Encoding Redundancy for Satisfaction-Driven Clause Learning[⋆]

Marijn J.H. Heule[1], Benjamin Kiesl[2,3], and Armin Biere[4]

[1] Department of Computer Science, The University of Texas at Austin, USA
[2] Institute of Logic and Computation, TU Wien, Vienna, Austria
[3] CISPA Helmholtz Center for Information Security, Saarbrücken, Germany
[4] Institute for Formal Models and Verification, JKU Linz, Austria

Abstract. Satisfaction-Driven Clause Learning (SDCL) is a recent SAT solving paradigm that aggressively trims the search space of possible truth assignments. To determine if the SAT solver is currently exploring a dispensable part of the search space, SDCL uses the so-called positive reduct of a formula: The positive reduct is an easily solvable propositional formula that is satisfiable if the current assignment of the solver can be safely pruned from the search space. In this paper, we present two novel variants of the positive reduct that allow for even more aggressive pruning. Using one of these variants allows SDCL to solve harder problems, in particular the well-known Tseitin formulas and mutilated chessboard problems. For the first time, we are able to generate and automatically check clausal proofs for large instances of these problems.

Introduction

Conflict-driven clause learning (CDCL) [26, 28] is the most successful paradigm for solving satisfiability (SAT) problems and therefore CDCL solvers are pervasively used as reasoning engines to construct and verify systems. However, CDCL solvers still struggle to handle some important applications within reasonable time. These applications include the verification of arithmetic circuits, challenges from cryptanalysis, and hard combinatorial problems. There appears to be a theoretical barrier to dealing with some of these applications efficiently.

At its core, CDCL is based on the resolution proof system, which means that the same limitations that apply to resolution also apply to CDCL. Most importantly, there exist only exponentially large resolution proofs for several seemingly easy problems [15, 33], implying that CDCL solvers require exponential time to solve them. A recent approach to breaking this exponential barrier is the *satisfaction-driven clause learning* (SDCL) paradigm [20], which can automatically find short proofs of pigeon-hole formulas in the PR proof system [19].

SDCL extends CDCL by pruning the search space of truth assignments more aggressively. While a pure CDCL solver learns only clauses that can be efficiently

derived via resolution, an SDCL solver also learns stronger clauses. The initial approach to learning these clauses is based on the so-called *positive reduct*: Given a formula and a partial truth assignment, the positive reduct is a simple propositional formula encoding the question of whether the assignment can be pruned safely from the search space. In cases where the positive reduct is satisfiable, the solver performs the pruning by learning a clause that blocks the assignment.

Although the original SDCL paradigm can solve the hard pigeon-hole formulas, we observe that it is not sophisticated enough to deal with other hard formulas that require exponential-size resolution proofs, such as Tseitin formulas over expander graphs [32, 33] or mutilated chessboard problems [1, 13, 27]. In this paper, we deal with this issue and present techniques that improve the SDCL paradigm. In particular, we introduce new variants of the above-mentioned positive reduct that allow SDCL to prune the search space even more aggressively.

In a first step, we explicitly formalize the notion of a *pruning predicate*: For a formula F and a (partial) assignment α, a pruning predicate is a propositional formula that is satisfiable if α can be pruned in a satisfiability-preserving way. Ideally, a pruning predicate is easily solvable while still pruning the search space as much as possible. We then present two novel pruning predicates of which one, the *filtered positive reduct*, is easier to solve and arguably more useful in practice while the other, the PR *reduct*, allows for stronger pruning.

In many applications, it is not enough that a solver just provides a simple yes/no answer. Especially when dealing with mathematical problems or safety-critical systems, solvers are required to provide automatically checkable proofs that certify the correctness of their answers. The current state of the art in proof generation and proof checking is to focus on *clausal proofs*, which are specific sequences of clause additions and clause removals. Besides the requirement that SAT solvers in the main track of the SAT competition must produce such clausal proofs, there also exist corresponding proof checkers whose correctness has been verified by theorem provers, as first proposed in a seminal TACAS'17 paper [12].

We implemented a new SDCL solver, called SaDiCaL, that can solve the pigeon-hole formulas, the Tseitin formulas, and the mutilated chessboard problems due to using the filtered positive reduct. Our solver also produces PR proofs [19]. We certify their correctness by translating them via DRAT proofs [17] to LRAT proofs, which are then validated by a formally verified proof checker [18].

Existing approaches to solving the Tseitin formulas are based on symmetry breaking [14] or algebraic reasoning, in particular Gaussian elimination [9, 31, 3, 21]. However, the respective tools do not output machine-checkable proofs. Moreover, approaches based on symmetry breaking and Gaussian elimination depend strongly on the syntactic structure of formulas to identify symmetries and cardinality constraints, respectively. They are therefore vulnerable to syntactic changes that do not affect the semantics of a formula. In contrast, SDCL reasons on the semantic level, making it less prone to syntactic changes.

The main contributions of this paper are as follows: (1) We explicitly formulate the notion of a pruning predicate, which was used only implicitly in the original formulation of SDCL. (2) We present two novel pruning predicates that

generalize the positive reduct. (3) We implemented a new SDCL solver, called SaDiCaL, that uses one of our new pruning predicates. (4) We show by an experimental evaluation that this new pruning predicate enables SaDiCaL to produce short proofs (without new variables) of Tseitin formulas and of mutilated chessboard problems.

Preliminaries

Propositional logic. We consider propositional formulas in *conjunctive normal form* (CNF), which are defined as follows. A *literal* is defined to be either a variable x (a *positive literal*) or the negation $\overline{x}$ of a variable x (a *negative literal*). The *complement* $\overline{l}$ of a literal l is defined as $\overline{l} = \overline{x}$ if $l = x$ and $\overline{l} = x$ if $l = \overline{x}$. Accordingly, for a set L of literals, we define $\overline{L} = \{\overline{l} \mid l \in L\}$. A *clause* is a disjunction of literals. A *formula* is a conjunction of clauses. We view clauses as sets of literals and formulas as sets of clauses. For a set L of literals and a formula F, we define $F_L = \{C \in F \mid C \cap L \neq \emptyset\}$. By $var(F)$ we denote the variables of a literal, clause, or formula F. For convenience, we treat $var(F)$ as a variable if F is a literal, and as a set of variables otherwise.

Satisfiability. An *assignment* is a function from a set of variables to the truth values 1 (*true*) and 0 (*false*). An assignment is *total* w.r.t. a formula F if it assigns a truth value to all variables $var(F)$ occurring in F; otherwise it is *partial*. A literal l is *satisfied* by an assignment α if l is positive and $\alpha(var(l)) = 1$ or if it is negative and $\alpha(var(l)) = 0$. A literal is *falsified* by an assignment α if its complement is satisfied by α. A clause is satisfied by an assignment α if it contains a literal that is satisfied by α. Finally, a formula is satisfied by an assignment α if all its clauses are satisfied by α. A formula is *satisfiable* if there exists an assignment that satisfies it. We often denote assignments by sequences of the literals they satisfy. For instance, $x\,\overline{y}$ denotes the assignment that assigns 1 to x and 0 to y. For an assignment α, $var(\alpha)$ denotes the variables assigned by α. For a set L of non-contradictory literals, we denote by α_L the assignment obtained from α by making all literals in L true and assigning the same value as α to other variables not in $var(L)$.

Formula simplification. We refer to the empty clause by $\bot$. Given an assignment α and a clause C, we define $C{\restriction}\alpha = \top$ if α satisfies C; otherwise, $C{\restriction}\alpha$ denotes the result of removing from C all the literals falsified by α. For a formula F, we define $F{\restriction}\alpha = \{C{\restriction}\alpha \mid C \in F \text{ and } C{\restriction}\alpha \neq \top\}$. We say that an assignment α *touches* a clause C if $var(\alpha) \cap var(C) \neq \emptyset$. A *unit clause* is a clause with only one literal. The result of applying the *unit clause rule* to a formula F is the formula $F{\restriction}l$ where (l) is a unit clause in F. The iterated application of the unit clause rule to a formula F, until no unit clauses are left, is called *unit propagation*. If unit propagation yields the empty clause $\bot$, we say that unit propagation applied to F derived a *conflict*.

```
      SDCL ( formula F )
 1  α := ∅
 2  forever do
 3      α := UnitPropagate (F, α)
 4      if  α falsifies a clause in F  then
 5          C := AnalyzeConflict()
 6          F := F ∧ C
 7          if  C is the empty clause ⊥  then return UNSAT
 8          α := BackJump(C, α)
 9      else if  the pruning predicate Pα(F) is satisfiable  then
10          C := AnalyzeWitness()
11          F := F ∧ C
12          α := BackJump(C, α)
13      else
14          if  all variables are assigned  then return SAT
15          l  := Decide ()
16          α := α ∪ {l}
```

Fig. 1. SDCL algorithm [20]. The lines 9 to 12 extend CDCL [26].

Formula relations. Two formulas are *logically equivalent* if they are satisfied by the same total assignments. Two formulas are *equisatisfiable* if they are either both satisfiable or both unsatisfiable. Furthermore, by $F \vDash_1 G$ we denote that for every clause $(l_1 \vee \cdots \vee l_n) \in G$, unit propagation applied to $F \wedge (\bar{l}_1) \wedge \cdots \wedge (\bar{l}_n)$ derives a conflict. If $F \vDash_1 G$, we say that F implies G via unit propagation. For example, $(\bar{a} \vee c) \wedge (\bar{b} \vee \bar{c})$ implies $(\bar{a} \vee \bar{b})$ via unit propagation since unit propagation derives a conflict on $(\bar{a} \vee c) \wedge (\bar{b} \vee \bar{c}) \wedge (a) \wedge (b)$.

Conflict-Driven Clause Learning (CDCL). To determine whether a formula is satisfiable a CDCL solver iteratively performs the following operations (obtained from the pseudo code in Fig. 1 by removing the lines 9 to 12): First, the solver performs unit propagation until either it derives a conflict or the formula contains no more unit clauses. If it derives a conflict, it analyzes the conflict to learn a clause that prevents it from repeating similar (bad) decisions in the future ("clause learning"). If this learned clause is the (unsatisfiable) empty clause ⊥, the solver can conclude that the formula is unsatisfiable. In case it is not the empty clause, the solver revokes some of its variable assignments ("backjumping") and then repeats the whole procedure again by performing unit propagation. If, however, the solver does not derive a conflict, there are two options: Either all variables are assigned, in which case the solver can conclude that the formula is satisfiable, or there are still unassigned variables, in which case the solver first assigns a truth value to an unassigned variable (the actual variable and the truth value are chosen based on a so-called *decision heuristic*) and then continues by again performing unit propagation. For more details see the chapter on CDCL [25] in the Handbook of Satisfiability [7].
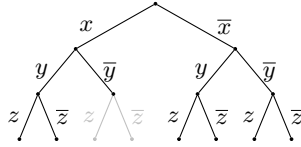
Fig. 2. By learning the clause $\overline{x} \vee y$, a solver prunes all branches where x is true and y is false from the search space. SDCL can prune satisfying branches too (unlike CDCL).

Satisfaction-Driven Clause Learning (SDCL). The SDCL algorithm [20], shown in Fig. 1, is a generalization of CDCL that is obtained by adding lines 9 to 12 to the CDCL algorithm. In CDCL, if unit propagation does not derive a conflict, the solver picks a variable and assigns a truth value to it. In contrast, an SDCL solver does not necessarily assign a new variable in this situation. Instead, it first checks if the current assignment can be pruned from the search space without affecting satisfiability. If so, the solver prunes the assignment by learning a new clause (Fig. 2 illustrates how clause learning can prune the search space). This clause is returned by the AnalyzeWitness() function and usually consists of the decision literals of the current assignment (although other ways of computing the clause are possible, c.f. [20]). If the assignment cannot be pruned, the solver proceeds by assigning a new variable—just as in CDCL. To check if the current assignment can be pruned, the solver produces a propositional formula that should be easier to solve than the original formula and that can only be satisfiable if the assignment can be pruned. Thus, an SDCL solver solves several easier formulas in order to solve a hard formula. In this paper, we call these easier formulas *pruning predicates*. We first formalize the pruning predicate used in the original SDCL paper before we introduce more powerful pruning predicates.

Pruning Predicates and Redundant Clauses

As already explained informally, a pruning predicate is a propositional formula whose satisfiability guarantees that an assignment can be pruned from the search space. The actual pruning is then performed by adding the clause that *blocks* the assignment (or a subclause of this clause, as explained in detail later):

Definition 1. *Given an assignment $\alpha = a_1 \ldots a_k$, the clause $(\overline{a}_1 \vee \cdots \vee \overline{a}_k)$ is the clause that* blocks *α.*

The clause that blocks α is thus the unique maximal clause falsified by α. Based on this notion, we define pruning predicates as follows:

Definition 2. *Let F be a formula and C the clause that blocks a given (partial) assignment α. A pruning predicate for F and α is a formula $P_\alpha(F)$ such that the following holds: if $P_\alpha(F)$ is satisfiable, then F and $F \wedge C$ are equisatisfiable.*

Thus, if a pruning predicate for a formula F and an assignment α is satisfiable, we can add the clause that blocks α to F without affecting satisfiability. We thus
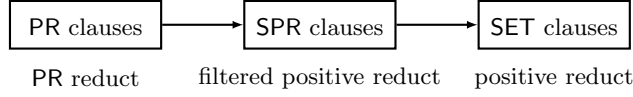
5

Fig. 3. Relationship between types of redundant clauses and the corresponding pruning predicates. An arrow from X to Y means that X is a superset of Y.

say that this clause is *redundant* with respect to F. In the paper that introduces SDCL [20], the so-called *positive reduct* (see Definition 3 below) is used as a pruning predicate. The positive reduct is obtained from satisfied clauses of the original formula by removing unassigned literals.

In the following, given a clause C and an assignment α, we write $\mathsf{touched}_\alpha(C)$ to denote the subclause of C that contains exactly the literals assigned by α. Analogously, we denote by $\mathsf{untouched}_\alpha(C)$ the subclause of C that contains the literals *not* assigned by α [20].

Definition 3. *Let F be a formula and α an assignment. Then, the* positive reduct $\mathsf{p}_\alpha(F)$ *of F and α is the formula $G \wedge C$ where C is the clause that blocks α and $G = \{\mathsf{touched}_\alpha(D) \mid D \in F \text{ and } D{\restriction}\alpha = \top\}$.*

Example 1. Let $F = (x \vee \overline{y} \vee z) \wedge (w \vee \overline{y}) \wedge (\overline{w} \vee \overline{z})$ and $\alpha = x\, y\, \overline{z}$. Then, the positive reduct $\mathsf{p}_\alpha(F)$ of F w.r.t. α is the formula $(x \vee \overline{y} \vee z) \wedge (\overline{z}) \wedge (\overline{x} \vee \overline{y} \vee z)$.

The positive reduct is satisfiable if and only if the clause blocked by α is a *set-blocked clause* [23], short **SET** clause, with respect to F. Since the addition of set-blocked clauses to a formula preserves satisfiability, it follows that the positive reduct is a pruning predicate. Moreover, since the problem of deciding whether a given clause is a set-blocked clause is **NP**-complete, it is natural to use a SAT solver for finding set-blocked clauses.

Although set-blocked clauses can be found efficiently with the positive reduct, there are more general kinds of clauses whose addition can prune the search space more aggressively, namely *propagation-redundant clauses* (**PR** clauses) and their subclass of *set-propagation-redundant clauses* (**SPR** clauses) [19].

In the following, we thus introduce two different kinds of pruning predicates. Given a formula F and an assignment α, the first pruning predicate, called the *filtered positive reduct*, is satisfiable if and only if the clause that blocks α is an **SPR** clause in F. The second pruning predicate, called **PR** *reduct*, is satisfiable if and only if the clause that blocks α is a **PR** clause; it allows us to prune more assignments than the filtered positive reduct but it is also harder to solve. The relationship between the redundant clause types and pruning predicates is shown in Fig. 3. According to [19], the definition of **PR** clauses is as follows:

Definition 4. *Let F be a formula, C a clause, and α the assignment blocked by C. Then, C is* propagation redundant (**PR**) *with respect to F if there exists an assignment ω such that $F{\restriction}\alpha \vdash_1 F{\restriction}\omega$ and ω satisfies C.*

6

The clause C can be seen as a constraint that prunes all assignments that extend α from the search space. Since $F\restriction\alpha$ implies $F\restriction\omega$ via unit propagation, every assignment that satisfies $F\restriction\alpha$ also satisfies $F\restriction\omega$, and so we say that F is *at least as satisfiable* under ω as it is under α. Moreover, since ω satisfies C, it must disagree with α. Consider the following example [19]:

Example 2. Let $F = (x \vee y) \wedge (\overline{x} \vee y) \wedge (\overline{x} \vee z)$ be a formula, $C = (x)$ a clause, and $\omega = x\,z$ an assignment. Then, $\alpha = \overline{x}$ is the assignment blocked by C. Now, consider $F\restriction\alpha = (y)$ and $F\restriction\omega = (y)$. Since unit propagation clearly derives a conflict on $F\restriction\alpha \wedge (\overline{y}) = (y) \wedge (\overline{y})$, we have $F\restriction\alpha \vdash_1 F\restriction\omega$ and thus C is propagation redundant with respect to F.

The key property of propagation-redundant clauses is that their addition to a formula preserves satisfiability [19]. A strict subclass of propagation-redundant clauses are set-propagation-redundant clauses, which have the additional requirement that ω must assign the same variables as α. For the following definition, recall (from the preliminaries) that α_L denotes the assignment obtained from α by assigning 1 to the literals in L [19]:

Definition 5. *Let F be a formula, C a clause, and α the assignment blocked by C. Then, C is* set-propagation redundant (SPR) *with respect to F if it contains a non-empty set L of literals such that $F\restriction\alpha \vdash_1 F\restriction\alpha_L$.*

If $F\restriction\alpha \vdash_1 F\restriction\alpha_L$, we say C is SPR *by L with respect to F.*

Example 3. Let $F = (x \vee y) \wedge (x \vee \overline{y} \vee z) \wedge (\overline{x} \vee z) \wedge (\overline{x} \vee u) \wedge (\overline{u} \vee x)$, $C = x \vee u$, and $L = \{x, u\}$. Then, $\alpha = \overline{x}\,\overline{u}$ is the assignment blocked by C, and $\alpha_L = x\,u$. Now, consider $F\restriction\alpha = (y) \wedge (\overline{y} \vee z)$ and $F\restriction\alpha_L = (z)$. Clearly, $F\restriction\alpha \vdash_1 F\restriction\alpha_L$ and so C is set-propagation redundant by L with respect to F.

Most known types of redundant clauses are SPR clauses [19]. This includes *blocked clauses* [24], *set-blocked clauses* [23], *resolution asymmetric tautologies* (RATs) [22], and many more. By introducing pruning predicates that allow us to add SPR clauses and even PR clauses to a formula, we thus allow for more effective pruning than with the positive reduct originally used in SDCL. We start by presenting our new *filtered* positive reduct.

The Filtered Positive Reduct

The original positive reduct of a formula F and an assignment α is obtained by first taking all clauses of F that are satisfied by α and then removing from these clauses the literals that are not touched (assigned) by α. The resulting clauses are then conjoined with the clause C that blocks α. We obtain the *filtered* positive reduct by not taking *all* satisfied clauses of F but only those for which the untouched part is not implied by $F\restriction\alpha$ via unit propagation:

Definition 6. *Let F be a formula and α an assignment. Then, the* filtered *positive reduct $\mathsf{f}_\alpha(F)$ of F and α is the formula $G \wedge C$ where C is the clause that blocks α and $G = \{\mathsf{touched}_\alpha(D) \mid D \in F \text{ and } F\restriction\alpha \not\vdash_1 \mathsf{untouched}_\alpha(D)\}$.*

Clearly the filtered positive reduct is a subset of the positive reduct because $F{\restriction}\alpha \not\vdash_1 \mathsf{untouched}_\alpha(D)$ implies $D{\restriction}\alpha = \top$. To see this, suppose $D{\restriction}\alpha \neq \top$. Then, $D{\restriction}\alpha$ is contained in $F{\restriction}\alpha$ and since $\mathsf{untouched}_\alpha(D) = D{\restriction}\alpha$, it follows that $F{\restriction}\alpha \vdash_1 \mathsf{untouched}_\alpha(D)$. Therefore, the filtered positive reduct is obtained from the positive reduct by removing ("filtering") every clause $D' = \mathsf{touched}_\alpha(D)$ such that $F{\restriction}\alpha \vdash_1 \mathsf{untouched}_\alpha(D)$.

The following example illustrates how the filtered positive reduct allows us to prune assignments that cannot be pruned when using only the positive reduct:

Example 4. Let $F = (x \vee y) \wedge (\overline{x} \vee y)$ and consider the assignment $\alpha = x$. The positive reduct $\mathsf{p}_\alpha(F) = (x) \wedge (\overline{x})$ is unsatisfiable and so it does not allow us to prune α. In contrast, the filtered positive reduct $\mathsf{f}_\alpha(F) = (\overline{x})$, obtained by filtering out the clause (x), is satisfied by the assignment $\overline{x}$. The clause (x) is not contained in the filtered reduct because $\mathsf{untouched}_\alpha(x \vee y) = (y)$ and $F{\restriction}\alpha = (y)$, which implies $F{\restriction}\alpha \vdash_1 \mathsf{untouched}_\alpha(x \vee y)$. Note that the clause $(\overline{x})$ is contained both in the positive reduct and in the filtered positive reduct since it blocks α.

The filtered positive reduct has a useful property: If a non-empty assignment α falsifies a formula F, then the filtered positive reduct $\mathsf{f}_\alpha(F)$ is satisfiable. To see this, observe that $\bot \in F{\restriction}\alpha$ and so $F{\restriction}\alpha \vdash_1 \mathsf{untouched}_\alpha(D)$ for every clause $D \in F$ because unit propagation derives a conflict on $F{\restriction}\alpha$ alone (note that this also holds if $\mathsf{untouched}_\alpha(D)$ is the empty clause $\bot$). Therefore, $\mathsf{f}_\alpha(F)$ contains only the clause that blocks α, which is clearly satisfiable. The ordinary positive reduct does not have this property.

Note that the filtered positive reduct contains only variables of $var(\alpha)$. Since it also contains the clause that blocks α, any satisfying assignment of the filtered positive reduct must disagree with α on at least one literal. Hence, every satisfying assignment of the filtered positive reduct is of the form α_L where L is a set of literals that are contained in the clause that blocks α. With the filtered positive reduct, we can identify exactly the clauses that are set-propagation redundant with respect to a formula:

Theorem 1. *Let F be a formula, α an assignment, and C the clause that blocks α. Then, C is SPR by an $L \subseteq C$ with respect to F if and only if the assignment α_L satisfies the filtered positive reduct $\mathsf{f}_\alpha(F)$.*

Proof. For the "only if" direction, suppose C is SPR by an $L \subseteq C$ in F, meaning that $F{\restriction}\alpha \vdash_1 F{\restriction}\alpha_L$. We show that α_L satisfies all clauses of $\mathsf{f}_\alpha(F)$. Let therefore $D' \in \mathsf{f}_\alpha(F)$. By definition, D' is either the clause that blocks α or it is of the form $\mathsf{touched}_\alpha(D)$ for some clause $D \in F$ such that $F{\restriction}\alpha \not\vdash_1 \mathsf{untouched}_\alpha(D)$. In the former case, D' is clearly satisfied by α_L since α_L must disagree with α. In the latter case, since $F{\restriction}\alpha \vdash_1 F{\restriction}\alpha_L$, it follows that either $F{\restriction}\alpha \vdash_1 D{\restriction}\alpha_L$ or α_L satisfies D. Now, if $D{\restriction}\alpha_L \neq \top$, it cannot be the case that $F{\restriction}\alpha \vdash_1 D{\restriction}\alpha_L$ since $var(\alpha_L) = var(\alpha)$ and thus $D{\restriction}\alpha_L = \mathsf{untouched}_\alpha(D)$, which would imply $F{\restriction}\alpha \vdash_1 \mathsf{untouched}_\alpha(D)$. Therefore, α_L must satisfy D. But then α_L must satisfy $D' = \mathsf{touched}_\alpha(D)$, again since $var(\alpha_L) = var(\alpha)$.

For the "if" direction, assume that α_L satisfies the filtered positive reduct $\mathsf{f}_\alpha(F)$. We show that $F{\upharpoonright}\alpha \vdash_1 F{\upharpoonright}\alpha_L$. Let $D{\upharpoonright}\alpha_L \in F{\upharpoonright}\alpha_L$. Since $D{\upharpoonright}\alpha_L$ is contained in $F{\upharpoonright}\alpha_L$, we know that α_L does not satisfy D and so it does not satisfy $\mathsf{touched}_\alpha(D)$. Hence, $\mathsf{touched}_\alpha(D)$ cannot be contained in $\mathsf{f}_\alpha(F)$, implying that $F{\upharpoonright}\alpha \vdash_1 \mathsf{untouched}_\alpha(D)$. But, $D{\upharpoonright}\alpha_L = \mathsf{untouched}_\alpha(D)$ since $var(\alpha_L) = var(\alpha)$ and thus it follows that $F{\upharpoonright}\alpha \vdash_1 D{\upharpoonright}\alpha_L$. □

When the (ordinary) positive reduct is used for SDCL solving, the following property holds [20]: Assume the solver has a current assignment $\alpha = \alpha_d \cup \alpha_u$ where α_d consists of all the assignments that were made by the decision heuristic and α_u consists of all assignments that were derived via unit propagation. If the solver then finds that the positive reduct of its formula and the assignment α is satisfiable, it can learn the clause that blocks α_d instead of the longer clause that blocks α, thus pruning the search space more effectively. This is allowed because the clause that blocks α_d is guaranteed to be propagation redundant.

The same holds for the filtered positive reduct and the argument is analogous to the earlier one [20]: Assume the filtered positive reduct of F and $\alpha = \alpha_d \cup \alpha_u$ is satisfiable. Then, the clause that blocks α is set-propagation redundant with respect to F and thus there exists an assignment α_L such that $F{\upharpoonright}\alpha \vdash_1 F{\upharpoonright}\alpha_L$. But then, since unit propagation derives all the assignments of α_u from $F{\upharpoonright}\alpha_d$, it must also hold that $F{\upharpoonright}\alpha_d \vdash_1 F{\upharpoonright}\alpha_L$, and so the clause that blocks α_d is propagation redundant with respect to F and (witness) assignment $\omega = \alpha_L$.

Finally, observe that the filtered positive reducts $\mathsf{f}_{\alpha_d}(F)$ and $\mathsf{f}_\alpha(F)$ are not always equisatisfiable. To see this, consider the formula $F = (\overline{x} \vee y) \wedge (x \vee \overline{y})$ and the assignments $\alpha = x\,y$ and $\alpha_d = x$. Clearly, the unit clause y is derived from $F{\upharpoonright}\alpha_d$. Now, observe that $\mathsf{f}_\alpha(F)$ is satisfiable while $\mathsf{f}_{\alpha_d}(F)$ is unsatisfiable. It thus makes sense to first compute the filtered positive reduct with respect to α and then—in case it is satisfiable—remove the propagated literals to obtain a shorter clause.

The PR Reduct

We showed in the previous section that the filtered positive reduct characterizes precisely the set-propagation-redundant clauses. Since set-propagation-redundant clauses are a subset of propagation-redundant clauses [19], it is natural to search for an encoding that characterizes the propagation-redundant clauses, which could possibly lead to an even more aggressive pruning of the search space. As we will see in the following, such an encoding must necessarily be large because it has to reason over all possible clauses of a formula. We thus believe that it is hardly useful for practical SDCL solving.

The positive reduct and the filtered positive reduct yield small formulas that can be easily solved in practice. The downside, however, is that nothing can be learned from their unsatisfiability. This is different for a pruning predicate that encodes propagation redundancy:

Theorem 2. *If a clause $l_1 \vee \cdots \vee l_k$ is not propagation redundant with respect to a formula F, then F implies $\bar{l}_1 \wedge \cdots \wedge \bar{l}_k$.*

Proof. Assume $l_1 \vee \cdots \vee l_k$ is not propagation redundant with respect to F, or equivalently that all assignments ω with $F{\upharpoonright}\bar{l}_1 \ldots \bar{l}_k \vdash_1 F{\upharpoonright}\omega$ agree with $\bar{l}_1 \ldots \bar{l}_k$. Then, no assignment that disagrees with $\bar{l}_1 \ldots \bar{l}_k$ can satisfy F. As a consequence, F implies $\bar{l}_1 \wedge \cdots \wedge \bar{l}_k$. $\qquad\square$

By solving a pruning predicate for propagation-redundant clauses, we thus not only detect if the current assignment can be pruned (in case the predicate is satisfiable), but also if the formula can only possibly be satisfied by extensions of the current assignment (in case the predicate is unsatisfiable). This is in contrast to the positive reduct and the filtered positive reduct, which often only need to consider a small subpart of the original formula. We thus believe that such an encoding is not useful in practice. In the following, we present a possible encoding which—due to the above reasons—we did not evaluate in practice. Nevertheless, performing such an evaluation is still part of our future work.

In the definition of propagation-redundant clauses, the assignment ω does not necessarily assign the same variables as α. To deal with this, we use the idea of the so-called *dual-rail encoding* [10, 30, 8]. In the dual-rail encoding, a given variable x is replaced by two new variables x^p and x^n. The intuitive idea is that x^p is true whenever the original variable x is supposed to be true and x^n is true whenever x is supposed to be false. If both x^p and x^n are false, then x is supposed to be unassigned. Finally, x^p and x^n cannot be true at the same time. Thus, the *dual-rail encodings* of a clause are defined as follows: Let $C = P \vee N$ be a clause with $P = x_1 \vee \cdots \vee x_k$ containing only positive literals and $N = \overline{x}_{k+1} \vee \cdots \vee \overline{x}_m$ containing only negative literals. Further, let $x_1^p, x_1^n, \ldots, x_m^p, x_m^n$ be new variables. Then, the *positive dual-rail encoding* C^p of C is the clause

$$x_1^p \vee \cdots \vee x_k^p \vee x_{k+1}^n \vee \cdots \vee x_m^n,$$

and the *negative dual-rail encoding* C^n of C is the clause

$$x_1^n \vee \cdots \vee x_k^n \vee x_{k+1}^p \vee \cdots \vee x_m^p.$$

We can now define the PR reduct as follows:

Definition 7. *Let F be a formula and α an assignment. Then, the PR reduct $\mathsf{pr}_\alpha(F)$ of F and α is the formula $G \wedge C$ where C is the clause that blocks α and G is the union of the following sets of clauses where all the s_i are new variables:*

$\{\overline{x^p} \vee \overline{x^n} \mid x \in var(F) \setminus var(\alpha)\},$

$\{\overline{s}_i \vee \mathsf{touched}_\alpha(D_i) \vee \mathsf{untouched}_\alpha(D_i)^p \mid D_i \in F\},$

$\{\overline{L^n} \vee s_i \mid D_i \in F \ \text{ and } \ L \subseteq \mathsf{untouched}_\alpha(D_i)$
$\qquad \text{such that } F{\upharpoonright}\alpha \nvdash_1 \mathsf{untouched}_\alpha(D_i) \setminus L\}.$

In the last set, if L is empty, we obtain a unit clause with the literal s_i.

We thus keep all the variables assigned by α but introduce the dual-rail variants for variables of F not assigned by α. The clauses of the form $\overline{x^p} \vee \overline{x^n}$ ensure that for a variable x, the two variables x^p and x^n cannot be true at the same time.

The main idea is that satisfying assignments of the PR reduct correspond to assignments of the formula F: from a satisfying assignment τ of the PR reduct we obtain an assignment ω over the variables of the original formula F as follows:

$$\omega(x) = \begin{cases} \tau(x) & \text{if } x \in var(\tau) \cap var(F), \\ 1 & \text{if } \tau(x^p) = 1, \\ 0 & \text{if } \tau(x^n) = 1. \end{cases}$$

Analogously, we obtain from ω a satisfying assignment τ of the filtered positive reduct $\mathsf{pr}_\alpha(F)$ as follows:

$$\tau(x) = \begin{cases} \omega(x) & \text{if } x \in var(\alpha); \\ 1 & \text{if } x = x^p \text{ and } \omega(x) = 1, \text{ or} \\ & \text{if } x = x^n \text{ and } \omega(x) = 0, \text{ or} \\ & \text{if } x = s_i \text{ and } \omega \text{ satisfies } D_i; \\ 0 & \text{otherwise.} \end{cases}$$

To prove that the clause that blocks an assignment α is propagation redundant w.r.t. a formula F if the PR reduct of F and α is satisfiable, we use the following:

Lemma 1. *Let F be a formula and let α and ω be two assignments such that $F{\restriction}\alpha \vdash_1 F{\restriction}\omega$. Then, $F{\restriction}\alpha \vdash_1 F{\restriction}\omega x$ for every literal x such that $var(x) \in var(\alpha)$.*

Proof. Let $D{\restriction}\omega x \in F{\restriction}\omega x$. We show that $F{\restriction}\alpha \vdash_1 D{\restriction}\omega x$. Clearly, $x \notin D$ for otherwise $D{\restriction}\omega x = \top$, which would imply $D{\restriction}\omega x \notin F{\restriction}\omega x$. Therefore, the only possible difference between $D{\restriction}\omega$ and $D{\restriction}\omega x$ is that $\overline{x}$ is contained in $D{\restriction}\omega$ but not in $D{\restriction}\omega x$. Now, since $var(x) \in var(\alpha)$, we know that $var(x) \notin F{\restriction}\alpha$. But then, $F{\restriction}\alpha \vdash_1 D{\restriction}\omega x$ if and only if $F{\restriction}\alpha \vdash_1 D{\restriction}\omega$ and thus $F{\restriction}\alpha \vdash_1 F{\restriction}\omega x$. $\square$

We can now show that the PR reduct precisely characterizes the propagation-redundant clauses:

Theorem 3. *Let F be a formula, α an assignment, and C the clause that blocks α. Then, C is propagation redundant with respect to F if and only if the PR reduct $\mathsf{pr}_\alpha(F)$ of F and α is satisfiable.*

Proof. For the "only if" direction, assume that C is propagation redundant with respect to F, meaning that there exists an assignment ω such that ω satisfies C and $F{\restriction}\alpha \vdash_1 F{\restriction}\omega$. By Lemma 1, we can without loss of generality assume that $var(\alpha) \subseteq var(\omega)$. Now consider the assignment τ that corresponds to ω as explained before Lemma 1. We show that τ satisfies $\mathsf{pr}_\alpha(F)$. Since the clause C that blocks α is in $\mathsf{pr}_\alpha(F)$, it must be satisfied by ω. Since ω satisfies C, τ satisfies C. Also, by construction, τ never satisfies both x^p and x^n for a variable

x and so it satisfies the clauses $\overline{x^p} \vee \overline{x^n}$. If, for a clause $\overline{s}_i \vee \mathsf{touched}_\alpha(D_i) \vee \mathsf{untouched}_\alpha(D_i)^p$, τ satisfies s_i, then we know that ω satisfies D_i and thus τ must satisfy $\mathsf{touched}_\alpha(D_i) \vee \mathsf{untouched}_\alpha(D_i)^p$.

It remains to show that τ satisfies the clause $\overline{L^n} \vee s_i$ for every $D_i \in F$ and every set $L \subseteq \mathsf{untouched}_\alpha(D_i)$ such that $F{\upharpoonright}_\alpha \not\vDash \mathsf{untouched}_\alpha(D_i) \setminus L$. Assume to the contrary that, for such a clause, $\tau(s_i) = 0$ and τ falsifies all literals in $\overline{L^n}$. Then, ω does not satisfy D_i and it falsifies all literals in L. But, from $var(\alpha) \subseteq var(\omega)$ we know that $D_i{\upharpoonright}\omega \subseteq \mathsf{untouched}_\alpha(D_i)$ and thus it follows that $D_i{\upharpoonright}\omega \subseteq \mathsf{untouched}_\alpha(D_i) \setminus L$. Hence, since $F{\upharpoonright}_\alpha \not\vDash \mathsf{untouched}_\alpha(D_i) \setminus L$, we conclude that $F{\upharpoonright}_\alpha \not\vDash D_i{\upharpoonright}\omega$, a contradiction.

For the "if" direction, assume that there exists a satisfying assignment τ of $\mathsf{pr}_\alpha(F)$ and consider the assignment ω that corresponds to τ as explained before Lemma 1. Since $C \in \mathsf{pr}_\alpha(F)$, ω must satisfy C. It remains to show that $F{\upharpoonright}_\alpha \vDash F{\upharpoonright}\omega$. Let $D_i{\upharpoonright}\omega \in F{\upharpoonright}\omega$. Then, ω does not satisfy D_i and so $\mathsf{touched}_\alpha(D_i) \vee \mathsf{untouched}_\alpha(D_i)^p$ is falsified by τ, implying that τ must falsify s_i. As $var(\alpha) \subseteq var(\omega)$, we know that $D_i{\upharpoonright}\omega \subseteq \mathsf{untouched}_\alpha(D_i)$, meaning that $D_i{\upharpoonright}\omega$ is of the form $\mathsf{untouched}_\alpha(D_i) \setminus L$ for some set $L \subseteq \mathsf{untouched}_\alpha(D_i)$ such that ω falsifies L. But then the clause $\overline{L^n} \vee s_i$ cannot be contained in $\mathsf{pr}_\alpha(F)$ since it would be falsified by τ. We thus conclude that $F{\upharpoonright}\omega \vDash \mathsf{untouched}_\alpha(D_i) \setminus L$ and so $F{\upharpoonright}\omega \vDash D_i{\upharpoonright}\omega$. $\square$

Note that the last set of clauses of the PR reduct, in principle has exponentially many clauses w.r.t. the length of the largest original clause. We leave it to future work to answer the question whether non-exponential encodings exist. But even if a polynomial encoding can be found, we doubt its usefulness in practice.

Implementation

We implemented a clean-slate SDCL solver, called SaDiCaL, that can learn PR clauses using either the positive reduct or the filtered positive reduct. It consists of around 3K lines of C and is based on an efficient CDCL engine using state-of-the-art algorithms, data structures, and heuristics, including a variable-move-to-front decision heuristic [5], a sophisticated restart policy [4], and aggressive clause-data-based reduction [2]. Our implementation provides a simple but efficient framework to evaluate new SDCL-inspired ideas and heuristics.

The implementation closely follows the pseudo-code shown in Fig. 1 and computes the pruning predicate before every decision. This is costly in general, but allows the solver to detect PR clauses as early as possible. Our goal is to determine whether short PR proofs can be found automatically. The solver produces PR proofs and we verified all the presented results using proof checkers. The source code of SaDiCaL is available at http://fmv.jku.at/sadical.

Two aspects of SDCL are crucial: the pruning predicate and the decision heuristics. For the pruning predicate we ran experiments with both the positive reduct and the filtered positive reduct. The initially proposed decision heuristics for SDCL [20] are as follows: Pick the variable that occurs most frequently in short clauses. Also, apart from the root-node branch, assign only literals that occur in clauses that are touched but not satisfied by the current assignment.

We added another restriction: whenever a (filtered) positive reduct is satisfiable, make all literals in the witness (i.e., the satisfying assignment of the pruning predicate) that disagree with the current assignment more important than any other literal in the formula. This restriction is removed when the solver backtracks to the root node (i.e., when a unit clause is learned) and added again when a new PR clause is found. The motivation of this restriction is as follows: we observed that literals in the witness that disagree with the current assignment typically occur in short PR clauses; making them more important than other literals increases the likelihood of learning short PR clauses.

Evaluation

In the following, we demonstrate that the filtered positive reduct allows our SDCL solver to prove unsatisfiability of formulas well-known for having only exponential-size resolution proofs. We start with Tseitin formulas [32, 11]. In short, a Tseitin formula represents the following graph problem: Given a graph with 0/1-labels for each vertex such that an odd number of vertices has label 1, does there exist a subset of the edges such that (after removing edges not in the subset) every vertex with label 0 has an even degree and every vertex with label 1 has an odd degree? The answer is *no* as the sum of all degrees is always even. The formula is therefore unsatisfiable by construction. Tseitin formulas defined over expander graphs require resolution proofs of exponential size [33] and also appear hard for SDCL when using the ordinary positive reduct as pruning predicate. We compare three settings, all with proof logging:

(1) plain CDCL,
(2) SDCL with the positive reduct $p_\alpha(F)$, and
(3) SDCL with the filtered positive reduct $f_\alpha(F)$.

Additionally, we include the winner of the 2018 SAT Competition: the CDCL-based solver `MapleLCMDistChronoBT` (short `MLBT`) [29]. The results are shown in Table 1. The last column shows the proof-validation times by the formally verified checker in ACL2. To verify the proofs for all our experiments, we did the following: We started with the PR proofs produced by our SDCL solver using the filtered positive reduct. We then translated them into DRAT proofs using the `pr2drat` tool [17]. Finally, we used the `drat-trim` checker to optimize the proofs (i.e., to remove redundant proof parts) and to convert them into the LRAT format, which is the format supported by the formally verified proof checker.

Table 1 shows the performance on small (Urquhart-s3*), medium (Urquhart-s4*), and large (Urquhart-s5*) Tseitin formulas running on a Xeon E5-2690 CPU 2.6 GHz with 64 GB memory.[5] Only our solver with the filtered positive reduct is able to efficiently prove unsatisfiability of all these instances. Notice that with the ordinary positive reduct it is impossible to solve any of the formulas. There may actually be a theoretical barrier here. The LSDCL solver also uses the

---

[5] Log files, benchmarks and source code are available at http://fmv.jku.at/sadical.

Table 1. Runtime Comparison (in Seconds) on the Tseitin Benchmarks [33, 11].

formula	MLBT [29]	LSDCL [20]	plain	$\mathsf{p}_\alpha(F)$	$\mathsf{f}_\alpha(F)$	ACL2
Urquhart-s3-b1	2.95	5.86	16.31	> 3600	**0.02**	0.09
Urquhart-s3-b2	1.36	2.4	2.82	> 3600	**0.03**	0.13
Urquhart-s3-b3	2.28	19.94	2.08	> 3600	**0.03**	0.16
Urquhart-s3-b4	10.74	32.42	7.65	> 3600	**0.03**	0.17
Urquhart-s4-b1	86.11	583.96	> 3600	> 3600	**0.32**	2.37
Urquhart-s4-b2	154.35	1824.95	183.77	> 3600	**0.11**	0.78
Urquhart-s4-b3	258.46	> 3600	129.27	> 3600	**0.16**	1.12
Urquhart-s4-b4	> 3600	> 3600	> 3600	> 3600	**0.14**	1.17
Urquhart-s5-b1	> 3600	> 3600	> 3600	> 3600	**1.27**	9.86
Urquhart-s5-b2	> 3600	> 3600	> 3600	> 3600	**0.58**	4.38
Urquhart-s5-b3	> 3600	> 3600	> 3600	> 3600	**1.67**	17.99
Urquhart-s5-b4	> 3600	> 3600	> 3600	> 3600	**2.91**	24.24

Table 2. Runtime Comparison (in Seconds) on the Pigeon-Hole Formulas.

formula	MLBT [29]	LSDCL [20]	plain	$\mathsf{p}_\alpha(F)$	$\mathsf{f}_\alpha(F)$	ACL2
hole20	> 3600	1.13	> 3600	**0.22**	0.55	6.78
hole30	> 3600	8.81	> 3600	**1.71**	4.30	87.58
hole40	> 3600	43.10	> 3600	**7.94**	20.38	611.24
hole50	> 3600	149.67	> 3600	**25.60**	68.46	2792.39

positive reduct, but only for assignments with at most two decision literals. As a consequence, the overhead of the positive reduct is small. In the future we plan to develop meaningful limits for SaDiCaL as well.

We also ran experiments with the pigeon-hole formulas. Although these formulas are hard for resolution, they can be solved efficiently with SDCL using the positive reduct [20]. Table 2 shows a runtime comparison, again including PR proof logging, for pigeon-hole formulas of various sizes. Notice that the computational costs of the solver with the filtered positive reduct are about 3 to 4 times as large compared to the solver with the positive reduct. This is caused by the overhead of computing the filtering. The sizes of the PR proofs produced by both versions are similar. Our solver with the positive reduct is about four times as fast compared to the SDCL version (only positive reduct) of Lingeling [20], in short LSDCL. As the heuristics and proof sizes of our solver and LSDCL are similar, the better performance is due to our dedicated SDCL implementation.

Finally, we performed experiments with the recently released 2018 SAT Competition benchmarks. We expected slow performance on most benchmarks due to the high overhead of solving pruning predicates before making decisions. However, our solver outperformed the participating solvers on mutilated chessboard problems [27] which were contributed by Alexey Porkhunov (see Table 3).

For example, our solver can prove unsatisfiability of the 18×18 mutilated chessboard in 43.88 seconds. The filtered positive reduct was crucial to obtain this result. The other solvers, apart from CaDiCaL solving it in 828 seconds,

Table 3. Runtime Comparison (in Seconds) on the Mutilated Chessboard Formulas.

formula	MLBT [29]	LSDCL [20]	plain	$\mathsf{p}_\alpha(F)$	$\mathsf{f}_\alpha(F)$	ACL2
mchess_15	51.53	1473.11	2480.67	> 3600	**13.14**	29.12
mchess_16	380.45	> 3600	2115.75	> 3600	**15.52**	36.86
mchess_17	2418.35	> 3600	> 3600	> 3600	**25.54**	57.83
mchess_18	> 3600	> 3600	> 3600	> 3600	**43.88**	100.71

timed out after 5000 seconds during the competition (on competition hardware). Resolution proofs of mutilated chessboard problems are exponential in size [1], which explains the poor performance of CDCL solvers. On these problems, like on the Tseitin formulas, our solver performed much better with the filtered positive reduct than with the positive reduct. The results are robust with respect to partially and completely scrambling formulas as suggested by [6], with the exception of the pigeon hole formulas, which needs to be investigated.

Conclusion

We introduced two new SAT encodings for pruning the search space in satisfaction-driven clause learning (SDCL). The first encoding, called the filtered positive reduct, is easily solvable and prunes the search space more aggressively than the positive reduct (which was used when SDCL was initially introduced). The second encoding, called the PR reduct, might not be useful in practice though it precisely characterizes propagation redundancy.

Based on the filtered positive reduct, we implemented an SDCL solver and our experiments show that the solver can efficiently prove the unsatisfiability of the Tseitin formulas, the pigeon-hole formulas, and the mutilated chessboard problems. For all these formulas, CDCL solvers require exponential time due to theoretical restrictions. Moreover, to the best of our knowledge, our solver is the first to generate machine-checkable proofs of unsatisfiability of these formulas. We certified our results using a formally verified proof checker.

Although our SDCL solver can already produce proofs of formulas that are too hard for CDCL solvers, it is still outperformed by CDCL solvers on many simpler formulas. This seems to suggest that also in SAT solving, there is no free lunch. Nevertheless, we believe that the performance of SDCL on simple formulas can be improved by tuning the solver more carefully, e.g., by only learning propagation-redundant clauses when this is really beneficial, or by coming up with a dedicated decision heuristic. To deal with these problems, we are currently investigating an approach based on reinforcement learning.

Considering our results, we believe that SDCL is a promising SAT-solving paradigm for formulas that are too hard for ordinary CDCL solvers. Finally, proofs of challenging problems can be enormous in size, such as the 2 petabytes proof of Schur Number Five [16]; SDCL improvements have the potential to produce proofs that are substantially smaller and faster to verify.

References

1. Alekhnovich, M.: Mutilated chessboard problem is exponentially hard for resolution. Theoretical Computer Science 310(1-3), 513–525 (2004)
2. Audemard, G., Simon, L.: Predicting learnt clauses quality in modern SAT solvers. In: Proc. of the 21st Int. Joint Conference on Artificial Intelligence (IJCAI 2019). pp. 399–404 (2009)
3. Biere, A.: Splatz, Lingeling, Plingeling, Treengeling, YalSAT Entering the SAT Competition 2016. In: Proc. of SAT Competition 2016 – Solver and Benchmark Descriptions. Dep. of Computer Science Series of Publications B, vol. B-2016-1, pp. 44–45. University of Helsinki (2016)
4. Biere, A., Fröhlich, A.: Evaluating CDCL restart schemes. In: Proc. of the 6th Pragmatics of SAT Workshop (PoS 2015) (2015), to be published.
5. Biere, A., Fröhlich, A.: Evaluating CDCL variable scoring schemes. In: Proc. of the 18th Int. Conference on Theory and Applications of Satisfiability Testing (SAT 2015). LNCS, vol. 9340, pp. 405–422. Springer (2015)
6. Biere, A., Heule, M.J.H.: The effect of scrambling CNFs. In: Proc. of the 9th Pragmatics of SAT Workshop (PoS 2018) (2018), to be published.
7. Biere, A., Heule, M.J.H., van Maaren, H., Walsh, T. (eds.): Handbook of Satisfiability. IOS Press (2009)
8. Bonet, M.L., Buss, S., Ignatiev, A., Marques-Silva, J., Morgado, A.: MaxSAT resolution with the dual rail encoding. In: Proc. of the 32nd AAAI Conference on Artificial Intelligence (AAAI 2018). AAAI Press (2018)
9. Brickenstein, M., Dreyer, A.: PolyBoRi: A framework for gröbner-basis computations with boolean polynomials. Journal of Symbolic Computation 44(9), 1326 – 1345 (2009), effective Methods in Algebraic Geometry
10. Bryant, R.E., Beatty, D., Brace, K., Cho, K., Sheffler, T.: COSMOS: A compiled simulator for MOS circuits. In: Proc. of the 24th ACM/IEEE Design Automation Conference (DAC 87). pp. 9–16. ACM (1987)
11. Chatalic, P., Simon, L.: Multi-resolution on compressed sets of clauses. In: Proc. of the 12th IEEE Int. Conference on Tools with Artificial Intelligence (ICTAI 2000). pp. 2–10 (2000)
12. Cruz-Filipe, L., Marques-Silva, J., Schneider-Kamp, P.: Efficient certified resolution proof checking. In: Proc. of the 23rd Int. Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2017). LNCS, vol. 10205, pp. 118–135 (2017)
13. Dantchev, S.S., Riis, S.: "Planar" tautologies hard for resolution. In: Proc. of the 42nd Annual Symposium on Foundations of Computer Science (FOCS 2001). pp. 220–229. IEEE Computer Society (2001)
14. Devriendt, J., Bogaerts, B., Bruynooghe, M., Denecker, M.: Improved static symmetry breaking for SAT. In: Proc. of the 19th Int. Conference on Theory and Applications of Satisfiability Testing (SAT 2016). LNCS, vol. 9710, pp. 104–122. Springer (2016)
15. Haken, A.: The intractability of resolution. Theoretical Computer Science 39, 297–308 (1985)
16. Heule, M.J.H.: Schur number five. In: Proc. of the 32nd AAAI Conference on Artificial Intelligence (AAAI 2018). AAAI Press (2018)
17. Heule, M.J.H., Biere, A.: What a difference a variable makes. In: Proc. of the 24th Int. Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2018). LNCS, vol. 10806, pp. 75–92. Springer (2018)

18. Heule, M.J.H., Jr., W.A.H., Kaufmann, M., Wetzler, N.: Efficient, verified checking of propositional proofs. In: Proc. of the 8th Int. Conference on Interactive Theorem Proving (ITP 2017). LNCS, vol. 10499, pp. 269–284. Springer (2017)
19. Heule, M.J.H., Kiesl, B., Biere, A.: Short proofs without new variables. In: Proc. of the 26th Int. Conference on Automated Deduction (CADE-26). LNCS, vol. 10395, pp. 130–147. Springer (2017)
20. Heule, M.J.H., Kiesl, B., Seidl, M., Biere, A.: PRuning through satisfaction. In: Proc. of the 13th Haifa Verification Conference (HVC 2017). LNCS, vol. 10629, pp. 179–194. Springer (2017)
21. Heule, M.J.H., van Maaren, H.: Aligning CNF- and equivalence-reasoning. In: Proc. of the 7th Int. Conference on Theory and Applications of Satisfiability Testing (SAT 2004) (2004)
22. Järvisalo, M., Heule, M.J.H., Biere, A.: Inprocessing rules. In: Proc. of the 6th Int. Joint Conference on Automated Reasoning (IJCAR 2012). LNCS, vol. 7364, pp. 355–370. Springer (2012)
23. Kiesl, B., Seidl, M., Tompits, H., Biere, A.: Super-blocked clauses. In: Proc. of the 8th Int. Joint Conference on Automated Reasoning (IJCAR 2016). LNCS, vol. 9706, pp. 45–61. Springer (2016)
24. Kullmann, O.: On a generalization of extended resolution. Discrete Applied Mathematics 96-97, 149–176 (1999)
25. Marques-Silva, J., Lynce, I., Malik, S.: Conflict-driven clause learning SAT solvers. In: Handbook of Satisfiability, pp. 131–153. IOS Press (2009)
26. Marques-Silva, J.P., Sakallah, K.A.: GRASP: A search algorithm for propositional satisfiability. IEEE Transactions on Computers 48(5), 506–521 (1999)
27. McCarthy, J.: A tough nut for proof procedures. Memo 16, Stanford Artificial Intelligence Project (July 1964)
28. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient SAT solver. In: Proc. of the 38th Design Automation Conference (DAC 2001). pp. 530–535. ACM (2001)
29. Nadel, A., Ryvchin, V.: Chronological backtracking. In: Proc. of the 21st Int. Conference on Theory and Applications of Satisfiability Testing (SAT 2018). LNCS, vol. 10929, pp. 111–121. Springer (2018)
30. Palopoli, L., Pirri, F., Pizzuti, C.: Algorithms for selective enumeration of prime implicants. Artificial Intelligence 111(1), 41 – 72 (1999)
31. Soos, M., Nohl, K., Castelluccia, C.: Extending SAT solvers to cryptographic problems. In: Proc. of the 12th Int. Conference on Theory and Applications of Satisfiability Testing (SAT 2009). LNCS, vol. 5584, pp. 244–257. Springer (2009)
32. Tseitin, G.S.: On the complexity of derivation in propositional calculus. In: Automation of Reasoning: 2: Classical Papers on Computational Logic 1967–1970. pp. 466–483. Springer (1983)
33. Urquhart, A.: Hard examples for resolution. Journal of the ACM 34(1), 209–219 (1987)