorithm test uses minisat2.2 as the SAT solver. Our computer has 2.4GHZ CPU, 8GB memory, and 500GB disk. We test every example 10 time and use the average time as the final time (every operation is certain, the repeat is just to decrease the fluctuate error).

Apply two test per time algorithm (algorithm 2), one test per time algorithm (algorithm 3) and solved backbone algorithm (algorithm 4) for 100 3-SAT data examples and get the result as Figure 1 and 2.

VI. SUMMARY

This article is mainly about the rapid computation method for backbone, aiming to get new backbone computation algorithm through finding some propositions about backbone and the backbone property in order to compute backbone faster. This article's contribution is finding one proposition about backbone and proving it, creating new algorithm for backbone computing according to this proposition, proving its practicability by experiments. We use algorithm 3 as the reference algorithm. The experiment indicates that our algorithm is 10% slower than algorithm 3 at the data with 80% backbone component content or lower and is 10% faster than algorithm 3 at the data with 80% backbone component content or higher.

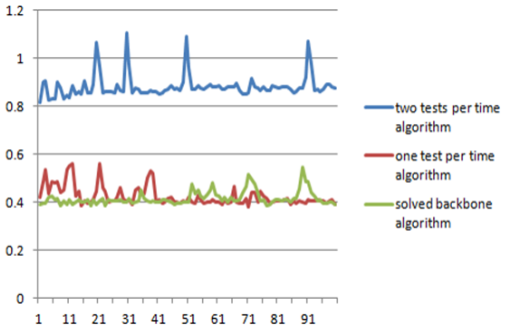

FIGURE I

We can get the solution that the algorithm can have optimization effect only when the data has large amount of backbone, because only in this situation can the formula have many situation which the proposition 1 needs. So when the backbone component content is high, the algorithm 4 can have 10% optimization effect and have application value.

Figure.1 the experiment result of applying two tests per time algorithm, one test per time algorithm and solved backbone algorithm on 90% backbone component content data. (100 instances)

REFERENCES

[1] A. J. Parkes. Clustering at the phase transition[C]. In Proc of the 14th AAAI, 1997, 340-345.

[2] J. Marques-Silva, M. Janota, I. Lynce. On computing backbones of propositional theories[C]. In Proc of the 19th ECAI, 2010, 15-20

[3] C. S. Zhu, G. Weissenbacher, D. Sethi, et al. SAT-based techniques for determining backbones for post-silicon fault localisation[C]. In Proc of HLDVT, 2011, 84-91.

[4] M. Janota, I. Lynce, J. Marques-Silva. Algorithms for computing backbones of propositional formulae[J]. AI Communications, 2015, 28(2): 161-177.

[5] O. Telelis, P. Stamatopoulos. Heuristic backbone sampling for maximum satisfiability. In Proc of the 2nd Hellenic Conference on Artificial Intelligence[C], 2002, 129-139.

[6] Zou Peng, Zhou Zhi, Chen Guo-Liang, Jiang He, Gu Jun. Approximate-Backbone Guided Fast Ant Algorithms to QAP[J]. Journal of Software, 2005, 16(10): 1691-1698.

[7] P. Kilby, J. Slaney, T. Walsh. The backbone of the travelling salesperson[C]. In Proc of the 19th IJCAI, 2005, 175-180.

[8] A. Belov, M. Janota, I. Lynce, et al. Algorithms for computing minimal equivalent subformulas[J]. Artificial Intelligence, 2014, 216: 309-326.

[9] P. Kilby, J.K. Slaney, S. Thiébaux and T. Walsh, Backbones and backdoors in satisfiability. In: AAAI Conference on Artificial Intelligence, 2005, pp. 1368–1373.

[10] Joao Marques-Silva, Mikol´a˘s Janota and In^es Lynce, On Computing Backbones of Propositional Theorie, 2010

[11] J. Marques-Silva, M. Janota and I. Lynce, On computing backbones of propositional theories, in: ECAI, 2010, pp. 15–20.