PRuning Through Satisfaction*

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

[1] Department of Computer Science, The University of Texas at Austin
[2] Institute of Information Systems, TU Wien
[3] Institute for Formal Models and Verification, JKU Linz

Abstract. The classical approach to solving the satisfiability problem of propositional logic prunes unsatisfiable branches from the search space. We prune more agressively by also removing certain branches for which there exist other branches that are more satisfiable. This is achieved by extending the popular conflict-driven clause learning (CDCL) paradigm with so-called PR-*clause learning*. We implemented our new paradigm, named *satisfaction-driven clause learning* (SDCL), in the SAT solver LINGELING. Experiments on the well-known pigeon hole formulas show that our method can automatically produce proofs of unsatisfiability whose size is cubic in the number of pigeons while plain CDCL solvers can only produce proofs of exponential size.

1 Introduction

Conflict-driven clause learning (CDCL) [11] is the leading paradigm for solving the satisfiability problem of propositional logic (SAT). It is well-known that CDCL solvers are able to generate resolution proofs but this useful ability comes at a price because it means that CDCL solvers suffer from the same restrictions as the resolution proof system. For instance, there are seemingly simple formula families that admit only exponential-size resolution proofs, implying that solving these formulas with CDCL takes exponential time [14].

To deal with the limitations of resolution, stronger proof systems have been proposed [19]. Popular examples of such proof systems are *extended resolution* [18] and an even more general system based on blocked clauses [10]. These systems extend resolution by allowing the introduction of short definition clauses over new variables. As shown by Cook [3], the introduction of these clauses already suffices to obtain short proofs of the famous pigeon hole formulas—a class of formulas known for admitting no short resolution proofs [4]. But the introduction of new variables has a downside: The search space of possible variables is infinite in general, which complicates the search for useful definition clauses. This may explain the limited success of GLUCOSER [1], a CDCL solver that uses extended resolution. To cope with this drawback, we recently introduced a proof system, called PR (short for *propagation redundancy*), that allows for short proofs of the pigeon hole formulas without the need to introduce new variables [6].

In this paper, we enhance the CDCL paradigm by extending it in such a way that it can exploit the strengths of the PR proof system. To do so, we introduce *satisfaction-driven clause learning* (SDCL), a SAT solving paradigm that extends CDCL as follows: If the usual unit propagation does not lead to a conflict, we do not immediately decide for a new variable assignment (as would be the case in CDCL). Instead, we first try to prune the search space of possible truth assignments by learning a so-called PR clause.

Intuitively, a PR clause is a clause that might not be implied by the current formula but whose addition preserves satisfiability. As we show in this paper, deciding whether a given clause is a PR clause is NP-complete. We therefore use an additional SAT solver for finding such clauses. Finding useful PR clauses is a non-trivial problem as it is not immediately clear which clauses should be added to improve solver performance. To gain further insight, we develop a strong theory that relates our SAT encoding for finding PR clauses with two concepts from the literature: *autarkies* [9] and *set-blocked clauses* [8].

The main contributions of this paper are as follows: (1) We introduce satisfaction-driven clause learning, a paradigm that extends CDCL by performing the addition of PR clauses. (2) We prove that the problem of deciding whether a given clause is a PR clause is NP-complete. (3) We use a SAT solver for finding PR clauses and show that the corresponding SAT encoding is strongly related to the concepts of autarkies and set-blocked clauses. (4) We implement SDCL as an extension of the award-winning SAT solver LINGELING [2], which is developed by the last author of this paper. An experimental evaluation shows that our approach can generate proofs for much larger pigeon hole formulas than two existing tools based on extended resolution.

2 Preliminaries

Below we present the most important background concepts related to this paper.

Propositional logic. We consider propositional formulas in *conjunctive normal form* (CNF), which are defined as follows. A *literal* is either a variable x (a *positive literal*) or the negation $\overline{x}$ of a variable x (a *negative literal*). The *complementary literal* $\overline{l}$ 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 $F_L = \{C \in F \mid C \cap L \neq \emptyset\}$. For a literal, clause, or formula F, $var(F)$ denotes the variables in 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 if it assigns a truth value to all variables occurring in the formula; otherwise it is *partial*. A literal l is *satisfied* (*falsified*) by an assignment α if l is positive and $\alpha(var(l)) = 1$

($\alpha(var(l)) = 0$, resp.) or if it is negative and $\alpha(var(l)) = 0$ ($\alpha(var(l)) = 1$, resp.). We often denote assignments by sequences of 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. Further, α_L denotes the assignment obtained from α by flipping the truth values of the literals in L. 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.

Formula simplification. We denote the empty clause by $\perp$ and the satisfied clause by $\top$. Given an assignment α and a clause C, we define $C|\alpha = \top$ if α satisfies C; otherwise, $C|\alpha$ denotes the result of removing from C all the literals falsified by α. For a formula F, we define $F|\alpha = \{C|\alpha \mid C \in F \text{ and } C|\alpha \neq \top\}$. We say that an assignment α *touches* a clause C if $var(\alpha) \cap var(C) \neq \emptyset$. Given an assignment α, the clause $\{x \mid \alpha(x) = 0\} \cup \{\overline{x} \mid \alpha(x) = 1\}$ is the clause that *blocks* α. 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 $F|l$ where (l) is a unit clause in F. The iterated application of the unit clause rule to a formula, until no unit clauses are left, is called *unit propagation.* If unit propagation yields the empty clause $\perp$, we say that it derived a *conflict.*

Formula relations. Two formulas are *logically equivalent* if they are satisfied by the same assignments. Two formulas are *satisfiability equivalent* if they are either both satisfiable or both unsatisfiable. Given two formulas F and F', we denote by $F \vDash F'$ that F implies F', i.e., all assignments satisfying F also satisfy F'. Furthermore, by $F \vdash_1 F'$ we denote that for every clause $(l_1 \vee \cdots \vee l_n) \in F'$, unit propagation on $F \wedge (\overline{l}_1) \wedge \cdots \wedge (\overline{l}_n)$ derives a conflict. If $F \vdash_1 F'$, we say that F implies F' through unit propagation. For example, $(x) \wedge (y) \vdash_1 (x \vee z) \wedge (y)$, since unit propagation of the unit clauses $(\overline{x})$ and $(\overline{z})$ derives a conflict with (x), and unit propagation of $(\overline{y})$ derives a conflict with (y).

Conflict-driven clause learning (CDCL) in a nutshell. To evaluate the satisfiability of a formula, a CDCL solver iteratively performs the following operations: First, the solver performs unit propagation. Then, it tests whether it has reached a conflict, meaning that the formula is falsified by the current assignment. If no conflict has been reached and all variables are assigned, the formula is satisfiable. Otherwise, the solver chooses an unassigned variable based on some decision heuristic, assigns a truth value to it, and continues by again performing unit propagation. If, however, a conflict has been reached, the solver learns a short clause that prevents it from repeating similar (bad) decisions in the future ("clause learning"). In case this clause is the (unsatisfiable) empty clause, the unsatisfiability of the formula can be concluded. 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.

3 Searching for Propagation-Redundant Clauses

As already mentioned in the introduction, the addition of so-called PR clauses (short for *propagation-redundant clauses*) to a formula can lead to short proofs for hard formulas without the introduction of new variables. In this section, we present an approach for finding PR clauses. Although PR clauses are not necessarily implied by the formula, their addition preserves satisfiability [6]. The intuitive reason for this is that the addition of a PR clause prunes the search space of possible assignments in such a way that there still remain assignments under which the formula is as satisfiable as under the pruned assignments. In the following definition, assignments can be partial with respect to the formula [6]:

Definition 1. *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 ω satisfies C and $F|_\alpha \vdash_1 F|_\omega$.*

The clause C can be seen as a constraint that prunes from the search space all assignments that extend α. Since $F|_\alpha$ implies $F|_\omega$, every assignment that satisfies $F|_\alpha$ also satisfies $F|_\omega$, meaning that F is at least as satisfiable under ω as it is under α. Moreover, since ω satisfies C, it must disagree with α on at least one variable. We refer to ω as the *witness*, since it witnesses the propagation-redundancy of the clause. Consider the following example [6]:

Example 1. Let $F = (x \vee y) \wedge (\overline{x} \vee y) \wedge (\overline{x} \vee z)$, $C = (x)$, and let $\omega = x\,z$ be an assignment. Then, $\alpha = \overline{x}$ is the assignment blocked by C. Now, consider $F|_\alpha = (y)$ and $F|_\omega = (y)$. Clearly, unit propagation on $F|_\alpha \wedge (\overline{y})$ derives a conflict. Thus, $F|_\alpha \vdash_1 F|_\omega$ and so C is propagation redundant w.r.t. F. $\qquad\square$

Most known types of redundant clauses are PR clauses [6]. This includes *blocked clauses* [10], *set-blocked clauses* [8], *resolution asymmetric tautologies* (RATs) [7], and many more. As a new result, we show next that deciding whether a given clause is a PR clause is NP-complete, which complicates the search for PR clauses.

Definition 2. *The* PR problem *is the following decision problem: Given a formula F and a clause C, decide if C is propagation-redundant w.r.t. F.*

Theorem 1. *The* PR *problem is* NP*-complete.*

Proof. MEMBERSHIP IN NP: Let α be the assignment blocked by C. To decide whether or not C is propagation-redundant with respect to F, just guess an assignment ω and check if $F|_\alpha \vdash_1 F|_\omega$.

NP-HARDNESS: We present a polynomial reduction from the SAT problem. Let F be an input formula (in CNF) for the SAT problem and let v be a fresh variable that does not occur in F. Now, let $C = \overline{v}$ and obtain the formula F' from F by adding to each clause the literal v. We show that F is satisfiable if and only if C is propagation-redundant with respect to F'.

For the "only if" direction, assume that F is satisfied by some assignment ω and let $\alpha = v$ be the assignment blocked by C. Now, obtain a new assignment

ω' from ω by extending it as follows: $\omega'(x) = \omega(x)$ if $x \in var(F)$ and $\omega'(v) = 0$. Then, ω' disagrees with α on v. Moreover, since ω satisfies F, it satisfies F'. Hence, $F'|\omega' = \emptyset$ and thus $F'|\alpha \vdash_1 F'|\omega'$ trivially holds. It follows that C is propagation-redundant with respect to F'.

For the "if" direction, assume that C is propagation-redundant with respect to F' and let $\alpha = v$ be the assignment blocked by C. Then, there exists an assignment ω' such that $F'|\alpha \vdash_1 F'|\omega'$ and ω' disagrees with α, meaning that $\omega'(v) = 0$. Since every clause in F' contains v, it follows that α satisfies F' and so it must be the case that ω' satisfies F'. Since $\omega'(v) = 0$ and $F'|\overline{v} = F$, it follows that ω' satisfies F. $\qquad\square$

Since the identification of PR clauses is NP-hard, we use a SAT solver to search for PR clauses. We thus next introduce a SAT encoding which, for a given formula F and an assignment α, tries to find a witness ω that certifies the propagation redundancy of the clause that blocks α. We obtain the encoding, which we call the *positive reduct*, by selecting only a subpart of F:

Definition 3. *Let F be a formula, α an assignment, and C the clause that blocks α. The* positive reduct *$p(F, \alpha)$ of F with respect to α is the formula $G \wedge C$, where G is obtained from F by first removing all clauses that are not satisfied by α and then removing from the remaining clauses all literals that are not assigned by α.*

Example 2. 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 $p(F, \alpha)$ of F w.r.t. α is the formula $(x \vee \overline{y} \vee z) \wedge (\overline{z}) \wedge (\overline{x} \vee \overline{y} \vee z)$. $\quad\square$

We next show that the positive reduct is satisfiable if and only if the clause blocked by α is a set-blocked clause [8] (see Definition 4 below), meaning that it is also a PR clause [6] (note that deciding set-blockedness of a clause is also NP-complete [8]). We show later that we can usually shorten this set-blocked clause and thereby turn it into a PR clause that might not be set-blocked anymore.

Definition 4. *A clause C is* set-blocked *by a non-empty set $L \subseteq C$ in a formula F if, for every clause $D \in F_{\overline{L}}$, the clause $(C \setminus L) \cup \overline{L} \cup D$ contains two complementary literals.*

We say that a clause is set-blocked in a formula F if it is set-blocked by some of its literals in F. Consider the following example [8]:

Example 3. Let $C = (x \vee y)$ and $F = (\overline{x} \vee y) \wedge (x \vee \overline{y})$. Then, C is set-blocked by $L = \{x, y\}$: Clearly, $F_{\overline{L}} = F$ and $C \setminus L = \emptyset$. Therefore, for $D_1 = (\overline{x} \vee y)$ we get that $(C \setminus L) \cup \overline{L} \cup D_1 = (\overline{x} \vee \overline{y} \vee y)$ contains two complementary literals and the same holds for $D_2 = (x \vee \overline{y})$, for which we get $(C \setminus L) \cup \overline{L} \cup D_2 = (\overline{x} \vee \overline{y} \vee x)$. $\quad\square$

Assume we are given a clause C which blocks some assignment α. Our new result given in the following theorem implies that C is set-blocked in a formula F if and only if the positive reduct $p(F, \alpha)$ is satisfiable. Recall that for an assignment α and a set of literals L, α_L denotes the assignment obtained from α by flipping the truth values of the literals in L:

Theorem 2. *Let F be a formula, α an assignment, and C the clause that blocks α. Then, C is set-blocked by $L \subseteq C$ in F if and only if α_L satisfies the positive reduct $p(F, \alpha)$.*

Proof. For the "only if" direction, assume that C is set-blocked by L in F. We show that α_L satisfies $p(F, \alpha)$. Clearly, α_L satisfies C since α_L is obtained from α by flipping the truth values of the literals in L. Now, let D be a clause in $p(F, \alpha)$ that is different from C. We show that D is satisfied by α_L. By the definition of $p(F, \alpha)$, D is satisfied by α and thus, if D contains no literals of $\overline{L}$ (i.e., $D \notin F_{\overline{L}}$), it is also satisfied by α_L. Assume therefore that $D \in F_{\overline{L}}$. Then, since C is set-blocked by L in F, the clause $(C \setminus L) \cup \overline{L} \cup D$ contains two complementary literals.

Since C cannot contain two complementary literals (because it blocks the assignment α), there must be a literal $l \in D$ such that one of the following holds: (1) $\bar{l} \in D$, (2) $\bar{l} \in C \setminus L$, (3) $\bar{l} \in \overline{L}$. In the first case, D is clearly satisfied by α_L. In the second case, since α_L differs from α only on literals in L and since α falsifies C, it follows that α_L falsifies $\bar{l}$ and thus it satisfies l. Finally, in the third case, it follows that $l \in L$ and so α_L satisfies l since it satisfies all the literals in L. It follows that D is satisfied by α_L. Therefore, α_L satisfies $p(F, \alpha)$.

For the "if" direction, assume that α_L satisfies $p(F, \alpha)$. We show that L set-blocks C in F. Let $D \in F_{\overline{L}}$. Since α falsifies C, it falsifies L. Therefore, α satisfies $\overline{L}$ and thus $p(F, \alpha)$ contains the clause D', obtained from D by removing all literals that are not assigned by α. By assumption, α_L satisfies D' and since it falsifies $\overline{L}$, it must satisfy some literal $l \in D' \setminus \overline{L}$. But then $\bar{l} \in C \setminus L$ and thus the clause $(C \setminus L) \cup \overline{L} \cup D$ contains two complementary literals. Hence, C is set-blocked by L in F. $\square$

Thus, if the SAT solver finds an assignment α for which the positive reduct with respect to F is satisfiable, then the clause that blocks α is a set-blocked clause and so its addition to F preserves satisfiability. Even better, when using a CDCL solver, we can usually add a shorter clause: If α is the current assignment of the solver, it consists of two parts—a part α_d of variable assignments that were decisions by the solver and a part α_u of assignments that were derived from these decisions via unit propagation. This means that $F|_{\alpha_d} \vdash_1 F|_\alpha$. Since C is set-blocked—and thus propagation-redundant—with respect to F, we know that there exists some assignment ω such that $F|_\alpha \vdash_1 F|_\omega$. But then $F|_{\alpha_d} \vdash_1 F|_\omega$ and so the clause that blocks α_d, which is a subclause of the clause that blocks α, is a PR clause with respect to F. We conclude:

Theorem 3. *Let C be a PR clause w.r.t. a formula F and let $\alpha = \alpha_d \cup \alpha_u$ be the assignment blocked by C. Assume furthermore that the assignments in α_u are derived via unit propagation on $F|_{\alpha_d}$. Then, the clause that blocks α_d is propagation-redundant w.r.t. to F.*

We can thus efficiently find short PR clauses by using an additional SAT solver for finding a set-blocked clause and then shortening the clause by removing literals that are not decision literals.

4 Conditional Autarkies

We have seen that the positive reduct can be used to determine whether a clause is set-blocked with respect to a given formula. As we show in this section, searching for satisfying assignments of the positive reduct is actually the same as searching for certain kinds of partial assignments [12]:

Definition 5. *A partial assignment ω is an* autarky *for a formula F if ω satisfies every $C \in F$ for which $var(\omega) \cap var(C) \neq \emptyset$.*

In other words, an autarky is a (partial) assignment that satisfies every clause it touches. For example, if a literal l is pure in a formula (i.e., $\bar{l}$ does not occur in the formula), then the assignment $\omega = l$ is an autarky for the formula. But also the empty assignment as well as every assignment that satisfies the whole formula are autarkies. If we are given an autarky ω for a formula F, we can use ω to simplify F because $F|\omega$ and F are satisfiability equivalent, although they are not necessarily logically equivalent [12]. Autarkies yield PR clauses as follows:

Theorem 4. *Let F be a formula and ω an autarky for F. Then, every clause C such that ω satisfies C and $var(C) \subseteq var(\omega)$ is a* PR *clause with respect to F.*

Proof. Let α be the assignment blocked by C. We show that $F|\alpha \vdash_1 F|\omega$. Let $D|\omega \in F|\omega$ for $D \in F$. Since D is not satisfied by ω, it follows that D is not touched by ω and thus—since $var(\alpha) \subseteq var(\omega)$—it is also not touched by α. Hence, $D|\alpha = D|\omega = D$ is contained in $F|\alpha$ and so $F|\alpha \vdash_1 D|\omega$. It follows that C is a PR clause with respect to F. □

Suppose a SAT solver has found a partial assignment ω_{con} for some formula F. We can then try to search for autarkies in the simplified formula $F|\omega_{con}$. Given an autarky ω_{aut} for $F|\omega_{con}$, we call $\omega = \omega_{con} \cup \omega_{aut}$ a *conditional autarky* for F:

Definition 6. *A partial assignment ω is a* conditional autarky *for a formula F if there exists a subassignment $\omega_{con} \subset \omega$ such that ω is an autarky for $F|\omega_{con}$. We call ω_{con} the* conditional part *of ω.*

If $\omega \setminus \omega_{con}$ assigns exactly one variable, we call the literal satisfied by $\omega \setminus \omega_{con}$ a *conditional pure literal with respect to ω_{con}.*

Example 4. Consider the formula $F = (x \vee y) \wedge (\overline{x} \vee z) \wedge (\overline{y} \vee \overline{z})$. The assignment $\omega = \overline{y}\,z$ is a conditional autarky with conditional part $\omega_{con} = \overline{y}$: By applying ω_{con} to F, we obtain the formula $F|\overline{y} = (x) \wedge (\overline{x} \vee z)$. The only clause of $F|\overline{y}$ that is touched by ω is the clause $(\overline{x} \vee z)$, which is satisfied ω. The literal z is a conditional pure literal with respect to ω_{con}. □

Note that every autarky ω is a conditional autarky where ω_{con} is the empty assignment. However, as illustrated by Example 4, the converse does not hold: Although the assignment $\omega = \overline{y}\,z$ is a conditional autarky for F, it is not an autarky for F because the clause $(x \vee y)$ is touched but not satisfied by ω. The following theorem shows that satisfying assignments of the positive reduct are nothing else than conditional autarkies:

Theorem 5. *Let F be a formula and α a partial assignment. Then, an assignment ω over $var(\alpha)$ satisfies the positive reduct $p(F, \alpha)$ if and only if ω is a conditional autarky for F with $\omega_{\mathrm{con}} = \alpha \cap \omega$.*

Proof. For the "only if" direction, assume that ω is a satisfying assignment of $p(F, \alpha)$. First, note that ω disagrees with α on at least one variable since ω satisfies the clause that blocks α. Therefore, $\omega_{\mathrm{con}} \subset \omega$. It remains to show that ω is an autarky for $F|_{\omega_{\mathrm{con}}}$. Let $D|_{\omega_{\mathrm{con}}}$ be a clause in $F|_{\omega_{\mathrm{con}}}$ such that $D \in F$ and assume that $D|_{\omega_{\mathrm{con}}}$ is touched by ω. Since $var(\omega) = var(\alpha)$, it follows that $D|_{\omega_{\mathrm{con}}}$ is also touched by α. Now, if α does not satisfy $D|_{\omega_{\mathrm{con}}}$, then ω satisfies $D|_{\omega_{\mathrm{con}}}$ since ω disagrees with α on all variables in $var(\alpha) \setminus var(\omega_{\mathrm{con}})$. In contrast, if α satisfies $D|_{\omega_{\mathrm{con}}}$, then the clause D', which contains only those literals of D that are touched by α, is contained in $p(F, \alpha)$. Hence, ω satisfies D' and thus it satisfies $D|_{\omega_{\mathrm{con}}}$. It follows that ω is a conditional autarky for F.

For the "if" direction, assume that ω is a conditional autarky for F with $\omega_{\mathrm{con}} = \alpha \cap \omega$ and let $D' \in p(F, \alpha)$. If D' is the clause that blocks α, then ω satisfies D' since ω disagrees with α (note that by definition $\omega_{\mathrm{con}} \subset \omega$). Assume thus that D' is not the clause that blocks α. Then, there exists a clause $D \in F$ such that α satisfies D and D' is obtained from D by removing all literals that are not assigned by α. Assume now that ω_{con} does not satisfy D'. Then, $D|_{\omega_{\mathrm{con}}} \in F|_{\omega_{\mathrm{con}}}$ (note that ω_{con} cannot satisfy D since the literals in $D \setminus D'$ are not assigned by α and thus also not by ω). Since α satisfies D, it satisfies $D|_{\omega_{\mathrm{con}}}$. Hence, $D|_{\omega_{\mathrm{con}}}$ is touched by α and thus also by ω. But then ω satisfies $D|_{\omega_{\mathrm{con}}}$ since it is a conditional autarky for F. Hence, since D' contains all literals of D that are assigned by α (and thus by ω), ω satisfies D'. It follows that ω satisfies $p(F, \alpha)$. $\square$

Combining Theorem 5 with Theorem 2, which states that a clause C is set-blocked by $L \subseteq C$ in a formula F if and only if α_L satisfies $p(F, \alpha)$, we obtain the following relationship between conditional autarkies and set-blocked clauses:

Corollary 1. *Let F be a formula, C a clause, and α the assignment blocked by C. Then, C is set-blocked by $L \subseteq C$ in F if and only if α_L is a conditional autarky for F with conditional part $\alpha_L \cap \alpha$.*

This correspondence between set-blocked clauses and conditional autarkies reveals an interesting relationship between set-blocked clauses and PR clauses:

Theorem 6. *Let $C \vee L$ be a clause that is set-blocked by L with respect to a formula F. Any clause $C \vee l$ with $l \in L$ is a PR clause with respect to F.*

Proof. Let α be the assignment that is blocked by $C \vee l$. We need to show that there exists an assignment ω such that ω satisfies $C \vee l$ and $F|_\alpha \vdash_1 F|_\omega$. Let ω be α_L. Clearly, ω satisfies $C \vee l$ on l. Moreover, we know from Theorem 1 that ω is a conditional autarky for F with conditional part $\omega_{\mathrm{con}} = \alpha \cap \omega$. Now, let $F' = F|_{\omega_{\mathrm{con}}}$, $\alpha' = \alpha \setminus \omega_{\mathrm{con}}$, and $\omega' = \omega \setminus \omega_{\mathrm{con}}$. Then, $F|_\alpha = F'|_{\alpha'}$ and $F|_\omega = F'|_{\omega'}$. Since $var(\alpha') \subseteq var(\omega')$ and ω' is an autarky for F' (Lemma 4), it follows that $F'|_{\alpha'} \vdash_1 F'|_{\omega'}$. But then $F|_\alpha \vdash_1 F|_\omega$ and thus $C \vee l$ is a PR clause w.r.t. F. $\square$

```
SDCL (formula F)
1     α := ∅
2     forever do
3       α := Simplify (F, α)
4       if F|α contains a falsified clause then
5         C := AnalyzeConflict ()
6         if C is the empty clause then return unsatisfiable
7         F := F ∪ {C}
8         α := BackJump (C, α)
9       else if p(F, α) is satisfiable then
10        C := AnalyzeWitness ()
11        F := F ∪ {C}
12        α := BackJump (C, α)
13      else
14        l := Decide ()
15        if l is undefined then return satisfiable
16        α := α ∪ {l}
```

Fig. 1. Pseudo-code of the *SDCL* procedure

This basically means that if a clause $C \vee L$ is set-blocked by L in a formula F, we can add any clause $C \vee l$ such that $l \in L$ to F without affecting its satisfiability.

Example 5. Consider the formula $F = (x \vee y \vee z) \wedge (\overline{x} \vee \overline{y} \vee z) \vee (x \vee \overline{y} \vee \overline{z})$. The clause $(\overline{x} \vee y \vee z)$ is set-blocked by $\{\overline{x}, y\}$ in F and so $(\overline{x} \vee z)$ and $(y \vee z)$ are PR clauses w.r.t. F as both $\alpha = x\,\overline{y}\,\overline{z}$ and $\omega = \overline{x}\,y\,\overline{z}$ are autarkies for F. Therefore, the addition of $(\overline{x} \vee z)$ or $(y \vee z)$ to F preserves satisfiability. □

We have seen different approaches to finding and adding PR clauses to a formula. In the following, we make use of these approaches when introducing our extension of conflict-driven clause learning.

5 Satisfaction-Driven Clause Learning

Our *satisfaction-driven clause learning* (SDCL) paradigm extends the CDCL paradigm in the following way: Whenever the CDCL solver is required to pick a new decision, we first check whether the current assignment and all its extensions can be *pruned* from the search space by learning the clause which contains the negation of all previous decisions. As explained in Section 3, such a learned clause can be obtained by searching for a satisfying assignment of the positive reduct with respect to the formula and the current assignment of the solver.

Figure 1 shows the pseudo code of the *SDCL* procedure. Removing lines 9 to 12 would result in the classical CDCL algorithm, which consists of three phases: *simplify*, *learn*, and *decide*. The simplify phase uses unit propagation to extend the current assignment α (line 3). The main reasoning of CDCL is performed in the learn phase, which kicks in when a *conflict* is reached, i.e., when a clause

is falsified by the current assignment α (i.e., when the if condition in line 4 is true). In this case, a so-called *conflict clause* is computed by the *AnalyzeConflict* procedure (line 5). A conflict clause serves as a constraint that should prevent the solver from investigating unsatisfiable parts of the search space in the future by encoding the reasons for the current conflict. The naive approach for this is to use the clause that blocks α as conflict clause. A stronger conflict clause can be obtained by computing the conflict clause that blocks only the decision literals of α. In practice, there are several approaches for learning even smaller clauses [11,17].

If the conflict clause is the empty clause, then the solver can conclude that the formula is unsatisfiable (line 6); otherwise, the clause is added to the formula (line 7). After adding the conflict clause to the formula (*clause learning*), the solver backjumps (line 8) to the level where the conflict clause contains a literal that is not falsified. Finally, the decide phase (lines 14 to 16) extends α by selecting a literal and making it true. In case all variables are assigned and no clause is falsified, the formula is identified as satisfiable (line 15).

The *SDCL* related lines (9 to 12) work as follows: If the current assignment α does *not* lead to a conflict (i.e., the if condition on line 4 fails), we check (optionally in a limited way) whether the positive reduct $p(F, \alpha)$ is satisfiable. If not, a new decision is made (line 14). Otherwise, we conclude that the clause that blocks α is set-blocked and thus redundant with respect to F. Similar to the *AnalyzeConflict* procedure, which shortens conflict clauses in practice, we can learn a clause that is smaller than the one that blocks α. This is done in the *AnalyzeWitness* procedure, which analyzes the assignment that satisfies the positive reduct (the witness). As shown in Theorem 3, we can add the clause that blocks only the decision literals of α since it is a PR clause. Alternatively, we can add PR clauses based on conditional autarkies as described in Theorem 6. After the clause addition, we backjump by unassigning all variables up to the last decision (line 12) and continue with a new iteration of the procedure.

A crucial part of the algorithm is the underlying decision heuristic of the *Decide* procedure. Most practical implementations of CDCL use the so-called *VSIDS* (*Variable State Independent Decaying Sum*) [13] heuristic which selects the variable that occurs most frequently in recent conflict clauses. In our early experiments, VSIDS turned out to be a poor heuristic for SDCL. A possible explanation is that VSIDS can select variables as early decisions that occur in different parts of the formula, thereby making it impossible to satisfy the resulting positive reduct.

In order to select variables in the same part of the formula, we propose the *autarky decision heuristic*: Given a formula F and the current assignment α, the autarky decision heuristic selects the variable that occurs most frequently in clauses in $F|_\alpha \setminus F$ (i.e., in the clauses of F that are touched but not satisfied by α). Occurrences are weighted based on the length of clauses—the smaller the clause, the larger the weight. If $F|_\alpha \setminus F$ is empty, then α is an autarky for F, hence the name. So this heuristic tries to guide the solver to an autarky.

We expect that this heuristic helps with finding conditional autarkies—and thus with satisfying the positive reduct formulas—more efficiently.

The autarky heuristic can only be used for non-empty assignments. We therefore need a special heuristic for the first decision. This turned out to be challenging and is still part of current research. A heuristic that works really well for the pigeon hole formulas is to select the variable x that is least constrained, i.e., either x or $\overline{x}$ occurs least frequently in the formula. The rationale behind this heuristic is that it creates an initial positive reduct with as few unit clauses as possible: Notice that a clause which is satisfied by the first decision becomes a unit clause in the positive reduct unless unit propagation assigns other literals in that clause. Such a unit clause makes it impossible to satisfy the positive reduct for the first decision. Also, this heuristic finds pure literals and fixes them using a PR clause: The positive reduct has only the clause that blocks the current assignment and can thus be trivially satisfied. However, it is unlikely that this heuristic is effective for a wide spectrum of benchmark families.

In the next section, we illustrate how short proofs of the pigeon hole formulas can be produced manually by combining the addition of set-blocked clauses with resolution. With this we want to illustrate why short proofs can be found automatically by our implementation of SDCL.
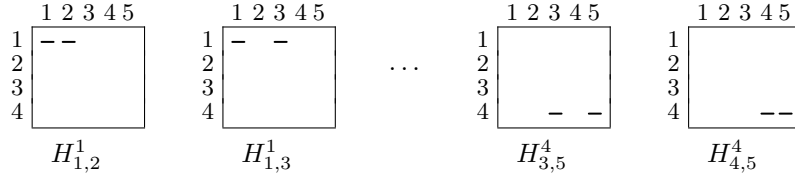
6 Solving Pigeon Hole Formulas using SDCL

A pigeon hole formula PHP_n intuitively encodes that $n + 1$ pigeons have to be assigned to n holes such that no hole contains more than one pigeon. In the encoding, a variable $x_{i,k}$ denotes that pigeon i is assigned to hole k:

$$PHP_n := \bigwedge_{1 \leq i \leq n+1} \overbrace{(x_{i,1} \vee \cdots \vee x_{i,n})}^{P_i} \wedge \bigwedge_{1 \leq i < j \leq n+1} \bigwedge_{1 \leq k \leq n} \overbrace{(\overline{x}_{i,k} \vee \overline{x}_{j,k})}^{H_{i,j}^k} \quad (1)$$

Clearly, pigeon hole formulas are unsatisfiable. Following Haken [4], we use array notation for clauses: Every clause is represented by an array of $n + 1$ columns and n rows. An array contains a "+" ("−") in the i-th column and k-th row if and only if the variable $x_{i,k}$ occurs positively (negatively, respectively) in the corresponding clause. The representation of PHP_n in array notation has for every clause $(x_{i,1} \vee \cdots \vee x_{i,n})$, an array in which the i-th column is filled with "+". Moreover, for every clause $(\overline{x}_{i,k} \vee \overline{x}_{j,k})$, there is an array that contains two "−" in row k—one in column i and the other in column j. For instance, PHP_4 in array notation looks as follows:

	1	2	3	4	5
1	+				
2	+				
3	+				
4	+				

P_1

	1	2	3	4	5
1		+			
2		+			
3		+			
4		+			

P_2

	1	2	3	4	5
1			+		
2			+		
3			+		
4			+		

P_3

	1	2	3	4	5
1				+	
2				+	
3				+	
4				+	

P_4

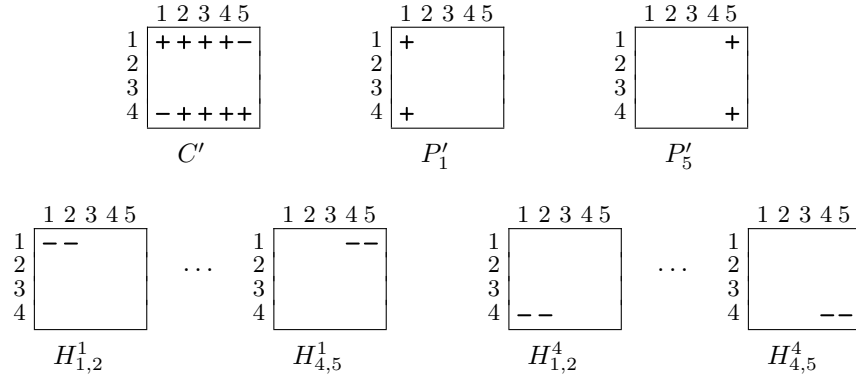|	1	2	3	4	5
1					+
2					+
3					+
4					+

P_5

We use array notation to describe a method for learning binary clauses. For the explanation, we pick $(\overline{x}_{1,5} \vee \overline{x}_{4,1})$ in PHP_4 as it allows an easy formulation of the proof of pigeon hole formulas. The proof idea is similar to that of Cook: We reduce a pigeon hole formula PHP_n to the smaller formula PHP_{n-1}. The main difference is that in our case PHP_{n-1} still uses the same variables as PHP_n.

Again, we pick the clause $C = (\overline{x}_{1,5} \vee \overline{x}_{4,1}) \in PHP_4$. Let $\alpha_d = x_{1,5}\, x_{4,1}$ be the assignment blocked by C. Then, $\alpha = \overline{x}_{1,1}\, \overline{x}_{1,2}\, \overline{x}_{1,3}\, \overline{x}_{1,4}\, x_{1,5}\, x_{4,1} \overline{x}_{4,2}\, \overline{x}_{4,3}\, \overline{x}_{4,4}\, \overline{x}_{4,5}$ is obtained from α_d by applying unit propagation. Let C' be the clause that blocks α. The clauses and assignments in array notation are as follows:
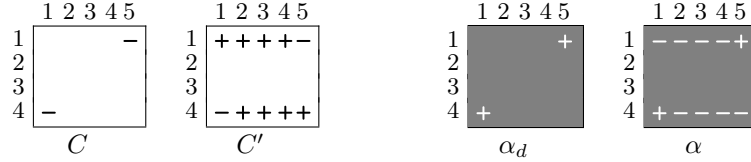
Next, we construct the positive reduct $p(PHP_4, \alpha)$. The positive reduct contains C' and all clauses of PHP_4 that are satisfied by α, which are the following 22 clauses: $P_1, P_5, H^1_{1,2}, \ldots, H^1_{4,5}, H^4_{1,2}, \ldots H^4_{4,5}$. From these clauses, we remove the literals that are not assigned by α and obtain the positive reduct $p(PHP_4, \alpha)$:

The positive reduct is satisfied by the following witness and conditional autarky:
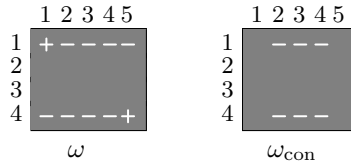
According to Theorem 3, we can learn clause C and, according to Theorem 6, we can learn the clauses A_1, A_2, A_3, and A_4:

```
    1 2 3 4 5        1 2 3 4 5        1 2 3 4 5        1 2 3 4 5        1 2 3 4 5
  1 ┌       −┐    1 ┌− + + +┐    1 ┌  + + + +┐    1 ┌  + + +  ┐    1 ┌  + + +  ┐
  2 │        │    2 │        │    2 │          │    2 │         │    2 │         │
  3 │        │    3 │        │    3 │          │    3 │         │    3 │         │
  4 └−       ┘    4 └  + + + ┘    4 └  + + +   ┘    4 └+ + + +  ┘    4 └  + + + −┘
        C              A_1             A_2              A_3              A_4
```

After learning $(\overline{x}_{1,5} \vee \overline{x}_{4,1})$, we can learn the clause $(\overline{x}_{2,5} \vee \overline{x}_{4,1})$ in a similar way. Now the assignment $\alpha_d = x_{4,1}$ can be extended using unit propagation to obtain $\alpha = \overline{x}_{3,1} \overline{x}_{3,2} \overline{x}_{3,3} \overline{x}_{3,4} x_{3,5} x_{4,1} \overline{x}_{4,2} \overline{x}_{4,3} \overline{x}_{4,4} \overline{x}_{4,5}$ The positive reduct of the extended formula, $p(PHP_4 \wedge (\overline{x}_{1,5} \vee \overline{x}_{4,1}) \wedge (\overline{x}_{2,5} \vee \overline{x}_{4,1}), \alpha)$, is satisfiable, allowing us to learn the unit clause $(\overline{x}_{4,1})$. By repeating the same procedure three times, we can learn the clauses $(\overline{x}_{4,2})$, $(\overline{x}_{4,3})$, and $(\overline{x}_{4,4})$ in a similar way. Now the clauses P_1 to P_4 can be shortened because their last literal is falsified. We have thus reduced PHP_4 to PHP_3.

7 Evaluation

We implemented a prototype[4] of SDCL on top of the plain LINGELING solver [2] (no pre- or inprocessing), including proof-logging support in the PR proof format [6]. We focus on solving large pigeon hole formulas efficiently and automatically, although we envision that the paradigm will be broadly applicable.

Apart from the standard encoding of the pigeon hole formulas, we ran experiments on two alternative, more compact, encodings. Both encodings replace the at-most-one constraint $\leq_1 (x_{1,k}; \ldots; x_{n+1,k})$, i.e., the $H_{i,j}^k$ clauses in formula (1): The first alternative replaces the direct encoding by a sequential counter encoding [15]. The second alternative, the minimal encoding, iteratively replaces three literals of the at-most-one constraint with a new literal and adds an at-most-one constraint between the three replaced literals and the new literal.

We compared our method with two tools that can reason using extended resolution: EBDDRES [16] and GLUCOSER [1]. Both tools can refute pigeon hole formulas efficiently compared to CDCL solvers. EBDDRES solves a given formula using BDDs and optionally converts the BDD proof into an extended-resolution proof, linear in the number of BDD nodes, which in turn can be checked using the TraceCheck tool [5]. GLUCOSER is an extension of the GLUCOSE solver that allows *extended learning*—a method that adds definitions based on conflict clauses. Proof logging is not supported by GLUCOSER, but it could in theory be added with reasonable effort.[5]

Table 1 shows the results of our experiments. Each benchmark was executed on a compute node with two Intel(R) Xeon(R) E5-2620 v4 @ 2.10GHz CPUs and 128 GB of main memory, running Ubuntu 16.04.2 64-bit.

---

[4] The tools and files are available at `http://fmv.jku.at/prune/`

[5] Because we were unable to compile the sources, we could not add proof logging. Instead, we used a statically compiled binary.

Table 1. The number of variables and clauses of pigeon hole formulas—standard (-std), sequential counter (-seq), and minimal (-min)—as well as the runtime (in seconds) and proof size (in BDD nodes or lemmas) for solving the formulas with EBDDRES, GLUCOSER, and our SDCL variant of LINGELING. "TO" means a timeout after 9000 seconds and "OF" means 32-bit index overflow ($\geq 2^{30}$ cache lines) for EBDDRES.

formula	input #var	#cls	EBDDRES time	#node	GLUCOSER time	#lemma	LINGELING (PR) time	#lemma
PHP_{10}-std	110	561	1.00	3,182,495	22.71	329,470	0.07	329
PHP_{11}-std	132	738	3.47	9,493,302	146.61	1,514,845	0.11	439
PHP_{12}-std	156	949	10.64	27,351,195	307.29	2,660,358	0.16	571
PHP_{13}-std	182	1,197	30.81	76,513,832	982.84	6,969,736	0.22	727
PHP_{20}-std	420	4,221	OF	——	TO	——	1.61	2,659
PHP_{30}-std	930	13,981	OF	——	TO	——	13.45	8,989
PHP_{40}-std	1,640	32,841	OF	——	TO	——	67.41	21,319
PHP_{50}-std	2,550	63,801	OF	——	TO	——	241.14	41,649
PHP_{10}-seq	220	311	OF	——	1.62	25,712	0.07	327
PHP_{11}-seq	264	375	OF	——	6.94	77,747	0.10	437
PHP_{12}-seq	312	445	OF	——	19.40	174,084	0.14	569
PHP_{13}-seq	364	521	OF	——	172.76	1,061,318	0.18	725
PHP_{20}-seq	840	1,221	OF	——	TO	——	1.05	2,657
PHP_{30}-seq	1,860	2,731	OF	——	TO	——	6.55	8,987
PHP_{40}-seq	3,280	4,841	OF	——	TO	——	27.10	21,317
PHP_{50}-seq	5,100	7,551	OF	——	TO	——	86.30	41,647
PHP_{10}-min	180	281	28.60	81,490,141	0.64	15,777	0.06	329
PHP_{11}-min	220	342	143.92	399,014,970	1.82	34,561	0.10	439
PHP_{12}-min	264	409	OF	——	9.87	121,321	0.13	571
PHP_{13}-min	312	482	OF	——	57.66	483,789	0.18	727
PHP_{20}-min	760	1,161	OF	——	TO	——	1.03	2,659
PHP_{30}-min	1,740	2,641	OF	——	TO	——	6.30	8,989
PHP_{40}-min	3,120	4,721	OF	——	TO	——	26.65	21,319
PHP_{50}-min	4,900	7,401	OF	——	TO	——	85.00	41,649

Our version of LINGELING is the only tool that can solve pigeon hole formulas with 20 or more pigeons. Over 99% of the learned clauses ("lemmas") produced by LINGELING are PR clauses; the remaining ones are conflict clauses. The number of lemmas for PHP_n is cubic in n, while the number of variables and the size of the formula are at least quadratic in n. All PR clauses that are found by LINGELING are also added to the proofs. The size of the automatically produced PR proofs is similar to that of our manual proofs [6]. For the proofs returned by LINGELING, we observed that between around 5 % and 40 % of the clauses are not required for proving unsatisfiability. Moreover, our approach is robust: Performance varies only minimally across the different encodings of the pigeon hole formulas.

If we turn off our SDCL code, LINGELING requires exponential runtime on PHP_n formulas. Runtimes are similar but in all cases larger than for GLUCOSER, e.g., 153 seconds for PHP_{13}-min. We also want to highlight that the autarky

decision heuristic is essential for proving the unsatisfiability of the pigeon hole formulas—without this heuristic, Lingeling could not find proofs within the given time limit.

8   Conclusions

We proposed a theoretical and a practical approach to searching for PR clauses. First, we showed that searching for a PR clause is an NP-complete problem. As a consequence, a SAT solver that performs the addition of PR clauses has to solve multiple NP-complete problems instead of only one. To make this approach feasible and efficient, we turned the problem of finding a PR clause into a SAT encoding that is significantly easier than the original problem. We called this encoding the *positive reduct* and showed that satisfying the positive reduct yields a set-blocked clause, or, equivalently, a conditional autarky. We also demonstrated how this set-blocked clause can be shortened. Based on our theoretical results, we introduced SDCL—a new SAT-solving paradigm that generalizes CDCL so that it produces not only conflict clauses but also PR clauses. Finally, we implemented SDCL in the solver Lingeling and performed preliminary experiments with the pigeon hole formulas that are very promising.

In future work, we want to focus on making the SDCL approach effective on a wide spectrum of formulas. There are several challenges ahead. First and foremost, a heuristic needs to be developed that facilitates finding PR clauses with few decisions. Our autarky decision heuristic appears to be a useful first step. Second, experiments should be performed to find out which PR clauses prune the search space most effectively. In our evaluation, we selected PR clauses based on decisions. An alternative is to use PR clauses based on conditional autarkies. It is also not yet clear when and how often an SDCL solver should search for PR clauses to achieve the best performance. Moreover, it could make sense to restrict the time spent on solving the positive reduct. Finally, we observed that although the computational costs for solving the positive reducts are low, the costs of generating the reducts are very high. Reducing the generation costs could thus improve the performance significantly.

References

1. Audemard, G., Katsirelos, G., Simon, L.: A Restriction of Extended Resolution for Clause Learning SAT Solvers. In: Proc. of the 24th AAAI Conference on Artificial Intelligence (AAAI 2010). pp. 15–20. AAAI Press (2010)
2. 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)
3. Cook, S.A.: A short proof of the pigeon hole principle using extended resolution. SIGACT News 8(4), 28–32 (Oct 1976)
4. Haken, A.: The intractability of resolution. Theoretical Computer Science 39, 297–308 (1985)

5. Heule, M.J.H., Biere, A.: Proofs for satisfiability problems. In: All about Proofs, Proofs for All (APPA), Math. Logic and Foundations, vol. 55. College Pub. (2015)
6. 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)
7. 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, Heidelberg (2012)
8. 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, Cham (2016)
9. Kleine Büning, H., Kullmann, O.: Minimal unsatisfiability and autarkies. In: Handbook of Satisfiability, pp. 339–401. IOS Press (2009)
10. Kullmann, O.: On a generalization of extended resolution. Discrete Applied Mathematics 96-97, 149–176 (1999)
11. Marques Silva, J.P., Sakallah, K.A.: GRASP: A search algorithm for propositional satisfiability. IEEE Trans. Computers 48(5), 506–521 (1999)
12. Monien, B., Speckenmeyer, E.: Solving satisfiability in less than 2^n steps. Discrete Applied Mathematics 10(3), 287–295 (1985)
13. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient SAT solver. In: Proceedings of the 38th Design Automation Conference (DAC 2001). pp. 530–535. ACM (2001)
14. Nordström, J.: On the interplay between proof complexity and SAT solving. SIGLOG News 2(3), 19–44 (2015)
15. Sinz, C.: Towards an optimal CNF encoding of boolean cardinality constraints. In: Proc. of the 11th Int. Conference on Principles and Practice of Constraint Programming (CP 2005). LNCS, vol. 3709, pp. 827–831. Springer (2005)
16. Sinz, C., Biere, A.: Extended Resolution Proofs for Conjoining BDDs. In: Proc. of the 1st Int. Computer Science Symposium in Russia (CSR 2006). LNCS, vol. 3967, pp. 600–611. Springer (2006)
17. Sörensson, N., Biere, A.: Minimizing learned clauses. In: Proc. of the 12th Int. Conference on Theory and Applications of Satisfiability Testing (SAT 2009). LNCS, vol. 5584, pp. 237–243. Springer (2009)
18. 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, Heidelberg (1983)
19. Urquhart, A.: The complexity of propositional proofs. In: Current Trends in Theoretical Computer Science, pp. 332–342. World Scientific (2001)