Learning Shorter Redundant Clauses in SDCL Using MaxSAT

Albert Oliveras
Technical University of Catalonia, Barcelona, Spain

Chunxiao Li
University of Waterloo, Canada

Darryl Wu
University of Waterloo, Canada

Jonathan Chung
University of Waterloo, Canada

Vijay Ganesh
University of Waterloo, Canada

Abstract

In this paper we present the design and implementation of a Satisfaction-Driven Clause Learning (SDCL) SAT solver, MapleSDCL, which uses a MaxSAT-based technique that enables it to learn shorter, and hence better, redundant clauses. We also perform a thorough empirical evaluation of our method and show that our SDCL solver solves Mutilated Chess Board (MCB) problems significantly faster than CDCL solvers, without requiring any alteration to the branching heuristic used by the underlying CDCL SAT solver.

2012 ACM Subject Classification Theory of computation

Keywords and phrases SAT, SDCL, MaxSAT

Digital Object Identifier 10.4230/LIPIcs.SAT.2023.18

Funding *Albert Oliveras*: Supported by grant PID2021-122830OB-C43, funded by MCIN/AEI/10.13039/501100011033 and by "ERDF: A way of making Europe".

1 Introduction

Conflict-Driven Clause Learning (CDCL) SAT solvers are routinely used to solve large industrial problems obtained from variety of applications in software engineering [7], formal methods [8], security [11, 25] and AI [6], even though the underlying Boolean satisfiability (SAT) problem is well known to be NP-complete [9] and believed to be intractable in general. Despite this, solver research has made significant progress in improving CDCL solvers' components and heuristics [19].

It is well known that CDCL SAT solvers are polynomially equivalent to resolution [20, 1], and consequently it follows that classes of formulas, such as the pigeon hole principle (PHP), that are hard for resolution are also hard for CDCL SAT solvers. In order to address such limitations, researchers are actively designing and implementing solvers that correspond to stronger propositional proof systems.

One such class of solvers is called Satisfaction-Driven Clause Learning (SDCL) solvers [15, 14, 13], which are based on the propagation redundancy (PR) property [12, 14]. The SDCL paradigm extends CDCL in the following way: unlike CDCL solvers, SDCL solvers may learn clauses even when an assignment trail α is consistent. To be more precise, an SDCL solver first computes a new formula $P_\alpha(F)$, known as a pruning predicate. Then, it checks the satisfiability of $P_\alpha(F)$. If it satisfiable, it means $\neg\alpha$ is *redundant* with respect to the formula, and the solver can learn the clause $(\neg\alpha)$. Even though the intuition is clear and procedures for computing a possible $P_\alpha(F)$ are very well defined, it is still an extremely challenging task to automate SDCL.

There are two main problems in this setting: first, the satisfiability check for the formula $P_\alpha(F)$ is NP-complete and is hard to solve in general. It essentially requires the SDCL solver to call another SAT solver that we refer to as a sub-solver. Given that this sub-solver call can be expensive, one needs to be strategic about when to invoke it during the run of an SDCL solver. Second, the clauses learned by SDCL can be large, and we want to learn shorter clauses whenever possible.

To solve this second problem, we propose a novel MaxSAT encoding of the problem of "what is the smallest subset γ of trail α, such that $P_\gamma(F)$ is satisfiable", to get the shortest clause $(\neg\gamma)$ to learn. We also apply a resolution-based technique inspired by conflict analysis to further shorten the clause. We refer to the SDCL solver augmented with our MaxSAT and clause minimization technique as MapleSDCL. Our experimental evaluation shows that MapleSDCL performs well on mutilated chess board (MCB) and bipartite perfect matching problems, that are known to be hard for CDCL solvers.

1.1 Contributions

(I) First, we make a theoretical contribution by introducing a new type of pruning predicate and a proof that it allows one to detect blocked clauses. This extends the spectrum of pruning predicates with redundancy notions associated with them. However, we remark that this is not implemented in our system as we consider it to have little practical impact.

(II) Second, we prove that when an assignment has a satisfiable positive reduct, finding a small sub-assignment with the same property is an NP-hard problem. Such small assignments summarize the reasons for the redundancy and lead to learning smaller redundant clauses. In essence, we believe that this is the equivalent to conflict analysis in CDCL solvers.

(III) Third, we introduce a MaxSAT encoding of the above-stated problem. Experimental results show that calling a MaxSAT solver within the SDCL architecture is not as expensive as one might expect, and more importantly, significant improvements in the size of the learned redundant clauses are achievable in practice. These improvements are even larger after applying conflict analysis techniques to convert the clause into an asserting one.

(IV) Finally, we show that the resulting SDCL solver can solve mutilated chess board problems without the need to alter the decision heuristic used by the underlying CDCL SAT solver. This is a very important property of our approach: the chances of learning (good) redundant clauses depend much less on choosing exactly the right decision literals, thus overcoming a serious roadblock for SDCL solver design.

2 Preliminaries on CDCL SAT Solving

CNF formulas. Let $\mathcal{X}$ be a finite set of propositional *variables*. A *literal* is a propositional variable (x) or the negation of one $(\neg x)$. The *negation* of a literal l, denoted $\neg l$, is x if $l = \neg x$ and is $\neg x$ if $l = x$. A *clause* is a disjunction of distinct literals $l_1 \vee \ldots \vee l_n$ (interchangeably denoted with or without brackets). A *CNF formula* is a conjunction of distinct clauses $C_1 \wedge \ldots \wedge C_m$. When convenient, we consider a clause to be the set of its literals, and a CNF to be the set of its clauses. In the rest of the paper we assume that all formulas are CNF.

Satisfaction. An *assignment* is a set of non-contradictory literals. A *total* assignment contains, for each variable $x \in \mathcal{X}$, either x or $\neg x$. Otherwise, it is a *partial* assignment. We denote by $\neg\alpha$ the clause consisting of the negation of all literals in the assignment α. An assignment α satisfies a literal l if $l \in \alpha$, it satisfies a clause C if it satisfies at least one of the literals in C, and it satisfies a formula F if it satisfies all the clauses in F. We denote these as $\alpha \models l$, $\alpha \models C$, and $\alpha \models F$, respectively. A *model* for a formula is an assignment that satisfies it. A formula with at least one model is *satisfiable*; otherwise, it is *unsatisfiable*. Given a formula F, the *SAT* problem consists of determining whether F is satisfiable. An assignment α falsifies a literal l if $\neg l \in \alpha$, falsifies a clause if it falsifies all its literals, and falsifies a formula if it falsifies at least one of its clauses. The truth values of literals, clauses, and formulas are *undefined* for an assignment if they are neither falsified nor satisfied. Given a clause C and an assignment α, we denote by $touched_\alpha(C)$ the disjunction of all literals of C that are either satisfied or falsified by α, by $untouched_\alpha(C)$ the disjunction of all undefined literals, and by $satisfied_\alpha(C)$ the disjunction of all satisfied literals.

Unit propagation. Given a formula F and an assignment α, unit propagation extends α by repeatedly applying the following rule until reaching a fixed point: if there is a clause with all literals falsified by α except one literal l, which is undefined, add l to α. If, as a result, a clause is found that is falsified by α (called *conflict*), the procedure stops and reports that a conflict clause has been found.

Formula relations. Two formulas F and G are *equisatisfiable*, denoted $F \equiv_{SAT} G$, if F is satisfiable if and only if G is satisfiable, and they are *equivalent*, denoted $F \equiv G$, if they are satisfied by the same total assignments. We write $F \vdash_1 G$ (*F implies G by unit propagation*) if for every clause $C \in G$ of the form $l_1 \vee \ldots \vee l_n$, it holds that unit propagation applied to $F \wedge \neg l_1 \wedge \ldots \wedge \neg l_n$ results in a conflict. We say that G is a logical consequence of F (written $F \models G$) if all models of F are models of G.

CDCL. The Conflict-Driven Clause Learning (CDCL) algorithm is the most successful procedure to-date for determining whether certain types of industrial formulas are satisfiable [19]. Let F denote such a formula. The CDCL procedure starts with an empty assignment α, which is extended and reduced in a last-in first-out (LIFO) way, by the following three steps until the satisfiability of formula is determined (see Algorithm 1 removing lines 9-12):

1. Unit propagation is applied.
2. If a conflict is found, a *conflict analysis* procedure [26] derives a clause C (called a *lemma*) which is a logical consequence of F. If C is the empty clause, we can conclude that F is unsatisfiable. Otherwise, it is guaranteed that by removing enough literals from α, a new unit propagation is possible due to C. This process is called *backjump*. Additionally, lemma C is conjuncted (*learnt*) with F, and the procedure returns to step (i).
3. If no conflict is found in unit propagation, either α is a total assignment (and hence it satisfies the formula), or an undefined literal is chosen and added to α (the branching step). The choice of this literal, called a *decision literal*, is determined by sophisticated heuristics [4] that can have a huge impact on performance of the CDCL procedure.

MaxSAT. Given a formula F, the *MaxSAT* problem consists of finding the assignment that satisfies the maximum number of clauses of F. Sometimes the clauses in F are split into *hard* and *soft clauses*, and in this case, the *Partial MaxSAT* problem consists of finding the assignment that satisfies all hard clauses and the maximum number of soft clauses.

3 Propagation Redundancy and SDCL

Despite their success on a variety of real-world applications [23, 21, 5, 16], CDCL SAT solvers have well-known limitations. Proof complexity techniques have established the polynomial equivalence between CDCL and general resolution [20, 1], the proof system with the inference rule that allows one to derive $C \vee D$ given two clauses of the form $l \vee C$ and $\neg l \vee D$. An important consequence of this equivalence is that if an unsatisfiable formula does not have a polynomial size proof by resolution, no run of CDCL can determine the unsatisfiability of the formula in polynomial time.

3.1 Propagation Redundancy

This limitation has motivated the search for extensions of CDCL solvers that may allow the resultant method to simulate more powerful proof systems. One example is the extended resolution proof system [24]: by allowing the introduction of new variables to resolution, it can produce polynomial size proofs of the pigeon-hole principle [10], which requires exponential-size resolution proofs otherwise. However, adding new variables would exponentially increase the search space of the formula. A newer direction [12, 14] tries to avoid the addition of new variables, and is instead based on the well-known notion of redundancy:

▶ **Definition 1.** *A clause C is* **redundant** *with respect to a formula F if F and $F \wedge C$ are equisatisfiable.*

In order to provide a more useful characterization of redundancy, we need some definitions.

▶ **Definition 2.** *Given an assignment α and a clause C, we define $\mathbf{C}|_\alpha = \top$ if $\alpha \models C$; otherwise $\mathbf{C}|_\alpha$ is the clause consisting of all literals of C that are undefined in α. For a formula F, we define the formula $\mathbf{F}|_\alpha = \{C|_\alpha \mid C \in F \text{ and } \alpha \not\models C\}$.*

▶ **Theorem 3** ([12], Theorem 1)**.** *A non-empty clause C is redundant with respect to a formula F if and only if there exists an assignment ω such that $\omega \models C$ and $F \wedge \neg C \models F|_\omega$.*

From a practical point of view, this characterization does not help much, because even if we know ω (known as the *witness*) it is hard to check whether the property holds. This is why a more limited notion of redundancy has been defined [12]:

▶ **Definition 4.** *A clause C is* **propagation redundant (PR)** *with respect to a formula F if there exists an assignment ω such that $\omega \models C$ and $F \wedge \neg C \vdash_1 F|_\omega$*

Note that since $F \wedge \neg C \vdash_1 F|_\omega$ implies $F \wedge \neg C \models F|_\omega$, any PR clause is redundant. Hence, we can add PR clauses to our formula in order to make it easier to solve without affecting its satisfiability. If we force ω to assign all variables in C but no other variable, we can obtain weaker but simpler notions of redundancy: if we force ω to satisfy exactly one literal of C, we obtain *literal-propagation redundant (LPR)* clauses; if allow ω to satisfy more than one literal of C, we obtain *set-propagation redundant (SPR)* clauses. Obviously, any LPR clause is SPR, and any SPR clause is PR, but none of these three notions are equivalent as the following examples show.

▶ **Example 5** ([12])**.** Let $F = \{x \vee y,\ \ x \vee \neg y \vee z,\ \ \neg x \vee z,\ \ \neg x \vee u,\ \ x \vee \neg u\}$ and $C = x \vee u$. The witness $\omega = \{x, u\}$ satisfies C and, since $F|_\omega = \{z\}$, it holds that $F \wedge \neg C \vdash_1 F|_\omega$, that is, unit propagation on $F \wedge \neg x \wedge \neg u \wedge \neg z$ results in a conflict. Hence, C is SPR w.r.t. F.

However, it is not LPR. The reason is that there are only two possible witnesses that satisfy exactly one literal of C: $\omega_1 = \{x, \neg u\}$ and $\omega_2 = \{\neg x, u\}$. But we have that both $F|_{\omega_1}$ and $F|_{\omega_2}$ contain, among others, the empty clause. Hence, $F \wedge \neg C \vdash_1 F|_{\omega_1}$ and $F \wedge \neg C \vdash_1 F|_{\omega_2}$ require that unit propagation on $F \wedge \neg C$, that is, $F \wedge \neg x \wedge \neg u$, results in a conflict, which is not the case.

▶ **Example 6** ([12]). Let $F = \{x \vee y, \neg x \vee y, \neg x \vee z\}$ and $C = (x)$. If we consider the witness $\omega = \{x, z\}$, we have that $F|_\omega = \{y\}$. It is obvious that $\omega \models C$ and also $F \wedge \neg x \vdash_1 y$. Thus, C is PR w.r.t. F. However it is not SPR because the only possible witness would be $\omega_1 = \{x\}$, but $F|_{\omega_1} = \{y, z\}$ and it does not hold that $F \wedge \neg x \vdash_1 z$.

3.2 SDCL and Reducts

It was proved in [12] that the proof system that combines resolution with the addition of PR clauses admits polynomial-sized proofs for the pigeon hole principle. However, it is not a trivial task to add this capability to CDCL solvers. This question was addressed with the development of Satisfiability-Driven Clause Learning (SDCL) [15]. The key notion in this new solving paradigm is the one of *pruning predicate*:

▶ **Definition 7.** *Let F be a formula and α an assignment. A* **pruning predicate** *for F and α is a formula $P_\alpha(F)$ such that if it is satisfiable, then the clause $\neg\alpha$ is redundant w.r.t. F.*

SDCL extends CDCL in the following way (See also Algorithm 1). Before making a decision, a pruning predicate for the assignment α and formula F is constructed. If satisfiable, we can learn $\neg\alpha$ and use it for backjump and continuing the search, hence pruning away the search tree without needing to find a conflict. This leads to the simple code in Algorithm 1, where removing lines 9 to 12 results in the standard CDCL algorithm, and where we can assume, for simplicity, that $analyzeWitness()$ returns $\neg\alpha$. More sophisticated versions of $analyzeWitness$ are discussed in the next Section.

Algorithm 1 The SDCL algorithm. Note that removing lines 9–12 results in the CDCL algorithm.

```
1  $\alpha := \emptyset$
2  while $true$ do
3      $\alpha := unitPropagate(F, \alpha)$
4      if $conflict\ found$ then
5          $C := analyzeConflict()$
6          $F := F \wedge C$
7          if $C$ is the empty clause then return UNSAT
8          $\alpha := backjump(C, \alpha)$
9      else if $P_\alpha(F)$ is satisfiable then
10         $C := analyzeWitness()$
11         $F := F \wedge C$
12         $\alpha := backjump(C, \alpha)$
13     else
14         if all variables are assigned then return SAT
15         $\alpha := \alpha \cup Decide()$
16     end
17 end
```

We can understand SDCL as a parameterized algorithm, since the use of different pruning predicates $P_\alpha(F)$ leads to distinct types of SDCL algorithms with possibly different underlying proof systems. In the following, we summarize the contributions of [15, 13] and explain the different pruning predicates and the corresponding proof systems that are known.

▶ **Definition 8.** *Given formula F and a (partial) assignment α, the* **positive reduct** *$p_\alpha(F)$ is the formula $\neg\alpha \wedge G$, where $G = \{touched_\alpha(D) \mid D \in F \text{ and } \alpha \models D\}$.*

That is, we only consider clauses satisfied by α, and among them, only the literals that are assigned. In [15] it is proved that $p_\alpha(F)$ is a valid pruning predicate. Moreover, a precise characterization of the redundancy achieved by $p_\alpha(F)$ is given: $p_\alpha(F)$ is satisfiable if and only if $\neg\alpha$ is set-blocked in F.

▶ **Definition 9.** *A clause C is* **set-blocked** *in a formula F if there exists a subset $L \subseteq C$ such that, for every clause D containing the negation of some literal in C, the clause $(C \setminus L) \vee \neg L \vee D$ contains two complementary literals.*

The results in [15] imply that a proof system based on resolution and set-blocked clauses has polynomial size proofs for the pigeon hole principle. It is also known [12] that set-blocked clauses are a particular case of SPR clauses. If one wants to obtain the full power of SPR clauses, the following pruning predicate is needed:

▶ **Definition 10.** *Given formula F and a (partial) assignment α, the* **filtered positive reduct** *$f_\alpha(F)$ is the formula $\neg\alpha \wedge G$, where $G = \{touched_\alpha(D) \mid D \in F \text{ and } F \wedge \alpha \not\vdash_1 untouched_\alpha(D)\}$.*

Again, a precise characterization of the power of $f_\alpha(F)$ is known [13]: $f_\alpha(F)$ is satisfiable if and only if $\neg\alpha$ is SPR with respect to F. Despite being harder to compute than $p_\alpha(F)$, the fact that $f_\alpha(F)$ is a subset of the clauses in $p_\alpha(F)$ makes it easier to check for satisfiability. Finally, another pruning predicate is given in [13] that achieves the full power of PR clauses, but it is not considered to be practical. We close this sequence of pruning predicates and their corresponding redundancy characterization with a novel pruning predicate and its corresponding redundancy notion.

▶ **Definition 11.** *Given formula F and a (partial) assignment α, the* **purely positive reduct** *$pp_\alpha(F)$ is the formula $\neg\alpha \wedge G$, where $G = \{satisfied_\alpha(D) \mid D \in F \text{ and } \alpha \models D\}$.*

Since all clauses in $pp_\alpha(F)$ are subclauses of clauses in $p_\alpha(F)$, whenever $pp_\alpha(F)$ is satisfiable, $p_\alpha(F)$ is also satisfiable. This proves that $pp_\alpha(F)$ is a pruning predicate, but we can be more precise about the notion of redundancy it corresponds to.

▶ **Definition 12.** *We say that a literal $l \in C$* **blocks** *C in F if an only if for every clause D in F containing literal $\neg l$, resolution between C and D gives a tautology. A clause C is* **blocked** *in F if and only if there exists some literal $l \in C$ that blocks C in F.*

▶ **Theorem 13.** *Given a formula F and an assignment α, the formula $pp_\alpha(F)$ is satisfiable if and only if the clause $\neg\alpha$ is blocked in F.*

Proof.

Left to right. let β be a model of $pp_\alpha(F)$. Since $\beta \models \neg\alpha$, we can take any literal $\neg l$ in $\neg\alpha$ satisfied by β. We now prove that $\neg l$ blocks $\neg\alpha$ in F. Let us consider a clause of the form $l \vee C \in F$. Since $l \in \alpha$ we have that $\alpha \models l \vee C$, and hence there is a clause of the form $l \vee satisfied_\alpha(C)$ in $pp_\alpha(F)$. Since $\beta \models pp_\alpha(F)$ and $\beta \models \neg l$, necessarily $\beta \models satisfied_\alpha(C)$. This means that C contains a literal from α different from l, and hence if we apply resolution between the clause $\neg\alpha$ and $l \vee C$ we obtain a tautology.

Right to left. Assume w.l.o.g. that the clause $\neg\alpha$ is blocked w.r.t. $\neg l$ in F. We prove that $\hat{\alpha} := \alpha \setminus \{l\} \cup \{\neg l\}$ is a model of $pp_\alpha(F)$. It is obvious that $\hat{\alpha}$ satisfies the clause $\neg\alpha \in pp_\alpha(F)$. Any other clause $D \in pp_\alpha(F)$ is of the form $satisfied_\alpha(C)$ for some $C \in F$ such that $\alpha \models C$. There are now in principle two cases:

(i) if D is not the unit clause l, it necessarily contains a literal from α different from l, and hence $\hat{\alpha}$ satisfies it.

(ii) If D is the unit clause l, this means that clause $C \in F$ does not contain any literal from α except for l. Thus, applying resolution between $\neg\alpha$ and C cannot give a tautology, contradicting the fact that $\neg\alpha$ is blocked w.r.t $\neg l$ in F. Hence, this case cannot take place. ◀

We finish this section with one important remark about the computation of reducts in SDCL: we need to add all already computed redundant clauses in the reduct computation when trying to find additional ones. Let us show why not doing this is incorrect. Given the satisfiable formula $(x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2) \wedge (x_3 \vee x_4)$, the SDCL solver might first build the assignment $\alpha = \{x_1, \neg x_2\}$. Its positive reduct is $(\neg x_1 \vee x_2) \wedge (x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2)$, which is satisfiable, and hence we learn the redundant clause $\neg x_1 \vee x_2$. If the solver now builds the assignment $\{\neg x_1, x_2\}$, the positive reduct w.r.t F is $(x_1 \vee \neg x_2) \wedge (x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2)$, which is again satisfiable and allows us to learn the clause $x_1 \vee \neg x_2$. However, adding the two learned redundant clauses to F makes it unsatisfiable. The solution is to build the second positive reduct w.r.t. F conjuncted with the first learned redundant clause. The corresponding reduct is $(x_1 \vee \neg x_2) \wedge (x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_2)$, which is now unsatisfiable and hence does not allow us to learn the second redundant clause.

A natural question that arises now is whether we also need to add all clauses that were derived using CDCL-style conflict analysis in a reduct. The answer is that we do not need to do so. The reason is that, given two formulas $G_1 \equiv G_2$, it holds that C is redundant w.r.t. G_1 if and only if C is redundant w.r.t G_2. Now, if the current formula that the SDCL solver has in its database is $F \wedge L \wedge R$, where F is the original formula, L are the lemmas derived by CDCL-style conflict analysis and R are the learned redundant clauses, it holds that $F \wedge L \wedge R \equiv F \wedge R$. Therefore, it is sufficient to compute redundant clauses w.r.t. $F \wedge R$ only. Having said that, it is better to compute reduntant clauses w.r.t $F \wedge R \wedge U$, where U denotes CDCL-derived unit clauses, because it results in smaller reducts and faster sub-solver calls. Note that for correctness, clauses in R are never deleted. This design decision prevents us from using off-the-shelf proof checkers like dpr-trim[1]. However, as we mention at the end of Section 5, this checker can be easily adapted.

4 Minimizing SDCL Learned Clauses

In Algorithm 1, we considered the function $AnalyzeWitness$ to always return $\neg\alpha$, which was correct due to the results presented in Section 3. However, adding the negation of the whole assignment results in a very large clause, and it is not a surprise that this is far from being useful in practice. Already in [15] it was proven that one can learn a much shorter clause: the negation of all decisions in α. We provide a simple proof that we use to justify that learning other clauses is also correct:

▶ **Theorem 14.** *Let F be a formula and C a clause that is redundant with respect to F. Any clause D obtained via resolution steps from $F \wedge C$ is also redundant with respect to F.*

[1] `https://github.com/marijnheule/dpr-trim`

Proof. Let us assume that F is satisfiable and prove that $F \wedge D$ also is. Since C is redundant w.r.t. F we know that $F \wedge C$ is satisfiable. We know that resolution generates logical consequences, and hence any model of $F \wedge C$ is also a model of $F \wedge C \wedge D$, which proves that $F \wedge D$ is satisfiable. ◀

It is well known that, if α is an assignment, starting from $\neg\alpha$ one can apply a series of resolution steps in order to derive a clause that only consists of decisions. If $\neg\alpha$ is redundant, the theorem proves that the decision-only clause is also redundant. However, learning the negation of all decisions is not the ideal situation for at least two reasons. The first one is that, according to experience from CDCL SAT solving, forcing the solver to learn clauses that only contain decisions leads to very poor performance in practice. It is certainly true that these clauses are small, but that is probably their only good property. The second reason is that not all decisions in α need to be present in the redundant clause. Similarly to what happens in CDCL, where usually not all decisions are responsible for a conflict, here not all decisions are responsible for the pruning predicate to be satisfiable. In order to fix these two issues, we modify $AnalyzeWitness$ so that it finds the smallest subset $\gamma \subseteq \alpha$ for which $P_\gamma(F)$ is satisfiable. This allows us to learn the hopefully much shorter clause $\neg\gamma$ and address one of the open problems mentioned in [14]: "checking if a subset of a conflict clause is propagation redundant with respect to the formula under consideration."

▶ **Example 15.** Consider $F = (x_1 \vee x_2 \vee x_4) \wedge (x_1 \vee \neg x_2 \vee x_5) \wedge (\neg x_1 \vee \neg x_4 \vee x_5) \wedge (x_2 \vee x_4 \vee \neg x_5) \wedge (x_3 \vee x_6 \vee \neg x_5)$ and assignment, $\alpha = \{x_1, x_4, x_5, \neg x_2\}$, where x_5 is the only non-decision. The positive reduct $p_\alpha(F)$ is $(\neg x_1 \vee \neg x_4 \vee \neg x_5 \vee x_2) \wedge (x_1 \vee x_2 \vee x_4) \wedge (x_1 \vee \neg x_2 \vee x_5) \wedge (\neg x_1 \vee \neg x_4 \vee x_5) \wedge (x_2 \vee x_4 \vee \neg x_5)$ and is satisfiable. Hence we could learn the redundant clause $\neg x_1 \vee \neg x_4 \vee x_2$ consisting of the negation of the decisions. However the subset $\gamma = \{x_1, \neg x_2\} \subseteq \alpha$ also has satisfiable positive reduct: $(\neg x_1 \vee x_2) \wedge (x_1 \vee x_2) \wedge (x_1 \vee \neg x_2)$ and hence we could learn the shorter clause $\neg x_1 \vee x_2$.

4.1 Hardness of Minimization

Unfortunately, as we prove, the problem of finding such small γ is NP-hard. Let us first formalize it as a decision problem:

▶ **Definition 16.** *Trail-Minimization: given a formula F, an assignment α and an integer $k \geq 0$, we want to know whether there is a subset $\gamma \subseteq \alpha$ of size k such that $p_\gamma(F)$ is satisfiable.*

Note that in the rest of the paper we focus on the positive reduct of the formula, and hence our approach allows us to obtain (short) set-blocked clauses. Before introducing the NP-hard problem that we use to prove the NP-hardness of Trail-Minimization, we need one definition:

▶ **Definition 17.** *Given α and β two assignments over the same variables $\{x_1, \ldots, x_n\}$, we say that $\beta < \alpha$ if $\beta \neq \alpha$ and for each $\neg x_i \in \alpha$ we also have that $\neg x_i \in \beta$.*

In other words, $\beta < \alpha$ if, considering an assignment as a sequence of n bits, the sequence of bits of β is pointwise smaller than the one of α.

▶ **Definition 18.** *Smaller-Model[17]: given a formula F and a total model α of F, we want to know whether there is a total model β of F such that $\beta < \alpha$.*

In [17] it is proved that Smaller-Model is NP-hard. Hence, a polynomial reduction from Smaller-Model to Trail-Minimization proves that the latter is also NP-hard.

▶ **Theorem 19.** *Trail-Minimization is NP-hard.*

Proof. Given (F, α) an instance of SMALLER-MODEL, we can partition $\alpha = \alpha^+ \cup \alpha^-$, where α^+ contains all positive literals in α, and α^- contains all negative literals. The reduction amounts to constructing an instance of TRAIL-MINIMIZATION as follows: the formula is $\hat{F} := F \cup \alpha^-$ (that is, we add to F all negative literals in α as unit clauses), the assignment is $\hat{\alpha} := \alpha$ and the integer $k := |\alpha|$. This can be computed in polynomial time and has polynomial size. Let us now check that (F, α) is a positive instance of SMALLER-MODEL if and only if $(\hat{F}, \hat{\alpha}, k)$ is a positive instance of TRAIL-MINIMIZATION.

Left to right: we know, by definition of SMALLER-MODEL, that there exists an assignment $\beta \models F$ such that $\beta < \alpha$. Since obviously $\hat{\alpha} \subseteq \hat{\alpha}$ and $|\hat{\alpha}| = k$, if we prove that $\beta \models p_{\hat{\alpha}}(\hat{F})$ we are done. Clause $\neg\alpha$ is satisfied by β because $\beta \neq \alpha$. Since $\hat{\alpha}$ is a total assignment, we have that $touched_{\hat{\alpha}}(C) = C$ for any clause $C \in \hat{F}$, hence any clause in $p_{\hat{\alpha}}(\hat{F})$ is either (i) a unit clause consisting of a literal in α^-, which is satisfied by β because $\beta < \alpha$ implies that $\beta \supseteq \alpha^-$ or (ii) a clause $C \in F$, which is of course satisfied by β since β is a model of F.

Right to left: the only subset of $\hat{\alpha}$ of size k is $\hat{\alpha}$ itself. Let us assume that $p_{\hat{\alpha}}(\hat{F})$ is satisfied by a model β. Since $\neg\alpha$ is a clause in $p_{\hat{\alpha}}(\hat{F})$, we know that $\beta \neq \alpha$. Also, since $p_{\hat{\alpha}}(\hat{F})$ contains all negative literals of α as unit clauses, we know that $\beta \supseteq \alpha^-$. Altogether, this proves that $\beta < \alpha$. The only missing piece is to prove that β satisfies F. This is not difficult to see: since α is a model of F, and $touched_\alpha(C) = C$ for any clause C, all clauses in F belong to $p_{\hat{\alpha}}(\hat{F})$ and β necessarily satisfies them. ◀

4.2 A MaxSAT Encoding for Trail-Minimization

Knowing that TRAIL-MINIMIZATION is a difficult optimization problem, and being somehow similar to SAT, it is very natural to try solving it using MaxSAT. Given a formula F, and an assignment α, we describe a partial MaxSAT formula $mp_\alpha(F)$ whose solutions correspond to a smallest $\gamma \subseteq \alpha$ such that $p_\gamma(F)$ is satisfiable.

Before formally defining $mp_\alpha(F)$, let us explain the intuition behind it. The main idea is that we have to determine which literals we can remove from α, giving a new assignment γ, such that $p_\gamma(F)$ is satisfiable. For each literal l in α, we add an additional variable r_l that indicates whether l is removed. Hence, a truth assignment over these variables induces an assignment $\gamma \subseteq \alpha$. The key point is to construct a formula such that when restricted with r_l's, it is essentially equivalent to $p_\gamma(F)$.

More formally, let us assume $\alpha = \{\alpha_1, \alpha_2, \ldots, \alpha_m\}$. We introduce three sets of additional variables:

- $\{r_1, r_2, \ldots, r_m\}$: indicate whether α_i is **r**emoved from α.
- $\{p_1, p_2, \ldots, p_m\}$: replace "**p**ositive" occurrences of α_i in $p(F, \alpha)$.
- $\{n_1, n_2, \ldots, n_m\}$: replace "**n**egative" occurrences of α_i in $p(F, \alpha)$.

Given a clause C, we denote by $\hat{C}$ the result of replacing in C, for $i = 1 \ldots m$, every occurrence of literal α_i by p_i and every occurrence of literal $\neg\alpha_i$ by n_i.

Our Partial MaxSAT formula $mp_\alpha(F)$ contains the following hard formulas (that can be easily converted into clauses), that enforce the semantics of the r, p and n variables:

$$\begin{array}{rcll} r_i & \rightarrow & \neg p_i & \text{for all } i = 1 \ldots m \\ r_i & \rightarrow & \neg n_i & \text{for all } i = 1 \ldots m \\ \neg r_i & \rightarrow & p_i = \alpha_i & \text{for all } i = 1 \ldots m \\ \neg r_i & \rightarrow & n_i = \neg\alpha_i & \text{for all } i = 1 \ldots m \end{array}$$

Intuitively, if r_i is false, and hence we do not to remove α_i from the assignment, then p_i is equivalent to α_i and n_i is equivalent to $\neg\alpha_i$. Otherwise, if r_i is removed, we force p_i and n_i to be false.

The rest of $mp_\alpha(F)$ is constructed by iterating over all clauses of $p_\alpha(F)$. For each clause $C \in p_\alpha(F)$, we add a set of hard clauses to $mp_\alpha(F)$, constructed as follows. If C is the clause $\neg\alpha$, we add the hard clause $\widehat{\neg\alpha}$. Otherwise C is of the form $S \vee D$, where S is the non-empty set of literals satisfied by α and D contains the remaining literals, which are touched, but not satisfied by α. The clauses to be added are:

$$\{\hat{S} \vee \hat{D} \vee r_i \mid i = 1 \dots m \text{ and } \alpha_i \in S\}$$

The idea here is that if we remove all literals in S from α, then C would not be satisfied and hence it should not appear in the positive reduct. The addition of the r_i's in the clauses guarantee that, if all of them are removed, and hence all r_i's are set to true, these clauses are all satisfied by the r_i's and hence they do not constrain the formula at all. On the other hand, if some literal in S is not removed, then the corresponding r_i is false and we essentially have the clause $\hat{S} \vee \hat{D}$, that is what we wanted to impose.

We want to note that we can obtain a smaller formula by, instead of adding multiple clauses of the form of $\hat{S} \vee \hat{D} \vee r_i$, introducing one auxiliary variable a_C for each clause $C = S \vee D$ and adding the clauses:

$$\begin{array}{ll} \hat{S} \vee \hat{D} \vee a_C & \\ \neg r_i \to \neg a_C & \text{for all } i = 1 \dots m \text{ such that } \alpha_i \in S \end{array}$$

Apart from these hard clauses and the hard ones imposing the semantics of r, p and n, our formula $mp_\alpha(F)$ is completed with the set of soft unit clauses $\{r_i \mid i = 1 \dots m\}$, expressing that we want to remove as many literals as possible while still satisfying the rest of the formula, which are hard clauses.

▶ **Example 20.** Let us revisit Example 15, where $\alpha = \{x_1, x_4, x_5, \neg x_2\}$ had $p_\alpha(F)$ satisfiable, but there was a smaller subset $\gamma = \{x_1, \neg x_2\}$ for which $p_\gamma(F)$ was also satisfiable. We use this example to illustrate our encoding. Let us consider that the variables related with x_i are p_i, n_i, r_i for $i \in \{1, 2, 4, 5\}$. We only show the hard clauses in $mp_\alpha(F)$ that are constructed from $p_\alpha(F)$. The implications defining the semantics of p, n, r are ignored, as well as the soft clauses, since those should be easy to understand.

$\mathbf{p_\alpha(F)}$	$\mathbf{mp_\alpha(F)}$
$\neg x_1 \vee \neg x_4 \vee \neg x_5 \vee x_2$	$n_1 \vee n_4 \vee n_5 \vee n_2$
$x_1 \vee x_2 \vee x_4$	$p_1 \vee n_2 \vee p_4 \vee r_1$
	$p_1 \vee n_2 \vee p_4 \vee r_4$
$x_1 \vee \neg x_2 \vee x_5$	$p_1 \vee p_2 \vee p_5 \vee r_1$
	$p_1 \vee p_2 \vee p_5 \vee r_2$
	$p_1 \vee p_2 \vee p_5 \vee r_5$
$\neg x_1 \vee \neg x_4 \vee x_5$	$n_1 \vee n_4 \vee p_5 \vee r_5$
$x_2 \vee x_4 \vee \neg x_5$	$n_2 \vee p_4 \vee n_5 \vee r_4$

Note that any literal x_1 is replaced by p_1 since $x_1 \in \alpha$. On the other hand, any literal x_2 is replaced by n_2 because $\neg x_2 \in \alpha$. The interesting fact is that if we set r_4, r_5 to true and r_1, r_2 to false, which means that we are removing x_4 and x_5 from α (hence obtaining γ) and propagate the implications, the hard clauses in $mp_\alpha(F)$ become equivalent to $p_\gamma(F)$. For example, take the first clause $n_1 \vee n_4 \vee n_5 \vee n_2$. Setting r_4 and r_5 to true causes the implications to unit propagate $\neg n_4$ and $\neg n_5$. Hence the clause is equivalent to $n_1 \vee n_2$. However, setting r_1 to false makes $n_1 = \neg x_1$ and setting r_2 to false makes $n_2 = x_2$. All in all, the clause is equivalent to $\neg x_1 \vee x_2$, which is the first clause of $p_\gamma(F)$.

If we take clause $n_2 \vee p_4 \vee n_5 \vee r_4$ we can see that it is satisfied due to r_4 and thus is not constraining the other variables. This is as expected, because if we remove x_4 from α, it no longer satisfies the clause $x_2 \vee x_4 \vee \neg x_5$ and hence it should not appear in the reduct.

Finally, the last case is a clause like $p_1 \vee n_2 \vee p_4 \vee r_1$. In this case r_1 is false and hence the last literal in the clause disappears. Also, since r_4 is true, it makes p_4 false due to the implications, and r_1, r_2 being false unit propagates $p_1 = x_1$ and $n_2 = x_2$, hence making the clause equivalent to $x_1 \vee x_2$, which is precisely the clause that appears in $p_\gamma(F)$. All in all, if we set the r variables to the appropriate values we can obtain the positive reduct of any subset of α. Below, we formally prove that this encoding is indeed correct.

▶ **Theorem 21.** *Given a formula F and an assignment $\alpha = \{\alpha_1, \ldots, \alpha_m\}$, it holds that the smallest subset $\gamma \subseteq \alpha$ such that $p_\gamma(F)$ is satisfiable has size $m - k$ if and only if the optimal solution to $mp_\alpha(F)$ satisfies k soft clauses.*

Proof. We prove something slightly stronger: there exists $\gamma \subseteq \alpha$ of size $m - k$ such that $p_\gamma(F)$ is satisfiable if and only if there exists an assignment that satisfies all hard clauses in $mp_\alpha(F)$ and exactly k soft clauses.

Left to right. let us consider $\gamma \subseteq \alpha$ of size $m - k$ with $p_\gamma(F)$ satisfiable, and let δ be a model for it. We build an assignment the satisfies all hard clauses in $mp_\alpha(F)$ and exactly k soft clauses as follows. The first remark is that $mp_\alpha(F)$ only consists of the variables r_i, p_i, n_i and the ones appearing in α_i, for $i = 1 \ldots m$ and hence we have to build an assignment β over those. For $i = 1 \ldots m$ we add r_i to β if $\alpha_i \notin \gamma$, and add $\neg r_i$ otherwise. Since there are k literals α_i not belonging to γ, it is clear that β satisfies exactly k soft clauses. The assignment β is completed by making it coincide with δ on the variables of γ and take arbitrary values for the variables of $\alpha \setminus \gamma$. If we now unit propagate these values on the implications that define the semantics of r, p and n, we complete β to define values for all p_i and n_i.

Let us now see that β satisfies the hard clauses in $mp_\alpha(F)$. The implications defining the semantics of the variables are obviously satisfied. Clause $\widehat{\neg\alpha}$ is also satisfied: we know that this clause is of the form $n_1 \vee n_2 \vee \ldots \vee n_m$. Since $\delta \models \neg\gamma$, there is a literal $\alpha_k \in \gamma$ such that $\delta \models \neg\alpha_k$. By the definition of β, we know that $\neg r_k \in \beta$ and hence the formula $\neg r_k \;\rightarrow\; n_k = \neg\alpha_k$ propagates n_k to be true in β and hence satisfy $\widehat{\neg\alpha}$.

Let us now take another clause $C \in mp_\alpha(F)$, which is necessarily of the form $\hat{S} \vee \hat{D} \vee r_i$, with $S \vee D \in p_\alpha(F)$ and r_i be such that $\alpha_i \in S$. If $\beta \models r_i$ we are done. Otherwise, it is because $\alpha_i \in \gamma$. Hence, the clause $S \vee D$ is satisfied by γ due to literal $\alpha_i \in S$ and $p_\gamma(F)$ contains the clause $touched_\gamma(S \vee D)$. Thus, $\delta \models touched_\gamma(S \vee D)$. Let us consider that case where δ satisfies $\alpha_j \in touched_\gamma(S \vee D)$ (the other case is that is satisfies some $\neg\alpha_j$ and the proof is similar). Since $\alpha_j \in \gamma$, we have that $r_j \notin \beta$ and the formula $\neg r_j \;\rightarrow\; p_j = \alpha_j$ guarantees that $\beta \models p_j$. We only have to realize that p_j is a literal in $\hat{S} \vee \hat{D}$, to conclude that $\beta \models C$.

Right to left. let us consider an assignment β that satisfies all hard clauses in $mp_\alpha(F)$ and exactly k soft clauses. We build a subset $\gamma \subseteq \alpha$ of size $m - k$ such that $p_\gamma(F)$ is satisfiable. As expected, γ is constructed by removing from α all α_i such that $\beta \models r_i$. It is obvious that $|\gamma| = m - k$, because β satisfies exactly k unit clauses of the form r_i.

In order to prove that $p_\gamma(F)$ is satisfiable, let us build an assignment δ that coincides with β over all variables in γ and prove that it is a model. The first clause in $p_\gamma(F)$ to consider is $\neg\gamma$. Since $\beta \models n_1 \vee \ldots \vee n_m$ and it satisfies the clauses $r_i \;\rightarrow\; \neg n_i$ it necessarily satisfies some n_i such that r_i is false. Due to the clause $\neg r_i \rightarrow n_i = \neg\alpha_i$, it also satisfies $\neg\alpha_i$. Since r_i is false in β we have that $\alpha_i \in \gamma$ and hence, by the definition of δ, it satisfies $\neg\alpha_i$. This proves that $\delta \models \neg\gamma$.

Let us now take another clause in $p_\gamma(F)$, which is necessarily of the form $\mathit{touched}_\gamma(C)$ for some $C \in F$ such that $\gamma \models C$. Since α is a superset of γ, obviously $\alpha \models C$, and hence a clause of the form $\mathit{touched}_\alpha(C)$ belongs to $p_\alpha(F)$. This clause in $p_\alpha(F)$ is of the form $S \vee D$, with S containing all literals satisfied by α, and thus we have in $mp_\alpha(F)$ hard clauses of the form $\hat{S} \vee \hat{D} \vee r_i$ for every i with $\alpha_i \in S$. If $\gamma \models C$ it is because it satisfies some $\alpha_i \in C$ with r_i being false in β. Hence, the existence of the clause $\hat{S} \vee \hat{D} \vee r_i$ implies that $\beta \models \hat{S} \vee \hat{D}$. We know that $\hat{S} \vee \hat{D}$ is a disjunction of positive p's and n's literals. Let us assume that it satisfies some p_k (the case n_k is similar). Note that r_k has to be false because otherwise the implications force p_k to be false. Hence β satisfies some α_k such that r_k is false and hence $\alpha_k \in \gamma$, which means that δ also satisfies α_k because they coincide over γ. Since $\alpha_k \in \gamma$, it belongs to $\mathit{touched}_\gamma(C)$ which is the clause that we wanted δ to satisfy. ◀

4.3 Practical Remarks

The previous encoding would allow us to learn the redundant clause $C := \neg\gamma$. However, SDCL (see Algorithm 1) requires C to be asserting (i.e. containing exactly one literal of the last decision level, and hence allowing it to unit propagate after backjumping). In order to achieve this property, we first observe that, being the negation of a subset of the current assignment, clause $\neg\gamma$ is a conflict. Hence, we can apply standard CDCL conflict analysis to it and obtain a clause that is asserting. For those familiar with SMT, this is essentially what DPLL(T)-based SMT solvers do when they analyze theory conflicts. Thanks to Theorem 14, we can guarantee that the final clause we obtain in this process is redundant and hence can be safely added. Moreover, as can be seen in Section 5, the size of this clause tends to be even smaller than $\neg\gamma$. In addition, this method allows us to learn clauses that are stronger than set-blocked clauses.

▶ **Example 22.** Let us consider $F = (\neg x_0 \vee x_1) \wedge (\neg x_0 \vee \neg x_2) \wedge (x_0 \vee x_2) \wedge (\neg x_1 \vee x_2 \vee x_4) \wedge (x_3 \vee x_6 \vee \neg x_5)$. Assume the SAT solver builds the assignment, from left to right, $\{\mathbf{x_0}, x_1, \neg x_2, x_4, \mathbf{x_5}\}$ where literals in bold are decisions. If we pick $\gamma = \{x_0, x_1, \neg x_2\}$, we can see that its reduct $(\neg x_0 \vee \neg x_1 \vee x_2) \wedge (\neg x_0 \vee x_1) \wedge (\neg x_0 \vee \neg x_2) \wedge (x_0 \vee x_2)$ is satisfiable. This mean that we can learn $\neg x_0 \vee \neg x_1 \vee x_2$. Now, in two resolution steps with the reasons of x_1 and $\neg x_2$ which are $\neg x_0 \vee x_1$ and $\neg x_0 \vee \neg x_2$, respectively, we can derive the redundant clause $\neg x_0$. However, assignment x_0 does not have satisfiable positive reduct. In fact, clause $\neg x_0$ is not even SPR. It can be checked that it is indeed PR (a possible witness is $w = \{\neg x_0, x_2, x_3\}$). This shows that by combining the positive reduct with posterior resolution steps, we can obtain clauses with stronger redundancy properties than set-blocked clauses, which is the one obtained by using the positive reduct alone.

One final question that we want to address is whether, in an SDCL implementation, we should (i) first ask a SAT solver whether $p_\alpha(F)$ is satisfiable, and then, if this is the case, ask a MaxSAT to possibly find a smaller $\gamma \subseteq \alpha$ for which p_γ is also satisfiable, or (ii) directly ask a MaxSAT solver whether there exists a subset of $\gamma \subseteq \alpha$ for which p_γ is satisfiable. The following result sheds some light on this:

▶ **Proposition 23.** *Given a formula F and an assignment α, if $mp_\alpha(F)$ has some solution, then $p_\alpha(F)$ is satisfiable.*

Proof. By Theorem 21, if $mp_\alpha(F)$ has some solution satisfying k soft clauses, we can build an assignment $\gamma \subseteq \alpha$ for which $p_\gamma(F)$ is satisfiable, and let β be a model for it. It is now easy to prove that $\delta := \beta \cup \alpha\backslash\gamma$ is a model for $p_\alpha(F)$. The first clause in $p_\alpha(F)$ is $\neg\alpha$, of which the clause $\neg\gamma \in p_\gamma(F)$ is a subclause and hence $\beta \models \neg\alpha$. This implies that $\delta \models \neg\alpha$. Now, any other clause C in $p_\alpha(F)$ is of the form $touched_\alpha(D)$ for some $D \in F$ such that $\alpha \models D$. Expressing $touched_\alpha(D)$ as $touched_\gamma(D) \vee touched_{\alpha\backslash\gamma}(D)$ helps in our reasoning. If $\gamma \models D$ then $touched_\gamma(D) \in p_\gamma(F)$ and hence β satisfies it. In this case $\delta \models C$. Otherwise, $\gamma \not\models D$ but since $\alpha \models D$ it has to be that $\alpha\backslash\gamma \models D$. This implies that $\alpha\backslash\gamma \models touched_{\alpha\backslash\gamma}(D)$ and hence $\delta \models C$. ◀

This result shows that by directly calling the MaxSAT solver on $mp_\alpha(F)$, the solver cannot learn more redundant clauses than if we call the SAT solver on $p_\alpha(F)$. Hence, it makes sense to first call the SAT solver, which should be faster and then, only if $p_\alpha(F)$ has been found to be satisfiable, call the MaxSAT solver to possibly learn a shorter redundant clause. If we make an analogy with CDCL, checking $p_\alpha(F)$ for satisfiable would be the equivalent of unit propagation and solving the MaxSAT formula $mp_\alpha(F)$ the equivalent of conflict analysis.

5 Experimental Evaluation

5.1 Implementation

We implemented SDCL with the clause minimization techniques described in the previous section on top of the SAT solver MapleSAT [18]. In order to solve the MaxSAT queries, we have used EvalMaxSAT [2], an efficient solver that provides a very convenient C++ API.

The changes in Algorithm 1 are limited to $analyzeWitness$. Once we know that $p_\alpha(F)$ is satisfiable, we construct $mp_\alpha(F)$ and obtain the optimal solution with EvalMaxSAT. This induces a clause $\neg\gamma$, to which standard CDCL conflict analysis is applied in order to derive an asserting clause, which is learned and used to backjump.

This general idea is refined in different directions. First of all, we do not apply this procedure before every decision. Without redundant-clause minimization, this might be a bad decision design, since the length of the learned clause coincides with the decision level, and hence we should apply it as soon as possible. With clause minimization enabled, applying this procedure at high decision levels can still give short redundant clauses. Since, as a consequence of Proposition 23, we know that long assignments are more likely to produce redundant clauses, it makes sense to delay the check until the assignment is large enough. However, there is a certain trade-off because at high decision levels, $p_\alpha(F)$ and $mp_\alpha(F)$ are larger formulas and hence can be more difficult to solve. Our strategy relies on defining a decision level goal and trying to derive a redundant clause only when we are at this decision level. We compute the ratio of success (i.e. a redundant clause has been derived) of the procedure calls; if this ratio is lower than a certain amount (e.g. 15%), we increment the decision level goal; if it is higher, we decrement it. The rationale for this strategy is to achieve a predefined ratio of successful calls but not invoking the technique too often.

The second refinement is that our final asserting clause is not always shorter than the clause obtained by negating all decisions. In those rare situations, we learn the only-decisions clause. A final refinement consists of only learning clauses of size at most 3. In our SDCL implementation, we cannot delete redundant clauses we have learned in SDCL unless we also delete all CDCL learned clauses that have been derived using them. This is why we have to be very cautious and only learn high-quality redundant clauses.

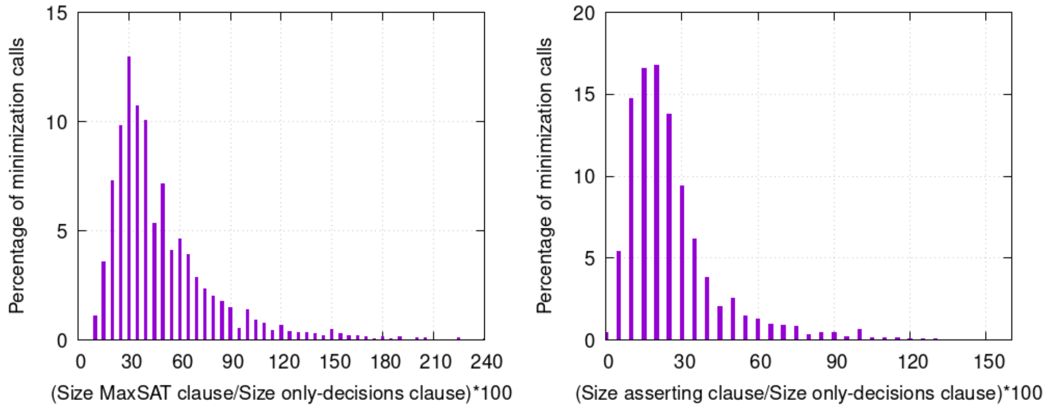

Figure 1 Distribution of the amount of minimization achieved in the clause returned by the MaxSAT solver (left) and in the final asserting clause (right).

One final remark is that, unlike previous SDCL implementations [15, 13], we have not modified the decision heuristics of the solver. We believe that, due to our conflict minimization techniques, picking the exact right variable at low decision levels is not so critical.

5.2 Experimental Results

We have evaluated our system on the benchmarks used in [13, 22]. In order to assess the impact of our Max-SAT based minimization technique, we have presented in Figure 1 results about one execution of our system on a mutilated chess board benchmark of size 20. Data for other benchmarks follow along the same lines. On the left-hand histogram, a bar over the x-point 30 with height 10 means that, in 10% of the calls to minimization, the percentage (Size MaxSAT clause / Size Only-decisions clause)*100 is between 30% and 35%. That is, the size of the MaxSAT clause was around one third the size of the only-decisions clause. The histogram on the right plots the same data, but comparing the final asserting clause with respect to the only-decisions clause. One can observe that the percentage of reduction is important and comes from the MaxSAT invocation as well as from the subsequent conflict analysis call that returns the final asserting clause.

We also studied the cost of calling the SAT solver for checking the satisfiability of $p_\alpha(F)$ and the MaxSAT solver for processing $mp_\alpha(F)$. Our experiments revealed that the cost of the SAT solver call never exceeds 2% of the total runtime, whereas the calls to MaxSAT are more expensive and they can account for almost 30% of the total runtime.

Finally, we present in Table 1 results on the performance of our system compared to others. We want to remark that no change to the decision heuristic of the baseline solver has been made. We chose Kissat [3] as a representative of a state-of-the-art CDCL SAT solver; SaDiCaL [15, 13] as the only other existing SDCL system; and our system MapleSDCL. For SaDiCaL, we used two versions, one using the positive reduct and one using the filtered positive reduct. Regarding MapleSDCL, we present three configurations: CDCL corresponds to the standard MapleSAT solver, implementing CDCL; SDCL represents a configuration using the positive reduct but no MaxSAT-based minimization, i.e., learning the only-decisions clause, and finally, SDCL-min uses the MaxSAT-based minimization presented in this paper. The table reports the number of seconds needed to solve each benchmark for each system. Due to the use of internal time limits in EvalMaxSAT, the exact behavior of SDCL-min is not reproducible. In order to have a higher confidence in its results we have run it 10

Table 1 Performance of different systems on mutilated chess board and bipartite perfect matching problems. Times are in seconds.

		SaDiCaL		MapleSDCL		
Benchmark	Kissat	Positive	Filtered	CDCL	SDCL	SDCL-min
mchess14	4.6	5682	3.6	11.7	7.3	2.7 (10)
mchess15	30.1	> 7200	13.8	54.3	24.7	5.5 (10)
mchess16	107	> 7200	19.5	439	191	9 (10)
mchess17	2293	> 7200	64.8	5038	517	25.8 (10)
mchess18	352	> 7200	71.8	> 7200	3803	52.8 (10)
mchess19	> 7200	> 7200	> 7200	> 7200	3578	128 (10)
mchess20	3720	> 7200	> 7200	> 7200	> 7200	369 (10)
mchess21	> 7200	> 7200	> 7200	> 7200	> 7200	977 (10)
mchess22	> 7200	> 7200	> 7200	> 7200	> 7200	4507 (7)
mchess23	> 7200	> 7200	> 7200	> 7200	> 7200	6041 (2)
randomG-Mix-17	> 7200	> 7200	> 7200	2837	1916	257 (10)
randomG-Mix-18	> 7200	> 7200	> 7200	> 7200	> 7200	1683 (10)
randomG-n17	> 7200	> 7200	> 7200	1266	688	157 (10)
randomG-n18	> 7200	> 7200	> 7200	> 7200	> 7200	2350 (10)

times on each benchmark. For this system, the number in parenthesis corresponds to the number of executions that solved that instance within the time limit of 7200 seconds, and the runtime is the average over those successful executions.

For pigeon-hole problems, we observed the same behavior reported in [13], dedicated decision heuristics are needed and they do not even work if the formula is scrambled. Tseitin formulas, and other benchmarks from the SAT competition used in [22] are out of reach of our system, probably to the fact that our current minimization procedure uses the positive reduct, and not the filtered one. All in all, we observed that our technique gives important benefit on mutilated chess board and bipartite perfect matching problems, outperforming all other competitors. We want to remark that the data showed in [22] indicate that their preprocessing-based technique for detecting PR clauses is able to achieve better performance. However, our goal was to show how far the SDCL framework can be improved, and we believe that results confirm that there is still a large space for improvement.

Finally, we would like to mention that MapleSDCL is able to produce proofs that are checkable with **dpr-trim**. However, this checker assumes that PR clauses are computed with respect to the current formula, including all learned lemmas. As already explained, we compute clauses that are PR with respect to $F \wedge R \wedge U$, where F is the initial formula, R contains all redundant clauses we have learned, and U is the set of all CDCL-like unit lemmas. This has forced us to add simple 6 lines of code to the checker that control which clauses have to be used when checking that the added PR clauses are correct.

6 Conclusions and Future Work

We have shown how redundant clauses learned within the SDCL approach can be shortened by encoding the problem as a partial MaxSAT formula. Via extensive empirical evaluation we show that our technique greatly improves the performance of SDCL over families of formulas for which it was theoretically known that SDCL had a competitive advantage with respect to CDCL. We outline several directions for future work. First of all, we could adapt the technique to also work for the filtered positive reduct. Secondly, there is a very interesting research opportunity in developing sophisticated adaptive strategies aimed at deciding as to when the SDCL solver should attempt to learn a redundant clause. Finally, parallelization of the MaxSAT calls would greatly improve the runtime of SDCL-based systems.

References

1 Albert Atserias, Johannes Klaus Fichte, and Marc Thurley. Clause-learning algorithms with many restarts and bounded-width resolution. *J. Artif. Intell. Res.*, 40:353–373, 2011. doi:10.1613/jair.3152.

2 Florent Avellaneda. A short description of the solver EvalMaxSAT. In Fahiem Bacchus, Jeremias Berg, Matti Järvisalo, and Ruben Martins, editors, *MaxSAT Evaluation 2020*, pages 8–9, 2020.

3 Armin Biere, Katalin Fazekas, Mathias Fleury, and Maximillian Heisinger. CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020. In Tomas Balyo, Nils Froleyks, Marijn Heule, Markus Iser, Matti Järvisalo, and Martin Suda, editors, *Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions*, volume B-2020-1 of *Department of Computer Science Report Series B*, pages 51–53. University of Helsinki, 2020.

4 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.

5 Armin Biere and Daniel Kröning. Sat-based model checking. In Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors, *Handbook of Model Checking*, pages 277–303. Springer, 2018. doi:10.1007/978-3-319-10575-8_10.

6 Avrim L Blum and Merrick L Furst. Fast planning through planning graph analysis. *Artificial intelligence*, 90(1-2):281–300, 1997.

7 Cristian Cadar, Vijay Ganesh, Peter M Pawlowski, David L Dill, and Dawson R Engler. EXE: Automatically Generating Inputs of Death. *ACM Transactions on Information and System Security (TISSEC)*, 12(2):1–38, 2008.

8 Edmund M Clarke Jr, Orna Grumberg, Daniel Kroening, Doron Peled, and Helmut Veith. *Model Checking.* MIT press, 2018.

9 Stephen A. Cook. The Complexity of Theorem-Proving Procedures. In *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing*, pages 151–158, 1971. doi:10.1145/800157.805047.

10 Stephen A. Cook. A short proof of the pigeon hole principle using extended resolution. *SIGACT News*, 8(4):28–32, 1976. doi:10.1145/1008335.1008338.

11 Julian Dolby, Mandana Vaziri, and Frank Tip. Finding Bugs Efficiently With a SAT Solver. In *Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 195–204, 2007. doi:10.1145/1287624.1287653.

12 Marijn J. H. Heule, Benjamin Kiesl, and Armin Biere. Short proofs without new variables. In Leonardo de Moura, editor, *Automated Deduction – CADE 26 – 26th International Conference on Automated Deduction, Gothenburg, Sweden, August 6-11, 2017, Proceedings*, volume 10395 of *Lecture Notes in Computer Science*, pages 130–147. Springer, 2017. doi:10.1007/978-3-319-63046-5_9.

13 Marijn J. H. Heule, Benjamin Kiesl, and Armin Biere. Encoding redundancy for satisfaction-driven clause learning. In Tomás Vojnar and Lijun Zhang, editors, *Tools and Algorithms for the Construction and Analysis of Systems – 25th International Conference, TACAS 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings, Part I*, volume 11427 of *Lecture Notes in Computer Science*, pages 41–58. Springer, 2019. doi:10.1007/978-3-030-17462-0_3.

14 Marijn J. H. Heule, Benjamin Kiesl, and Armin Biere. Strong extension-free proof systems. *J. Autom. Reason.*, 64(3):533–554, 2020. doi:10.1007/s10817-019-09516-0.

15 Marijn J. H. Heule, Benjamin Kiesl, Martina Seidl, and Armin Biere. Pruning through satisfaction. In Ofer Strichman and Rachel Tzoref-Brill, editors, *Hardware and Software: Verification and Testing – 13th International Haifa Verification Conference, HVC 2017, Haifa, Israel, November 13-15, 2017, Proceedings*, volume 10629 of *Lecture Notes in Computer Science*, pages 179–194. Springer, 2017. doi:10.1007/978-3-319-70389-3_12.

16 Marijn J. H. Heule, Oliver Kullmann, and Victor W. Marek. Solving and verifying the boolean pythagorean triples problem via cube-and-conquer. In Nadia Creignou and Daniel Le Berre, editors, *Theory and Applications of Satisfiability Testing – SAT 2016 – 19th International Conference, Bordeaux, France, July 5-8, 2016, Proceedings*, volume 9710 of *Lecture Notes in Computer Science*, pages 228–245. Springer, 2016. doi:10.1007/978-3-319-40970-2_15.

17 Lefteris M. Kirousis and Phokion G. Kolaitis. The complexity of minimal satisfiability problems. *Inf. Comput.*, 187(1):20–39, 2003. doi:10.1016/S0890-5401(03)00037-3.

18 Jia Hui Liang, Vijay Ganesh, Pascal Poupart, and Krzysztof Czarnecki. Learning rate based branching heuristic for SAT solvers. In Nadia Creignou and Daniel Le Berre, editors, *Theory and Applications of Satisfiability Testing – SAT 2016 – 19th International Conference, Bordeaux, France, July 5-8, 2016, Proceedings*, volume 9710 of *Lecture Notes in Computer Science*, pages 123–140. Springer, 2016. doi:10.1007/978-3-319-40970-2_9.

19 João Marques-Silva, Inês Lynce, and Sharad Malik. Conflict-driven clause learning SAT solvers. 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 133–182. IOS Press, 2021. doi:10.3233/FAIA200987.

20 Knot Pipatsrisawat and Adnan Darwiche. On the power of clause-learning SAT solvers as resolution engines. *Artif. Intell.*, 175(2):512–525, 2011. doi:10.1016/j.artint.2010.10.002.

21 Mukul R. Prasad, Armin Biere, and Aarti Gupta. A survey of recent advances in sat-based formal verification. *Int. J. Softw. Tools Technol. Transf.*, 7(2):156–173, 2005. doi:10.1007/s10009-004-0183-4.

22 Joseph E. Reeves, Marijn J. H. Heule, and Randal E. Bryant. Preprocessing of propagation redundant clauses. In Jasmin Blanchette, Laura Kovács, and Dirk Pattinson, editors, *Automated Reasoning – 11th International Joint Conference, IJCAR 2022, Haifa, Israel, August 8-10, 2022, Proceedings*, volume 13385 of *Lecture Notes in Computer Science*, pages 106–124. Springer, 2022. doi:10.1007/978-3-031-10769-6_8.

23 João P. Marques Silva and Karem A. Sakallah. Invited tutorial: Boolean satisfiability algorithms and applications in electronic design automation. In E. Allen Emerson and A. Prasad Sistla, editors, *Computer Aided Verification, 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000, Proceedings*, volume 1855 of *Lecture Notes in Computer Science*, page 3. Springer, 2000. doi:10.1007/10722167_3.

24 Grigori S Tseitin. On the complexity of derivation in propositional calculus. *Automation of reasoning: 2: Classical papers on computational logic 1967–1970*, pages 466–483, 1983.

25 Yichen Xie and Alexander Aiken. Saturn: A SAT-Based Tool for Bug Detection. In *Proceedings of the 17th International Conference on Computer Aided Verification, CAV 2005*, pages 139–143, 2005. doi:10.1007/11513988_13.

26 Lintao Zhang and Sharad Malik. Conflict driven learning in a quantified boolean satisfiability solver. In Lawrence T. Pileggi and Andreas Kuehlmann, editors, *Proceedings of the 2002 IEEE/ACM International Conference on Computer-aided Design, ICCAD 2002, San Jose, California, USA, November 10-14, 2002*, pages 442–449. ACM / IEEE Computer Society, 2002. doi:10.1145/774572.774637.