Experimental Analysis of Backbone Computation Algorithms*

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

[1] INESC-ID/IST, Technical University of Lisbon, Portugal
[2] CASL/University College Dublin, Ireland

Abstract. In a number of applications, it is not sufficient to decide whether a given propositional formula is satisfiable or not. Often, one needs to infer specific properties about a formula. This paper focuses on the computation of backbones, which are variables that have the same value in all models of a Boolean formula. Backbones find theoretical applications in characterization of SAT problems but they also find practical applications in product configuration or fault localization. This paper presents an extensive evaluation of existing backbone computation algorithms on a large set of benchmarks. This evaluation shows that it is possible to compute the set of backbones for a large number of instances and it also demonstrates that a large number of backbones appear in practical instances.

1 Introduction

Backbones of a propositional formula φ are literals that take value true in all models of φ [23,4,15]. 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 [8] and in optimization problems [7,27,16,28], including Maximum Satisfiability (MaxSAT) [29,22]. 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 backbones prevents the user from choosing values that cannot be extended to a model (or configuration). A more recent application is post-silicon

* This paper is based on, but significantly extends, a paper presented at ECAI 2010 on the same subject [20].

Proceedings of the 19th RCRA workshop on *Experimental Evaluation of Algorithms for Solving Problems with Combinatorial Explosion* (RCRA 2012).
In conjunction with AI*IA 2012, Rome, Italy, June 14–16, 2012.

fault localization in integrated circuits [31,30]. Besides uses in practical applications, backbones provide relevant information that can be used when addressing other decision, enumeration and optimization problems related to propositional theories. Concrete examples include model enumeration, minimal model computation and prime implicant computation, among others.

This paper has the following main contributions. First, the paper overviews recent algorithms for backbone computation [20,31]. Second, the paper provides a unifying algorithm for backbone computation, and shows that some of the recent algorithms for backbone computation are special cases of this unifying algorithm. Third, the paper describes a comprehensive experimental evaluation of the best backbone computation algorithms. This experimental evaluation is carried out on an extensive set of problem instances from practical applications and past SAT competitions. The experimental results support early data [20] that large practical problem instances can contain large backbones, in many cases close to 90% of the variables. In addition, the experimental results confirm 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 representative practical applications and from recent SAT competitions[3]. 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 [21]. 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, $\text{SAT}(\varphi) = 1$ if there exists an assignment ν with $\nu(\varphi) = 1$. Similarly, $\text{SAT}(\varphi) = 0$ if for all complete assignments ν, $\nu(\varphi) = 0$. In what follows, *true*

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

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]). 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 [25,6,17]. 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:

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

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 [26]. Moreover, it is well-known that the computation of prime implicants can be modeled with minimum-cost satisfiability [24].

2.2 Backbones

The most widely used definition of backbone is given in [27] (see [7] 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 [23,15]. For the algorithms described in this paper, the backbone for unsatisfiable instances is defined to be the empty set.

Furthermore, backbones can be related to 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 [24].

Moreover, additional alternative approaches can be devised. Kilby et al. [16] indicate that the backbone problem is NP-equivalent, and that deciding whether a literal 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.*

Input : CNF formula φ
Output: Backbone of φ, ν_R

```
1  ν_R ← ∅
2  repeat
3  |  (outc, ν) ← SAT(φ)                          // SAT solver call
4  |  if outc = false then
5  |  |  return ν_R                               // Terminate if unsatisfiable
6  |  if ν_R = ∅ then
7  |  |  ν_R ← ν                                  // Initial backbone estimate
8  |  else
9  |  |  ν_R ← ν_R ∩ ν                            // Update backbone estimate
10 |  ω_B ← BlockClause(ν)                        // Block model
11 |  φ ← φ ∪ ω_B
12 until outc = false or ν_R = ∅
13 return ∅
```

Algorithm 1: Enumeration-based backbone computation

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 starts by overviewing algorithms for backbone computation [20,31]. Next, a new unified algorithm is proposed, which exploits the notion of subsets (or *chunks*) of backbone candidates. Two representative algorithms described in this section are shown to be special cases of the unified algorithms.

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. [26]) is added to the propositional theory. A blocking clause represents the complement of the computed satisfying assignment, and prevents the same

Input : CNF formula φ, with variables X
Output: Backbone of φ, ν_R

1 $\nu_R \leftarrow \emptyset$
2 **foreach** $x \in X$ **do**
3 $\quad (\mathsf{outc}_1, \nu) \leftarrow \mathtt{SAT}(\varphi \cup \{x\})$
4 $\quad (\mathsf{outc}_0, \nu) \leftarrow \mathtt{SAT}(\varphi \cup \{\bar{x}\})$
5 $\quad$ **if** $\mathsf{outc}_1 = \mathsf{false}$ **and** $\mathsf{outc}_0 = \mathsf{false}$ **then**
6 $\quad\quad$ **return** $\emptyset$
7 $\quad$ **if** $\mathsf{outc}_1 = \mathsf{false}$ **then**
8 $\quad\quad \nu_R \leftarrow \nu_R \cup \{\bar{x}\}$ `// ` $\bar{x}$ `is backbone`
9 $\quad\quad \varphi \leftarrow \varphi \cup \{\bar{x}\}$
10 $\quad$ **if** $\mathsf{outc}_0 = \mathsf{false}$ **then**
11 $\quad\quad \nu_R \leftarrow \nu_R \cup \{x\}$ `// ` x `is backbone`
12 $\quad\quad \varphi \leftarrow \varphi \cup \{x\}$
13 **return** ν_R

Algorithm 2: Iterative algorithm (two tests per variable)

satisfying assignment from being computed again. In order to improve the efficiency of the algorithm, the blocking clauses are heuristically minimized using standard techniques, e.g. variable lifting [26]. 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 [21].

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 [21]. Clearly, some variable assignments may be irrelevant for satisfying the CNF formula. More importantly, these irrelevant variable assignments

Input : CNF formula φ, with variables X
Output: Backbone of φ, ν_R

```
1  (outc, ν) ← SAT(φ)
2  if outc = false then
3      return ∅
4  Λ ← {l | l̄ ∈ ν}                                  // SAT tests planned
5  ν_R ← ∅
6  foreach l ∈ Λ do
7      (outc, ν) ← SAT(φ ∪ {l})
8      if outc = false then
9          ν_R ← ν_R ∪ {l̄}                          // Backbone identified
10         φ ← φ ∪ {l̄}
11     else
12         foreach x ∈ X do
13             if x ∉ ν ∧ x̄ ∉ ν then
14                 Λ ← Λ − {x, x̄}                    // Var filtering
15         foreach l_ν ∈ ν do
16             if l_ν ∈ Λ then
17                 Λ ← Λ − {l_ν}                     // Var filtering
18 return ν_R
```

Algorithm 3: Iterative algorithm (one test per variable)

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 any variable that takes more than one truth value in previous iterations of the algorithm (lines 15 to 17)

In contrast with 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 approaches can be used to provide approximate upper and lower bounds on the size of the backbone, respectively.

Input : CNF formula φ, with variables X
Output: Backbone of φ, ν_R

```
1  (outc, ν) ← SAT(φ)
2  if outc = false then
3      return ∅
4  ν_R ← ν                                   // Initial backbone estimate
5  ω_N ← {l̄ | l ∈ ν_R}                       // Negate backbone estimate
6  while ω_N ≠ ∅ do
7      (outc, ν) ← SAT(φ ∪ ω_N)
8      if outc = false then
9          return ν_R                        // Terminate if unsatisfiable
       // Instance is sat
10     foreach x ∈ X do
11         if x ∉ ν ∧ x̄ ∉ ν then
12             ν_R ← ν_R − {x, x̄}            // Variable filtering
13     foreach l_ν ∈ ν_R do
14         if l̄_ν ∈ ν then
15             ν_R ← ν_R − {l_ν}             // Refine backbone estimate
16     ω_N ← {l̄ | l ∈ ν_R}                   // Negate backbone estimate
```

Algorithm 4: Iterative algorithm with complement of backbone estimate

3.3 Integrating the Complemented Backbone Estimate

An algorithm that complements the algorithms described in the previous sections was recently proposed in [31]. Although in practice this algorithm is less efficient than the algorithms described in the previous section, namely Algorithm 3, it is guaranteed to require fewer SAT solver calls. Indeed, the algorithm described in [31] is also based on iterative SAT testing, but only a single SAT solver call is required to prove the set of backbone literals. This section studies this algorithm, and proposes optimizations targeting improved efficiency. Algorithm 4 shows the new algorithm developed in [31]. As can be observed, the complement of the backbone estimate is added as a clause to the formula. If the formula is satisfiable, then the computed truth assignment includes at least one satisfied literal in the complement of the backbone estimate. Hence, the backbone estimate is refined (as is its complement). The process is repeated until the backbone estimate represents the actual backbone, in which case the formula is unsatisfiable.

Proposition 4. *Let $|BB|$ denote the backbone size. Then, the number of SAT tests in Algorithm 4 is at most* $(|X| - \max(|BB|, 1) + 1) + 1 \leq |X| + 1$.

Proposition 5. *There is exactly one unsatisfiable SAT test for Algorithm 4. The number of satisfiable SAT tests is at most* $|X| - |BB| \leq |X|$.

Input : CNF formula φ, with variables X; K chunk size
Output: Backbone of φ, ν_R

```
1  (outc, ν) ← SAT(φ)
2  if outc = false then
3      return ∅
4  ν_R ← ν                                          // Initial backbone estimate
5  ω_N ← {l̄ | l ∈ ν_R}                              // Negate backbone estimate
6  while ν_R ≠ ∅ do
7      k ← min(K, |ω_N|)
8      Γ ← pick k literals from ω_N
9      (outc, ν) ← SAT(φ ∪ ⋁_{l∈Γ} l)
10     if outc = false then
11         return ν_R                                // Terminate if unsatisfiable
       // Instance is sat
12     foreach x ∈ X do
13         if x ∉ ν ∧ x̄ ∉ ν_R then
14             ν_R ← ν_R − {x, x̄}                    // Variable filtering
15     foreach l_ν ∈ ν_R do
16         if l̄_ν ∈ ν then
17             ν_R ← ν_R − {l_ν}                     // Refine backbone estimate
18     ω_N ← {l̄ | l ∈ ν_R}                           // Negate backbone estimate
```

Algorithm 5: Chunking algorithm

As observed in [31], Algorithm 4 can perform poorly when compared with the algorithms described in earlier sections. This results from negating the complete backbone estimate, that can result in difficult instances of SAT.

A solution to the problem of negating the complete backbone estimate is to iteratively negate and analyze subsets of the backbone estimate. This process consists of splitting the backbone estimate into *chunks* of some size K as presented in Algorithm 5. The algorithm has the same structure as Algorithm 4 but instead of adding a clause of the size of the whole backbone estimate, a clause of the size K is added. The intuition behind this clause is "show that at least one of the respective literals is not a backbone."

Interestingly, the use of chunks covers both Algorithm 4, when a single chunk is used, and Algorithm 3, when chunks of size 1 are used. The use of chunks provides added flexibility. For example, one can consider adapting the chunk size given the set of problem instances being considered. Alternatively, one can envision adaptive chunk sizes, that are selected given both properties of the problem instance and the run times of previously analyzed chunks.

Proposition 6. *Algorithm 4 corresponds to a chunk of size n. By adapting Algorithm 4 to use chunks, and considering chunks of size 1 yields Algorithm 3.*

3.4 Practical Implementation & Configurations

Earlier work [20,31] carried out extensive evaluations of backbone computation algorithms, including different approaches for exploiting the incremental interface of SAT solvers. The overall conclusion is that Algorithm 3 is the most efficient solution for backbone computation. The experimental analysis in this paper focuses on the unified approach outlined earlier, i.e. Algorithm 4 with chunks of fixed size. Special cases of the unified algorithm are Algorithm 3, i.e. using chunks of size 1, and Algorithm 4, i.e. using a single chunk.

3.5 Additional Solutions

Besides the algorithms outlined above, 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.

4 Results

The presented algorithms were implemented using `minisat2.2` as the underlying SAT solver [9], availing of its incremental interface in all algorithms. The experiments were conducted on a compute cluster where each node is a dual quad-core

algorithm	Chunk-1	Chunk-10	Chunk-100	Chunk-500	Chunk-unbounded	VBS
solved	667	664	662	587	485	683
wins	507	448	258	220	176	–

Table 1. Overview of the results including the **v**irtual **b**est **s**olver (VBS). A win is counted if the algorithm is not worse than the best time on the instance by 1 s.

Xeon E5450 3 GHz with 32 GB of memory. Each instance was run with a timeout of 800 s and memory limit 2 GB.

To evaluate the effect of the size of a chunk on the overall performance, problem instances from practical application domains were selected from the past SAT competitions and races[4]. In total, 779 problem instances were selected. The selection was guided by the goals of finding instances that are easy for SAT solvers but also are practically motivated.

Table 1 provides an overview of the results. Chunks of size 1, 10, 100, and 500 were used accompanied by the "unbounded" chunk, i.e. the algorithm where the chunk comprises the whole set of variables [31]. The table clearly shows that the basic algorithm (Chunk-1) performs the best. Increasing the chunk size always leads to an overall decrease in performance, with the unbounded version being the worst. However, Chunk-10 solves only 3 fewer instances than Chunk-1. Moreover, the results for the virtual best solver (VBS) show that the other algorithms enabled solving 24 more instances on top of those solved by Chunk-1. Given the results for the VBS, a parallel portfolio using different backbone computation algorithms is expected to substantially outperform the best individual configuration, i.e. Chunk-1.

Figure 1 focuses on the *hard instances* from the whole instance set. These instances were chosen by eliminating those that could be solved by all the algorithms in less than 50 seconds. The results are congruent with those in the table above. Chunk-1 is the best approach and Chunk-10 closely follows. The results for the virtual best solver here not only indicate that more instances can be solved but also that there is a noticeable improvement in time if the right algorithm were to be used.

The number of backbone variables was surprisingly high. Out of the 779 instances, 477 had over 50% backbone variables; 96 instances had over 74%; 52 over 90%. More detailed information for individual instances can be found on the authors' website[5].

5 Conclusions

This paper develops improvements to algorithms for backbone computation. Whereas some of the algorithms are based on earlier work [20,31], others are

[4] http://www.satcompetition.org/.
[5] http://sat.inesc-id.pt/~mikolas/rcra12_bb

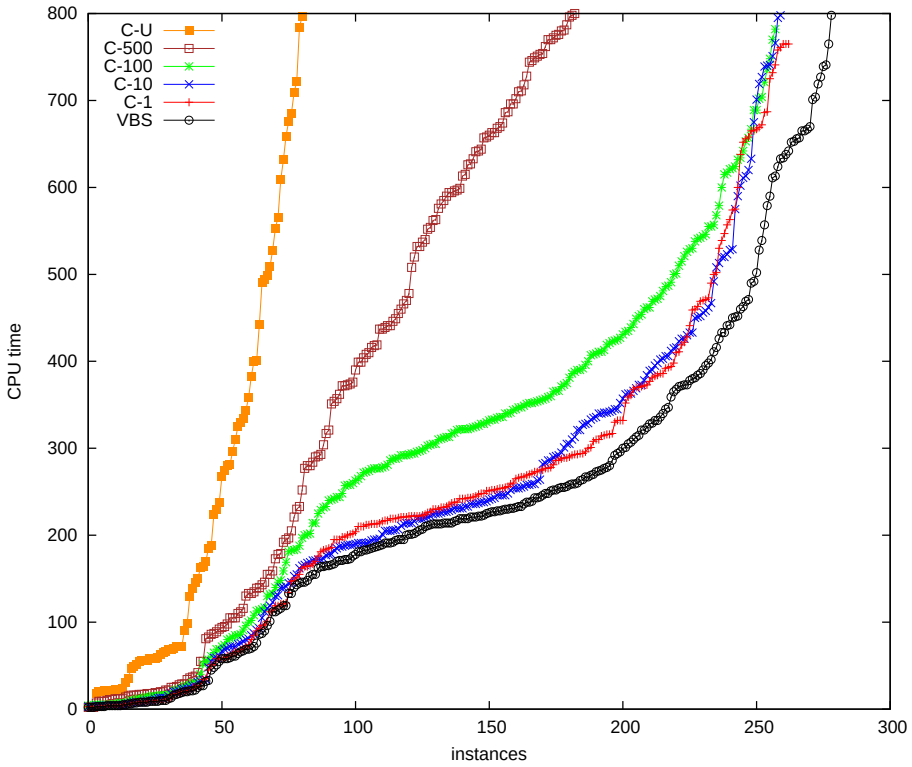

Fig. 1. Cactus plot for hard instances, i.e. instances where at least one algorithm took over 50 s

novel, and unify some of the most representative earlier algorithms. In addition, the paper extends 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 [20], are the most efficient. However, the unified algorithm developed in this paper, with a fixed chunk size, can provide performance gains for some problems instances. 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 50% of the number of variables.

The experimental results confirm that backbone computation is feasible for large practical instances. This observation motivates further work on applying backbone information for solving decision and optimization problems related to propositional theories, including model enumeration, minimal model computa-

tion and prime implicant computation. Moreover, more efficient backbone computation algorithms are expected to impact practical applications [18,14,13,30].

Future improvements to backbone computation algorithms include automatic identification of chunk size and parallel portfolios with different chunk sizes. 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), FCT through grants ATTEST (CMU-PT/ELE/0009/2009), POLARIS (PTDC/ EIA-CCO/123051/2010) and ASPEN (PTDC /EIA-CCO/110921/2009), and by INESC-ID multiannual funding from the PIDDAC program funds.

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. J. C. Culberson and I. P. Gent. Frozen development in graph coloring. *Theor. Comput. Sci.*, 265(1-2):227–264, 2001.
8. 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.
9. N. Eén and N. Sörensson. An extensible SAT-solver. In *SAT*, 2003.
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, M. Janota, and I. Lynce. On computing backbones of propositional theories. In *ECAI*, pages 15–20, 2010.
21. 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.
22. 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.
23. R. Monasson, R. Zecchina, S. Kirkpatrick, B. Selman, and L. Troyansk. Determining computational complexity from characteristic 'phase transitions'. *Nature*, 400:133–137, July 1999.
24. L. Palopoli, F. Pirri, and C. Pizzuti. Algorithms for selective enumeration of prime implicants. *Artificial Intelligence*, 111(1-2):41–72, 1999.
25. W. V. Quine. The problem of simplifying truth functions. *American Mathematical Monthly*, 59:521–531, October 1952.
26. 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.
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.
30. C. Zhu, G. Weissenbacher, and S. Malik. Post-silicon fault localisation using maximum satisfiability and backbones. In *Formal Methods in Computer-Aided Design (FMCAD)*, pages 63–66, 2011.
31. C. Zhu, G. Weissenbacher, D. Sethi, and S. Malik. SAT-based techniques for determining backbones for post-silicon fault localisation. In *High Level Design Validation and Test Workshop (HLDVT)*, pages 84–91, 2011.