Efficient All-UIP Learned Clause Minimization (Extended Version)

Mathias Fleury and Armin Biere

Johannes Kepler University, Linz, Austria
{armin.biere, mathias.fleury}@jku.at

Abstract. In 2020 Feng & Bacchus revisited variants of the all-UIP learning strategy, which considerably improved performance of their version of CaDiCaL submitted to the SAT Competition 2020, particularly on large planning instances. We improve on their algorithm by tightly integrating this idea with learned clause minimization. This yields a clean shrinking algorithm with complexity linear in the size of the implication graph. It is fast enough to unconditionally shrink learned clauses until completion. We further define trail redundancy and show that our version of shrinking removes all redundant literals. Independent experiments with the three SAT solvers CaDiCaL, Kissat, and Satch confirm the effectiveness of our approach.

1 Introduction

Learned clause minimization [19] is a standard feature in modern SAT solvers. It allows to learn shorter clauses which not only reduces memory usage but arguably also helps to prune the search space. However, completeness of minimization was never formalized nor proven. Using Horn SAT [9] we define trail redundancy through entailment with respect to the reasons in the trail and show that the standard minimization algorithm removes all redundant literals (Sect. 2).

Minimization, in its original form [19], only removes literals from the initial *deduced clause* during conflict analysis, i.e., the 1st-unique-implication-point clause [22]. In 2020 Feng & Bacchus [12] revisited the *all-UIP* heuristics with the goal to reduce the size of the deduced clause even further by allowing to add new literals. In this paper we call such advanced minimization techniques *shrinking*. In order to avoid spending too much time in such shrinking procedures the authors of [12] had to limit its effectiveness. They also described and implemented several variants in the SAT solver CaDiCaL [2]. One variant was winning the planning track of the SAT Competition 2020. The benchmarks in this track require to learn clauses with many literals on each decision level.

As Feng & Bacchus [12] consider minimization and all-UIP shrinking separately, they apply minimization first, then all-UIP shrinking, and finally again minimization (depending on the deployed strategy/variant), while we integrate both techniques into one simple algorithm. In contrast, their variants process literals of the deduced clause from highest to lowest decision level and eagerly

https://doi.org/10.35011/fmvtr.2021-3
Technical Report 21/3, May 2021, FMV Reports Series
Institute for Formal Models and Verification, Johannes Kepler University
Altenbergerstr. 69, 4040 Linz, Austria

introduce literals on lower levels. Thus their approach has to be guarded against actually producing larger clauses and can not be run unconditionally (Sect. 3).

We integrate minimization and shrinking in one procedure with linear complexity in the size of the implication graph (Sect. 4). Processing literals of the deduced clause from lowest to highest level allows us to reuse the minimization cache, without compromising on completeness, thus making it possible to run the shrinking algorithm unconditionally until completion. On the theoretical side we prove that our form of shrinking fulfills the trail redundancy criteria.

Experiments with our SAT solvers Kissat, CaDiCaL, and Satch show the effectiveness of our approach and all-UIP shrinking in general. Shrinking decreases the number of learned literals, particularly on the recent planning track. We also study the amount of time used by the different parts of the transformation from a conflicting clause to the shrunken learned clause (Sect. 5).

Regarding related work we refer to the *Handbook of Satisfiability* [7], particularly for an introduction to CDCL [18], the main algorithm used by state-of-the-art SAT solvers. This work is based on the classical minimization algorithm [19], which Van Gelder [20] improved by making it linear (in the number of literals) in the implication graph without changing the resulting minimized clause. The original all-UIP scheme [22] was never considered to be efficient enough to be part of SAT solvers, until the work by Feng & Bacchus [12]. We refer to their work for a detailed discussion on all-UIPs. Note that, Feng & Bacchus [12] consider their algorithm to be independent of minimization, more like a post-processing step, while we combine shrinking and minimization for improved efficiency. This is the extend version (with the proofs) of our SAT paper [14]

2 Minimization

We first present a formalization of what minimization actually achieves through the notion of "trail redundancy". Then the classical deduced clause minimization algorithm is revisited. It identifies literals that are removable and others literals called *poison* that are not. The algorithm uses a syntactic criterion, but removes exactly the trail redundant literals. We present five existing criteria to detect (ir)redundancy earlier and prove their correctness.

When a SAT solver identifies a conflicting clause, i.e., a clause in which all literals are assigned to false, it analyzes the clause and first deduces a 1st-unique-implication-point clause [18, 22]. This *deduced clause* is the starting point for minimization and shrinking. The goal is to reduce the size of this clause by removing as many literals as possible. The following redundancy criterion specifies if a literal is removable from the deduced clause.

Definition 1 (Semantic Trail Redundancy). *Given the formula F_M composed only of the reason annotating propagated literals in the trail M and the conflicting clause D such that $M \vDash \neg D$. The literal $L \in \neg M$ is called* redundant *iff* $F_M \vDash \neg L \vee (D \setminus \{L\})$.

For this definition we only consider redundancy with respect to the reasons in the trail (ignoring other clauses in the formula). Note that, most SAT solvers only use the first clause in the watch lists to propagate, even though "better" clauses might trigger the same propagation. For instance PrecoSAT scans watch lists to find such cases [3]. However, due to potential cyclic dependencies, deducing the shortest learned clause is difficult [21].

Theorem 2 (Redundant Literals are Removable). *If $L \vee D$ is the deduced clause and L is redundant, then D is conflicting and entailed.*

Proof. Since $F_M \subseteq F$ we can apply self-subsuming resolution [10] on the deduced clause, applying the redundancy criterion on L. □

Our next theorem states that the order of removal does not impact the outcome and that it is possible to cache whether a literal is (ir)redundant.

Theorem 3. *Literals stay (ir)redundant after removal of redundant literals.*

Proof. By self-subsuming resolution on the removed literals. □

The reason $L \vee C$ annotates the propagation literal $L^{L \vee C}$ in the trail. Minimizing the deduced clause consists in recursively resolving with the reasons: If the clause becomes smaller, it is used. Duplicate literals are removed from the clause. Algorithm 1 shows a recursive implementation that resolve away the literal L without addition of literals. The minimization algorithm applies to every conflicting clause but is only applied to the *deduced clause* [22], namely the deduced clause after the first unique implication point was derived.

The minimization algorithm is standard in SAT solvers with several improvements. First, they use efficient data structures to efficiently check if a literal is in the deduced clause. Second, they use caching: if a literal was deemed (un)removable before, the same outcome is used again. Caching successes and failures [20] make the algorithm linear in the size of the implication graph. Literals that can not be removed are called *poison*.

Our definition of trail redundancy is semantic, while the minimization algorithm uses relies on syntactic criteria to determine if a literal is removable or not. We show that both criteria are equivalent by using a result of Horn satisfiability.

Definition 4 (Transition System by Dowling and Gallier [9]). *Consider the following rewriting system defined for Horn formulas, starting from the start symbol I*

1. *For every clause $L \vee \neg L_1 \vee \cdots \vee \neg L_n$, we consider the associated rewrite rule $\neg L \to \neg L_1 \cdots \neg L_n$ (where n can be zero).*
2. *For every clause $\neg L_1 \vee \cdots \vee \neg L_n$, we consider the rewrite rule $I \to \neg L_1 \cdots \neg L_n$.*

In Definition 4, given our SAT context the step $\neg L_1 \cdots \neg L_n$, represents the entailed clause $\neg L_1 \vee \cdots \vee \neg L_n$. One rewriting step is a resolution step.

Theorem 5 (Dowling and Gallier [9]). *Given a satisfiable Horn formula, a literal is true iff it can be rewritten to $\bot$.*

Function `IsLiteralRedundant`(L, d, C)

Input: Literal L assigned to *true*, recursion depth d, deduced clause C
Output: Whether L can be removed

if L is a decision **then**
  return *false*
$D \vee L \longleftarrow$ reason(L);
foreach *literal* $K \in D$ **do**
  if $\neg$`IsLiteralRedundant`($\neg K$, $d+1$, C) **then**
    return *false*
return *true*

Function `MinimizeSlice`(B, C)

Input: A clause C (passed by reference) and a subset B of C to minimize
Output: The minimized clause with redundant literals in B removed

foreach $K \in B$ **do**
  $R \longleftarrow \emptyset$
  if `IsLiteralRedundant`($\neg K$, 0, C) **then**
    $R \longleftarrow R \cup \{K\}$
$C \longleftarrow C \backslash R$

Algorithm 1: Basic recursive minimization algorithm similar to [19].

The transition system from Definition 4 is not linear. As far we are aware, this is the first description of minimization algorithm in terms of Horn SAT.

Theorem 6. *Algorithm 1 is the same as the transition system from Definition 4.*

Proof. Up to renaming literals to their polarity in the trail, reason clauses are Horn. When removing the literal L, the other literals are duplicates and therefore removed and rewritten to $\bot$. Resolving with the reason $K \vee C$ is the same as rewriting $\neg K \to C$. Removing other literals from D is equivalent to rewriting those literals to $\bot$. The only rule from I is $I \to \neg L$. □

Theorem 7 (Equivalence Syntactic and Semantic Redundancy). *Both notions of redundancy are equivalent. In particular, every redundant literal is also removable.*

Proof. Semantic redundancy is equivalent to proving $F_M, \neg D \setminus \{L\} \vDash \neg L$, i.e. $\neg L$ is true. We know that the reasons are satisfiable (the trail is model of it). After converting the system to Horn, by Theorem 5 the literal $\neg L$ is true iff it can be rewritten away. We conclude using Theorem 6. □

In our formalization of learned clause minimization for our verified SAT solver IsaSAT [13], we use a different definition of redundancy, namely $F_M \vDash \neg L \vee D_{<_M L}$ where $D_{<_M L}$ are all the literals of D that appear before L in the trail M. This definition is equivalent but it makes more explicit that only literals that appear before L are relevant. We have not formalized completeness while working on IsaSAT since we only cared about correctness.

```
Function IsLiteralRedundantEfficient(L, d, C)
    Input: Literal L assigned to true, recursion depth d, deduced clause C
    Output: Whether L can be removed
    if status of L is cached in minimization cache then
        return cached value
    if any advanced poison criterion from Theorem 9 applies (uses d ) then
        return false
    if L is root-level assigned (unit) or ¬L ∈ C then
        return true
    if L is a decision then
        return false
    D ∨ L ⟵ reason(L)
    foreach K ∈ D do
        if ¬IsLiteralRedundantEfficient(¬K, d + 1, C) then
            Cache false for L
            return false
    Cache true for L
    return true
```

Algorithm 2: Advanced minimization algorithm equivalent to Algorithm 1.

Theorem 8. *A literal L is redundant iff $F_M \models \neg L \vee D_{<_M L}$.*

Proof. By induction over the derivation from Theorem 6, literals that occur after L are never introduced, because they never appear in the reasons. □

Our implementation relies on the alternative definition: It sorts the literals in the clause by its position on the trail. Each literal, starting from the lowest position, is checked. If it is not redundant, it is marked as present in the deduced clause for efficient checking. This reduces the number of flags (like testing if a literal is present in the deduced clause) to reset. Instead we could use d: When $d = 0$, the condition "L is in the deduced clause" does not apply.

Thanks to caching both successes and failures, the complexity is linear in the number of literals of the trail. Compared to our simple break conditions, more advanced criteria are possible.

Theorem 9 (Poison Criteria).

1. *If a literal appears on the trail before any other literal of the deduced clause on a decision level, then it is not redundant.*
2. *Literals with a decision level not in the deduced clause are not redundant.*
3. *Literals that are alone on a given decision level are not redundant (Knuth).*

Proof. 1. Propagation is done eagerly. Therefore, every reason depends on a literal of current level down to the decision literal that is not redundant.
2. Similar to previous point.
3. Resolve with the reason and apply first point. □

Function `MinAllUIPShrinkSlice`(B, C)

Input: Slice B of literals of the deduced clause C on the (slice) level
Output: B unchanged or shrunken if *min-alluip* is successful

$E \longleftarrow \emptyset$
while $|B| > 1$ **do**
  Remove from B last assigned literal $\neg L$
  $D \vee L \longleftarrow \text{reason}(L)$
  if $\exists K \in D \backslash C$ assigned at lower level not already in C **then**
    $E \longleftarrow E \cup \{L\}$
  else
    $B \longleftarrow B \cup \{K \in D \mid K \text{ assigned on slice level}\}$
Replace in deduced clause C original B with $B \cup E$

Function `MinAllUipShrinking`(C)

Input: The deduced clause C (passed by reference)
Output: The shrunken clause using the *min-alluip* strategy

$C' \longleftarrow C$
foreach Level i of literals in the deduced clause – highest to lowest **do**
  $B \longleftarrow \{L \in C \mid L \text{ assigned at level } i\}$
  `MinAllUIPShrinkSlice`(B, C)
Replace C with saved original deduced clause C' unless $|C| < |C'|$

Algorithm 3: Shrinking algorithm *min-alluip* from Feng&Bacchus [12].

The proof relies on the fact that the SAT solver propagates literals eagerly. This is not the case globally if the SAT solver uses chronological backtracking [16,17] but remains correct for the reason clauses. The second and third point are widely used (e.g., in MiniSAT and Glucose), whereas the first one is a novelty of CaDiCaL and is not described so far. Root-level assigned false literals can also appear in deduced clauses and be removed without recursing over their reasons.

Theorem 10. *Literals at level 0 are redundant.*

Algorithm 2 combines the two ideas that are described here, the caching and the advanced poison criteria. The ideas 1 and 3 from Theorem 9 require data structures that are not present in every SAT solver, namely the position τ of each literal in the trail. Doing so was not necessary until now, but it is required for shrinking. In our solvers, we also use the depth to limit the number of the recursive calls and avoid stack overflows. The implementation in MiniSAT [11] (and all derived solvers like Glucose [1]) uses a non-recursive version, but it requires two functions, one for depth zero and another for the recursive case.

3 Shrinking

After detecting conflicting clauses, the SAT solver analyzes them and deduces the first unique-implication point or 1-UIP [7], where only one literal remains

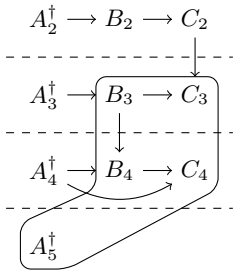

Fig. 1. Conflict example

on the current (largest) decision level. This is the first point where the clause is propagating, fixing the current search direction. The idea of 1-UIP can be applied on every level in order to produce shorter clauses. We call this process *shrinking*. It differs from minimization because it adds new literals to the deduced clause.

If fully applied, shrinking derives a subset of the decision-only clause. Therefore, it is limited. Feng & Bacchus [12] (abbreviated F&B from now on) have used various heuristics like not adding literals of low importance, without a clear winner across all implementations. We focus on their *min-alluip* variant. It applies the 1-UIP on every level. For each literal in the clause, the solver resolves with its reason unless a literals from a new level is added, thus making sure that the LBD or "glue" [1] is not increased, an important metric, which seems to relate well to the "quality" of learned clauses. In their implementation, if the clause becomes longer, the minimized clause would be used instead.

Algorithm 3 shows the implementation of *min-alluip*. It considers the set of all literals of the deduced clause on the same level, or *slice* (same as a block if no chronological backtracking [16, 17] is allowed). Each slice is shrunken starting from the highest level. It resolves each literal of the slice with its reason or fails when adding new literals on lower levels. Because SAT solvers propagate eagerly, $|B| \geq 1$ is an invariant of the while loop (and L cannot be a decision literal).

The key difference between shrinking and minimization is that reaching the UIP is a *global* property, namely of all literals on a level, and not of a single literal. This means that testing redundancy is a depth-first search algorithm while shrinking is a breadth-first search algorithm on the implication graph.

Example 11. Consider the implication graph from Figure 1. The algorithm starts with the highest level, namely with B_4 and A_4. The level is reduced to A_4 introducing the already present B_3. On the next level, C_3 cannot be removed because it would import level 2. The resulting clause $\neg A_5 \vee \neg A_4 \vee \neg A_3$ is smaller and is used instead of the original clause.

F&B unfortunately do not provide source code nor binaries used in their experiments. Therefore we focus on their version of CaDiCaL submitted [15] to the SAT Competition 2020. It implements only one of their strategies, which,

Function `ShrinkingSlice(`B, C`)`

Input: Slice B of literals of the deduced clause C on a single (slice) level
Output: B unchanged or shrunken to UIP if our new method is successful

while $|B| > 1$ **do**
  Remove from B last assigned literal $\neg L$
  $D \vee L \longleftarrow \text{reason}(L)$
  if $\exists K \in D \backslash C$ at lower level and $\neg$`IsLiteralRedundant(`$\neg K$, 1, C`)`
  then
    return with failure (keep original B in C)
  else
    $B \longleftarrow B \cup \{K \in D \mid K \text{ on slice level}\}$
Replace in deduced clause C original B with the remaining UIP in B

Function `Shrinking(`C`)`

Input: The deduced clause C (passed by reference)
Output: The shrunken and minimized clause using our new strategy

foreach Level i of literals in the deduced clause – lowest to highest **do**
  $B \longleftarrow \{L \in C \mid L \text{ assigned at level } i\}$
  `ShrinkingSlice(`B, C`)`
  if shrinking the slice failed **then** `MinimizeSlice(`B, C`)`;

Algorithm 4: Our new method for integrated shrinking with minimization.

as far we can tell, matches the variant *min-alluip* [12] described above, while code for the other variants is incomplete or missing.

4 Minimizing and Shrinking

In contrast to F&B our algorithm minimizes literal slices of the deduced clause assigned on a certain level starting from the lowest to highest level. This enables us to remove all redundant literals on-the-fly. After presenting our algorithm we study its complexity and then discuss its implementation in our SAT solvers CaDiCaL, Kissat, and Satch.

The main loop of our Algorithm 4 interleaves shrinking and (if shrinking failed) minimization. For each slice of literals in the deduced clauses assigned on a certain level we then attempt to reach the 1-UIP, similarly to Algorithm 3. If this fails, we minimize the slice. This also allows to lift some restriction on shrinking: only non-redundant literals interrupt the search for the 1-UIP. We start from the lowest level to keep completeness of minimization.

Example 12. Consider the implication graph from Figure 1. The algorithm starts with the slice of literals on the lowest decision level, namely with B_3 and C_3. No UIP can be found because it would import level 2. Level 1 is shrunken to A_4. The shrunken clause is $\neg A_5 \vee \neg A_4 \vee \neg B_3 \vee C_3$

Theorem 13. *Shrinkable literals are redundant.*

Proof. Removing the literal L amounts to resolving with its reason $L \vee C$. If the deduced clause is $\neg L \vee D$, L is redundant with respect to $C \vee D$. □

This theorem indicates that a literal that was removed by shrinking is also redundant for the minimization. Therefore, when shrinking succeeds, we can directly mark the literals as shrinkable.

As mentioned before, for efficiency a cache is maintained during minimization to know whether a literal is redundant or not.

Theorem 14 (Shrinking and Redundancy). *Redundant literals remain redundant during shrinking.*

Proof. Similar to the proof of Theorem 13.

Theorem 14 ignores irredundant literals because new literals are added to the deduced clause, allowing for more removable literals. This explains why F&B propose (in one variant of shrinking) to minimize again after shrinking. For the same reason we do not check if literals are redundant on the current level, since added literals (e.g., new 1st UIPs) invalidate the literals marked as "poisoned". Instead, we check for redundancy of literals on lower levels and on current level only after shrinking them, when the literals on the slice level are fixed.

Example 15 (Minimization during shrinking). Consider the following trail

$$A_1^\dagger B_1^{B_1 \vee \neg A_1} \quad A_2^\dagger B_2^{B_2 \vee \neg B_1 \vee \neg A_2} \quad A_3^\dagger$$

where † marks a decision and the deduced clause is $\neg A_1 \vee \neg B_2 \vee \neg A_3$. Shrinking cannot remove B_2 because it would introduce the new literal B_1 on lower levels, unless it is determined to actually be redundant (A_1 is in the deduced clause).

To keep the complexity linear, when interleaving minimization with shrinking as shown in Algorithm 4, we maintain a global shared minimization cache, not reset between minimizing different slices. A more complicated solution consists in minimizing up-front (as in the implementation of F&B in [15]), followed by shrinking, and if shrinking succeeds, reset the poison literals on the current level.

Resetting only literals on the current level is important for reducing the runtime complexity from quadratic to linear in the size of the implication graph. As we are shrinking "in order" (from lowest to highest decision level) we can keep cached poisoned (and removable) literals from previous levels, thus matching the overall linear complexity of (advanced) minimization.

Our solution also avoids updating the minimization cache more than once during shrinking. When a slice is successfully reduced to a single literal, all shrunken literals are marked as redundant in the minimization cache. The process is complete in the sense that no redundant literals remain.

Theorem 16 (Completeness). *All redundant literals are removed.*

Proof. During execution of Algorithm 4 the minimization cache remains valid for all lower levels. Since minimization is applied on every level, all redundant literals are removed. Note that, UIPs are never redundant (Theorem 9, point 3). □

This result relies on the fact that during the outer loop no literal on a lower level is added to the deduced clause. If this would be allowed (as in Algorithm 3), the poisoned flag has to be reset and minimization redone, yielding a quadratic algorithm. However, the theorem says nothing about minimality of the shrunken clause if we allow to add new literals, as in the following example.

Example 17 (Smaller Deduced Clause). Consider the trail

$$A_1^\dagger B_1^{B_1 \vee \neg A_1} C_1^{C_1 \vee \neg B_1} \quad A_2^\dagger B_2^{B_2 \vee \neg A_2} C_2^{C_2 \vee \neg B_2 \vee \neg B_1} \quad A_3^\dagger$$

and the deduced clause $\neg C_1 \vee \neg B_2 \vee \neg C_2 \vee \neg A_3$. The clause is neither minimized nor shrunken by our algorithm, but could be to the smaller $\neg B_1 \vee \neg B_2 \vee \neg A_3$.

In Algorithm 4, on the one hand, shrinking could use a priority queue (implemented as binary heap) to determine the last assigned literal in B. Then for each slice, we have a complexity of $\mathcal{O}(n_b \log n_b)$ for shrinking where n_b is the number of literals at the slice level in the implication graph. On the other hand, minimization of all slices is linear in the size of the implication graph. Overall the complexity is $\mathcal{O}(\text{glue} \cdot n \log n)$ where the "glue" is the number of different slices (and a number that SAT solvers try to reduce heuristically) and n the maximum of the n_b. However, note that, bumping heuristics require sorting of the involved literals anyhow either implicitly or explicitly [6].

Instead of representing the slice B as a priority queue, implemented as binary heap, to iterate over its literals, it is also possible to iterate over the trail directly as it is common in conflict analysis to deduce the 1st-UIP clause. Without chronological backtracking, the slices on the trail are disjoint and iterating over the trail is efficient and gives linear complexity $\mathcal{O}(|\text{glue}| \times |\text{max\_trail\_slice\_length}|)$, i.e., linear in the size of the implication graph.

With chronological backtracking slices on the trail are not guaranteed to be disjoint. Therefore, in the worst case, iterating over a slice along the trail might require to iterate over the complete trail. In principle, this could give a quadratic complexity for chronological backtracking without using a priority queue for B. In our experiments both variants produced almost identical run-times and thus we argue that the simpler variant of going over the trail should be preferred.

We have implemented the algorithm from the previous section in our SAT solvers CaDiCaL [5], Kissat [5], and Satch [4]. The implementation is part of our latest release in the file `shrink.c` (`shrink.cpp` for CaDiCaL).[1] Note that, Satch is a simple implementation of the CDCL loop with restarts and was written to explain CDCL. It does not feature any in- nor preprocessing yet.

We either traverse the trail directly or use a radix heap [8] as priority queue. Unlike the implementation by F&B, our priority queue contains only the literals from the current slice until either shrinking fails or the 1-UIP is found. It allows for efficient popping and pushing trail positions. Note that, radix heaps require popped elements to be strictly decreasing, and as the analysis follows reverse trail order, we first compute the maximum trail position of literals in the considered

[1] Source code and log files are available at `http://fmv.jku.at/sat_shrinking`.

slice and then index literals by their offsets on the tail from this maximum trail position. The literal position in the trail is not cached in every SAT solver, but was already maintained in Kissat and CaDiCaL.

5 Experiments

We have implemented our algorithm in the SAT solvers CaDiCaL, Kissat (the winner of the SAT Competition 2020), and Satch and evaluated them on benchmark instances from the SAT Competition 2020 on an 8-core Intel Xeon E5-2620 v4 CPUs running at 2.10 GHz (turbo-mode disabled). For both tracks we used a memory limit of 128 GB (as in the SAT Competition 2020). We tested 3 configurations, shrink (shrinking and minimizing), minimize, and no-minimize (neither shrinking nor minimizing). Due to space constraint we only give graphs for some solvers but findings are consistent across all of them.

Tables 1 for Kissat and Satch show that minimization is more important than shrinking, but the latter still improves performance for Kissat. In the planning track, running time decreases significantly, whereas the impact on the main track is smaller. Compared to the main track, the planning problems require much more memory and memory usage drops substantially with shrinking. For Satch, we observe a slight performance decrease. Figures 2 and 3 show that even if shrinking solves only a few more problems, the speedup is significant.

In all our SAT solvers we distinguish between *focused mode* (many restarts) and *stable mode* (few restarts). Note that CaDiCaL uses the number of conflicts to switch between these modes which is rather imprecise: in stable mode decision frequency is lower while the conflicts frequency is higher compared to focused mode and accordingly the fraction of running time spent in conflict analysis and thus minimization and shrinking increases in stable mode compared to focused mode. To improve precision both Kissat and Satch measure the time by estimating the number of possible cache misses instead, called "ticks" [5]. By default Kissat also counts the number of such ticks during shrinking and minimization. To avoid the bias introduced by this technique in terms of influencing mode switching we deactivated this feature in our experiments (only for Kissat).

We analyzed the results on the main track in more details over all instances (i.e., until timeout or memory out), not only over solved instances. The amount of time (in percentage of the total) more than doubles when activating shrinking: it goes from 6.3 % to 14.3 % of the total amount of time (Figure 5). However, the size of the clauses is reduced with a similar ratio: It drops from 110 to 46 (183 without minimization). On the planning track, it drops from 13 076 to 5 398 literals on average (16 637 without minimization).

To compare our method to the *min-alluip* implementation, which is based on CaDiCaL 1.2.1, we backported our *shrinking* algorithm to this 3-year-old version of CaDiCaL 1.2.1 too. The results are in Table. 2. The only difference is the shrinking algorithm, hence there are not differences for the minimize and no-minimize configuration. The F&B version performs slightly better than our version. An interesting observation is that CaDiCaL 1.2.1 learns much larger

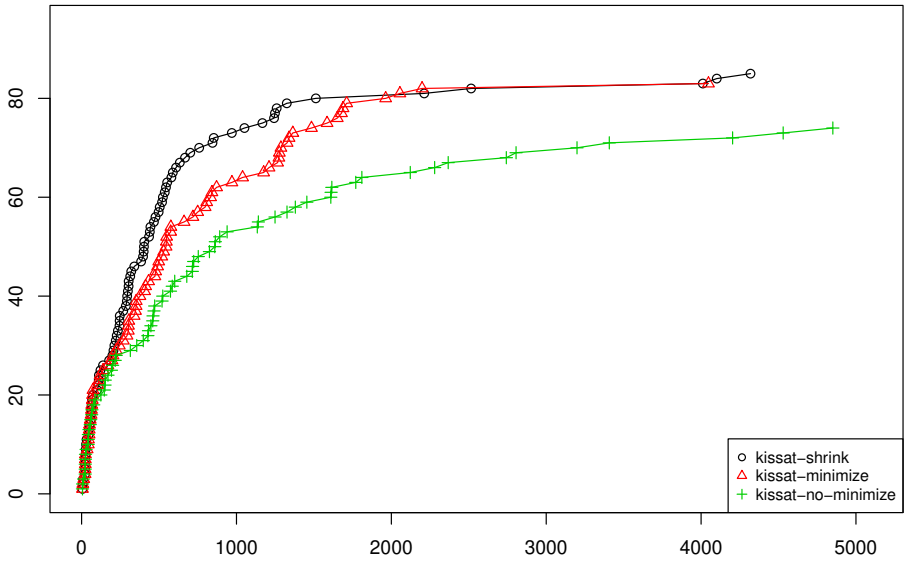

Fig. 2. KISSAT solving time on the planning track.

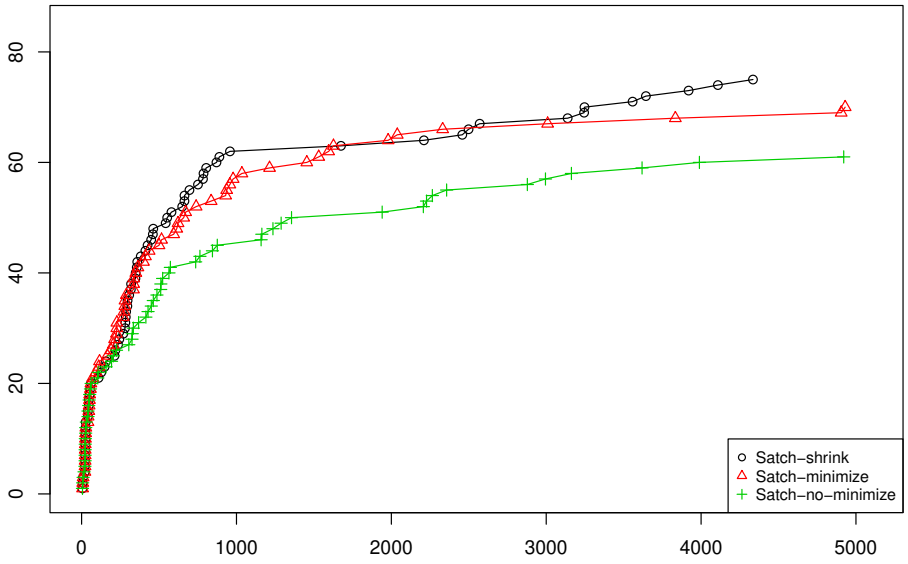

Fig. 3. SATCH solving time on the planning track.

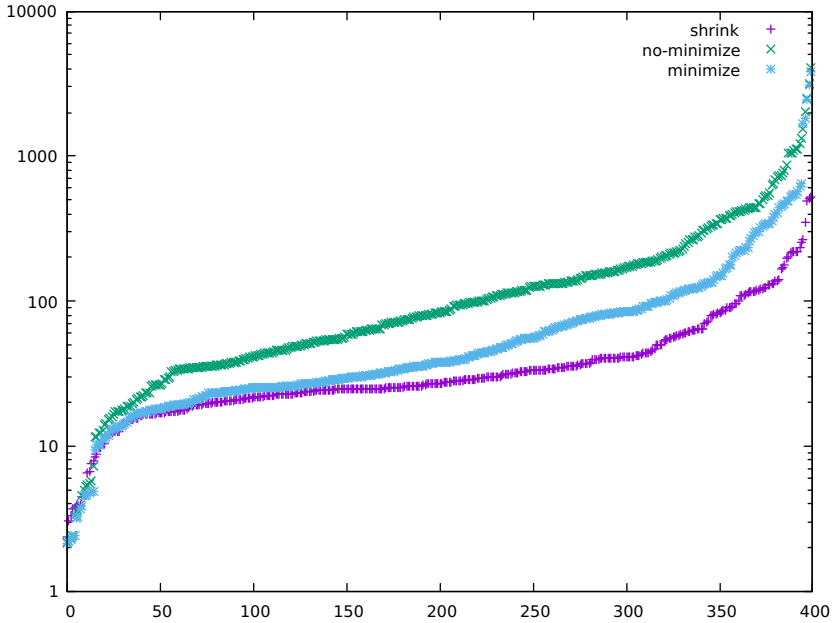

Fig. 4. Absolute sizes of learned clauses of KISSAT on main track.

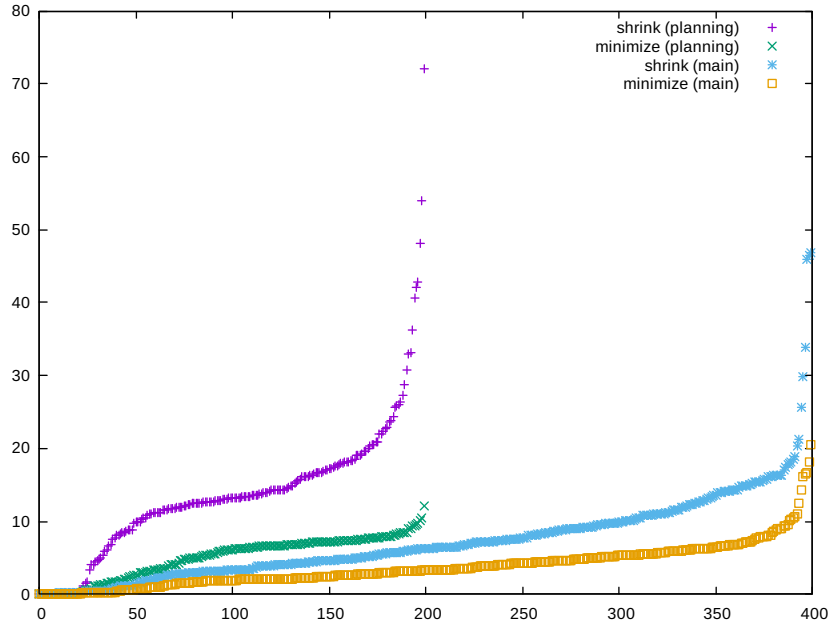

Fig. 5. Amount of time in percent spent during shrinking and minimization of KISSAT.

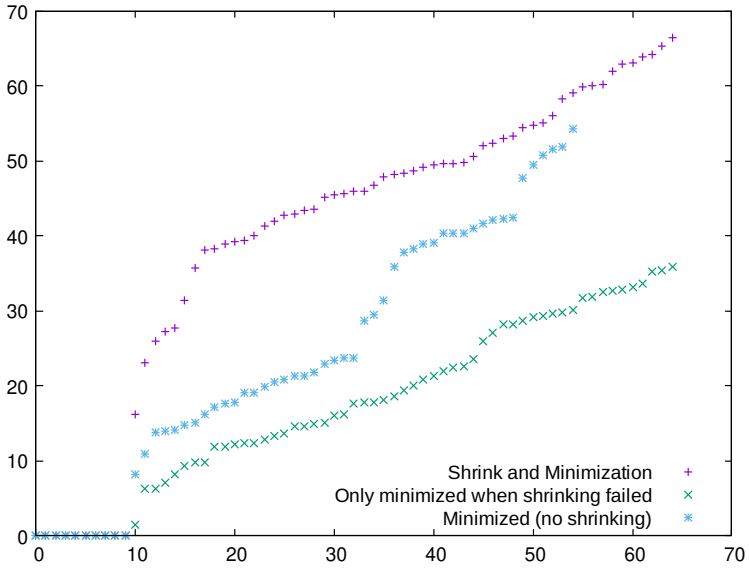

Fig. 6. Percentage removed literals in learned clauses for CaDiCaL in planning track.

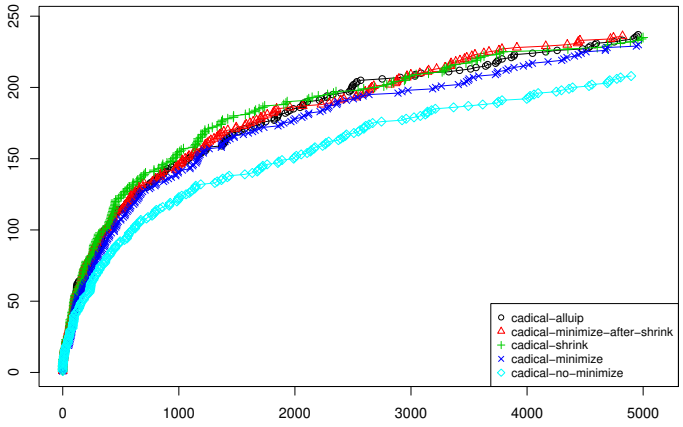

Fig. 7. Comparison between CaDiCaL-1.2.1 with shrinking (this paper) and the F&B version on the main track.

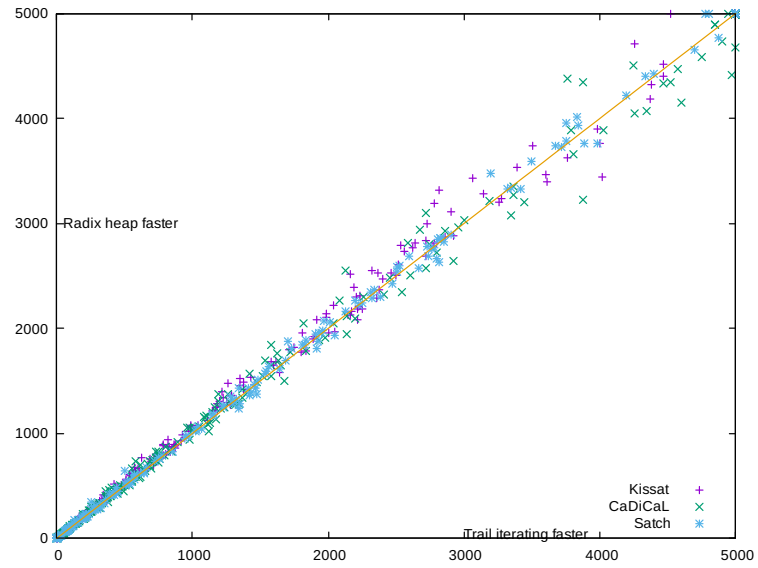

Fig. 8. Speed difference with and without radix heap for all our solvers.

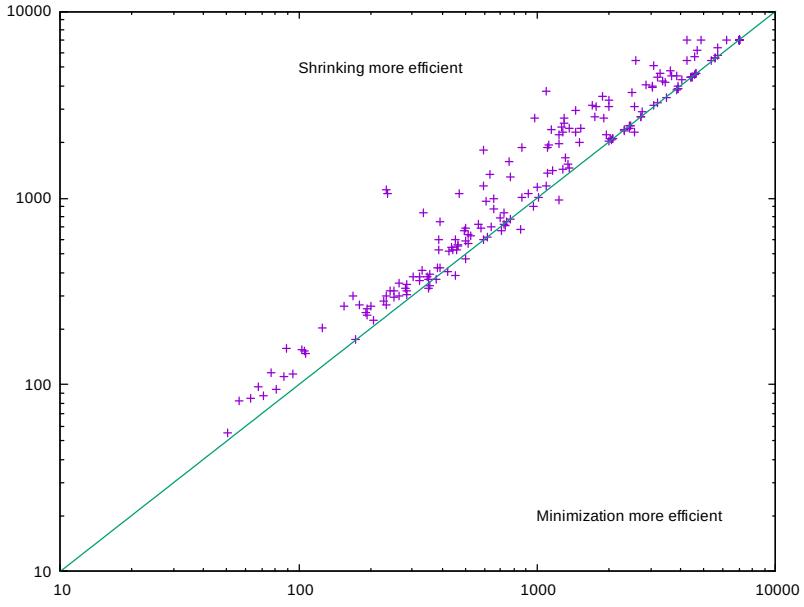

Fig. 9. Memory comparison of SATCH on the planning track.

Table 1. Results for new solvers on the SAT Competition 2020 benchmarks

Solver	Track	Configuration	Solved	PAR-2	Average clause size
Kissat	Main Track (400 problems)	shrink	**270**	**1561735**	**46**
		minimize	267	1 566 688	110
		no-minimize	235	1 891 872	183
	Planning Track (200 problems)	shrink	**85**	**1197799**	**5 398**
		minimize	83	1 222 535	13 076
		no-minimize	74	1 325 957	16 637
Satch	Main Track (400 problems)	shrink	196	2 271 119	**46**
		minimize	**203**	**2240351**	144
		no-minimize	159	2 621 070	370
	Planning Track (200 problems)	shrink	**85**	**1212977**	**5 043**
		minimize	80	1 250 861	11 854
		no-minimize	72	1 338 592	15 474
CaDiCaL 1.4.0	Main Track (400 problems)	shrink	**240**	**1 870 484**	**90**
		minimize	233	1 939 998	121
		no-minimize	194	2 280 897	153
	Planning Track (200 problems)	shrink	**73**	**1 334 718**	**4 885**
		minimize	64	1 454 186	7 799
		no-minimize	42	1 615 676	11 767

clauses than Kissat and Satch but also larger than the latest CaDiCaL version. The effect can be partially explained by the stable mode that is much longer than on the other solvers.During stable mode, much longer clauses are learned (with much higher glue too). We have also experimented with minimizing separately from shrinking instead of combining them. As long as the cache is shared there is very little performance difference. Figure 7 shows the CDF for the main track and Figure 10 for the planning track.

Figure 6 shows percentages of removable literals on the planning track. Shrinking removes more literals than the subsequent minimization (and more than minimization alone).

We have mentioned the complexity difference between using a radix heap and iterating over the trail. We have implemented both versions in our three SAT solvers. We compare both version but could not observe any significant difference (see Figure 8). We believe that this is due to the fact that finding the next literal is actually very efficient: it is in the trail (that is in cache anyways) and we check a single flag. We attempted to force the worst case by enforcing chronological backtracking, but performance remained similar.

We also compared the memory impact of activating shrinking on the problems of the planning track solved by both shrinking and minimizing (see Figure 9)

Table 2. Results for solvers based on CaDiCaL 1.2.1 on the SAT Competition 2020 benchmarks with a memory limit of 128 GB, following the SAT Competition

Solver	Track	Configuration	Solved	PAR-2	Average clause size
shrinking (this paper)	Main Track (400 problems)	shrink	235	**1 897 387**	**92**
		minimize	230	1 972 949	135
		no-minimize	208	2 184 920	187
	Planning Track (200 problems)	shrink	73	1 351 542	5 373
		minimize	63	1 454 871	6 433
		no-minimize	39	1 643 665	9 874
min-alluip [12, 15]	Main Track	shrink	**237**	1 904 745	104
	Planning Track	shrink	**81**	**1 271 930**	**3 261**

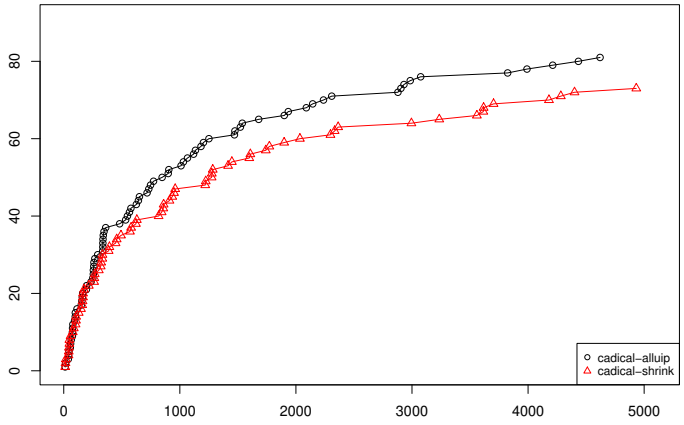

Fig. 10. Comparison between CaDiCaL-1.2.1 with shrinking (this paper) and the F&B version on the planning track.

6 Conclusion

We presented a simple linear algorithm which integrates minimization and shrinking and is guaranteed to remove all redundant literals. In practice it can be run to completion unconditionally. Our implementation and evaluation with several SAT solvers show the benefit of our approach and confirm effectiveness of shrinking in general.

An open question is how to extend our notion of trail redundancy to capture that new literals can be added in order to reduce size. This would allow to formulate completeness of shrinking in the same way as we did for minimization.

Acknowledgment

This work is supported by Austrian Science Fund (FWF), NFN S11408-N23 (RiSE), and the LIT AI Lab funded by the State of Upper Austria. We also thank Sibylle Möhle and the anonymous reviewers for suggesting textual improvements.

References

1. Audemard, G., Simon, L.: Predicting learnt clauses quality in modern SAT solvers. In: IJCAI. pp. 399–404 (2009), http://ijcai.org/Proceedings/09/Papers/074.pdf
2. Biere, A.: CaDiCaL, Lingeling, Plingeling, Treengeling and YalSAT entering the SAT Competition 2018. In: Heule, M., Järvisalo, M., Suda, M. (eds.) Proc. of SAT Competition 2018 – Solver and Benchmark Descriptions. Department of Computer Science Series of Publications B, vol. B-2018-1, pp. 13–14. University of Helsinki (2018)
3. Biere, A.: Lingeling, Plingeling, PicoSAT and PrecoSAT at SAT Race 2010. Tech. rep., FMV Reports Series, Institute for Formal Models and Verification, Johannes Kepler University (aug 2021)
4. Biere, A.: The SAT solver Satch. Git repository (2021), https://github.com/arminbiere/satch, last Accessed: March 2021
5. Biere, A., Fazekas, K., Fleury, M., Heisinger, M.: CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020. In: Balyo, T., Froleyks, N., Heule, M., Iser, M., Järvisalo, M., Suda, M. (eds.) Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions. Department of Computer Science Report Series B, vol. B-2020-1, pp. 51–53. University of Helsinki (2020)
6. Biere, A., Fröhlich, A.: Evaluating CDCL variable scoring schemes. In: Heule, M., Weaver, S.A. (eds.) SAT 2015. LNCS, vol. 9340, pp. 405–422. Springer (2015). https://doi.org/10.1007/978-3-319-24318-4_29
7. Biere, A., Heule, M.J.H., van Maaren, H., Walsh, T. (eds.): Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications, vol. 185. IOS Press (2009)
8. Cherkassky, B.V., Goldberg, A.V., Silverstein, C.: Buckets, heaps, lists, and monotone priority queues. SIAM J. Comput. **28**(4), 1326–1346 (1999). https://doi.org/10.1137/S0097539796313490
9. Dowling, W.F., Gallier, J.H.: Linear-time algorithms for testing the satisfiability of propositional Horn formulae. J. Log. Program. **1**(3), 267–284 (1984). https://doi.org/10.1016/0743-1066(84)90014-1
10. Eén, N., Biere, A.: Effective preprocessing in SAT through variable and clause elimination. In: Bacchus, F., Walsh, T. (eds.) SAT 2005. LNCS, vol. 3569, pp. 61–75. Springer (2005). https://doi.org/10.1007/11499107_5
11. Eén, N., Sörensson, N.: An extensible SAT-solver. In: SAT. LNCS, vol. 2919, pp. 502–518. Springer (2003). https://doi.org/10.1007/978-3-540-24605-3_37
12. Feng, N., Bacchus, F.: Clause size reduction with all-UIP learning. In: SAT. LNCS, vol. 12178, pp. 28–45. Springer (2020). https://doi.org/10.1007/978-3-030-51825-7_3
13. Fleury, M.: Formalization of logical calculi in Isabelle/HOL. Ph.D. thesis, Saarland University, Saarbrücken, Germany (2020), https://tel.archives-ouvertes.fr/tel-02963301

14. Fleury, M., Biere, A.: Efficient all-UIP learned clause minimization. In: SAT. LNCS, Springer (2021), to appear
15. Hickey, R., Feng, N., Bacchus, F.: Cadical-trail, Cadical-alluip, Cadical-alluip-trail and Maple-LCM-Dist-alluip-trail at the SAT Competition. In: Balyo, T., Froleyks, N., Heule, M., Iser, M., Järvisalo, M., Suda, M. (eds.) Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions. Department of Computer Science Report Series B, vol. B-2020-1, p. 10. University of Helsinki (2020)
16. Möhle, S., Biere, A.: Backing backtracking. In: SAT. LNCS, vol. 11628, pp. 250–266. Springer (2019). https://doi.org/10.1007/978-3-030-24258-9_18
17. Nadel, A., Ryvchin, V.: Chronological backtracking. In: SAT. LNCS, vol. 10929, pp. 111–121. Springer (2018). https://doi.org/10.1007/978-3-319-94144-8_7
18. Silva, J.P.M., Lynce, I., Malik, S.: Conflict-driven clause learning SAT solvers. In: Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.) Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications, vol. 185, pp. 131–153. IOS Press (2009). https://doi.org/10.3233/978-1-58603-929-5-131
19. Sörensson, N., Biere, A.: Minimizing learned clauses. In: SAT. LNCS, vol. 5584, pp. 237–243. Springer (2009). https://doi.org/10.1007/978-3-642-02777-2_23
20. Van Gelder, A.: Improved conflict-clause minimization leads to improved propositional proof traces. In: SAT. LNCS, vol. 5584, pp. 141–146. Springer (2009). https://doi.org/10.1007/978-3-642-02777-2_15
21. Van Gelder, A.: Generalized conflict-clause strengthening for satisfiability solvers. In: SAT. Lecture Notes in Computer Science, vol. 6695, pp. 329–342. Springer (2011). https://doi.org/10.1007/978-3-642-21581-0_26
22. Zhang, L., Madigan, C.F., Moskewicz, M.W., Malik, S.: Efficient conflict driven learning in Boolean satisfiability solver. In: ICCAD. pp. 279–285. IEEE Computer Society (2001). https://doi.org/10.1109/ICCAD.2001.968634