ECAI 2010
H. Coelho et al. (Eds.)
IOS Press, 2010
© 2010 The authors and IOS Press. All rights reserved.
doi:10.3233/978-1-60750-606-5-15

On Computing Backbones of Propositional Theories

Joao Marques-Silva[1] and **Mikoláš Janota**[2] and **Inês Lynce**[3]

Abstract. Backbones of propositional theories are literals that are true in every model. Backbones have been used for characterizing the hardness of decision and optimization problems. Moreover, backbones find other applications. For example, backbones are often identified during product configuration. Backbones can also improve the efficiency of solving computational problems related with propositional theories. These include model enumeration, minimal model computation and prime implicant computation. This paper investigates algorithms for computing backbones of propositional theories, emphasizing the integration of these algorithms with modern SAT solvers. Experimental results, obtained on representative problem instances, indicate that the proposed algorithms are effective in practice and can be used for computing the backbones of large propositional theories. In addition, the experimental results indicate that propositional theories can have large backbones, often representing a significant percentage of the total number of variables.

1 Introduction

Backbones of a propositional formula φ are literals that take value true in all models of φ [22, 4, 15]. Interest in backbones was originally motivated by the study of phase transitions in Boolean Satisfiability (SAT) problems, where the backbone size was related with search complexity. In addition, backbones have also been studied in random 3-SAT [9] and in optimization problems [8, 27, 16, 28], including Maximum Satisfiability (MaxSAT) [29, 21]. Finally, backbones have been the subject of recent interest, in the analysis of backdoors [11] and in probabilistic message-passing algorithms [12].

Besides the theoretical work, backbones have been studied (often with other names) in practical applications of SAT. One concrete example is SAT-based product configuration [1], where the identification of variables with necessary values has been studied in the recent past [18, 14, 13]. In configuration, the identification of the backbone prevents the user from choosing values that cannot be extended to a model (or configuration). Besides uses in practical applications, backbones provide relevant information that can be used when addressing other decision, enumeration and optimization problems related with propositional theories. Concrete examples include model enumeration, minimal model computation and prime implicant computation, among others.

This paper has three main contributions. First, it develops several algorithms for computing backbones. Some algorithms are based on earlier work [14, 13, 11], whereas others are novel. Moreover, several new techniques are proposed for improving overall performance of backbone computation. Second, the paper evaluates the proposed algorithms in computing the backbone of large practical SAT problem instances, many of which taken from recent SAT competitions. Third, and somewhat surprisingly, the results show that large practical problem instances can contain large backbones, in many cases close to 90% of the variables. In addition, the experimental results show that, by careful implementation of some of the proposed algorithms, it is feasible to compute the backbone of large problem instances.

The paper is organized as follows. Section 2 introduces the notation and definitions used throughout the paper. Section 3 develops two main algorithms for backbone computation, one based on model enumeration and the other based on iterative SAT testing. Also, this section details techniques that are relevant for improving the performance of backbone computation algorithms, and suggests alternative algorithms. Moreover, a number of algorithm configurations are outlined, which are then empirically evaluated. Section 4 analyzes experimental results on large practical instances of SAT, taken from recent SAT competitions[4]. Finally, Section 5 concludes the paper.

2 Preliminaries

A propositional theory (or formula) φ is defined on a set of variables X. φ is represented in conjunctive normal form (CNF), as a conjunction of disjunctions of literals. φ will also be viewed as a set of sets of literals, where each set of literals denotes a clause ω, and a literal is either a variable x or its complement $\bar{x}$. The following definitions are assumed [20]. An assignment ν is a mapping from X to $\{0, u, 1\}$, $\nu : X \rightarrow \{0, u, 1\}$. ν is a *complete* assignment if $\nu(x) \in \{0, 1\}$ for all $x \in X$; otherwise, ν is a *partial* assignment. u is used for variables for which the value is left *unspecified*, with $0 < u < 1$. Given a literal l, $\nu(l) = \nu(x)$ if $l = x$, and $\nu(l) = 1-\nu(x)$ if $l = \bar{x}$. ν is also used to define $\nu(\omega) = \max_{l\in\omega} \nu(l)$ and $\nu(\varphi) = \min_{\omega\in\varphi} \nu(\omega)$. A *satisfying assignment* is an assignment ν for which $\nu(\varphi) = 1$. Given φ, SAT$(\varphi) = 1$ if there exists an assignment ν with $\nu(\varphi) = 1$. Similarly, SAT$(\varphi) = 0$ if for all complete assignments ν, $\nu(\varphi) = 0$. In what follows, *true variables* represent variables assigned value 1 under a given assignment, whereas *false variables* represent variables assigned value 0.

2.1 Models and Implicants

In many settings, a *model* of a propositional theory is interpreted as a satisfying assignment. However, in the remainder of this paper, it is convenient to represent a model as a set of variables M, defined as follows. Given a satisfying assignment ν, for each $x \in X$, add x to M if $\nu(x) = 1$. Hence, models are represented solely with the *true* variables in a satisfying assignment (see for example [6, 19]).

[1] CASL/CSI, University College Dublin, Ireland, email: jpms@ucd.ie
[2] INESC-ID, Lisbon, Portugal, email: mikolas.janota@gmail.com
[3] INESC-ID/IST, TU Lisbon, Portugal, email: ines@sat.inesc-id.pt

[4] `http://www.satcompetition.org/`.

An *implicant* I is defined as a set of literals. Given a satisfying assignment ν, for each $x \in X$, (i) if $\nu(x) = 1$, then include x in I; (ii) if $\nu(x) = 0$, then include $\bar{x}$ in I. This in turn leads to the following definitions.

Definition 1 (Minimal Model) *A model M_1 of φ is* minimal *if there is no other model M_2 of φ such that $M_2 \subsetneq M_1$.*

Minimal models find many applications in artificial intelligence, including knowledge representation and non-monotonic reasoning [2, 6, 17].

Definition 2 (Prime Implicant) *An implicant I_1 of φ is* prime *if there is no other implicant I_2 of φ such that $I_2 \subsetneq I_1$.*

Prime implicants also find many applications in computer science, including knowledge compilation in artificial intelligence and Boolean function minimization in switching theory [24, 6, 17]. Besides a wide range of practical applications, prime implicants and minimal models have also been studied in computational complexity theory. The identification of a minimum-size minimal model is in $\Delta_2^p[log\, n]$ [19]. Minimal models can be computed with algorithms for minimum-cost satisfiability (also referred to as the Binate Covering Problem (BCP)) [5, 19, 10]). Prime implicants can be obtained from computed satisfying assignments. Suppose ν is a satisfying assignment, which can either be complete or partial. For each $\omega \in \varphi$, let $\mathcal{T}(\omega, \nu)$ denote the true literals of ω, and let $\mathcal{T}(\varphi, \nu) = \cup_{\omega\in\varphi}\mathcal{T}(\omega, \nu)$. Moreover, define the following minimum cost satisfiability problem:

$$\begin{array}{ll} \min & \sum_{l\in\mathcal{T}(\varphi,\nu)} l \\ \text{s.t.} & \wedge_{\omega\in\varphi} \left(\vee_{l\in\mathcal{T}(\omega,\nu)}\, l\right) \end{array} \tag{1}$$

The solution to the above set covering problem represents the smallest number of true literals (among the true literals specified by ν) that satisfy the propositional theory. Hence, this solution represents a prime implicant of φ.

Proposition 1 *Given a satisfying assignment ν of a propositional theory φ, the solution of (1) is a prime implicant of φ.*

This result summarizes the main arguments of [25]. Moreover, it is well-known that the computation of prime implicants can be modeled with minimum-cost satisfiability [23].

2.2 Backbones

The most widely used definition of backbone is given in [27] (see [8] for an alternative definition):

Definition 3 (Backbone) *Let φ be a propositional theory, defined on a set of variables X. A variable $x \in X$ is a* backbone variable *of φ if for every model ν of φ, $\nu(x) = v$, with $v \in \{0, 1\}$. Let $l_x = \bar{x}$ if $v = 0$ and $l_x = x$ if $v = 1$. Then l_x is a* backbone literal.

In addition, the computation of the backbone literals of φ is referred to as the *backbone problem*. In the remainder of the paper, backbone variables and backbone literals will be used interchangeably, and the meaning will be clear from the context. Although the focus of this paper are satisfiable instances of SAT, there are different definitions of backbone for the unsatisfiable case [22, 15]. For the algorithms described in this paper, the backbone for unsatisfiable instances is defined to be the empty set.

Furthermore, backbones can be related with the prime implicants of a propositional theory.

Proposition 2 (Backbones and Prime Implicants) *$x \in X$ is a backbone variable of a propositional theory φ if and only if either x or $\bar{x}$ (but not both) occur in all* prime implicants *of φ.*

Following the definition of backbone, a possible solution for computing the backbone of a propositional theory consists in intersecting all of its models. The final result represents the backbone. Propositions 1 and 2 can be used for developing procedures for solving the backbone problem, including: (i) intersection of the prime implicants based on enumeration of satisfying assignments; and (ii) intersection of the prime implicants based on enumeration of the minimal models of a modified propositional theory [23].

Moreover, additional alternative approaches can be devised. Kilby et al. [16] indicate that the backbone problem is NP-equivalent, and that deciding whether a variable is a backbone of a propositional theory is NP-easy, because this can be decided with a SAT test. Clearly, this suggests computing the backbone of a propositional theory with a sequence of SAT tests that grows with $|X|$. Hence, the backbone problem can be solved by a polynomial number of calls to a SAT solver, and so the backbone problem is in Δ_2^P. The basic result can be stated as follows:

Proposition 3 *Let φ be a propositional theory, defined on a set of variables X, and consider the modified theories $\varphi_P = \varphi \cup \{x\}$ and $\varphi_N = \varphi \cup \{\bar{x}\}$. Then one of the following holds:*

1. *If φ_P and φ_N are both unsatisfiable, then φ is also unsatisfiable.*
2. *If φ_P is satisfiable and φ_N is unsatisfiable, then $x \in X$ is a backbone such that φ is satisfiable if and only if $x = 1$ holds.*
3. *If φ_N is satisfiable and φ_P is unsatisfiable, then $x \in X$ is a backbone such that φ is satisfiable if and only if $x = 0$ holds.*
4. *If both φ_N and φ_P are satisfiable, then $x \in X$ is not a backbone.*

Proposition 3 can be used to develop algorithms that compute the backbone of a propositional theory with a number of SAT tests that grows with $|X|$, as suggested for example in [14, 13, 11]. The different approaches outlined in this section for solving the backbone problem are described in more detail in the next section.

3 Computing Backbones

This section develops algorithms for backbone computation. The first algorithm follows the definition of backbone literal. Hence, it enumerates and intersects the satisfying assignments of the propositional theory. As will be shown in Section 4, this algorithm does not scale for large propositional theories. The second algorithm consists of iteratively performing satisfiability tests, considering one or two truth values for each variable. This algorithm follows earlier work [14, 13, 11], and is amenable to a number of optimizations. This section also outlines a number of different algorithm configurations, which will be evaluated in Section 4.

3.1 Model Enumeration

An algorithm for computing the backbone of a propositional theory based on model enumeration is shown in Algorithm 1. The algorithm consists in enumerating the satisfying assignments of a propositional theory. For each satisfying assignment, the backbone estimate is updated. In addition, a *blocking clause* (e.g. [25]) is added to the propositional theory. A blocking clause represents the complement of the computed satisfying assignment, and prevents the same satisfying assignment from being computed again. In order

```
Input  : CNF formula $\varphi$
Output: Backbone of $\varphi$, $\nu_R$
1  $\nu_R \leftarrow \emptyset$
2  repeat
3      $(\text{outc}, \nu) \leftarrow \texttt{SAT}(\varphi)$          // SAT solver call
4      if outc = false then
5          return $\nu_R$  // Terminate if unsatisfiable
6      if $\nu_R = \emptyset$ then
7          $\nu_R \leftarrow \nu$     // Initial backbone estimate
8      else
9          $\nu_R \leftarrow \nu_R \cap \nu$ // Update backbone estimate
10     $\omega_B \leftarrow \texttt{BlockClause}(\nu)$         // Block model
11     $\varphi \leftarrow \varphi \cup \omega_B$
12 until outc $=$ false or $\nu_R = \emptyset$
13 return $\emptyset$
```

Algorithm 1: Enumeration-based backbone computation

```
Input  : CNF formula $\varphi$, with variables $X$
Output: Backbone of $\varphi$, $\nu_R$
1  $\nu_R \leftarrow \emptyset$
2  foreach $x \in X$ do
3      $(\text{outc}_1, \nu) \leftarrow \texttt{SAT}(\varphi \cup \{x\})$
4      $(\text{outc}_0, \nu) \leftarrow \texttt{SAT}(\varphi \cup \{\bar{x}\})$
5      if outc$_1$ $=$ false and outc$_0$ $=$ false then
6          return $\emptyset$
7      if outc$_1$ $=$ false then
8          $\nu_R \leftarrow \nu_R \cup \{\bar{x}\}$            // $\bar{x}$ is backbone
9          $\varphi \leftarrow \varphi \cup \{\bar{x}\}$
10     if outc$_0$ $=$ false then
11         $\nu_R \leftarrow \nu_R \cup \{x\}$            // $x$ is backbone
12         $\varphi \leftarrow \varphi \cup \{x\}$
13 return $\nu_R$
```

Algorithm 2: Iterative algorithm (two tests per variable)

```
Input  : CNF formula $\varphi$, with variables $X$
Output: Backbone of $\varphi$, $\nu_R$
1  $(\text{outc}, \nu) \leftarrow \texttt{SAT}(\varphi)$
2  if outc $=$ false then
3      return $\emptyset$
4  $\nu \leftarrow \texttt{ReduceModel}(\nu)$      // Simplify ref model
5  $\Lambda \leftarrow \{l \mid \bar{l} \in \nu\}$            // SAT tests planned
6  $\nu_R \leftarrow \emptyset$
7  foreach $l \in \Lambda$ do
8      $(\text{outc}, \nu) \leftarrow \texttt{SAT}(\varphi \cup \{l\})$
9      if outc $=$ false then
10         $\nu_R \leftarrow \nu_R \cup \{\bar{l}\}$       // Backbone identified
11         $\varphi \leftarrow \varphi \cup \{\bar{l}\}$
12     else
13         $\nu \leftarrow \texttt{ReduceModel}(\nu)$    // Simplify model
14         foreach $x \in X$ do
15             if $x \notin \nu \wedge \bar{x} \notin \nu$ then
16                 $\Lambda \leftarrow \Lambda - \{x, \bar{x}\}$      // Var filtering
17         foreach $l_\nu \in \nu$ do
18             if $l_\nu \in \Lambda$ then
19                 $\Lambda \leftarrow \Lambda - \{l_\nu\}$        // Var filtering
20 return $\nu_R$
```

Algorithm 3: Iterative algorithm (one test per variable)

to improve the efficiency of the algorithm, the blocking clauses are heuristically minimized using standard techniques, e.g. variable lifting [25]. In addition, a SAT solver with an incremental interface [3] is used. The incremental interface reduces significantly the communication overhead with the SAT solver, and automatically implements clause reuse [20].

It is interesting to observe that Algorithm 1 maintains a superset of the backbone after the first satisfying assignment is computed. Hence, at each iteration of the algorithm, and after the first satisfying assignment is computed, the size of ν_R represents an *upper bound* on the size of the backbone.

3.2 Iterative SAT Testing

The algorithm described in the previous section can be improved upon. As shown in Proposition 3, a variable is a backbone provided exactly one of the satisfiability tests $\text{SAT}(\varphi \cup \{x\})$ and $\text{SAT}(\varphi \cup \{\bar{x}\})$ is unsatisfiable. This observation allows devising Algorithm 2. This algorithm is inspired by earlier solutions [14, 13]. Observe that if a literal is declared a backbone, then it can be added to the CNF formula, as shown in lines 9 and 12; this is expected to simplify the remaining SAT tests. Clearly, the worst case number of SAT tests for Algorithm 2 is $2 \cdot |X|$.

Analysis of Algorithm 2 reveals a number of possible optimizations. First, it is unnecessary to test variable x if there exist at least two satisfying assignments where x takes different values. Also, modern SAT solvers compute complete assignments [20]. Clearly, some variable assignments may be irrelevant for satisfying the CNF formula. More importantly, these irrelevant variable assignments are *not* backbone literals. These observations suggest a different organization, corresponding to Algorithm 3. The first SAT test provides a reference satisfying assignment, from which at most $|X|$ SAT tests are obtained. These $|X|$ SAT tests (denoted by Λ in the pseudo-code) are iteratively executed, and serve to decide which literals are backbones and to reduce the number of SAT tests that remain to be considered. The organization of Algorithm 3 guarantees that it executes at most $|X| + 1$ SAT tests. Besides the reduced number of SAT tests, Algorithm 3 filters from backbone consideration (i) any variable that takes more than one truth value in previous iterations of the algorithm (lines 17 to 19), and (ii) any variable that can be removed from the computed satisfying assignment (lines 14 to 16).

Different techniques can be used for removing variables from computed satisfying assignments. One example is *variable lifting* [25]. Lifting consists of analyzing each variable and discarding the variable if it is not used for satisfying any clause. Another example is (approximate) *set covering* [25]. The set covering model is created by associating with each variable the subset of clauses it satisfies. The goal is then to select a minimal set of variables that satisfies all clauses (see (1) in Section 2.1). Since the set covering problem is NP-hard, approximate solutions are often used. One example is a greedy approximation algorithm for the set covering problem (e.g. [7]). The integration of either of these two techniques is shown in lines 4 and 13.

In contrast to the enumeration-based approach, iterative algorithms refine a subset of the backbone. Hence, at each iteration of the algorithm, the size of ν_R represents a *lower bound* on the size of the backbone. For complex instances of SAT, the enumeration-based and the iteration-based can be used to provide approximate upper and lower bounds on the size of the backbone, respectively.

3.3 Implementation & Configurations

The previous sections outlined two main algorithmic solutions for computing the backbone of a propositional theory. In addition, a number of optimizations was proposed. Nevertheless, in order to achieve the best possible performance, the practical implementation of the algorithms involves essential optimizations. For algorithms that require iterated calls to a SAT solver, a well-known technique is the use of an incremental interface (e.g. [20]). For the results in this paper, the incremental interface of the PicoSAT [3] solver was considered. Nevertheless, an incremental interface is standard in modern SAT solvers [20]. For backbone computation, the incremental interface allows specifying a target assumption (i.e. the value to assign to a variable) in each iteration. As a result, there is no need to recreate the internal data structures of the SAT solver. One additional advantage of using an incremental interface is that clause reuse [20] is implemented by default. Hence, unit clauses from backbones are automatically inferred.

Table 1 summarizes the algorithm configurations to be evaluated in Section 4. *Enumeration* denotes an implementation of Algorithm 1. *Iteration* with 2 tests denotes an implementation of Algorithm 2. *Iteration* with 1 test denotes an implementation of Algorithm 3. *Incremental* denotes implementing repeated SAT tests through an incremental interface. *Variable filtering* represents the elimination of unnecessary SAT tests using the pseudo-code in lines 17 to 19 in Algorithm 3. *Variable lifting* represents the elimination of unnecessary SAT tests obtained by simplifying computed satisfying assignments using standard variable lifting [25]. *Appr set covering* represents the elimination of unnecessary SAT tests obtained by simplifying computed satisfying assignments using an approximation of set covering [25]. These two techniques correspond to calling function ReduceModel in lines 4 and 13 of Algorithm 3, and serve for further elimination of unnecessary SAT tests, as shown in lines 14 to 16 of Algorithm 3. In Table 1, both bb3, bb8, and bb9 correspond to Algorithm 3. The main differences are (i) bb3 does not use the SAT solver's incremental interface, and (ii) the satisfying assignment simplification algorithm used differs.

3.4 Additional Solutions

Besides the algorithms outlined in the previous sections, and which will be evaluated in Section 4, a number of additional algorithms and techniques can be envisioned. A simple technique is to consider k initial SAT tests that implement different branching heuristics, different default truth assignments and different initial random seeds. A similar technique would be to consider local search to list a few initial satisfying assignments, after the first satisfying assignment is computed. Both techniques could allow obtaining satisfying assignments with more variables assuming different values. This would allow set Λ to be further reduced. The experiments in Section 4 indicate that in most cases the number of SAT tests tracks the size of the backbone, and so it was deemed unnecessary to consider multiple initial SAT tests. Another approach consists of executing enumeration and iteration based algorithms in parallel, since enumeration refines upper bounds on the size of the backbone, and iteration refines lower bounds. Such algorithm could terminate as soon as both bounds become equal. The experiments in Section 4 suggest that a fine-tuned iterative algorithm, integrating the techniques outlined above, is a fairly effective solution, and enumeration tends to perform poorly on large practical instances. Finally, as suggested in Section 2.2 and Proposition 2, an alternative algorithm would involve the enumeration of prime implicants, instead of model enumeration. Algorithm 1 could be modified to invoke a procedure for computing prime implicants. However, given the less promising results of model enumeration, prime implicant enumeration is unlikely to outperform the best algorithms described in earlier sections.

Feature	bb1	bb2	bb3	bb4	bb5	bb6	bb7	bb8	bb9
Enumeration	X								
Iteration, 2 tests				X	X				
Iteration, 1 test		X	X			X	X	X	X
Incremental	X			X	X	X	X	X	X
Variable filtering			X		X		X	X	X
Variable lifting	X		X					X	
Appr set covering									X

Table 1. Summary of algorithm configurations

4 Results

The nine algorithm configurations outlined in Section 3.3 were evaluated on representative SAT problem instances. First, a few simple satisfiable instances were taken from standard encodings of *planning* into SAT [26]. These instances provide a baseline for comparing all algorithms. In addition, a few *2dlx* instances were selected from the SAT 2002 competition. Finally, instances from the SAT 2005, 2007 and 2009 competitions were selected. These include instances from the *maris*, *grieu*, *narain*, *ibm* and *aprove* classes of benchmarks. The selected instances are solved by a modern SAT solver in a few seconds (usually less than 20s), to allow computing the backbone in a reasonable time limit. Nevertheless, some of the instances considered have in excess of 70,000 variables, and a few hundred thousand clauses. In total, 97 satisfiable instances were evaluated. All experimental results were obtained on an Intel Xeon 5160 3GHz server, running RedHat Enterprise Linux WS4. The experiments were obtained with a memory limit of 2GB and a time limit of 1,000 seconds. In the results below, TO indicates that the CPU time limit was exceeded. Figure 1 presents a plot by increasing run times of the problem instances for each configuration. The x-axis represents the number of instances solved for a given run time, which is shown in the y-axis (in seconds). In addition, Table 2 presents the results in more detail for a representative subset of the instances. The first column gives the instance name, the second one its number of variables, the third one the percentage of variables which belong to the backbone, and the following ones the CPU time (in seconds) required to run each of the algorithm configurations.

One main conclusion of the experimental results, is that backbone computation for large practical instances is feasible. Some algorithm configurations allow computing the backbone for problem instances with more than 70,000 variables (and more than 250,000 clauses). Another main conclusion is that the size of the backbone for these large problem instances can represent a significant percentage of the number of variables. For some of the large problem instances, the backbone can represent 90% of the variables, and for a few other examples, the backbone can exceed 90%. Moreover, the backbone size is never below 10%. The identification of large backbones on non-random instances agrees with, but significantly extends, earlier

Instance	#vars	%bb	bb1	bb2	bb3	bb4	bb5	bb6	bb7	bb8	bb9
crawford-4blocksb	410	86.3	**0.1**	9.4	8.6	0.6	0.5	0.4	0.4	0.5	0.4
dimacs-hanoi5	1931	100.0	0.6	805.9	800.9	1.8	1.7	**1.5**	**1.5**	**1.5**	**1.5**
selman-f7hh.15	5315	13.2	TO	335.3	62.4	98.9	45.7	54.5	25.2	**11.2**	11.9
selman-facts7hh.13	4315	15.6	TO	165.4	34.7	44.6	22.3	23.6	12.6	**5.4**	**5.4**
2dlx_cc_mc_ex_bp_f2_bug001	4821	36.6	TO	TO	322.4	78.0	21.1	41.4	15.1	14.9	**14.8**
2dlx_cc_mc_ex_bp_f2_bug005	4824	44.7	TO	TO	TO	64.8	25.3	44.4	22.1	**17.9**	18.3
2dlx_cc_mc_ex_bp_f2_bug009	4824	34.8	TO	489.2	290.2	65.6	16.7	35.1	12.3	12.4	**12.1**
maris-sat05-depots3_v01a	1498	82.6	TO	86.1	73.6	7.6	5.7	6.5	**5.4**	**5.4**	5.5
maris-sat05-ferry8_v01i	1745	63.3	TO	TO	TO	40.9	26.5	33.5	**18.8**	19.4	18.9
maris-sat05-rovers5_ks99i	1437	23.7	TO	30.0	15.3	4.9	2.3	3.0	**1.8**	**1.8**	**1.8**
maris-sat05-satellite2_v01i	853	80.1	1.6	18.4	15.7	1.0	0.8	0.7	**0.6**	**0.6**	**0.6**
grieu-vmpc-s05-25	625	100.0	263.6	TO	TO	91.9	92.1	129.9	131.4	**131.1**	139.1
grieu-vmpc-s05-27	729	92.9	TO	TO	TO	**591.2**	602.4	882.2	853.9	859.3	742.2
narain-sat07-clauses-2	75528	89.3	TO	TO	TO	TO	TO	974.4	869.3	868.9	**865.8**
IBM_FV_01_SAT_dat.k20	15069	36.9	TO	TO	TO	526.5	367.1	564.0	**357.6**	379.2	406.5
IBM_FV_02_2_SAT_dat.k20	12088	19.4	TO	TO	203.9	303.1	41.9	158.4	24.1	23.3	**23.0**
IBM_FV_03_SAT_dat.k35	34174	59.8	TO	TO	TO	TO	553.4	931.6	323.7	322.1	**320.8**
IBM_FV_04_SAT_dat.k25	27670	78.4	TO	TO	TO	545.1	317.4	297.4	**163.6**	172.4	175.7
IBM_FV_04_SAT_dat.k30	33855	70.5	TO	TO	TO	898.5	454.2	513.1	224.5	**223.7**	224.7
IBM_FV_06_SAT_dat.k35	42801	50.8	TO	TO	TO	TO	TO	TO	669.3	728.1	**655.4**
IBM_FV_06_SAT_dat.k40	49126	45.0	TO	TO	TO	TO	TO	TO	TO	994.3	**977.9**
IBM_FV_1_02_3_SAT_dat.k20	15775	17.4	TO	TO	TO	566.2	59.7	316.1	43.9	**36.8**	37.0
IBM_FV_1_16_2_SAT_dat.k20	7410	29.7	TO	174.9	56.8	67.1	15.5	34.4	8.6	**8.1**	8.2
IBM_FV_1_16_2_SAT_dat.k50	19110	19.8	TO	TO	373.4	779.5	142.5	408.7	82.1	82.7	**77.3**
IBM_FV_19_SAT_dat.k30	73337	28.9	TO	TO	TO	TO	TO	TO	947.0	684.9	**634.7**
IBM_FV_2_16_2_SAT_dat.k20	7416	29.7	TO	182.0	60.3	35.3	8.7	18.1	**4.9**	**4.9**	**4.9**
IBM_FV_2_16_2_SAT_dat.k50	19116	19.8	TO	TO	378.3	483.5	88.9	242.3	47.6	**47.2**	**47.2**
IBM_FV_3_02_3_SAT_dat.k20	15775	17.5	TO	TO	TO	492.1	38.1	207.2	25.9	**24.4**	**24.4**
IBM_FV_4_16_2_SAT_dat.k20	10371	34.6	TO	395.6	137.4	69.4	15.6	35.7	**9.2**	**9.2**	**9.2**
IBM_FV_4_16_2_SAT_dat.k50	25971	25.1	TO	TO	786.3	952.9	152.8	487.6	83.5	83.5	**83.4**
IBM_FV_5_02_3_SAT_dat.k20	15775	17.5	TO	TO	TO	374.4	38.5	195.5	26.2	21.8	**21.7**
IBM_FV_5_16_2_SAT_dat.k50	25582	25.4	TO	TO	666.6	TO	206.2	669.5	**113.0**	116.1	115.5
AProVE09-03	59231	51.7	TO	TO	TO	TO	TO	TO	**743.3**	779.5	783.1
AProVE09-05	14685	76.3	**41.7**	TO	TO	146.5	72.0	97.2	61.8	61.6	61.6
AProVE09-07	8567	77.4	108.3	TO	TO	147.2	117.7	120.0	**106.2**	108.4	114.3
AProVE09-11	20192	50.5	TO	TO	TO	475.3	102.1	269.7	**79.4**	81.8	81.9
AProVE09-13	7606	64.5	TO	222.1	123.3	33.5	11.9	16.3	8.7	**8.4**	8.5
AProVE09-17	33894	65.4	TO	TO	TO	TO	895.2	TO	839.9	**629.8**	669.9
AProVE09-22	11557	45.5	TO	724.2	295.7	144.7	29.6	75.4	**19.1**	**19.1**	19.2
AProVE09-24	61164	18.0	TO	TO	TO	TO	897.0	TO	687.2	697.3	**648.0**

Table 2. Experimental results for the 9 algorithm configurations

results [11]. It should be emphasized that these large backbones are observed in problem instances originating from well-known practical applications of SAT, including planning (*maris*, and the initial set of benchmarks), formal verification (*2dlx*), model finding (*narain*), model checking (*ibm*), termination in term-rewriting (*aprove*) and cryptanalysis (*grieu*).

In addition, the experimental results allow drawing several general conclusions. With a few exceptions, it can be concluded that the enumeration-based algorithms do not scale for large practical problem instances. Despite the poor results, it should be noted that algorithm bb1 is fairly optimized. For example, blocking clauses are minimized with variable lifting [25], and the SAT solver's incremental interface is used [3], which also provides clause reuse. Iterative algorithms that do not use the incremental SAT solver interface also perform poorly. This is justified by (i) learned clauses are not reused, and (ii) repeated creation of the SAT solver's internal data structures.

The use of a single test per variable, with an additional initial test for computing a reference assignment, is an effective technique that can reduce the run times substantially. Some of the simplification techniques are key for solving larger problem instances. Concrete examples include filtering of variables with complementary values in different models, and recording backbone literals as unit clauses. The simplification of models for additional filtering of variables can be significant for some of the most difficult problem instances. Regarding Table 2, and with the exception of a few outliers, the performance improves (often significantly) with the integration of the techniques proposed in this paper. bb9, bb8 and bb7 are the best algorithms for 20, 18 and 14 instances, respectively. The remaining algorithms combined are the best performing for only 4 instances. Similarly, for Figure 1, out of the test set of 97 instances, bb8 solved 78 instances, closely followed by bb9 and bb7, that solve 76 and 75 instances, respectively.

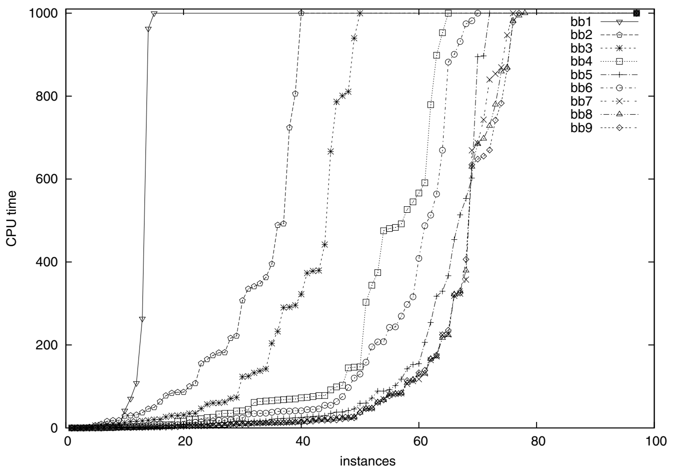

Figure 1. Run times of each algorithm configuration

5 Conclusions

This paper develops algorithms for backbone computation. The algorithms build on earlier work, but integrate new techniques, aiming improved performance. In addition, the paper conducts a comprehensive experimental study of backbones on practical instances of SAT. The experimental results suggest that iterative algorithms, requiring at most one satisfiability test per variable, are the most efficient. However, the best performance requires exploiting the incremental interface of modern SAT solvers, and the implementation of a number of key techniques. These techniques include learning unit clauses from identified backbones, clause reuse, variable filtering due to simplified models, and variables having more than one truth value in satisfying assignments. In addition, the experimental results show that the proposed algorithms allow computing the backbone for large practical instances of SAT, with variables in excess of 70,000 and clauses in excess of 250,000. Furthermore, the experimental results also show that these practical instances of SAT can have large backbones, in some cases representing more than 90% of the number of variables and, in half of the cases, representing more than 40% of the number of variables.

The experimental results confirm that backbone computation is feasible for large practical instances. This conclusion motivates further work on applying backbone information for solving decision and optimization problems related with propositional theories, including model enumeration, minimal model computation and prime implicant computation. Finally, the integration of additional model simplification techniques could yield additional performance gains.

ACKNOWLEDGEMENTS

This work is partially supported by SFI PI grant BEACON (09/IN.1/I2618) and European projects COCONUT (FP7-ICT-217069) and MANCOOSI (FP7-ICT-214898). The work was carried out while the second author was at UCD.

REFERENCES

[1] D. Batory. Feature models, grammars, and propositional formulas. In *International Software Product Line Conference*, pages 7–20, 2005.

[2] R. Ben-Eliyahu and R. Dechter. On computing minimal models. *Annals of Mathematics and Artificial Intelligence*, 18(1):3–27, 1996.

[3] A. Biere. PicoSAT essentials. *Journal on Satisfiability, Boolean Modeling and Computation*, 4(2-4):75–97, 2008.

[4] B. Bollobás, C. Borgs, J. T. Chayes, J. H. Kim, and D. B. Wilson. The scaling window of the 2-SAT transition. *Random Structures and Algorithms*, 18(3):201–256, 2001.

[5] R. K. Brayton and F. Somenzi. An exact minimizer for Boolean relations. In *International Conference on Computer-Aided Design*, pages 316–319, November 1989.

[6] M. Cadoli and F. M. Donini. A survey on knowledge compilation. *AI Communications*, 10(3-4):137–150, 1997.

[7] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms, 2nd Edition*. MIT Press, 2001.

[8] J. C. Culberson and I. P. Gent. Frozen development in graph coloring. *Theor. Comput. Sci.*, 265(1-2):227–264, 2001.

[9] O. Dubois and G. Dequen. A backbone-search heuristic for efficient solving of hard 3-SAT formulae. In *International Joint Conference on Artificial Intelligence*, pages 248–253, 2001.

[10] Z. Fu and S. Malik. Solving the minimum-cost satisfiability problem using SAT based branch-and-bound search. In *International Conference on Computer-Aided Design*, pages 852–859, 2006.

[11] P. Gregory, M. Fox, and D. Long. A new empirical study of weak backdoors. In *International Conference on Principles and Practice of Constraint Programming*, pages 618–623, 2008.

[12] E. I. Hsu, C. J. Muise, J. C. Beck, and S. A. McIlraith. Probabilistically estimating backbones and variable bias: Experimental overview. In *International Conference on Principles and Practice of Constraint Programming*, pages 613–617, 2008.

[13] M. Janota. Do SAT solvers make good configurators? In *Workshop on Analyses of Software Product Lines (ASPL)*, pages 191–195, 2008.

[14] A. Kaiser and W. Küchlin. Detecting inadmissible and necessary variables in large propositional formulae. In *Intl. Joint Conf. on Automated Reasoning (Short Papers)*, June 2001.

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

[16] P. Kilby, J. K. Slaney, and T. Walsh. The backbone of the travelling salesperson. In *International Joint Conference on Artificial Intelligence*, pages 175–180, 2005.

[17] J. Lang, P. Liberatore, and P. Marquis. Propositional independence: Formula-variable independence and forgetting. *Journal of Artificial Intelligence Research*, 18:391–443, 2003.

[18] D. Le Berre. Exploiting the real power of unit propagation lookahead. *Electronic Notes in Discrete Mathematics*, 9:59–80, 2001.

[19] P. Liberatore. Algorithms and experiments on finding minimal models. Technical report, DIS, Univ. Rome, La Sapienza, December 2000.

[20] J. Marques-Silva, I. Lynce, and S. Malik. Conflict-driven clause learning SAT solvers. In A. Biere, M. Heule, H. van Maaren, and T. Walsh, editors, *SAT Handbook*, pages 131–154. IOS Press, 2009.

[21] M. E. Menai. A two-phase backbone-based search heuristic for partial max-sat - an initial investigation. In *Industrial and Engineering Appl. of Artif. Intell. and Expert Systems*, pages 681–684, 2005.

[22] R. Monasson, R. Zecchina, S. Kirkpatrick, B. Selman, and L. Troyansk. Determining computational complexity from characteristic 'phase transitions'. *Nature*, 400:133–137, July 1999.

[23] L. Palopoli, F. Pirri, and C. Pizzuti. Algorithms for selective enumeration of prime implicants. *Artificial Intelligence*, 111(1-2):41–72, 1999.

[24] W. V. Quine. The problem of simplifying truth functions. *American Mathematical Monthly*, 59:521–531, October 1952.

[25] K. Ravi and F. Somenzi. Minimal assignments for bounded model checking. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 31–45, 2004.

[26] B. Selman and H. Kautz. Planning as satisfiability. In *European Conference on Artificial Intelligence*, pages 359–363, 1992.

[27] J. K. Slaney and T. Walsh. Backbones in optimization and approximation. In *International Joint Conference on Artificial Intelligence*, pages 254–259, 2001.

[28] W. Zhang and M. Looks. A novel local search algorithm for the traveling salesman problem that exploits backbones. In *International Joint Conference on Artificial Intelligence*, pages 343–350, 2005.

[29] W. Zhang, A. Rangan, and M. Looks. Backbone guided local search for maximum satisfiability. In *International Joint Conference on Artificial Intelligence*, pages 1179–1186, 2003.