Algorithms for Computing Backbones of Propositional Formulae*

Mikoláš Janota
INESC-ID Lisbon, Portugal

Inês Lynce
INESC-ID/IST, TU Lisbon, Portugal

Joao Marques-Silva
INESC-ID/IST, TU Lisbon, Portugal
CASL/CSI University College Dublin, Ireland

Abstract

The problem of propositional satisfiability (SAT) has found a number of applications in both theoretical and practical computer science. In many applications, however, knowing a formula's satisfiability alone is insufficient. Often, some other properties of the formula need to be computed. This article focuses on one such property: the *backbone* of a formula, which is the set of literals that are true in all the formula's models. Backbones find theoretical applications in characterization of SAT problems and they also find practical applications in product configuration or fault localization. This article overviews existing algorithms for backbone computation and introduces two novel ones. Further, an extensive evaluation of the algorithms is presented. This evaluation demonstrates that one of the novel algorithms significantly outperforms the existing ones.

1 Introduction

A backbone of a propositional formula ϕ is formed by literals that are true in all models of ϕ [27, 3, 17]. Alternatively, one can view a backbone as the set of *necessary* assignments: if a literal l is in the backbone of ϕ, any assignment satisfying ϕ *must* set l to true.

The term backbone was coined in research on the phase transitions in Boolean Satisfiability (SAT), where the size of a backbone was related with search complexity. In addition, backbones have also been studied in random 3-SAT [7] and in optimization problems [6, 31, 18, 34], including Maximum Satisfiability (MaxSAT) [35, 26]. Finally, backbones have been the subject of recent interest, in the analysis of backdoors [10] and in probabilistic message-passing algorithms [12].

Backbones appear in a number of practical applications of SAT. One concrete example is SAT-based *interactive product configuration* [2], where the identification of a backbone was utilized in the recent past [19, 16, 14, 15]. Identification of a backbone during the configuration process prevents the user from choosing values that cannot be extended to a model (or configuration). Another recent application of backbones is post-silicon *fault localization* in integrated circuits [37, 36]. Manolios and Papavasileiou show that backbones enable improving the solving of pseudo-Boolean constraints via compilation to SAT [21]. Lonsing and Biere use backbones to preprocess quantified Boolean formulas [20].

It is worth mentioning that throughout the literature, the concept of backbone appears under various terms. For instance, *inadmissible* and *necessary variables* [16], *bound literals* [15], *fixed assignments* [37], *units* [21], or *frozen variables* [1]. Also, we should note that the concept of a backbone is closely related to the concept of *failed literals*. A literal l is failed in a formula ϕ in conjunctive normal form if unit prop-

*This paper is based on, but significantly extends, papers presented at ECAI 2010 and RCRA 12 on the same subject [22, 23]. This is a preprint of a paper accepted to RCRA 2012 AI-Com special issue.

agation derives falsity for the formula $\phi \wedge l$ [9]. If a literal l is a failed literal, then the complementary literal $\bar{l}$ is part of the backbone of ϕ (but not the other way around).

Besides uses in practical applications, backbones provide relevant information exploitable in other problems related to propositional theories. Such problems can be decision problems but also enumeration or optimization problems. Concrete examples include model enumeration, minimal model computation and prime implicant computation, among others.

This article provides the following main contributions. 1) The article overviews recent algorithms for backbone computation [22, 37]. 2) The article 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. 3) It develops a novel algorithm based on *unsatisfiable cores*, which outperforms the existing ones. 4) The article develops a novel technique for backbone filtering from implicants, named *rotatable literals*. 5) The article 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 [22] that large instances, coming from practical applications, can have large backbones, in many cases close to 90% of the variables. In addition, the experimental results confirm that a careful implementation of some of the proposed algorithms enables us computing the backbone of large problem instances.

The article is organized as follows. Section 2 introduces notation and definitions used throughout the article. Section 3 studies several properties of backbones essential for the development of the algorithms. Section 4 describes several algorithms for computing backbones where two of these algorithms are novel. Relations between the presented algorithms are discussed. Section 5 proposes techniques for filtering literals from a backbone estimate given a model of the formula in question. Section 6 outlines possible heuristics and portfolio uses of the presented algorithms. Section 7 analyzes experimental results on large practical instances of SAT, taken from representative practical applications and from recent SAT competitions[1]. Finally, Section 8 concludes the article.

2 Preliminaries

Throughout the paper we assume a universe of Boolean variables X. A *literal* is a Boolean variable or its negation. A *clause* is a disjunction of zero or more literals. A formula in *conjunctive normal form (CNF)* is a conjunction of clauses. In this article all formulas are in CNF and therefore whenever we say formula, we automatically assume a formula in CNF. Whenever convenient, a clause is seen as a set of literals and a formula as a set of clauses.

For a literal l, we write $\bar{l}$ to denote its complement, i.e. $\bar{x} = \neg x$, $\overline{\neg x} = x$. For a literal l, we write $\text{var}(l)$ to denote the variable in the literal, i.e. $\text{var}(x) = x$, $\text{var}(\bar{x}) = x$. Analogously, for a clause ω, $\text{var}(\omega) = \{\text{var}(l) \mid l \in \omega\}$ and for a formula ϕ, $\text{var}(\phi) = \bigcup_{\omega \in \phi} \text{var}(\omega)$.

The following definitions are adopted from [24]. An assignment ν is a mapping from X to $\{0, u, 1\}$, $\nu : X \to \{0, u, 1\}$. The constant u has the meaning of an *unspecified* value, and we define $0 < u < 1$ and $1 - u = u$. The assignment ν is a *complete* assignment if $\nu(x) \in \{0, 1\}$ for all $x \in X$; otherwise, ν is a *partial* assignment. Given a literal l, $\nu(l) = \nu(x)$ if $l = x$, and $\nu(l) = 1 - \nu(x)$ if $l = \bar{x}$. An assignment ν is also applicable to a clause and a formula. The values are defined as $\nu(\omega) = \max_{l \in \omega} \nu(l)$ and $\nu(\phi) = \min_{\omega \in \phi} \nu(\omega)$.

A formula ϕ is *satisfiable* iff there exists an assignment ν such that $\nu(\phi) = 1$. A formula is *unsatisfiable* iff it is not satisfiable.

2.1 Models and Implicants

An assignment ν *satisfies* a formula ϕ iff the formula evaluates to 1 under the assignment, i.e. $\nu(\phi) = 1$. Analogously, we say that an assignment ν satisfies a literal/clause. We say that ν satisfies a set of literals L iff it satisfies all literals $l \in L$. An assignment ν is

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

a *model* of a formula ϕ iff it satisfies ϕ and it specifies a value for each variable in ϕ, i.e. $\nu(x) \in \{0, 1\}$ for all $x \in \text{var}(\phi)$.

Definition 1 (implicant)**.** *An* implicant ν *of a formula* ϕ *is a set of literals such that it does not contain two complementary literals and any assignment* μ *that satisfies* ν *satisfies* ϕ*.*

An implicant ν *of* ϕ *is a* prime implicant *iff there is no other implicant* ν' *of* ϕ *such that* $\nu' \subsetneq \nu$*.*

Observe that any implicant ν of ϕ has a nonempty intersection with each of the clauses in ϕ. Consequently, any set of non-contradictory literals ν' s.t. $\nu \subseteq \nu'$ is also an implicant of ϕ.

2.2 Backbones

A widely used definition of a backbone is given in [17] (see [6] for an alternative definition):

Definition 2 (backbone literal)**.** *Let* ϕ *be a satisfiable formula. A literal* l *is a* backbone literal of ϕ *iff* $\mu(l) = 1$ *for any model* μ *of* ϕ*.*

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

Example 1. *For the formula* $\phi = \{\bar{x} \vee \bar{y}, x, z \vee w\}$*, the literals* x *and* $\bar{y}$ *form the backbone of* ϕ*.*

Note that any satisfiable formula ϕ has a *unique* backbone and that any literal that is a backbone literal of ϕ must be part of that backbone. Hence, the expressions *"l is a backbone literal of ϕ"* and *"l is in the backbone of ϕ"* will be used interchangeably. Often, when ϕ is clear from the context, it is omitted.

In addition, the computation of the backbone of a formula is referred to as the *backbone problem.*

Remark 1. *This paper focuses on backbones of satisfiable formulas as in for instance [17]. It is possible, however, to define backbones for unsatisfiable instances [27] or optimization instances [18].*

3 Properties of Backbones

Directly following the definition of a backbone, a possible solution for computing the backbone of a formula consists in intersecting all of its models. The following simple propositions tell us that it is sufficient to focus only on implicants of the formula. First we observe that any backbone literal must appear in an arbitrary implicant.

Proposition 1 (backbone literals and implicants)**.** *Let* ϕ *be a satisfiable formula,* l *a literal, and* ν *an implicant of* ϕ*. If* l *is a backbone literal of* ϕ*, then* $l \in \nu$*. Conversely, if* $l \notin \nu$ *then* l *is* not *a backbone literal.*

Proof. Assume that $l \notin \nu$. Construct an assignments μ that assigns true to all literals in ν and $\mu(l) = 0$. Additionally, put μ to an arbitrary value for variables $\text{var}(\phi) \setminus (\text{var}(\nu) \cup \text{var}(l))$. Since ν is an implicant ϕ, any assignment that satisfies ν satisfies ϕ. Hence, μ is a model of ϕ that shows that l is not a backbone literal. □

Consequently, if we consider a set of implicants that covers all the models of the formula, it is sufficient to intersect only those to get the backbone.

Proposition 2 (backbone and implicant cover)**.** *Let* ϕ *be a satisfiable formula and* l *a literal. Consider a set of implicants* $\mathcal{I}$ *such that any model of* ϕ *satisfies at least one of the implicants in* $\mathcal{I}$*. The literal* l *is a backbone literal of* ϕ *iff* l *occurs in all implicants in* $\mathcal{I}$*.*

Proof. If the literal l is in the backbone, then it must occur in all implicants of ϕ due to Proposition 1.

For contradiction, assume that l is not a backbone literal. Therefore there is a model μ of ϕ such that $\mu(l) = 0$. Due to the condition on $\mathcal{I}$, this assignment μ satisfies some implicant $\nu \in \mathcal{I}$. However, this is a contradiction because $l \in \nu$. □

Proposition 2 gives us a recipe for how to compute a backbone through implicants: look for new implicants until they cover the given formula and compute an intersection of those implicants.

Another approach to computing the backbone is inspired by the complexity classification of the problem. Deciding whether a literal is in the backbone or not is co-NP-complete [15, Claim 2]. The functional problem of computing the backbone is NP-equivalent [17],

Proposition 3. *Let ϕ be a satisfiable formula and $x \in \text{var}(\phi)$. Consider the modified formulas $\phi_P = \phi \cup \{x\}$ and $\phi_N = \phi \cup \{\bar{x}\}$. Then one of the following holds:*

1. *ϕ_P is satisfiable and ϕ_N is unsatisfiable. Hence the literal x is a backbone literal of ϕ.*
2. *ϕ_N is satisfiable and ϕ_P is unsatisfiable. Hence the literal $\bar{x}$ is a backbone literal of ϕ.*
3. *Both ϕ_N and ϕ_P are satisfiable. Hence neither the literal $\bar{x}$ nor x is a backbone literal of ϕ.*

Proposition 3 shows that deciding whether a literal is a backbone literal or not is in co-NP. The following proposition shows that it is also co-NP-hard.

Proposition 4. *[15, Claim 2]. Let ϕ be a formula and l a literal. Deciding whether l is a backbone literal of ϕ is co-NP-hard.*

Proof. Let ψ be a CNF. Let x be some variable that does not appear in ψ. Construct the following CNF $\psi' = \{\omega \vee x \mid \omega \in \psi\}$. The formula ψ' is satisfiable because setting x to true satisfies all clauses of ψ'. Observe that an assignment τ to variables $\text{var}(\psi)$ is a model of ψ iff $\tau \cup \{\bar{x}\}$ is a model of $\psi' \wedge \bar{x}$.

We show that x is a backbone literal of ψ' if and only if ψ is unsatisfiable. If x is a backbone literal, there is no assignment satisfying ψ' setting x to false, and hence ψ has no models. If x is not a backbone literal, there is a model τ of ψ' with $\bar{x} \in \tau$. Consequently, $\tau \setminus \{\bar{x}\}$ is a model of ψ. Hence determining whether x is a backbone literal decides whether ψ is unsatisfiable, which is co-NP-complete. □

Propositions 3 and 4 show that determining whether a literal is a backbone literal is co-NP-complete. This suggests algorithms that compute the backbone of a formula with a sequence of SAT tests that grows with $|\text{var}(\phi)|$, as suggested for example in [16, 14, 10]. The idea is followed and extended in the upcoming section.

4 Computing Backbones

This section presents a number of algorithms for backbone computation. The presentation begins with an implicant enumeration algorithm (based on Proposition 2) and it continues with an algorithm that tests one literal at a time (based on Proposition 3) [22]. Further, the section discusses an algorithm based on negating a backbone estimate [37]. A novel algorithm is introduced, which uses the notion of subsets of a backbone estimate (*chunks*) and it is shown that two of the previous algorithms are a special case of this novel one. The last part of the section focuses on the second novel algorithm, which uses *unsatisfiable cores* ("reasons for unsatisfiability").

For the purpose of algorithm presentation we assume that a SAT solver is represented by a function $\text{SAT}(\phi)$, which returns a pair (outc, ν). The first component outc has either the value true or false, corresponding to satisfiability or unsatisfiability of ϕ, respectively. If ϕ is satisfiable, i.e. $\mathsf{outc} = \mathsf{true}$, the second component ν is an implicant of ϕ. We modify this representation slightly when describing the core-based algorithms in Section 4.4.

Remark 2. *In practice SAT solvers return assignments to all variables in the given formula rather than implicants. Clearly, any such assignment corresponds to an implicant that contains a literal for each variable. We choose to model the return value of a SAT solver as an implicant since it simplifies the presentation and moreover, makes the algorithms more general. Section 5 discusses how implicants can be reduced for the purpose of efficiency.*

4.1 Implicant Enumeration

An algorithm for computing the backbone of a propositional formula based on implicant enumeration is shown in Algorithm 1. The algorithm enumerates implicants one by one and updates the backbone estimate in each iteration. In order to avoid finding the same implicant again, the algorithm adds to the formula a *blocking clause* [29, 25]. A blocking clause for an implicate ν is defined as the clause $\bigvee_{l \in \nu} \bar{l}$, as shown in line 7. Adding the blocking clauses to the

Algorithm 1: Enumeration-based backbone computation

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

```
1  ν_R ← {x | x ∈ var(φ)} ∪ {x̄ | x ∈ var(φ)}          // Initial backbone upper bound
2  while ν_R ≠ ∅ do
3      (outc, ν) ← SAT(φ)                              // SAT solver call
4      if outc = false then
5          return ν_R                                  // Terminate if no more implicants
6      ν_R ← ν_R ∩ ν                                   // Update backbone estimate
       // Block implicant
7      ω_B ← ⋁_{l∈ν} l̄
8      φ ← φ ∪ ω_B
9  assert(ν_R = ∅)          // Backbone estimate became empty before enumeration finished
10 return ν_R
```

formula is both a necessary and a sufficient condition for the implicant ν not to be found again.

In order to mitigate the size of blocking clauses, the implicant returned by the SAT solver can be heuristically reduced by standard techniques, e.g. variable lifting [29]. We return to this topic in more detail in Section 5.

It is interesting to observe that Algorithm 1 maintains a superset of the backbone, i.e. it maintains an *upper bound* of the backbone (in terms of the subset ordering).

4.2 Iterative SAT Testing

Enumerating implicants has its clear limitations since the number of implicants is in the worst-case exponential in the number of variables (cf. [4]). An alternative to enumerating implicants is to focus at each literal separately and *test* whether it is a backbone literal or not.

Proposition 3 shows that a literal is in the backbone iff $\text{SAT}(\phi \cup \{\bar{l}\})$ is unsatisfiable. This observation allows us devising Algorithm 2. Observe that if a literal is decided to be a backbone literal, then it is correct to add it to the formula as a unit clause, as shown in lines 8 and 11. This addition is not required for the correctness of the algorithm, but it is expected to simplify the remaining SAT tests. The worst case number of SAT tests for Algorithm 2 is $2 \times |\text{var}(\phi)|$.

Recall that a SAT solver not only tells us whether the given formula is satisfiable or not, but it also gives us an implicant of a satisfiable formula. Recall also that any backbone literal must be in any implicant (Proposition 1). This gives us an opportunity to improve Algorithm 2. Once we obtain an implicant from a SAT call, we do not have to test anymore any of those literals that do *not* appear in the implicant.

This observation suggests a different organization, corresponding to Algorithm 3. The algorithm maintains a set of literals Λ of those literals that still need to be tested. The set is initialized by an implicant ϕ obtained by a SAT call. Hence, when the loop starts, the set Λ contains at most $|\text{var}(\phi)|$ literals. In each iteration of the loop, the algorithm picks a literal l to test and subsequently tests if l is in the backbone by the call $\text{SAT}(\phi \cup \{\bar{l}\})$. If l is in the backbone, $\phi \cup \{\bar{l}\}$ is unsatisfiable and l is stowed in ν_R. If l is *not* in the backbone, $\phi \cup \{\bar{l}\}$ is satisfied by some implicant ν, which is used to remove from Λ those literals that do not appear in it. Observe that the tested literal l is removed from Λ in line 12. This is because ν satisfies $\phi \cup \{\bar{l}\}$ and therefore $l \notin \nu$.

Algorithm 3 guarantees that the loop iterates at

Algorithm 2: Iterative algorithm (two tests per variable)

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

```
1  ν_R ← ∅                                              // Initial backbone lower bound
2  foreach x ∈ var(φ) do
3      (outc_1, ν) ← SAT(φ ∪ {x})
4      (outc_0, ν) ← SAT(φ ∪ {x̄})
5      assert ( outc_1 = true or outc_0 = true )        // φ must be satisfiable
6      if outc_1 = false then
7          ν_R ← ν_R ∪ {x̄}                              // x̄ is backbone
8          φ ← φ ∪ {x̄}
9      if outc_0 = false then
10         ν_R ← ν_R ∪ {x}                              // x is backbone
11         φ ← φ ∪ {x}
12 return ν_R
```

most $|\text{var}(\phi)|$ times. Hence, the algorithm performs at most $|\text{var}(\phi)| + 1$ SAT tests in total.

In contrast to the enumeration-based approach, Algorithm 2 refines a subset of the backbone. In each iteration of the algorithm, the set ν_R represents a *lower bound* of the backbone. Algorithm 3 integrates the two bounds, lower and upper, together. Even though Algorithm 3 does not have an explicit representation of the upper bound, it maintains its explicit representation in the form $\Lambda \cup \nu_R$. When the algorithm terminates, Λ becomes empty and ν_R consists of all the backbone literals.

4.3 Integrating the Complemented Backbone Estimate

An algorithm that complements the algorithms described in the previous sections was recently proposed in [37]. 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 [37] is also based on iterative SAT testing but only a single SAT solver call is required to prove that the current backbone estimate is indeed the backbone. This section studies this algorithm and proposes optimizations targeting improved efficiency.

Algorithm 4 shows the algorithm developed in [37]. In each iteration of the loop, a complement of the backbone estimate is conjoined to the formula and tested for satisfiability (line 4). If the formula is satisfiable, then the computed implicant includes *at least one* literal in the complement of the backbone estimate. Hence, the backbone estimate is refined (line 7). The process is repeated until the backbone estimate represents the actual backbone, in which case the formula is unsatisfiable.

Proposition 5. *Let $|BB|$ denote the backbone size. Then, the number of SAT tests in Algorithm 4 is at most* $(|\text{var}(\phi)| - \max(|BB|, 1) + 1) + 1 \leq |\text{var}(\phi)| + 1$.

Proposition 6. *There is exactly one unsatisfiable SAT test for Algorithm 4. The number of satisfiable SAT tests is at most* $|\text{var}(\phi)| - |BB| \leq |\text{var}(\phi)|$.

As observed in [37], Algorithm 4 mostly performs poorly when compared with the algorithms described in previous sections. This is a consequence of negating the *whole* backbone estimate, which tends to result in difficult instances of SAT.

A solution to the problem of negating the whole backbone estimate is to iteratively analyze its sub-

Algorithm 3: Iterative algorithm (one test per variable)

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

```
1  (outc, ν) ← SAT(φ)
2  Λ ← ν                                              // SAT tests planned
3  ν_R ← ∅                                            // Initial backbone lower bound
4  while Λ ≠ ∅ do
5      l ← pick a literal from Λ                      // Pick a literal to test
6      (outc, ν) ← SAT(φ ∪ {l̄})                       // Test if l is a backbone
7      if outc = false then
           // Backbone identified
8          ν_R ← ν_R ∪ {l}                            // Add l to the backbone estimate
9          Λ = Λ ∖ {l}                                // l does not need to be tested anymore
10         φ ← φ ∪ {l}
11     else
12         Λ ← Λ ∩ ν                                  // Literal filtering
13 return ν_R
```

sets. 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 size K is added. The intuition behind this clause is "show that at least one of the literals in the chunk is *not* a backbone literal."

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.

Proposition 7. *Algorithm 4 corresponds to Algorithm 5 with chunk size* $|\text{var}(\phi)|$. *Algorithm 3 corresponds to Algorithm 5 with chunk size* 1.

4.4 Core-based Algorithm

In the previous algorithms we compute the backbone using the following pattern. First, we find some implicant ν, which gives us an initial estimate of the backbone. All the literals that do *not* appear in ν are *not* in the backbone. The literals that do appear in ν *might* be in the backbone. To prove or disprove that certain literal $l \in \nu$ is in the backbone, we try to *flip it*. More precisely, we try to find a different implicant that contains the complementary literal.

The key question is how to look for this other implicant. In Algorithm 5 we look for an implicant that flips *at least one* of the literals in ν. In this section, we take a different approach that tries to flip *all* literals in ν *at the same time*. If we are lucky and we manage to do that, we show that there are no backbone literals and the algorithm terminates after 2 SAT calls. This is illustrated by the following example.

Example 2. *Let* $\phi = \{x \vee y, \bar{x} \vee \bar{y}\}$ *and* $\nu = \{x, \bar{y}\}$. *The set* ν *is an implicant of* ϕ*; flipping all its literals yields* $\nu' = \{\bar{x}, y\}$ *Since* ν' *is also an implicant of* ϕ, *the backbone of* ϕ *is empty.*

In general, however, it is not possible to flip *all* the literals at the same time. Consequently, the proposed algorithm gradually reduces the set of literals that it tries to flip. In order to decide which literals should no longer be flipped, the algorithm assumes that the SAT solver is capable of giving us a *core*—a set of clauses responsible for unsatisfiability.

Algorithm 4: Iterative algorithm with complement of backbone estimate

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

```
1 (outc, ν) ← SAT(φ)
2 ν_R ← ν                                          // Initial backbone estimate
3 while ν_R ≠ ∅ do
4 |   (outc, ν) ← SAT(φ ∪ { ⋁_{l∈ν_R} l̄ })
5 |   if outc = false then
6 |   |   return ν_R                               // Terminate if unsatisfiable
7 |   ν_R ← ν_R ∩ ν                                 // Refine backbone estimate
8 return ν_R
```

Definition 4 (core)**.** *For an unsatisfiable formula ϕ, a formula $\psi \subseteq \phi$ is called a core iff ψ is unsatisfiable.*

Remark 3. *Observe that the definition of a core does not guarantee any type of minimality. Indeed, if a formula ϕ is unsatisfiable, the whole ϕ is already* a *core. In practice, however, state-of-the-art SAT solvers return cores significantly smaller than the given formula. Nevertheless, any correct algorithm must account for cases where the returned core contains superfluous clauses, i.e. clauses whose removal yields another core.*

Consider some set of non-contradictory literals L that form a superset of the backbone. To determine if they can be flipped, we call a SAT solver on the formula $\phi \cup \bigcup_{l \in L} \{ \bar{l} \}$. If the call is unsatisfiable, i.e. the literals cannot be flipped, the corresponding core gives us some subset of literals that *cannot* be flipped.

There is one special case worth noting. If the core contains a negation of a single literal from L, then this literal is a backbone literal.

Proposition 8. *Let ϕ be a satisfiable formula and L be a set of literals such that $\phi \wedge \bigwedge_{k \in L} \bar{k}$ is unsatisfiable. A literal $l \in L$ is in the backbone of ϕ iff there is a core ψ of $\phi \wedge \bigwedge_{k \in L} \bar{k}$ for which $\psi \subseteq \phi \wedge \bar{l}$.*

There is another special case that we need to take into account and that is when the core contains *all* literals to be flipped. In such case the core-based algorithm cannot be used and it reverts to one of the algorithms described earlier.

Since the upcoming algorithm requires the SAT solver to return a core, we extend the function $\mathrm{SAT}(\phi)$ to return a triple (outc, ν, C), where outc and ν have the same meaning as before, and C is a core of ϕ if $\mathsf{outc} = \mathsf{false}$.

Algorithm 6 shows a pseudocode for the above presented ideas. The algorithm maintains its computation state in three variables. The variable Λ contains those literals that still might be in the backbone (but we are not sure). The variable ν_R contains literals that have been shown to be in the backbone. Finally, the variable ω_N contains the negation of some of the literals in Λ.

Initially ω_N contains the negation of all the literals in Λ. As the inner loop progresses, ω_N is gradually reduced based on the cores obtained from the SAT solver (line 17). Note that the algorithm utilizes Proposition 8 in order to identify backbone literals (lines 13–16).

The inner loop terminates either when some literals from Λ were flipped (the SAT call returns true), or the cores exhaust the set Λ, i.e. ω_N is empty. If ω_N is empty, Algorithm 6 reverts to some other algorithm to test whether the remaining literals are in the backbone or not (line 19). Algorithm 3 ("one test per variable") is particularly suitable for this task because it is easy to instruct it to test only a set of literals.

Algorithm 5: Chunking algorithm

Input : Satisfiable formula ϕ, with variables X; $K \in \mathbb{N}^+$ chunk size
Output: Backbone of ϕ, ν_R

```
1  (outc, ν) ← SAT(φ)
2  ν_R ← ∅                                              // Initial backbone lower bound
3  Λ ← ν                                                // Initial literals to test
4  while Λ ≠ ∅ do
5      k ← min(K, |Λ|)
6      Γ ← pick k literals from Λ
7      (outc, ν) ← SAT(φ ∪ { ⋁_{l∈Γ} l̄ })
8      if outc = false then
           // All literals in chunk are backbones
9          ν_R ← ν_R ∪ Γ                                // Add Γ to lower bound.
10         Λ ← Λ ∖ Γ                                    // Literals in Γ do not need to be tested anymore.
11         φ ← φ ∪ {{ l } | l ∈ Γ}
12     else
13         Λ ← Λ ∩ ν
14 return ν_R
```

Example 3. *Let $\phi = \{x \vee y, u \vee v, w\}$ and $\nu = \{x, y, \bar{u}, v, w\}$. Invoking SAT on $\phi \wedge \bar{x} \wedge \bar{y} \wedge u \wedge \bar{v} \wedge \bar{w}$ gives a core $\{w, \bar{w}\}$. The core lets us infer that w is a backbone literal by Proposition 8. Invoking SAT on $\phi \wedge \bar{x} \wedge \bar{y} \wedge u \wedge \bar{v}$ gives a core $\{x \vee y, \bar{x}, \bar{y}\}$ hence we remove the requirement $\bar{x} \wedge \bar{y}$. Invoking SAT on $\phi \wedge u \wedge \bar{v}$ yields an implicant flipping the value of both u and v thus showing they are not backbone literals.*

The next iteration of the outer loop invokes SAT on $\phi \wedge \bar{x} \wedge \bar{y}$ yielding again the core $\{x \vee y, \bar{x}, \bar{y}\}$, which contains all the remaining literals to be flipped and we revert to one of the other algorithms.

Algorithm 6 suffers from the fact that it tries to flip too many literals at the same time and therefore it might take too long before it finds a satisfying assignment. In response to this issue, again we apply the idea of chunking. The chunking version of the algorithm in each iteration picks a chunk of the literals that still might be in the backbone and tries to flip those at the same time. This is repeated until there are no literals that might be in the backbone.

Algorithm 7 presents a chunking version of Algorithm 6. The structure of the algorithm is similar to the one of Algorithm 6 with the exception that in each iteration of the outer loop it tries to flip only some chunk of literals Γ. Note that the inner loop operates on the complement of Γ stored in the variable ω_N. Using unsatisfiable cores, the set ω_N is being reduced until some subset of the literals in Γ is flipped. The special case is again when ω_N becomes empty and then we revert to another algorithm to test literals in Γ. Unit cores are treated just as in Algorithm 6.

Observe that the algorithm coincides with the iterative algorithm (Algorithm 3) when $K = 1$ just as the chunking algorithm (Algorithm 5).

Here we should make an important remark about the core computation. In general, computing a core might bring in some computational inefficiency. In this concrete case, however, there is no practical efficiency penalty. This is because we are computing a core of the formula $\phi \wedge \bigwedge_{l \in L} l$ for some set of literals L and moreover we are only interested in the intersection of the core with the literals from L. SAT solvers based on the interface of `minisat2.2` enable

Algorithm 6: Core-based Algorithm

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

```
1  (outc, ν, C) ← SAT(φ)
2  ν_R ← ∅                                              // Initial backbone lower bound
3  Λ ← ν                                                // Initial literals to test
4  while Λ ≠ ∅ do
5      ω_N ← {l̄ | l ∈ Λ}
6      while true do
7          (outc, ν, C) ← SAT(φ ∪ {{l} | l ∈ ω_N})
8          if outc = true then
9              Λ ← Λ ∩ ν
10             break // Move onto a different set of literals to flip
11         else
12             assert(C ∩ ω_N ≠ ∅)                      // φ must be satisfiable
13             if C ∩ ω_N = {l} then
                   // The core contains a single literal from ω_N
14                 ν_R ← ν_R ∪ {l̄}
15                 Λ ← Λ ∖ {l̄}
16                 φ ← φ ∪ {l̄}
17             ω_N ← {p | p ∈ ω_N ∧ {p} ∉ C}        // Remove from ω_N literals that appear in the core
18             if ω_N = ∅ then
19                 test literals in Λ by another algorithm
20                 return ν_R
21 return ν_R
```

computing precisely that by enabling passing the literals L as *assumptions*. Those assumptions that are part of the core are then returned in the *final conflict clause* [8].

5 Backbone Filtering

With the exception of Algorithm 2, all the algorithms utilize implicants to prune the backbone estimate. Indeed, according to Proposition 1, if ν is an implicant then any literal $l \notin \nu$ is *not* in the backbone. Consequently, the less literals an implicant contains, the more literals are filtered out from the backbone estimate. However, modern SAT solvers compute complete assignments, i.e. implicants that contain a literal for each variable [24]. Thus, for the purpose of backbone filtering it might be useful to see if some of these literals are not redundant.

Example 4. *Let $\phi = \{x \vee y, x \vee \bar{z}\}$ and $\nu = \{x, y, \bar{z}\}$. The set ν is an implicant of ϕ but so is the singleton set $\{x\}$.*

Different techniques can be used for removing literals from computed implicants. One example is *variable lifting* [29]. Lifting consists of analyzing each variable and discarding the variable if it is not used for satisfying any clause. Another technique is (approximate) *set covering* [29]. The goal is to compute an implicant $\nu' \subseteq \nu$ such that for any impli-

Algorithm 7: Core-based Algorithm with Chunking

Input : Satisfiable formula ϕ; $K \in \mathbb{N}^+$ chunk size
Output: Backbone of ϕ, ν_R

```
1  (outc, ν, C) ← SAT(φ)
2  ν_R ← ∅                                              // Initial backbone lower bound
3  Λ ← ν                                                // Initial literals to test
4  while Λ ≠ ∅ do
5      k ← min(K, |Λ|)
6      Γ ← pick k literals from Λ
7      ω_N ← { l̄ | l ∈ Γ }
8      while true do
9          (outc, ν, C) ← SAT(φ ∪ { { l } | l ∈ ω_N })
10         if outc = true then
11             Λ ← Λ ∩ ν
12             break // Done with the chunk
13         else
14             if C ∩ ω_N = { l } then
                   // The core contains a single literal from ω_N.
15                 ν_R ← ν_R ∪ { l̄ }
16                 Λ ← Λ ∖ { l̄ }
17                 φ ← φ ∪ { l̄ }
18             ω_N ← {p | p ∈ ω_N ∧ {p} ∉ C}        // Remove from ω_N literals that appear in the core.
19             if ω_N = ∅ then
20                 test literals in Γ by another algorithm
21                 Λ = Λ ∖ Γ
22                 break // Done with the chunk
23 return ν_R
```

cant $\nu'' \subseteq \nu$ it holds that $|\nu'| \leq |\nu''|$. This problem translates directly to the *set cover problem* because ν' has to be a minimal set of literals such that it has a nonempty intersection with each clause of ϕ. Since the set cover problem is NP-hard, approximate solutions are often used. One example is a greedy approximation algorithm for the set cover problem (e.g. [5]).

In the following we refer to these techniques (lifting, set cover) as *implicant reduction*, i.e. we say that an implicant ν was reduced to an implicant ν' and thus filtering out the literals $\nu \setminus \nu'$ from the backbone estimate.

Here we develop another technique for filtering non-backbone literals, which we call *rotatable literals*. We show that this technique is strictly stronger than implicant reduction.

Consider an implicant where flipping (rotating) the value of some variable x yields another implicant. This gives us two different implicants that show that neither x nor $\bar{x}$ is a backbone literal.

Definition 5 (rotatable literal)**.** *Let ν be an implicant of ϕ and l be a literal s.t. $l \in \nu$. Let ν' be defined as $\nu' = \nu \setminus \{l\} \cup \{\bar{l}\}$. The literal l is* rotatable *in ν iff ν' is an implicant of ϕ.*

Example 5. *Let* $\phi = \{x \vee y, z \vee w\}$ *and* $\nu = \{x, y, z, \bar{w}\}$. *The literals* x, y, *and* $\bar{w}$ *are rotatable in* ν *but* z *is not since flipping it unsatisfies the clause* $z \vee w$. *All of the four literals are rotatable in the implicant* $\{x, y, z, w\}$.

The intuition behind rotatable literals is that one can rotate (flip) the literal and obtain another implicant. Effectively, however, this is equivalent to the fact that removing the literal yields another implicant.

Proposition 9. *Let* ν *be an implicant of* ϕ *and* l *a literal such that* $l \in \nu$. *The literal* l *is rotatable in* ϕ *iff* $\nu \setminus \{ l \}$ *is an implicant of* ϕ.

Proof. If l is rotatable, extending $\nu \setminus \{l\}$ with either l or $\bar{l}$ yields and implicant and therefore $\nu \setminus \{l\}$ is an implicant as well.

If $\nu \setminus \{l\}$ is an implicant, construct ν' from ν as in Definition 5, i.e. $\nu' = \nu \setminus \{ l \} \cup \{ \bar{l} \}$. Since $\nu \setminus \{ l \}$ is an implicant of ϕ and $\nu \setminus \{ l \} \subseteq \nu'$, the set ν' is also an implicant of ϕ. □

Proposition 10. *If literal* l *is rotatable in an implicant* ν *of* ϕ, *then neither of the literals* l *and* $\bar{l}$ *is a backbone literal.*

Proof. Immediate consequence of Proposition 1. □

It is possible to compute the set of rotatable literals by directly relying on their definition. One goes through the list of all literals in the given implicant and for each literal tests whether flipping it yields another implicant or not. The complexity of this procedure is $\mathcal{O}(|\mathrm{var}(\phi)| \times |\phi|)$. Fortunately, it is possible to compute rotatable literals in a single traversal of the formula. For such we utilize the following definition.

Definition 6 (unit literal)**.** *A literal* l *is* unit in a clause ω for a set of literals ν *iff* $\omega \cap \nu = \{l\}$.

A literal l *is* unit in a formula ϕ for a set of literals ν *iff there is a clause* $\omega \in \phi$ *in which* l *is unit for* ν.

Proposition 11. *A literal* l *is rotatable in an implicant* ν *of* ϕ *iff neither* l *nor* $\bar{l}$ *is unit in* ϕ *for* ν.

Proof. For contradiction assume that l is rotatable and there is a clause ω where l is unit for ν. Since l is rotatable, $\nu' = \nu \setminus \{l\} \cup \{\bar{l}\}$ must be an implicant. However, this is a contradiction since $\omega \cap \nu' = \emptyset$, thus, ν' it is not an implicant.

To show that l is rotatable, we demonstrate that $\nu' = \nu \setminus \{l\} \cup \{\bar{l}\}$ is an implicant. We do so by considering the following three cases. 1) Consider a clause ω that contains the literal l. Since l is not unit in ω, there is another literal $k \in \omega \cap \nu$ and therefore also $k \in \omega \cap \nu'$. 2) Analogously, a clause $\omega \in \phi$ containing $\bar{l}$ is satisfied by ν'. 3) A clause not containing either of l and $\bar{l}$ is satisfied by ν' because $\nu' \cap \omega = \nu \cap \omega$. □

Proposition 9 relates rotatable literals and implicants. In particular, it says that if removing a literal from an implicant yields another implicant, then the literal is rotatable. An immediate consequence is that filtering the backbone estimate by literal rotation is at least as powerful as filtering by implicant reduction. Since if a literal can be filtered by implicant reduction, it can also be identified as rotatable (and filtered). The following proposition shows that identifying rotatable literals gives us a more powerful technique than implicant reduction.

Proposition 12. *There are instances where implicant reduction does not filter a literal, which is filtered by the technique of rotatable literals.*

Proof. Let $\phi = \{x \vee y\}$ and $\nu = \{x, y\}$. Both x and y are rotatable but any implicant reduction filters out at most one of the two literals. □

Remark 4. *The example in the proof of Proposition 12 shows another interesting property of rotatable literals. The implicant* $\{x, y\}$ *of* $\{x \vee y\}$ *can be reduced into two different implicants* $\{ x \}$ *and* $\{ y \}$; *each filtering out a different backbone literal. In general, an implicant can be reduced in exponentially many ways. Proposition 9 shows that filtering by rotatable literals filters out* all *literals obtained from all those possible reductions.*

6 Other Heuristics and Portfolios

Besides the algorithms outlined above, and which are evaluated in Section 7, a number of additional heuristics and techniques can be envisioned on top of these algorithms.

Diversification. Once a SAT solver gives us an implicant, we can remove from the backbone upper bound all the literals that do not appear in this implicant. Since throughout the course of the run of the algorithm many implicants are generated, it would be beneficial for these implicants to be as *diverse* as possible in order to reduce the backbone estimate more rapidly. Existing techniques for SAT solution diversification could be applied to this end [28].

Heuristic parametrization. The article introduces two novel chunking algorithms. Both of them are parametrized by the size of the chunk K. Hence, these algorithms can be fine-tuned by setting this parameter based on some properties of the given instance. Moreover, K could be changed dynamically throughout the run of the algorithm based on some heuristics. Since larger K leads to harder SAT calls but reduces the number of the calls, it is meaningful to increase K for formulas that are "easy" for the SAT solver and conversely decrease K for hard SAT formulas. One could search inspiration in frameworks like SATzilla [33, 32] or ParamILS [13] to heuristically determine the formula's hardness and determine an appropriate strategy for the value of K.

Portfolios. A natural way how to avail of the different algorithms and their parametrizations is to aggregate them into a portfolio. This can be done by picking heuristically the best solver for the given instance, as for example in SATzilla [33], or executing the algorithms in parallel and terminate once the winner terminates, as for instance in ManySAT [11].

Table 1: Legend for Configurations of Algorithms

element	**meaning**
number	size of chunk
u	chunk size var(ϕ)
cb	core-based algorithm
l	approximate set covering reduction
r	rotating variables
VBS	virtual best solver

7 Results

The presented algorithms were implemented using `minisat2.2` as the underlying SAT solver [8], availing of its incremental interface in all algorithms. The experiments were conducted on a computer cluster where each node is a dual quad-core Xeon E5450 3 GHz with 32 GB of memory. Each instance was run with a timeout of 800 s and memory limit of 2 GB. Each test was repeated 3 times and the presented execution times are obtained as the median of those (all implementations are deterministic, the repetitions are used only to counter fluctuations in the cluster). Results for each individual instance and algorithm configuration can be found on the authors' website[2].

Earlier work [22, 37] carried out extensive evaluations of Algorithms 1 ("implicant enumeration"), 2 ("two tests per variable"), 3 ("one test per variable"), and 4 ("complement of backbone estimate").

This earlier evaluation shows that Algorithm 3 consistently outperforms all the others. Indeed, the enumeration algorithm has to do at least as many iterations as there are prime implicants of the given formula and the two-tests algorithm simply does not avail of the implicants returned by the SAT solver as the one-test algorithm does.

Hence, the following experimental results focus on the following facets: 1) *iterative chunking algorithm* (Algorithm 5), 2) *core-based chunking algorithm* (Algorithm 7), and 3) *backbone filtering* (Section 5).

7.1 Means of Presentation

Table 1 describes the elements that identify algorithms within plots and tables. Hence, for instance 500 identifies the iterative algorithm with the chunk size 500 and cb100r the core-based algorithm with chunk size 100 and literal rotation (Definition 5).

A number of results are shown as *cactus plots*, which contain a point at the coordinate (x, y) if the considered algorithm solves x instances within y seconds. All the presented cactus plots consider only instances where at least one of the algorithms required over 30 seconds.

The cactus plots are accompanied by tables that contain for each algorithm the number of *solved* instances and the number of *wins*. For a given instance, an algorithm wins iff its runtime is not worse than the minimal time on that instance by more than 1 second. Like so the sum of wins is typically larger than the total number of considered instances. The constant of 1 second was chosen since we consider a smaller difference as insignificant, especially in the context of 800 second timeout.

7.2 Overall Results

Problem instances from practical application domains were selected from the past SAT competitions and races[3]. In total, 779 problem instances were selected. The selection was guided by the goals of finding instances that are easy for SAT solvers but are also practically motivated.

Table 2 and Figure 1 provide an overview of the results for all the considered instances. The first thing to observe is that the core-based algorithm outperforms the iterative one. Second, the effect of using chunk size $|\text{var}(\phi)|$ is detrimental for both iterative (u) and core-based algorithm (cbu).

The effect of chunk size is quite different in the two types of algorithms. For the Iterative algorithm, using chunks 30 and 100 enabled solving one more instance than the 1-Chunk configuration but the number of wins shows that the 1-Chunk was faster on a number of instances. The chunk size 500 is already too big.

In the case of core-based algorithm chunks give us unequivocally a gain (recall that Core-Based 1-Chunk is equivalent to Iterative 1-Chunk). The Core-Based 100-Chunk can solve one more instance than the Core-Based 30-Chunk. In the case of 500-Chunk, however, the performance drops. Interestingly enough, the Core-Based 500-Chunk is still better than any of the iterative configurations.

The results for the virtual best solver (VBS) show that the other algorithms enabled solving 24 more instances on top of those solved by Core-Based Chunk-100. Given this results for the VBS, a parallel portfolio using different backbone computation algorithms is expected to substantially outperform the best individual configuration.

Table 3: Categories according to backbone percentages

percentage range	number of instances
0–25%	191
25–50%	322
50–75%	123
75–100%	98

7.3 The Effect of Percentage of Backbone Literals

This section focuses on the effectiveness of the individual algorithms for different percentages of the backbone literals found in the instance. The 738 solved instances were split into four categories according to the percentage of backbone literals in the instance. Table 3 shows the number of instances in each category.

Figure 2 shows the cactus plots for each of the categories and Table 4 shows the number of solved instances and wins. Interestingly enough, the algorithms behave very similarly when the percentage of backbone is over 25%. In contrast, for the lower percentages the core-based algorithms are significantly better. This can be explained by the fact that eventually any algorithm has to prove for each backbone

[2] http://sat.inesc-id.pt/~mikolas/backbones
[3] http://www.satcompetition.org/.

Table 2: 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.

algorithm	u	cbu	500	100	30	1	cb500	cb30	cb100	VBS
solved	485	487	596	667	667	666	706	717	718	738
wins	168	183	200	220	232	282	317	434	431	—

Table 4: Behavior of the algorithms for different percentages of the backbone literals

(a) 0–25% backbone literals

algorithm	u	cbu	500	100	30	1	cb500	cb30	cb100	VBS
solved	58	60	108	124	124	119	161	171	172	191
wins	13	21	15	13	17	29	61	67	96	—

(b) 25–50% backbone literals

algorithm	u	cbu	500	100	30	1	cb500	cb30	cb100	VBS
solved	210	213	267	321	321	322	322	322	322	322
wins	17	25	27	44	57	88	99	205	191	—

(c) 50–75% backbone literals

algorithm	u	cbu	500	100	30	1	cb500	cb30	cb100	VBS
solved	119	114	122	122	123	123	123	123	123	123
wins	62	64	76	84	84	94	89	97	99	—

(d) 75–100% backbone literals

algorithm	u	cbu	500	100	30	1	cb500	cb30	cb100	VBS
solved	94	96	95	96	95	98	96	97	97	98
wins	72	70	79	80	76	81	81	86	82	—

literal that it is indeed in the backbone. Hence, for large backbones the "effort" is the same. For small backbones, however, an algorithm can gain by excluding non-backbones by being clever in the search for implicants that reduce the backbone estimate.

It is worth noting that the core-based algorithm dominates all categories except the last one. Both in terms of speed and the number of solved instances. In the category 75–100%, the configuration Iterative 1-Chunk solves one more instance than the configurations Core-Based 30/100-Chunk.

7.4 The Effect of Chunk Size

In the previous sections we have seen that increasing the chunk size in the iterative algorithm decreases its speed. However, we expect that larger the chunk, the less SAT calls the algorithm needs to perform. Hence, if the overall solving time is larger, it must mean that the SAT calls take longer.

Figure 3 shows the correlation between the number of SAT calls and the average time of the calls. The correlation is shown between 1 and 100 Chunk in the case of the iterative algorithm and between 30 and 500 Chunk for the Core-Based algorithm.

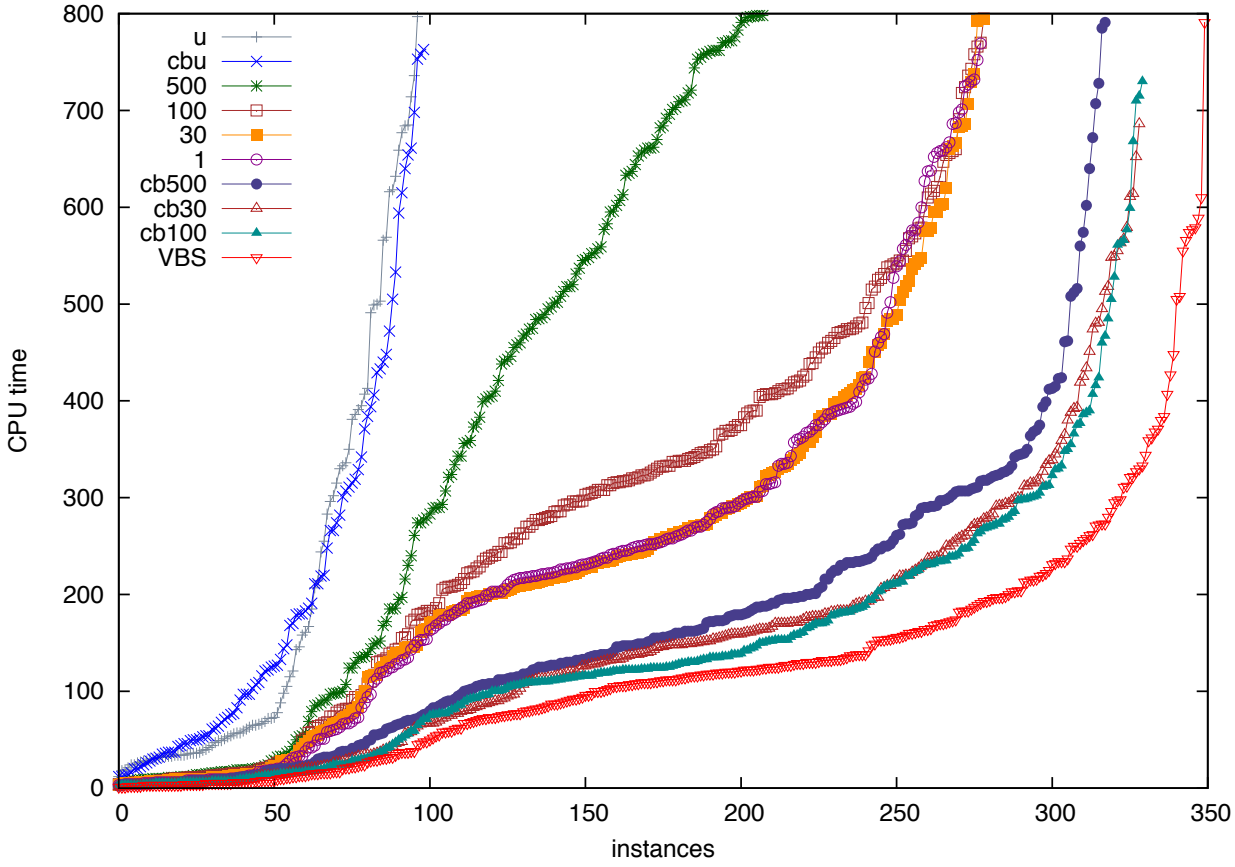


Figure 1: Cactus plot for hard instances, i.e. instances where at least one algorithm took over 30 s

Fig. 3(a) confirms our hypothesis. The Iterative 100-Chunk performs less number of SAT calls than the 1-Chunk but their average time grows significantly.

The behavior of the core-based algorithm, depicted in Fig. 3(b), is again quite different from the iterative algorithm. The average SAT execution times lie in the same area for the two configurations but the number of SAT calls tends to be smaller for the 30-Chunk configuration. This can be explained by the fact that the 500-Chunk configuration needs to do a large number of unsatisfiable calls before it reaches a satisfiable one.

7.5 The Effect of Backbone Filtering

Finally, we show the effect of backbone filtering (Section 5) on two representative configurations, Iterative 1-Chunk and Core-Based 100-Chunk. The results are again presented in the form of a cactus plot (Figure 4) and a table (Table 5).

The results show that in general, backbone filtering is not significantly helpful. As expected, implicate reduction realized by approximate set covering reduction significantly underperforms rotatable literals. However, the rotatable literals technique yields generally a slower algorithm than the one without it. This can be explained by several reasons that most likely occur in tandem. Firstly, even though

(e) 1-Chunk

algorithm	1	1r	1l	VBS
solved	666	665	521	680
wins	615	287	174	—

(f) Core-Based 100-Chunk

algorithm	cb100	cb100r	cb100l	VBS
solved	718	714	699	730
wins	642	326	202	—

Table 5: Backbone Filtering

the complexity of backbone filtering is polynomial, it is still expensive for large formulas and it might be effectively slowing down the computation. Secondly, many industrial CNF formulas contain many binary clauses, which reduce the likelihood of rotatable literals or implicant reduction to be effective. Thirdly, within the described algorithms, a backbone upper bound is being refined through implicants obtained from the SAT solver; this is computationally cheap, because the SAT calls need to be carried out in either way, and since many SAT calls are performed, many different implicants are obtained and this in turn is downplaying the effectiveness of backbone filtering.

However, as the virtual best solver shows, there are some instances where backbone filtering is important. Indeed, in the case of Iterative 1-Chunk, 14 more instances were solved and in the case of Core-Based 100-Chunk, 12 more instances were solved.

8 Conclusions

This paper develops improvements to algorithms for backbone computation. Whereas some of the algorithms are based on earlier work [22, 37], others are novel, and unify some of the most representative earlier algorithms. In addition, the paper extends a comprehensive experimental study of backbones on instances of SAT coming from practical applications.

The experimental results suggest that the novel Core-Based algorithm is the most efficient. However, the unified iterative algorithm developed in this paper, can provide performance gains for some problems instances. Thus, opening a possibility for portfolio-based solvers. In addition, the experimental results show that the proposed algorithms enable computing the backbone for large instances of SAT, with variables in excess of 70,000 and clauses in excess of 250,000.

This observation motivates further work on applying backbone information for solving decision and optimization problems related to propositional theories, including model enumeration, minimal model computation and prime implicant computation. Moreover, more efficient backbone computation algorithms are expected to impact practical applications [19, 16, 14, 36, 21, 20].

Future improvements to backbone computation algorithms include automatic identification of chunk size and parallel portfolios with different chunk sizes. Finally, the integration of additional implicant 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. Achlioptas and F. Ricci-Tersenghi. Random formulas have frozen variables. *SIAM J. Comput.*, 39(1):260–280, 2009.

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

[3] 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.

[4] Y. Boufkhad and O. Dubois. Length of prime implicants and number of solutions of random CNF formulae. *Theor. Comput. Sci.*, 215(1-2):1–30, 1999.

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

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

[7] 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.

[8] N. Eén and N. Sörensson. An extensible SAT-solver. In E. Giunchiglia and A. Tacchella, editors, *SAT*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2003.

[9] J. W. Freeman. *Improvements To Propositional Satisfiability Search Algorithms*. PhD thesis, University of Pennsylvania, 1995.

[10] 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.

[11] Y. Hamadi, S. Jabbour, and L. Sais. ManySAT: a parallel SAT solver. *JSAT*, 6(4):245–262, 2009.

[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] F. Hutter, H. H. Hoos, K. Leyton-Brown, and T. Stützle. ParamILS: An automatic algorithm configuration framework. *J. Artif. Intell. Res. (JAIR)*, 36:267–306, 2009.

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

[15] M. Janota. *SAT Solving in Interactive Configuration*. PhD thesis, University College Dublin, Nov. 2010.

[16] 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.

[17] 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.

[18] 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.

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

[20] F. Lonsing and A. Biere. Failed literal detection for QBF. In Sakallah and Simon [30], pages 259–272.

[21] P. Manolios and V. Papavasileiou. Pseudo-boolean solving by incremental translation to SAT. In P. Bjesse and A. Slobodová, editors, *FMCAD*, pages 41–45. FMCAD Inc., 2011.

[22] J. Marques-Silva, M. Janota, and I. Lynce. On computing backbones of propositional theories. In *ECAI*, pages 15–20, 2010.

[23] J. Marques-Silva, M. Janota, and I. Lynce. Experimental analysis of backbone computation algorithms. In *International Workshop on "Experimental Evaluation of Algorithms for solving problems with combinatorial explosion", RCRA*, 2012.

[24] 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.

[25] K. L. McMillan. Applying SAT methods in unbounded symbolic model checking. In E. Brinksma and K. G. Larsen, editors, *CAV*, volume 2404 of *Lecture Notes in Computer Science*, pages 250–264. Springer, 2002.

[26] M. E. Menaï. A two-phase backbone-based search heuristic for partial MAX-SAT - an initial investigation. In *Industrial and Engineering Applications of Artificial Intelligence and Expert Systems*, pages 681–684, 2005.

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

[28] A. Nadel. Generating diverse solutions in SAT. In Sakallah and Simon [30], pages 287–301.

[29] 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.

[30] K. A. Sakallah and L. Simon, editors. *Theory and Applications of Satisfiability Testing - SAT 2011 - 14th International Conference, SAT 2011, Ann Arbor, MI, USA, June 19-22. Proceedings*, volume 6695 of *Lecture Notes in Computer Science*. Springer, June 2011.

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

[32] L. Xu, H. H. Hoos, and K. Leyton-Brown. Hierarchical hardness models for SAT. In C. Bessiere, editor, *CP*, volume 4741 of *Lecture Notes in Computer Science*, pages 696–711. Springer, 2007.

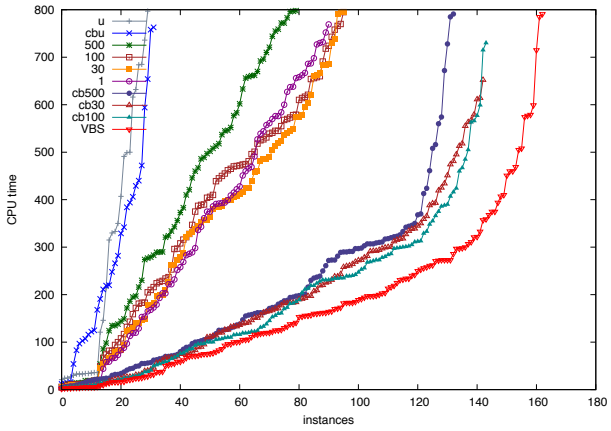
[33] L. Xu, F. Hutter, H. H. Hoos, and K. Leyton-Brown. SATzilla: Portfolio-based algorithm selection for SAT. *J. Artif. Intell. Res. (JAIR)*, 32:565–606, 2008.

[34] 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.

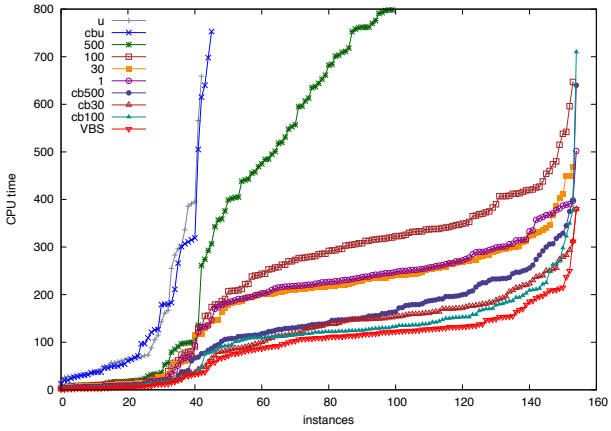
[35] W. Zhang, A. Rangan, and M. Looks. Backbone guided local search for maximum satisfiability. In *International Joint Conference on Artificial Intelligence*, pages 1179–1186, 2003.

[36] 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.

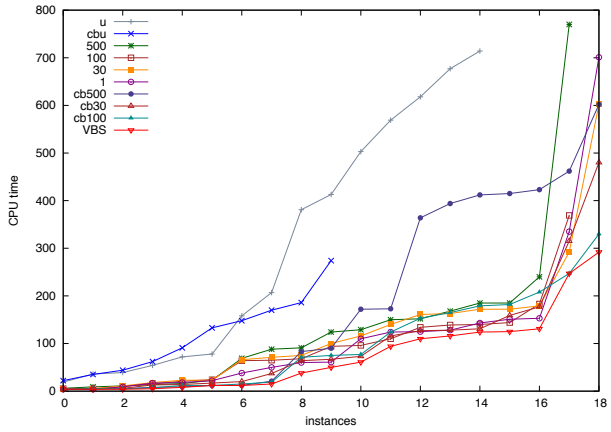
[37] C. Zhu, G. Weissenbacher, D. Sethi, and S. Malik. SAT-based techniques for determining backbones for post-silicon fault localisation. In *High Level Design Validation and Test Workshop (HLDVT)*, pages 84–91, 2011.



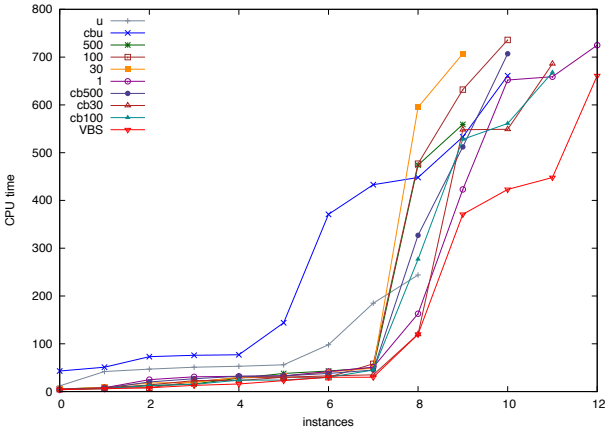
(a) 0–25% backbone literals



(b) 25–50% backbone literals

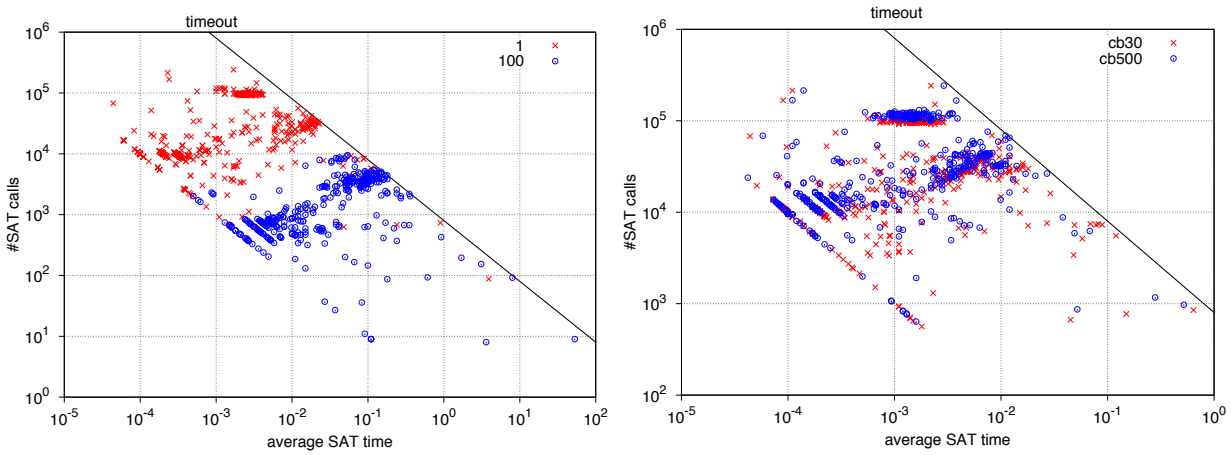


(c) 50–75% backbone literals



(d) 75-100% backbone literals

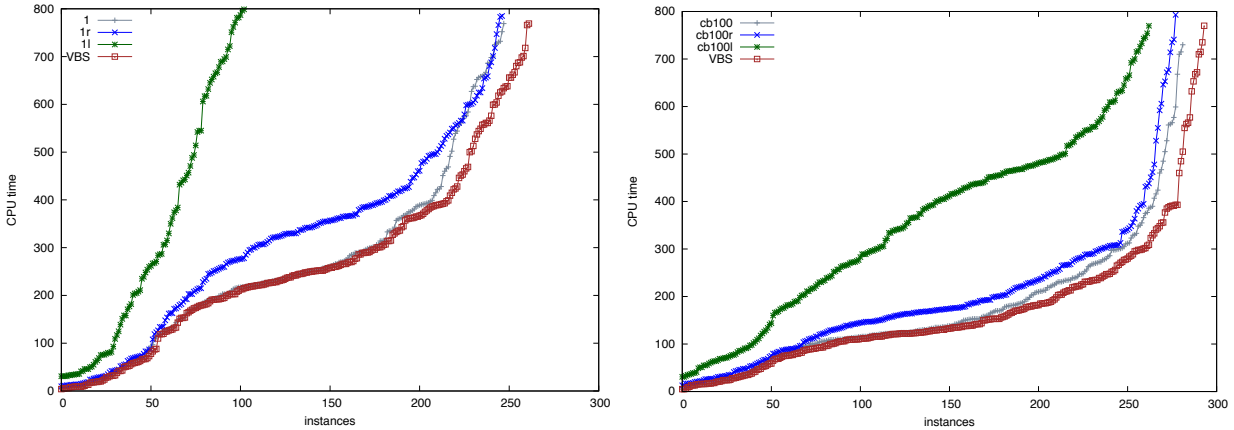
Figure 2: Behavior of the algorithms for different percentages of the backbone literals



(a) 1 vs 100 Chunk

(b) Core-Based, 30 vs 500 Chunk

Figure 3: Correlation between average SAT time and number of SAT calls



(a) 1-Chunk

(b) Core-Based, 100-Chunk

Figure 4: Filtering