CadiBack: Extracting Backbones with CaDiCaL

Armin Biere
Universität Freiburg, Germany

Nils Froleyks
Johannes Kepler Universität, Linz, Austria

Wenxi Wang
University of Texas at Austin, TX, USA

Abstract

The backbone of a satisfiable formula is the set of literals that are true in all its satisfying assignments. Backbone computation can improve a wide range of SAT-based applications, such as verification, fault localization and product configuration. In this tool paper, we introduce a new backbone extraction tool called CadiBack. It takes advantage of unique features available in our state-of-the-art SAT solver CaDiCaL including transparent inprocessing and single clause assumptions, which have not been evaluated in this context before. In addition, CaDiCaL is enhanced with an improved algorithm to support model rotation by utilizing watched literal data structures. In our comprehensive experiments with a large number of benchmarks, CadiBack solves 60% more instances than the state-of-the-art backbone extraction tool MiniBones. Our tool is thoroughly tested with fuzzing, internal correctness checking and cross-checking on a large benchmark set. It is publicly available as open source, well documented and easy to extend.

2012 ACM Subject Classification Theory of computation → Automated reasoning

Keywords and phrases Satisfiability, Backbone, Incremental Solving

Digital Object Identifier 10.4230/LIPIcs.SAT.2023.3

Supplementary Material *Software*: `https://github.com/arminbiere/cadiback`
archived at `swh:1:dir:9e9c1ac209792174194cde74eb3a385f48ceade2`

1 Introduction

In 1997, Parkes first defined the backbone of a propositional formula as the set of literals whose assignments are true in every satisfying assignment [24]. The size of the backbone is associated with the hardness of the corresponding propositional problem [23, 27]. Usually, the larger a backbone, the more tightly constrained the problem becomes, thus the harder for the solver to find a satisfying assignment [11, 30]. It is proved by Janota that deciding if a literal is in the backbone of a formula is co-NP complete [17]. Furthermore, Kilby et al. show that even approximating the backbone is intractable in general [20].

Nevertheless, the identification of the backbone (either in a partial or a complete way) has a number of practical applications, such as post-silicon fault localization in integrated circuits [34, 36, 35], interactive product configuration [17], facilitating the solving efficiency of MaxSAT [14, 28, 29, 31] and random 3-SAT problems [12], as well as improving the performance of chip verification [26]. Motivated by the wide range of applications, developing efficient algorithms for computing the backbone of a given propositional formula is important.

Indeed, numerous techniques to compute the backbone have been proposed during the past few decades. These approaches make use of four main techniques: (*i*) model enumeration, which enumerates all models of a satisfiable formula to identify the backbone; (*ii*) iterative SAT testing, which repeatedly filters out a candidate or include it in the backbone; (*iii*) upper bound checks, which try to identify multiple backbone literals at once; and (*iv*) the core-based method, which is guided by unsatisfiable cores and tries to eliminate as many candidates at once as possible. For example, Kaiser et al. [19] designed three model-enumeration algorithms. Climer et al. [10] propose a graph-based iterative SAT testing approach.

Later, Zhu et al. [34, 36] designed more efficient SAT testing approaches for post-silicon fault localization. Note that, the backbone extractor MiniBones [18, 22] implements both an iterative and a core-based approach. Despite recent attempts [25, 32, 33] to improve upon MiniBones, the corresponding tools are not publicly available, and no significant advances have been made so far which still leaves MiniBones as the state-of-the-art.

Our new backbone extractor CadiBack tries to improve the iterative algorithms of MiniBones [18, 22] and uses the state-of-the-art SAT solver CaDiCaL [4], extended with new flipping algorithms to support backbone extraction. Different configurations of these algorithms are implemented inside CadiBack, empirically evaluated and compared with MiniBones on a large set of satisfiable instances collected from the main track of the SAT Competitions from 2004 - 2022, on which CadiBack solves 60% more instances.

The paper is structured as follows. After this introduction, we discuss basic concepts and notations related to backbone extraction in Section 2. The relevant backbone extraction algorithms of MiniBones are introduced in Section 3. We then present our improvements over these algorithms and propose CadiBack in Section 4. The implementation details of CadiBack are provided in Section 5. Finally, we empirically evaluate CadiBack in Section 6 and draw conclusions in Section 7.

2 Basic Concepts and Notations

Consider a propositional *formula* φ in conjunctive normal form (CNF) over a fixed set of variables $\mathcal{V}$ and literals $\mathcal{L} = \mathcal{V} \cup \overline{\mathcal{V}}$, where $\overline{\mathcal{V}} = \{\bar{v} \mid v \in \mathcal{V}\}$ denotes the negated variables. For a literal $\ell \in \mathcal{L}$, we define $v = |\ell|$ as the variable of ℓ, i.e., $\ell \in \{v, \bar{v}\}$. In this paper, we mainly consider full *assignments* $\sigma \colon \mathcal{V} \to \{0, 1\}$ assigning variables to Boolean constants "0" (*false*) or "1" (*true*). For convenience, we use the set and logic notation interchangeably for formulas φ, clauses $C \in \varphi$ and literals $\ell \in C$, as well as assignments $\{\ell \mid \sigma(\ell) = 1\}$. The notion of assignments is lifted to literals, formulas and clauses in the natural way through substitution followed by Boolean expression simplification. A *model* of φ is an assignment σ with $\sigma(\varphi) = 1$ and also called satisfying assignment. A formula is *satisfiable* if it has a model. Otherwise, it is *unsatisfiable*. In this paper, we focus on satisfiable formulas φ.

A literal ℓ is a *backbone literal* of a formula φ iff there exists a model σ of φ with $\sigma(\ell) = 1$ and all other assignments σ' with $\sigma'(\ell) = 0$ do not satisfy φ, i.e., $\sigma'(\varphi) = 0$. The *backbone* $\mathcal{B}$ of a formula φ is the set of its backbone literals. We introduce two conditions that determine whether literals are included or not in the backbone $\mathcal{B}$.

The first condition is based on identifying *fixed* literals. A clause $C = \ell$ (or $C = \{\ell\}$ in set notation) having a single literal ℓ is called *unit* clause. If a unit clause $C \in \varphi$, the corresponding literal ℓ is clearly a backbone literal. We call such literals *fixed*. This also applies to unit clauses deduced by the SAT solver through for instance clause learning, simplification and preprocessing [7]. All such fixed literals are included in the backbone $\mathcal{B}$.

The second condition is called *disagreement condition*, stating that if there are two models σ and σ' *disagreeing* on ℓ, i.e., $\sigma'(\ell) = \overline{\sigma(\ell)}$, then neither ℓ nor its negation are backbone literals (i.e., $\ell, \bar{\ell} \notin \mathcal{B}$). This can be realized by using each newly discovered model σ' to *filter* the list of remaining backbone candidates. For instance, the empty formula over n variables has both constant assignments $\sigma \equiv 0$ and $\sigma' \equiv 1$ as models, disagreeing on all literals, and thus $\mathcal{B} = \emptyset$. Note that, there is a special case of the disagreement condition called *model rotation*, as described in [18]. Similar ideas have been used for MUS extraction [1]. The literal ℓ is *rotatable* [18] in a model σ of φ iff $\sigma(\ell) = 1$ and the assignment τ that differs from σ only in $|\ell|$ is a model of φ (τ can be taken as the special case of σ' in the disagreement condition). We also call such literals *flippable*, which applies for the rest of the paper.

Obviously, a literal which can be flipped is not a backbone literal, nor is its negation, and both can be dropped from the backbone candidate list. Example 1 below shows how a literal is determined to be flippable under the model rotation condition.

▶ **Example 1.** Consider $\varphi = (\bar{c} \vee t) \wedge (c \vee e) \wedge \varphi'$ which encodes "if-then-else(c, t, e)", where neither c nor $\bar{c}$ occur in φ' but e and t do (they are not "pure"). Assume that the constant true assignment $\sigma \equiv 1$ is a model, i.e., $\sigma(\varphi') = 1$. Both t and e are set to true, but only the literal c can be flipped. In the resulting model τ, all variables are set to true except for c, and $\bar{c}$ can be flipped (back) in τ to obtain the original model σ. Thus, literal c is flippable.

3 Algorithms in MiniBones

The backbone extraction algorithms of MiniBones [18] take advantage of incremental SAT solving (refer to [13, 15, 16] for details) to gradually augment the original formula with implied clauses (particularly learned clauses). These clauses are added implicitly to the single SAT solver instance during incremental queries, while assuming the negation of one or more remaining backbone candidate literals. Specifically, these iterative MiniBones algorithms (Algorithms 3, 4 and 5 in [18]) utilize discovered models and model rotation to refine the set of candidate literals $\Lambda \subseteq \mathcal{L}$ which is initialized as $\Lambda = \{\ell \mid \sigma(\ell) = 1\}$ by the first discovered model σ. On termination ($\Lambda = \emptyset$), the backbone $\mathcal{B}$ matches the fixed literals (of the augmented formula) and all other literals are dropped.

There are three iterative algorithms proposed for MiniBones in [18]. The basic algorithm (Algorithm 3 in [18]) needs at least as many iterations as the number of backbone literals, which is inefficient on formulas with exactly one solution but many variables. An improved algorithm (Algorithm 4 in [18]) assumes that at least one of the remaining candidate literals can be flipped (i.e., using activation literals a temporary clause is added that contains the disjunction of the negated candidates). If the SAT query under such assumption is unsatisfiable, all candidates are fixed and the backbone extraction is done. A more advanced algorithm (Algorithm 5 in [18]) only adds a subset of the remaining candidates, called a *chunk*, to the temporary clause. Chunks are limited in size to avoid thrashing the SAT solver with too large temporary clauses and make it more likely for a call to be unsatisfiable.

Furthermore, MiniBones proposes a new model rotation algorithm (Section 5 in [18]) to determine flippable (rotatable) literals based on the notion of *forcing*. A clause C *forces* a literal $\ell \in C$ under assignment σ, if $\sigma(C) = \sigma(\ell) = 1$ and $\tau(C) = 0$ with τ obtained from σ by flipping ℓ. A literal ℓ is *forced* in a formula φ under a model σ, if there is a clause $C \in \varphi$ which forces ℓ under σ. It is straightforward to see that literals which can be flipped in a model σ of φ are exactly those that are not forced. Based on this observation, the model rotation algorithm goes over all clauses whenever a new model is found and identifies literals that are not forced by any of them. If any of the remaining backbone candidates are not forced, they are dropped from the candidate list.

4 Improved Algorithms in CadiBack

CadiBack is built upon the state-of-the-art SAT solver CaDiCaL [4] which has been extended with additional algorithms to support backbone extraction. The general backbone extraction algorithm of CadiBack is shown in Algorithm 1 of Figure 1. It follows the iterative algorithms of MiniBones, which uses complements of backbone estimates (as constraints) and chunking, but with three key improvements.

```
// Assume φ is satisfiable and use
// K = 1   for one-by-one,
// K = 10 for chunking and
// K = ∞ as default (non-chunking).
backbone (CNF φ, chunk rate K = ∞)
1  (res, σ) ← SAT(φ)
2  assert σ(φ) = res = 1     // φ satisfiable!
3  Λ ← {ℓ ∈ σ | ¬flippable(ℓ, σ)}  // candidates
4  k ← 1,    B ← ∅
5  while Λ ≠ ∅ do
       // F ← ∅ for no-fixed, otherwise by default
6      F ← {ℓ ∈ Λ | ℓ is fixed by SAT in φ}
7      B ← B ∪ F,    Λ ← Λ \ F
8      Γ ← pick k' literals from Λ     // chunk
               with k' = min(k, |Λ|)
9      ρ ← ⋁_{ℓ∈Γ} ℓ̄  // constraint: flip one in chunk
       // Solve φ under ρ with "bool constrain"
       // or use activation literal for no-constrain.
10     (res, σ) ← SAT(φ | ρ)
11     if res then    // SAT call satisfiable
           // filter only a single literal for no-filter
12         Λ ← {ℓ ∈ Λ | σ(ℓ)}
13         Λ ← {ℓ ∈ Λ | ¬flippable(ℓ, σ)}
14         k ← 1  // reset chunk size to 1
15     else              // SAT call unsatisfiable
16         B ← B ∪ Γ
17         Λ ← Λ \ Γ
18         k ← K · k  // increase size geometrically
19 return B    // or print when literal is added
```

Algorithm 1 Extracting backbone of formula φ.

```
// Assume σ(φ) = σ(ℓ) = 1, unit clauses
// have exactly one watched literal
// and all other clauses are watched
// by two literals w1 ≠ w2 with
// σ(w1) = 1 if σ(w2) = 0 and vice versa.
flippable (CNF φ, literal ℓ, model σ)
1  // return 0 for no-flip
2  for all clauses C watched by ℓ in φ
3      if σ(C \ {ℓ}) = 0 then return 0
4  return 1
```

Algorithm 2 Checking if literal ℓ can be flipped in model σ.

```
// Given a single clausal constraint
// ρ = ℓ1 ∨ ··· ∨ ℓk and assignment σ
// determine whether ρ is conflicting.
// Otherwise pick new decision.
decide (constraint ρ, partial model σ)
1  ... // handle literal assumptions
2  if σ(ρ) = 1 then // constraint true
3      ℓ ← "first" literal in ρ with σ(ℓ)
       // speed-up future search for ℓ
4      move ℓ to the front of ρ
5  elif σ(ρ) = 0 then // constraint false
6      ... // handle conflicting constraint
7  else    // constraint undetermined
8      ℓ ← highest scored literal in σ(ρ)
9      pick ℓ as new decision and return
10 ... // fall back to default decisions
```

Algorithm 3 Picking the next decision literal under clausal constraint ρ and the partial model σ.

Figure 1 Our backbone algorithm combines all three iterative approaches from [18]. It simulates the basic iterative Algorithm 3 in [18] for $K = 1$ and comes close to the improved Algorithm 4 in [18] for $K > |\Lambda|$ and the most advanced Algorithm 5 in [18] for other values of K. The difference between our algorithm and the latter two is that we use a dynamic chunk size that is reset to 1 after a satisfiable call and grows geometrically as long SAT queries remain unsatisfiable. In any case, it first identifies an initial model σ and initializes the set of candidates Λ after filtering out flippable literals F. The remaining candidates are examined in chunks Γ. If all of the literals in the chunk are backbones, the chunk size is increased. Otherwise, the solver returns a new model σ which is used to filter the candidate list, as it is guaranteed to disagree with the previous model in at least one of the literals in the current chunk by assuming the constraint ρ. After that, another model rotation is performed and the chunk size is reset to 1. Note that, instead of including explicit insertions of backbones, we can assume that the SAT solver does the insertion implicitly. Flippable literals are identified by the new flippable algorithm which only traverses clauses watched by the remaining backbone candidates. The decide algorithm is an optimized version of the decision procedure in our SAT solver for more efficiently handling large constraints as they arise in this application, which picks the literal with the highest variables scores (i.e., EVSIDS scores [2] or VMTF stamps [5]).

First, CadiBack uses transparent incremental inprocessing [15], as CaDiCaL is able to effectively and efficiently simplify the formula (e.g., using variable elimination) during incremental queries completely transparent to the user, while MiniBones does not support inprocessing due to the limitation of its base solver MiniSat [13].

Second, to assume the disjunction of the complements, CadiBack utilizes single clause assumptions through the "`void constrain (int lit)`" API call in CaDiCaL [16], instead of adding a clause with the complement literals and an activation literal [13], as in MiniBones. The reason is that these added clauses and variables by MiniBones may risk to clog the SAT solver, and handling constraints explicitly can have the benefit to give the SAT solver more control on selecting decisions. Since the assumed clausal constraint contains a high number of literals in this application ($|\mathcal{V}|$ initially), we extended the existing implementation of single clause assumptions in CaDiCaL slightly, as shown in Algorithm 3 in Figure 1. After each restart the SAT solver is forced to decide on a literal to satisfy the constraint. CadiBack chooses the one with the highest variable score (EVSIDS scores [2] or VMTF stamps [5]) among all unassigned literals in the constraint.

Third, while in earlier work model rotation only had negative effects on MiniBones [18], we show that CadiBack benefits from using model rotation to improve efficiency of backbone computation. The key of this improvement is our fast flipping algorithm implemented in CaDiCaL, accessible through the new API call "`bool flippable (int lit)`". As described in Algorithm 2 in Figure 1, it uses watch lists to find individual "flippable" literals in models through propagation instead of going over the whole formula to find unit clauses. We also consider a variant of Algorithm 2 which eagerly flips flippable literals as they are found, with the goal to drop even more backbone candidates through flipping. The following example shows the possibility that flipping a flippable literal can yield additional flippable literals.

▶ **Example 2.** Continuing Example 1, assume that no clause in φ' forces literal t under $\sigma \equiv 1$, which is not the case for the first clause $(\bar{c} \vee t)$ in φ, as it forces t under σ. Thus, t cannot be flipped in σ. As c does not occur in φ', there is no clause forcing $\bar{c}$ under τ. In addition, the only other clause $(\bar{c} \vee t)$ with t is not forcing as it is satisfied by two literals. Thus, flipping c makes t flippable in τ (τ' obtained from τ by flipping t remains a model of φ). Therefore, neither c nor t are backbone literals.

To implement this idea we provide a new "`bool flip (int lit)`" API call in CaDiCaL which implements a variant of Algorithm 2 in Figure 1, inspired by propagation in SAT solvers. While for "`flippable`" we only need to check that there is another satisfied literal in all traversed clauses watched by the literal ℓ requested to be flipped, the "`flip`" implementation needs to unwatch ℓ in these clauses and watch that other satisfied literal instead (unless the second watched literal in the clause is also satisfied). If finding replacements is successful for all clauses watched by ℓ (or the other watched literal is satisfied), the value of ℓ is flipped. Otherwise, it remains unchanged and "`flip`" fails. Note that this variant of our flipping algorithm was previously implemented inside the sub-solver Kitten of Kissat to diversify models with the goal of speeding up the refinement process of SAT sweeping [3].

Algorithm 3 of [18] can be simulated precisely with our algorithm by setting $K = 1$. However, Algorithm 5 of [18], which uses a fixed chunk size limit can only be approximated by setting $K = 100$, as we change the chunk size k dynamically. Our adaptive scheme increases k geometrically with rate K as long as SAT queries remain unsatisfiable (which fixes all backbones in the chunk at once). If the SAT solver finds a model instead then the chunk size k is reset to one, i.e., the next constraint will only contain the negation of a single backbone candidate. With $K = \infty$ the SAT solver is either assuming the complement of a single or the disjunction of the negation of all remaining backbone candidates which is the setting of our algorithm closest to Algorithm 4 of [18] which does not limit chunk size at all.

5 Implementation

Our tool CadiBack uses the extended CaDiCaL [4] and is implemented in roughly 1200 lines of C++ code (counted after formatting with ClangFormat). The source code available at `https://github.com/arminbiere/cadiback` is concise and well-documented.

To check the correctness of algorithms and implementations, an internal backbone checker is implemented inside CadiBack. The checker can be enabled through the command line option "`--check`" and is simply a SAT solver instance of CaDiCaL, i.e., if checking is enabled, CadiBack obtains the checker instance as a copy of the main internal CaDiCaL solver through the "`copy`" API call provided by CaDiCaL.

First, it checks correctness of an identified backbone literal ℓ, by confirming that the input formula φ under the assumption $\neg\ell$ (negation of the backbone) is unsatisfiable. Second, it checks the correctness of dropping a literal ℓ from being a backbone candidate (removed from set Λ in Algorithm 1), by confirming that the input formula φ remains satisfiable under the assumption $\neg\ell$. Third, it checks whether the number of backbone literals found and the number of dropped literals sum up to the number of variables in the input formula.

Standard grammar-based black-box fuzz-testing was applied [9] with the backbone checking enabled on all 42 compatible pairs of options used in our experiments in Section 6. This pairwise combinatorial testing [21] through fuzzing was run for 50 hours in parallel using as many processes as configurations on a dual processor AMD EPYC 7343 machine (providing in total 64 virtual cores). In addition, sizes of the backbones of all our benchmarks (see Section 6) were sanity checked with the ones computed by 12 configurations of CadiBack and two configurations of MiniBones considered in our experiments.

In addition, for flipping information extraction, the library of CaDiCaL is extended to provide "`bool flippable (int)`" and "`bool flip (int)`" as discussed in the last section. The model based tester Mobical is also extended correspondingly for testing the new functionality. This extended version of CaDiCaL with improved constrain handling and flipping is also available at `https://github.com/arminbiere/cadical`.

6 Experiments

Benchmarks. To evaluate CadiBack empirically, we collected all benchmarks from the main track of the SAT competition 2004 to 2022 as our initial benchmark set. We noticed that benchmarks from one competition year often contain old benchmarks (sometimes arbitrarily renamed or commented by the competition organizers) from previous competition years. This caused our initial benchmark set to include several redundant benchmarks. To remove such duplicates, in a second step, we cleaned up each individual benchmark by removing comments using a simple DIMACS pretty printer, followed by identifying identical benchmarks through computing an MD5 checksum and removing redundant ones. Then we ran the state-of-the-art SAT solver Kissat 3.0.0 [3] with 5,000 second timeout on the no-duplicate benchmark set and selected benchmarks solved to be satisfiable. In total this yields 1798 benchmarks available at `https://cca.informatik.uni-freiburg.de/sc04to22sat.zip` (6 GB) and [6].

Baseline. We choose the state-of-the-art backbone solver MiniBones as our baseline (ported to support newer C++ compilers available at `https://github.com/arminbiere/minibones`). As suggested by [18], we use the configuration "`-e -c 100 -i`" (called minibones-core-based), which adopts the core-based approach with a fixed chunk size of 100 and inserts found backbone literals into the input formula explicitly. Additionally, to evaluate how our algorithm improves upon the Algorithm 5 in [18], we choose "`-u -c 100 -i`" (called minibones-iterative) which implements the algorithm and uses activation literals.

Platform. For benchmark collection we used a machine with an AMD Ryzen Threadripper 3970X 32-Core Processor at 4.5 GHz and 256 GB RAM. All other experiments were conducted in parallel on a cluster consisting of 32 machines, each with two 8-core Intel Xeon E5-2620 v4 CPUs running at 2.10 GHz (turbo-mode disabled) and 128 GB RAM. Each instance is allocated to one core with a timeout of 5,000 seconds and a memory limit of 7 GB.

Data Availability. Experimental data including source code and log files are available on `https://cca.informatik.uni-freiburg.de/cadiback`.

Overall Results. We run both CaDiBack and the baseline MiniBones on all benchmarks in our benchmark set. We consider an instance solved if the tool completes backbone computation, i.e., classifies all literals as either backbone or non-backbone. The number of instances solved over time are presented in Figure 2. It turns out that the best performing default configuration (default) of CaDiBack can solve 732 instances in total, which is 274 more (59.82%) than the best performing configuration of MiniBones, i.e., the iterative configuration minibones-iterative, which solves only 458 instances. Note that, 11 failing runs of CaDiBack and 61 failing runs of MiniBones hit the memory limit. It is also instructive to observe that over all selected 1798 instances CaDiBack was able to find the first model in 1573 cases, while MiniBones did so in only 1152 cases, which clearly shows the advantage of using CaDiCaL [4] versus MiniSat [13] in this application. It might be interesting to investigate whether this improvement transfers to other applications using MiniSat.

In addition, following the SAT manifesto v1.0 [8], we also compare the default configuration of CaDiBack with the best configuration of MiniBones on all satisfiable benchmarks in the main track of SAT competitions from 2020 to 2022. As a result, CaDiBack/MiniBones solved 22/9 instances in 2020, 52/17 in 2021 and 41/10 in 2022.

Configurations. We study the impact of different design options in CaDiBack by evaluating 12 configurations (see Figure 2) including an extension implementing the core-based approach of MiniBones (Algorithm 7 in [18]). Firstly, we observe that the effects of using smaller chunks were detrimental in our experiments. In fact, the infinite chunk size $K = \infty$ (default) has been very beneficial which solves 732 instances, while chunk size $K = 10$ (chunking) solves only 702 instances and chunk size $K = 1$ (one-by-one) even only 692.

Secondly, we study the impact of design options related to flipping. The experimental results indicate that removing flippable literals from the candidate list (no-flip) does not have a significant overall impact. This result differs from the one given by the authors of MiniBones where the model rotation was detrimental. We attribute this to the efficiency of using watch lists for the flippable check. The really-flip configuration uses "`flip`" and simply tries to flip literals of the candidate chunk in an arbitrary order. It performs similar to the default configuration which uses "`flippable`", but is better in the aspect that the default only found 30,780,841 flippable literals in total, while really-flip found 32,488,468. This directly leads to a reduction of the total number of SAT solver calls, which goes down from 2,070,166 calls in no-flip to 1,478,160 calls in default and even down to 992,404 calls in really-flip.

Thirdly, to evaluate the impact of CaDiCaL on CaDiBack in detail, including its more advanced inprocessing and its most recent "`constrain`" API to support single clause assumptions [16]. We observe that disabling the inprocessing in CaDiCaL (no-inprocessing) significantly degrades the performance from solving 732 instances to 690 instances. Disabling the single clause assumption support from CaDiCaL in configuration no-constrain and falling back to activation literals (as MiniBones does) degrades the efficiency of CaDiBack even more significantly to solving only 672 instances.

Lastly, we evaluate the impact of our core-based algorithm. The core-based preprocessing in CadiBack only solves 672 instances. However, since the core-based approach falls back to default if the considered literal set becomes empty after removing failed assumptions (see Algorithm 7 in [18] for details), our cores version is more sophisticated than MiniBones (minibones-core-based), thanks to its advanced features in default. In contrast, the core-based MiniBones configuration (minibones-core-based) is slightly better than its iterative version (minibones-iterative) for shorter run times, which matches observations of [18] with the lower time limit of 800 seconds. One can argue, that the reason probably is that the core-based algorithm in MiniBones can rely on literal assumptions [13] avoiding the overhead inflicted from adding temporary clauses and activation literals. However, this slight advantage degrades for long running instances, as can be seen in Figure 2.

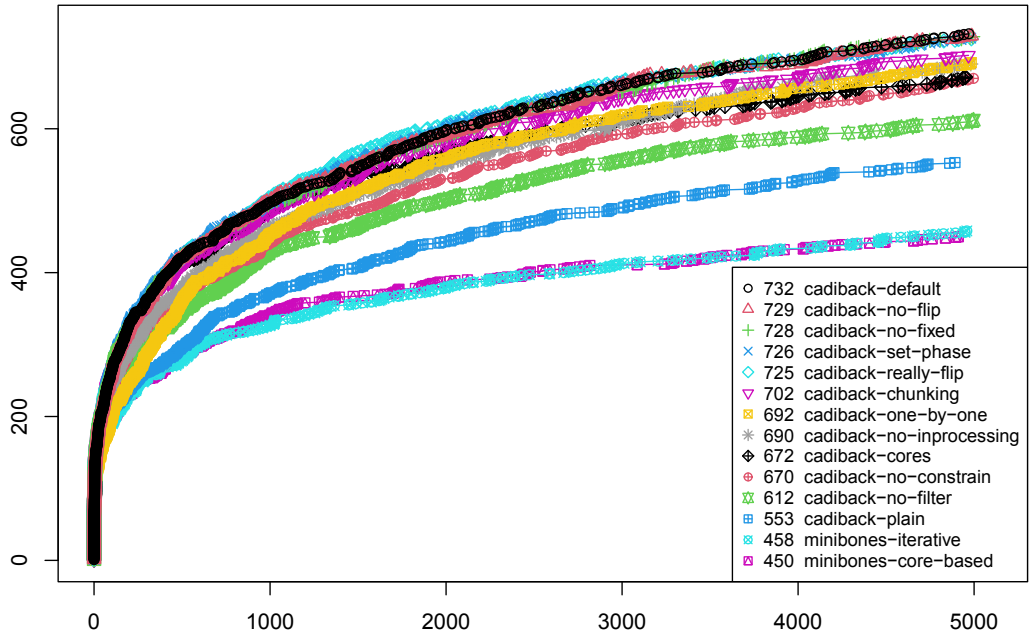

Figure 2 Benchmarks solved (vertical) over time in seconds (horizontal) where backbone extraction completed within 5,000 seconds by 12 CadiBack configurations: default denoting all optimizations enabled except for chunking and cores; no-flip denoting no model rotation; no-fixed representing no checking on candidates for being fixed explicitly; set-phase denoting picking decisions in SAT solver to falsify backbone candidates; really-flip denoting flipping flippable literals eagerly; chunking representing the fine-grained chunk size control ($K = 10$); one-by-one denoting single literal chunks ($K = 1$); no-inprocessing representing no SAT solver internal inprocessing; cores denoting core-based preprocessing; no-constrain meaning only using activation literals instead of using "`constrain`" API; no-filter disables filtering backbone candidates by the disagreement condition; and plain setting $K = 1$ (as one-by-one) and disabling all other optimizations. We also considered 2 MiniBones configurations: iterative implementing Algorithm 5 in [18]; and core-based implementing Algorithm 7 in [18].

7 Conclusion

We revisited backbone algorithms and implemented a new open-source backbone extraction tool CadiBack based on an extended version of the state-of-the-art SAT solver CaDiCaL. Our extensive evaluation on a large set of benchmarks shows a substantial performance improvement by solving 60% more benchmarks than the state-of-the-art MiniBones.

References

1 Anton Belov and João Marques-Silva. Accelerating MUS extraction with recursive model rotation. In Per Bjesse and Anna Slobodová, editors, *International Conference on Formal Methods in Computer-Aided Design, FMCAD '11, Austin, TX, USA, October 30 – November 02, 2011*, pages 37–40. FMCAD Inc., 2011.

2 Armin Biere. Adaptive restart strategies for conflict driven SAT solvers. In Hans Kleine Büning and Xishun Zhao, editors, *Theory and Applications of Satisfiability Testing – SAT 2008, 11th International Conference, SAT 2008, Guangzhou, China, May 12-15, 2008. Proceedings*, volume 4996 of *Lecture Notes in Computer Science*, pages 28–33. Springer, 2008. doi:10.1007/978-3-540-79719-7_4.

3 Armin Biere and Mathias Fleury. Gimsatul, IsaSAT and Kissat entering the SAT Competition 2022. In Tomas Balyo, Marijn Heule, Markus Iser, Matti Järvisalo, and Martin Suda, editors, *Proc. of SAT Competition 2022 – Solver and Benchmark Descriptions*, volume B-2022-1 of *Department of Computer Science Series of Publications B*, pages 10–11. University of Helsinki, 2022.

4 Armin Biere, Mathias Fleury, and Maximillian Heisinger. CaDiCaL, Kissat, Paracooba entering the SAT Competition 2021. In Tomas Balyo, Nils Froleyks, Marijn Heule, Markus Iser, Matti Järvisalo, and Martin Suda, editors, *Proc. of SAT Competition 2021 – Solver and Benchmark Descriptions*, volume B-2021-1 of *Department of Computer Science Report Series B*, pages 10–13. University of Helsinki, 2021.

5 Armin Biere and Andreas Fröhlich. Evaluating CDCL variable scoring schemes. In Marijn Heule and Sean A. Weaver, editors, *Theory and Applications of Satisfiability Testing – SAT 2015 – 18th International Conference, Austin, TX, USA, September 24-27, 2015, Proceedings*, volume 9340 of *Lecture Notes in Computer Science*, pages 405–422. Springer, 2015. doi:10.1007/978-3-319-24318-4_29.

6 Armin Biere, Nils Froleyks, and Wenxi Wang. Sampled and Normalized Satisfiable Instances from the main track of the SAT Competition 2004 to 2022, March 2023. doi:10.5281/zenodo.7750076.

7 Armin Biere, Matti Järvisalo, and Benjamin Kiesl. Preprocessing in SAT solving. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability – Second Edition*, volume 336 of *Frontiers in Artificial Intelligence and Applications*, pages 391–435. IOS Press, 2021. doi:10.3233/FAIA200992.

8 Armin Biere, Matti Järvisalo, Daniel Le Berre, Kuldeep S. Meel, and Stefan Mengel. The SAT practitioner's manifesto, September 2020. doi:10.5281/zenodo.4500928.

9 Robert Brummayer, Florian Lonsing, and Armin Biere. Automated testing and debugging of SAT and QBF solvers. In Ofer Strichman and Stefan Szeider, editors, *Theory and Applications of Satisfiability Testing – SAT 2010, 13th International Conference, SAT 2010, Edinburgh, UK, July 11-14, 2010. Proceedings*, volume 6175 of *Lecture Notes in Computer Science*, pages 44–57. Springer, 2010. doi:10.1007/978-3-642-14186-7_6.

10 Sharlee Climer and Weixiong Zhang. Searching for backbones and fat: A limit-crossing approach with applications. In *AAAI/IAAI*, pages 707–712, 2002.

11 Michael Codish, Yoav Fekete, and Amit Metodi. Backbones for equality. In Valeria Bertacco and Axel Legay, editors, *Hardware and Software: Verification and Testing – 9th International Haifa Verification Conference, HVC 2013, Haifa, Israel, November 5-7, 2013, Proceedings*, volume 8244 of *Lecture Notes in Computer Science*, pages 1–14. Springer, 2013. doi:10.1007/978-3-319-03077-7_1.

12 Olivier Dubois and Gilles Dequen. A backbone-search heuristic for efficient solving of hard 3-SAT formulae. In *IJCAI*, volume 1, pages 248–253, 2001.

13 Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003. Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2003. doi:10.1007/978-3-540-24605-3_37.

14 Mohamed El Bachir Menaï. A two-phase backbone-based search heuristic for partial MAX-SAT–an initial investigation. In *Innovations in Applied Artificial Intelligence: 18th International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems, IEA/AIE 2005, Bari, Italy, June 22-24, 2005. Proceedings 18*, pages 681–684. Springer, 2005.

15 Katalin Fazekas, Armin Biere, and Christoph Scholl. Incremental inprocessing in SAT solving. In Mikoláš Janota and Inês Lynce, editors, *Theory and Applications of Satisfiability Testing – SAT 2019 – 22nd International Conference, SAT 2019, Lisbon, Portugal, July 9-12, 2019, Proceedings*, volume 11628 of *Lecture Notes in Computer Science*, pages 136–154. Springer, 2019. doi:10.1007/978-3-030-24258-9_9.

16 Nils Froleyks and Armin Biere. Single clause assumption without activation literals to speed-up IC3. In *Formal Methods in Computer Aided Design, FMCAD 2021*, pages 72–76. IEEE, 2021. doi:10.34727/2021/isbn.978-3-85448-046-4_15.

17 Mikoláš Janota. *SAT solving in interactive configuration*. PhD thesis, University College Dublin, 2010.

18 Mikoláš Janota, Inês Lynce, and João Marques-Silva. Algorithms for computing backbones of propositional formulae. *AI Commun.*, 28(2):161–177, 2015. doi:10.3233/AIC-140640.

19 Andreas Kaiser and Wolfgang Küchlin. Detecting inadmissible and necessary variables in large propositional formulae. In *Intl. Joint Conf. on Automated Reasoning (Short Papers)*, pages 96–102. University of Siena, 2001.

20 Philip Kilby, John Slaney, Sylvie Thiébaux, Toby Walsh, et al. Backbones and backdoors in satisfiability. In *AAAI*, volume 5, pages 1368–1373, 2005.

21 D. Richard Kuhn, Dolores R. Wallace, and Albert M. Gallo. Software fault interactions and implications for software testing. *IEEE Trans. Software Eng.*, 30(6):418–421, 2004. doi:10.1109/TSE.2004.24.

22 João Marques-Silva, Mikoláš Janota, and Inês Lynce. On computing backbones of propositional theories. In Helder Coelho, Rudi Studer, and Michael J. Wooldridge, editors, *ECAI 2010 – 19th European Conference on Artificial Intelligence, Lisbon, Portugal, August 16-20, 2010, Proceedings*, volume 215 of *Frontiers in Artificial Intelligence and Applications*, pages 15–20. IOS Press, 2010.

23 Rémi Monasson, Riccardo Zecchina, Scott Kirkpatrick, Bart Selman, and Lidror Troyansky. Determining computational complexity from characteristic 'phase transitions'. *Nature*, 400(6740):133–137, 1999.

24 Andrew J. Parkes. Clustering at the phase transition. In Benjamin Kuipers and Bonnie L. Webber, editors, *Proceedings of the Fourteenth National Conference on Artificial Intelligence and Ninth Innovative Applications of Artificial Intelligence Conference, AAAI 97, IAAI 97, July 27-31, 1997, Providence, Rhode Island, USA*, pages 340–345. AAAI Press / The MIT Press, 1997.

25 Alessandro Previti and Matti Järvisalo. A preference-based approach to backbone computation with application to argumentation. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*, pages 896–902, 2018.

26 Miroslav N Velev. Formal verification of vliw microprocessors with speculative execution. In *Computer Aided Verification: 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000. Proceedings 12*, pages 296–311. Springer, 2000.

27 Ryan Williams, Carla P. Gomes, and Bart Selman. Backdoors to typical case complexity. In Georg Gottlob and Toby Walsh, editors, *IJCAI-03, Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence, Acapulco, Mexico, August 9-15, 2003*, pages 1173–1178. Morgan Kaufmann, 2003.

28 Guoqiang Zeng, Chongwei Zheng, Zhengjiang Zhang, and Yongzai Lu. An backbone guided extremal optimization method for solving the hard maximum satisfiability problem. In *2012 International Conference on Computer Application and System Modeling*, pages 1301–1304. Atlantis Press, 2012.

29 Weixiong Zhang. Configuration landscape analysis and backbone guided local search.: Part i: Satisfiability and maximum satisfiability. *Artificial Intelligence*, 158(1):1–26, 2004.

30 Weixiong Zhang. Phase transitions and backbones of the asymmetric traveling salesman problem. *Journal of Artificial Intelligence Research*, 21:471–497, 2004.

31 Weixiong Zhang, Ananda Rangan, Moshe Looks, et al. Backbone guided local search for maximum satisfiability. In *IJCAI*, volume 3, pages 1179–1186, 2003.

32 Yueling Zhang, Min Zhang, and Geguang Pu. Optimizing backbone filtering. *Science of Computer Programming*, 187:102374, 2020.

33 Yueling Zhang, Min Zhang, Geguang Pu, Fu Song, and Jianwen Li. Towards backbone computing: A greedy-whitening based approach. *AI Communications*, 31(3):267–280, 2018.

34 Charlie Shucheng Zhu, Georg Weissenbacher, and Sharad Malik. Post-silicon fault localisation using maximum satisfiability and backbones. In *2011 Formal Methods in Computer-Aided Design (FMCAD)*, pages 63–66. IEEE, 2011.

35 Charlie Shucheng Zhu, Georg Weissenbacher, and Sharad Malik. Silicon fault diagnosis using sequence interpolation with backbones. In Yao-Wen Chang, editor, *The IEEE/ACM International Conference on Computer-Aided Design, ICCAD 2014, San Jose, CA, USA, November 3-6, 2014*, pages 348–355. IEEE, 2014. doi:10.1109/ICCAD.2014.7001373.

36 Charlie Shucheng Zhu, Georg Weissenbacher, Divjyot Sethi, and Sharad Malik. SAT-based techniques for determining backbones for post-silicon fault localisation. In *2011 IEEE International High Level Design Validation and Test Workshop*, pages 84–91. IEEE, 2011.

A Appendix

This appendix provides more experimental details. The left plot in Figure 3 emphasizes why the most simplistic backbone algorithm, i.e., assuming the negation of exactly one remaining backbone candidate literal, does not scale, as it just takes way too many SAT calls.

Furthermore, in a number of applications it can be beneficial to get the backbones as soon as they are found, particularly if the backbone search does not terminate. To that end we evaluate MiniBones and CadiBack as anytime algorithms and compare the number of backbones they find over time. We modified the default configuration of MiniBones (minibones-core-based, i.e., corresponding to options "`-e -i -c 100`") to print a backbone as soon as it is found and evaluated it against the default version of CadiBack on the 2022 SAT competition benchmark set. The results are presented on the right in Figure 3.

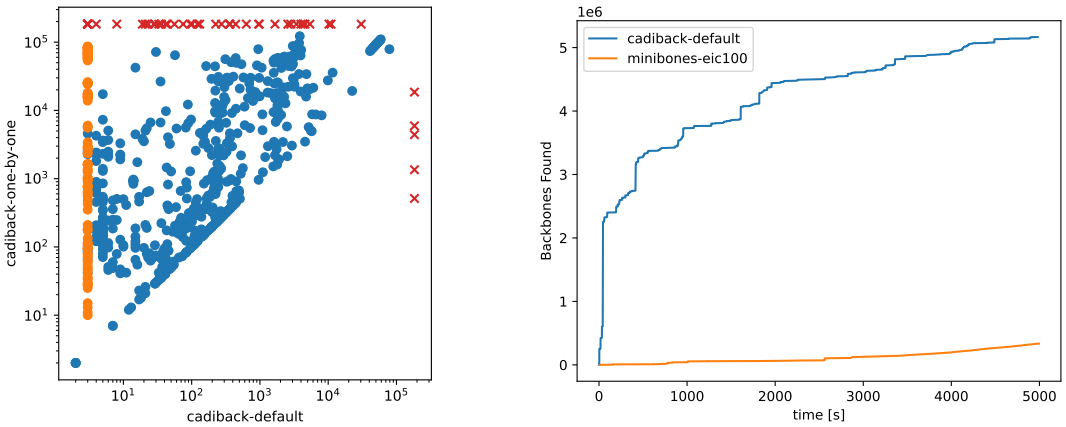

Figure 3 The *left plot* compares the one-by-one and the default configuration. Timeouts for one of the configurations are marked in the margin. Highlighted on the left are 133 (out of 1798) benchmarks that have exactly one model (every variable is in the backbone). Using an infinite chunk size (the default), such benchmarks are always solved in 3 SAT calls. The *right plot* compares MiniBones and CadiBack in an anytime setting. Shown are the number of backbones found combined across all instances in the SAT competition 2022 benchmark set.

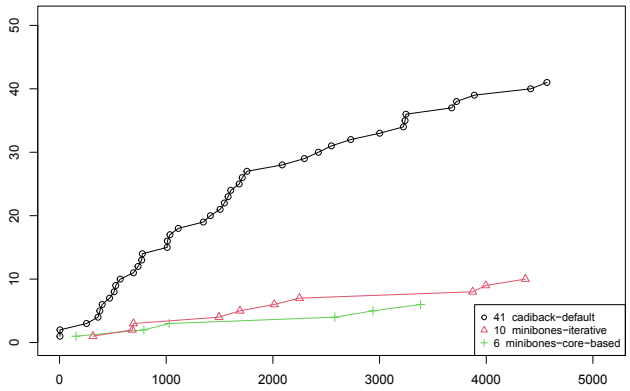

Figure 4 Benchmarks solved on satisfiable instances from the SAT Competition 2022.

Table 1 More detailed results for the runs plotted in Fig. 2 on the large SAT competition 2004–2022 benchmark set where: solved instances; failed to solved; to time out of 5,000 seconds hits; mo memory limit of 7 GB hit; time accumulated process time of solved instances (in seconds); space sum of the maximum memory usage over solved instances (in MB); max maximum memory usage on solved instances (in MB); best number of instances with best shortest solving time; unique uniquely solved number of instances. For the description of the configurations see caption of Fig. 2.

	solved	failed	to	mo	time	space	max	best	unique
cadiback-default	732	842	831	11	694 027	110 614	2600	53	1
cadiback-no-flip	729	845	837	8	686 832	103 021	2600	58	0
cadiback-no-fixed	728	846	835	11	682 242	106 129	2600	70	2
cadiback-set-phase	726	848	838	10	657 492	108 737	2565	163	4
cadiback-really-flip	725	849	838	11	640 447	105 963	2600	46	1
cadiback-chunking	702	872	861	11	630 633	93 625	2600	108	0
cadiback-one-by-one	692	882	871	11	715 101	86 152	2600	30	0
cadiback-no-inprocessing	690	884	873	11	688 418	93 947	2628	116	7
cadiback-cores	672	902	891	11	570 724	100 362	2600	78	1
cadiback-no-constrain	670	904	890	14	693 284	93 836	2546	41	0
cadiback-no-filter	612	962	951	11	562 853	72 360	2600	9	0
cadiback-plain	553	1021	1010	11	544 688	58 500	2655	13	0
minibones-iterative	458	1340	1279	61	402 645	110 138	5281	54	17
minibones-core-based	450	1348	1283	65	348 856	72 793	3542	52	2

Finally we show in Figure 4 the performance of the default version of CadiBack versus the iterative and core-based versions of MiniBones on the last SAT Competition 2022. Table 1 gives more details about the runs plotted in Fig. 2.