Preprocessing of Propagation Redundant Clauses[*]

Joseph E. Reeves, Marijn J. H. Heule, and Randal E. Bryant

Carnegie Mellon University, Pittsburgh, Pennsylvania, United States
{jereeves,mheule,randy.bryant}@cs.cmu.edu

Abstract. The *propagation redundant* (PR) proof system generalizes the *resolution* and *resolution asymmetric tautology* proof systems used by *conflict-driven clause learning* (CDCL) solvers. PR allows short proofs of unsatisfiability for some problems that are difficult for CDCL solvers. Previous attempts to automate PR clause learning used hand-crafted heuristics that work well on some highly-structured problems. For example, the solver SADICAL incorporates PR clause learning into the CDCL loop, but it cannot compete with modern CDCL solvers due to its fragile heuristics. We present PRELEARN, a preprocessing technique that learns short PR clauses. Adding these clauses to a formula reduces the search space that the solver must explore. By performing PR clause learning as a preprocessing stage, PR clauses can be found efficiently without sacrificing the robustness of modern CDCL solvers. On a large portion of SAT competition benchmarks we found that preprocessing with PRELEARN improves solver performance. In addition, there were several satisfiable and unsatisfiable formulas that could only be solved after preprocessing with PRELEARN. PRELEARN supports proof logging, giving a high level of confidence in the results.

1 Introduction

Conflict-driven clause learning (CDCL) [27] is the standard paradigm for solving the satisfiability problem (SAT) in propositional logic. CDCL solvers learn clauses implied through *resolution* inferences. Additionally, all competitive CDCL solvers use pre- and in-processing techniques captured by the *resolution asymmetric tautology* (RAT) proof system [21]. As examples, the well-studied pigeonhole and mutilated chessboard problems are challenging benchmarks with exponentially-sized resolution proofs [1, 12]. It is possible to construct small hand-crafted proofs for the pigeonhole problem using *extended resolution* (ER) [8], a proof system that allows the introduction of new variables [32]. ER can be expressed in RAT but has proved difficult to automate due to the large search space. Even with modern inprocessing techniques, many CDCL solvers struggle on these seemingly simple problems. The *propagation redundant* (PR) proof system allows short proofs for these problems [14, 15], and unlike in ER, no new variables are required. This makes PR an attractive candidate for automation.

At a high level, CDCL solvers make decisions that typically yield an unsatisfiable branch of a problem. The clause that prunes the unsatisfiable branch from the search space is learned, and the solver continues by searching another branch. PR extends this

[*]The authors are supported by the NSF under grant CCF-2108521.

paradigm by allowing more aggressive pruning. In the PR proof system a branch can be pruned as long as there exists another branch that is at least as satisfiable. As an example, consider the mutilated chessboard. The mutilated chessboard problem involves finding a covering of 2×1 dominos on an $n \times n$ chessboard with two opposite corners removed (see Section 5.4). Given two horizontally oriented dominoes covering a 2×2 square, two vertically oriented dominos could cover the same 2×2 square. For any solution that uses the dominos in the horizontal orientation, replacing them with the dominos in the vertical orientation would also be a solution. The second orientation is as satisfiable as the first, and so the first can be pruned from the search space. Even though the number of possible solutions may be reduced, the pruning is satisfiability preserving. This is a powerful form of reasoning that can efficiently remove many symmetries from the mutilated chessboard, making the problem much easier to solve [15].

The *satisfaction-driven clause learning* (SDCL) solver SADICAL [16] incorporates PR clause learning into the CDCL loop. SADICAL implements hand-crafted decision heuristics that exploit the canonical structure of the pigeonhole and mutilated chessboard problems to find short proofs. However, SADICAL's performance deteriorates under slight variations to the problems including different constraint encodings [7]. The heuristics were developed from a few well-understood problems and do not generalize to other problem classes. Further, the heuristics for PR clause learning are likely ill-suited for CDCL, making the solver less robust.

In this paper, we present PRELEARN, a preprocessing technique for learning PR clauses. PRELEARN alternates between finding and learning PR clauses. We develop multiple heuristics for finding PR clauses and multiple configurations for learning some subset of the found PR clauses. As PR clauses are learned we use failed literal probing [11] to find unit clauses implied by the formula. The preprocessing is made efficient by taking advantage of the inner/outer solver framework in SADICAL. The learned PR clauses are added to the original formula, aggressively pruning the search space in an effort to guide CDCL solvers to short proofs. With this method PR clauses can be learned without altering the complex heuristics that make CDCL solvers robust. PRELEARN focuses on finding short PR clauses and failed literals to effectively reduce the search space. This is done with general heuristics that work across a wide range of problems.

Most SAT solvers support logging proofs of unsatisfiability for independent checking [17, 20, 33]. This has proved valuable for verifying solutions independent of a (potentially buggy) solver. Modern SAT solvers log proofs in the DRAT proof system (RAT [21] with deletions). DRAT captures all widely used pre- and in-processing techniques including bounded variable elimination [10], bounded variable addition [26], and extended learning [4,32]. DRAT can express the common symmetry-breaking techniques, but it is complicated [13]. PR can compactly express some symmetry-breaking techniques [14, 15], yielding short proofs that can be checked by the proof checker DPR-TRIM [16]. PR gives a framework for strong symmetry-breaking inferences and also maintains the highly desirable ability to independently verify proofs.

The contributions of this paper include: (1) giving a high-level algorithm for extracting PR clauses, (2) implementing several heuristics for finding and learning PR clauses, (3) evaluating the effectiveness of different heuristic configurations, and (4) assessing the impact of PRELEARN on solver performance. PRELEARN improves the performance of the CDCL solver KISSAT on a quarter of the satisfiable and unsatisfiable competition benchmarks we considered. The improvement is significant for a number of instances that can only be solved by KISSAT after preprocessing. Most of them come from hard combinatorial problems with small formulas. In addition, PRELEARN directly produces refutation proofs for the mutilated chessboard problem containing only unit and binary PR clauses.

2 Preliminaries

We consider propositional formulas in *conjunctive normal form* (CNF). A CNF formula ψ is a conjunction of *clauses* where each clause is a disjunction of *literals*. A literal l is either a variable x (positive literal) or a negated variable $\overline{x}$ (negative literal). For a set of literals L the formula $\psi(L)$ is the clauses $\{C \in \psi \mid C \cap L \neq \emptyset\}$.

An *assignment* is a mapping from variables to truth values 1 (*true*) and 0 (*false*). An assignment is *total* if it assigns every variable to a value, and *partial* if it assigns a subset of variables to values. The set of variables occurring in a formula, assignment, or clause is given by $\mathsf{var}(\psi)$, $\mathsf{var}(\alpha)$, or $\mathsf{var}(C)$. For a literal l, $\mathsf{var}(l)$ is a variable.

An assignment α *satisfies* a positive (negative) literal l if α maps $\mathsf{var}(l)$ to true (α maps $\mathsf{var}(l)$ to false, respectively), and *falsifies* it if α maps $\mathsf{var}(l)$ to false (α maps $\mathsf{var}(l)$ to true, respectively). We write a finite partial assignment as the set of literals it satisfies. An assignment satisfies a clause if the clause contains a literal satisfied by the assignment. An assignment satisfies a formula if every clause in the formula is satisfied by the assignment. A formula is *satisfiable* if there exists a satisfying assignment, and *unsatisfiable* otherwise. Two formula are *logically equivalent* if they share the same set of satisfying assignments. Two formulas are *satisfiability equivalent* if they are either both satisfiable or both unsatisfiable.

If an assignment α satisfies a clause C we define $C|_\alpha = \top$, otherwise $C|_\alpha$ represents the clause C with the literals falsified by α removed. The empty clause is denoted by $\bot$. The formula ψ reduced by an assignment α is given by $\psi|_\alpha = \{C|_\alpha \mid C \in \psi \text{ and } C|_\alpha \neq \top\}$. Given an assignment $\alpha = l_1 \dots l_n$, $C = (\bar{l}_1 \vee \dots \vee \bar{l}_n)$ is the clause that *blocks* α. The assignment *blocked* by a clause is the negation of the literals in the clause. The literals touched by an assignment is defined by $\mathsf{touched}_\alpha(C) = \{l \mid l \in C \text{ and } \mathsf{var}(l) \in \mathsf{var}(\alpha)\}$ for a clause. For a formula ψ, $\mathsf{touched}_\alpha(\psi)$ is the union of touched variables for each clause in ψ. A *unit* is a clause containing a single literal. The *unit clause rule* takes the assignment α of all units in a formula ψ and generates $\psi|_\alpha$. Iteratively applying the unit clause rule until fixpoint is referred to as *unit propagation*. In cases where unit propagation yields $\bot$ we say it derived a *conflict*. A formula ψ *implies* a formula ψ', denoted $\psi \models \psi'$, if every assignment satisfying ψ satisfies ψ'. By $\psi \vdash_1 \psi'$ we denote that for every clause $C \in \psi'$, applying unit propagation to the assignment blocked by C in ψ derives a conflict. If unit propagation derives a conflict on the formula $\psi \cup \{\{l\}\}$, we say l is a *failed literal* and the unit $\bar{l}$ is logically implied by the formula. Failed literal probing [11] is the process of successively assigning literals to check if units are implied by the formula. In its simplest form, probing involves assigning a literal l and learning the unit $\bar{l}$ if unit propagation derives a conflict, otherwise l is unassigned and the next literal is checked.

To evaluate the satisfiability of a formula, a CDCL solver [27] iteratively performs the following operations: First, the solver performs unit propagation, then tests for a conflict. Unit propagation is made efficient with two-literal watch pointers [28]. If there is no conflict and all variables are assigned, the formula is satisfiable. Otherwise, the solver chooses an unassigned variable through a variable decision heuristic [6, 25], assigns a truth value to it, and performs unit propagation. If, however, there is a conflict, the solver performs conflict analysis potentially learning a short clause. In case this clause is the empty clause, the formula is unsatisfiable.

3 The PR Proof System

A clause C is *redundant* w.r.t. a formula ψ if ψ and $\psi \cup \{C\}$ are *satisfiability equivalent*. The clause sequence $\psi, C_1, C_2, \ldots, C_n$ is a clausal proof of C_n if each clause C_i $(1 \leq i \leq n)$ is redundant w.r.t. $\psi \cup \{C_1, C_2, \ldots, C_{i-1}\}$. The proof is a refutation of ψ if C_n is $\bot$. Clausal proof systems may also allow deletion. In a refutation proof clauses can be deleted freely because the deletion cannot make a formula less satisfiable.

Clausal proof systems are distinguished by the kinds of redundant clauses they allow to be added. The standard SAT solving paradigm CDCL learns clauses implied through *resolution*. These clauses are logically implied by the formula, and fall under the *reverse unit propagation* (RUP) proof system. The *Resolution Asymmetric Tautology* (RAT) proof system generalizes RUP. All commonly used inprocessing techniques emit DRAT proofs. The *propagation redundant* (PR) proof system generalizes RAT by allowing the pruning of branches *without loss of satisfaction*.

Let C be a clause in the formula ψ and α the assignment blocked by C. Then C is PR w.r.t. ψ if and only if there exists an assignment ω such that $\psi|_\alpha \vdash_1 \psi|_\omega$ and ω satisfies C. Intuitively, this allows inferences that block a partial assignment α as long as another assignment ω is as satisfiable. This means every assignment containing α that satisfies ψ can be transformed to an assignment containing ω that satisfies ψ.

Clausal proofs systems must be checkable in polynomial time to be useful in practice. RUP and RAT are efficiently checkable due to unit propagation. In general, determining if a clause is PR is an NP-complete problem [18]. However, a PR proof is checkable in polynomial time if the witness assignments ω are included. A clausal proof with witnesses will look like $\psi, (C_1, \omega_1), (C_2, \omega_2), \ldots, (C_n, \omega_n)$. The proof checker DPR-TRIM can efficiently check PR proofs that include witnesses. Further, DPR-TRIM can emit proofs in the LPR format. They can be validated by the formally-verified checker CAKE-LPR [31], which was used to validate results in recent SAT competitions.

4 Pruning Predicates and SADICAL

Determining if a clause is PR is NP-complete and can naturally be formulated in SAT. Given a clause C and formula ψ, a *pruning predicate* is a formula such that if it is satisfiable, the clause C is redundant w.r.t. ψ. SADICAL uses two pruning predicates to determine if a clause is PR: *positive reduct* and *filtered positive reduct*. If either predicate is satisfiable, the satisfying assignment serves as the witness showing the clause is PR.

Given a formula ψ and assignment α, the *positive reduct* is the formula $G \wedge C$ where C is the clause that blocks α and $G = \{\mathsf{touched}_\alpha(D) \mid D \in \psi \text{ and } D|_\alpha = \top\}$. If the positive reduct is satisfiable, the clause C is PR w.r.t. ψ. The positive reduct is satisfiable iff the clause blocked by α is a *set-blocked* clause [23].

Given a formula ψ and assignment α, the *filtered positive reduct* is the formula $G \wedge C$ where C is the clause that blocks α and $G = \{\mathsf{touched}_\alpha(D) \mid D \in \psi \text{ and } D|_\alpha \nvdash_1 \mathsf{touched}_\alpha(D)\}$. If the filtered positive reduct is satisfiable, the clause C is PR w.r.t. ψ. The filtered positive reduct is a subset of the positive reduct and is satisfiable iff the clause blocked by α is a *set-propagation redundant* clause [14]. Example 1 shows a formula for which the positive and filtered positive reducts are different, and only the filtered positive reduct is satisfiable.

Example 1. Given the formula $(x_1 \vee x_2) \wedge (\overline{x}_1 \vee x_2)$, the positive reduct with $\alpha = x_1$ is $(x_1) \wedge (\overline{x}_1)$, which is unsatisfiable. The clause (x_1) can be filtered, giving the filtered positive reduct $(\overline{x}_1)$, which is satisfiable.

SADICAL [16] uses satisfaction-driven clause learning (SDCL) that extends CDCL by learning PR clauses [18] based on (filtered) positive reducts. SADICAL uses an inner/outer solver framework. The outer solver attempts to solve the SAT problem with SDCL. SDCL diverges from the basic CDCL loop when unit propagation after a decision does not derive a conflict. In this case a reduct is generated using the current assignment, and the inner solver attempts to solve the reduct using CDCL. If the reduct is satisfiable, the PR clause blocking the current assignment is learned, and the SDCL loop continues. The PR clause can be simplified by removing all non-decision variables from the assignment. SADICAL emits PR proofs by logging the satisfying assignment of the reduct as the witness, and these proofs are verified with DPR-TRIM. The key to SADICAL finding good PR clauses leading to short proofs is the decision heuristic, because variable selection builds the candidate PR clauses. Hand-crafted decision heuristics enable SADICAL to find short proofs on pigeonhole and mutilated chessboard problems. However, these heuristics differ significantly from the score-based heuristics in most CDCL solvers. Our experiences with SaCiDaL suggest that improving the heuristics for SDCL reduces the performance of CDCL and the other way around. This may explain why SADICAL performs worse than standard CDCL solvers on the majority of the SAT competition benchmarks. While SADICAL integrates finding PR clauses of arbitrary size in the main search loop, our tool focuses on learning short PR clauses as a preprocessing step. This allows us to develop good heuristics for PR learning without compromising the main search loop.

5 Extracting PR Clauses

The goal of PRELEARN is to find useful PR clauses that improve the performance of CDCL solvers on both satisfiable and unsatisfiable instances. Figure 1 shows how a SAT problem is solved using PRELEARN. For some preset time limit, PR clauses are found and then added to the original formula. Interleaved in this process is failed literal probing to check if unit clauses can be learned. When the preprocessing stage ends, the new formula that includes learned PR clauses is solved by a CDCL solver. If the

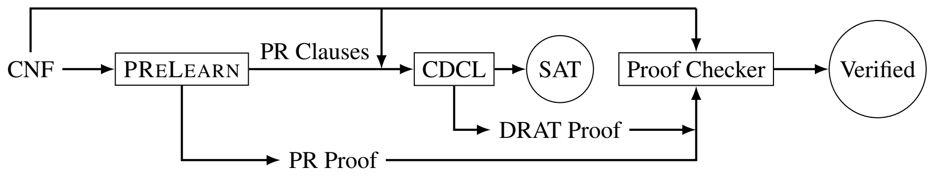


Fig. 1. Solving a formula with PRELEARN and a CDCL solver.

formula is satisfiable, the solver will produce a satisfying assignment. If the formula is unsatisfiable, a refutation proof of the original formula can be computed by combining the satisfaction preserving proof from PRELEARN and the refutation proof emitted by the CDCL solver. The complete proof can be verified with DPR-TRIM.

PRELEARN alternates between finding PR clauses and learning PR clauses. Candidate PR clauses are found by iterating over each variable in the formula, and for each variable constructing clauses that include that variable. To determine if a clause is PR, the positive reduct generated by that clause is solved. It can be costly to generate and solve many positive reducts, so heuristics are used to find candidate clauses that are more likely to be PR. It is possible to find multiple PR clauses that conflict with each other. PR clauses are conflicting if adding one of the PR clauses to the formula makes the other no longer PR. Learning PR clauses involves selecting PR clauses that are non-conflicting. The selection may maximize the number of PR clauses learned or optimize for some other metric. Adding PR clauses and units derived from probing may cause new clauses to become PR, so the entire process is iterated multiple times.

5.1 Finding PR Clauses

PR clauses are found by constructing a set of candidate clauses and solving the positive reduct generated by each clause. In SADICAL the candidates are the clauses blocking the partial assignment of the solver after each decision in the SDCL loop that does not derive a conflict. In effect, candidates are constructed using the solver's variable decision heuristic. We take a more general approach, constructing sets of candidates for each variable based on unit propagation and the partial assignment's neighbors.

For a variable x, $\mathsf{neighbors}(x)$ denotes the set of variables occurring in clauses containing literal x or $\overline{x}$, excluding variable x. For a partial assignment α, $\mathsf{neighbors}(\alpha)$ denotes $\bigcup_{x \in \mathsf{var}(\alpha)} \mathsf{neighbors}(x) \setminus \mathsf{var}(\alpha)$. Candidate clauses for a literal l are generated in the following way:

- Let α be the partial assignment found by unit propagation starting with the assignment that makes l true.
- Generate the candidate PR clauses $\{(\bar{l} \vee y), (\bar{l} \vee \overline{y}) \mid y \in \mathsf{neighbors}(\alpha)\}$.

Example 2 shows how candidate binary clauses are constructed using both polarities of an initial variable x. In Example 3 the depth is expanded to reach more variables and create larger sets of candidate clauses. The depth parameter is used in Section 5.4.

Example 2. Consider the following formula: $(x_1 \vee \overline{x}_2) \wedge (\overline{x}_1 \vee x_3) \wedge (x_1 \vee x_4 \vee x_5) \wedge (x_2 \vee x_6 \vee x_7) \wedge (x_3 \vee x_7 \vee x_8) \wedge (x_8 \vee x_9)$,
Case 1: We start with $\mathsf{var}(x_1) = 1$ and perform unit propagation resulting in $\alpha = \{x_1 x_3\}$. Observe that $\mathsf{neighbors}(\alpha) = \{x_2, x_4, x_5, x_7, x_8\}$. The generated candidate clauses are $(\overline{x}_1 \vee x_2), (\overline{x}_1 \vee \overline{x}_2), (\overline{x}_1 \vee x_4), (\overline{x}_1 \vee \overline{x}_4), \ldots, (\overline{x}_1 \vee x_8), (\overline{x}_1 \vee \overline{x}_8)$.
Case 2: We start with $\mathsf{var}(x_1) = 0$ and perform unit propagation resulting in $\alpha = \{\overline{x}_1 \overline{x}_2\}$. Observe that $\mathsf{neighbors}(\alpha) = \{x_3, x_4, x_5, x_6, x_7\}$. The generated candidate clauses are $(x_1 \vee x_3), (x_1 \vee \overline{x}_3), (x_1 \vee x_4), (x_1 \vee \overline{x}_4), \ldots, (x_1 \vee x_7), (x_1 \vee \overline{x}_7)$.

Example 3. Take the formula from Example 2 and assignment of $\mathsf{var}(x_1) = 1$ as in case 1. The set of candidate clauses can be expanded by also considering the unassigned neighbors of the variables in $\mathsf{neighbors}(\alpha)$. For example, $\mathsf{neighbors}(x_8) = \{x_3, x_7, x_9\}$, of which x_9 is new and unassigned. This adds $(\overline{x}_1 \vee x_9)$ and $(\overline{x}_1 \vee \overline{x}_9)$ to the set of candidate clauses. This can be iterated by including neighbors of new unassigned variables from the prior step.

We consider both polarities when constructing candidates for a variable. After all candidates for a variable are constructed, the positive reduct for each candidate is generated and solved in order. Note that propagated literals appearing in the partial assignment do not appear in the PR clause. The satisfying assignment is stored as the witness and the PR clause may be learned immediately depending on the learning configuration.

This process is naturally extended to ternary clauses. The binary candidates are generated, and for each candidate $(x \vee y)$, x and y are assigned to false in the first step. The variables $z \in \mathsf{neighbors}(\alpha)$ yield clauses $(x \vee y \vee z)$ and $(x \vee y \vee \overline{z})$. This approach can generate many candidate ternary clauses depending on the connectivity of the formula since each candidate binary clause is expanded. A filtering operation would be useful to avoid the blow-up in number of candidates. There are likely diminishing returns when searching for larger PR clauses because (1) there are more possible candidates, (2) the positive reducts are likely larger, and (3) each clause blocks less of the search space. We consider only unit and binary candidate clauses in our main evaluation.

Ideally, we should construct candidate clauses that are likely PR to reduce the number of failed reducts generated. Note, the (filtered) positive reduct can only be satisfiable if given the partial assignment there exists a reduced, satisfied clause. By focusing on neighbors, we guarantee that such a clause exists. The *reduced* heuristic in SaDiCaL finds variables in all reduced but unsatisfied clauses. The idea behind this heuristic is to direct the assignment towards conditional autarkies that imply a satisfiable positive reduct [18]. The neighbors approach generalizes this to variables in all reduced clauses whether or not they are unsatisfied. A comparison can be found in our repository.

5.2 Learning PR Clauses

Given multiple clauses that are PR w.r.t. the same formula, it is possible that some of the clauses conflict with each other and cannot be learned simultaneously. Example 4 shows how learning one PR clause may invalidate the witness of another PR clause. It may be that a different witness exists, but finding it requires regenerating the positive reduct to include the learned PR clause and solving it. The simplest way to avoid conflicting PR clause is to learn PR clauses as they are found. When a reduct is satisfiable, the PR clauses is added to the formula and logged with its witness in the proof. Then subsequent reducts will be generated from the formula including all added PR clauses. Therefore, a satisfiable reduct ensures a PR clause can be learned.

Alternatively, clauses can be found in batches, then a subset of nonconflicting clauses can be learned. The set of conflicts between PR clauses can be computed in polynomial time. For each pair of PR clauses C and D, if the assignment that generated the pruning predicate for D touches C and C is not satisfied by the witness of D, then C conflicts with D. In some cases reordering the two PR clauses may avoid a conflict. In Example 4 learning the second clause would not affect the validity of the first clauses' witness. Once the conflicts are known, clauses can be learned based on some heuristic ordering. Batch learning configurations are discussed more in the following section.

Example 4. Assume the following clause witness pairs are valid in a formula ψ: $\{(x_1 \lor x_2 \lor x_3), x_1\overline{x}_2\overline{x}_3\}$, and $\{(x_1 \lor \overline{x}_2 \lor x_4), \overline{x}_1\overline{x}_2x_4\}$. The first clause conflicts with the second. If the first clause is added to ψ, the clause $(x_1 \lor x_2)$ would be in the positive reduct for the second clause, but it is not satisfied by the witness of the second clause.

5.3 Additional Configurations

The sections above describe the PRELEARN configuration used in the main evaluation, i.e., finding candidate PR clauses with the neighbors heuristic and learning clauses instantly as the positive reducts are solved. In this section we present several additional configurations. The time-constrained reader may skip ahead to Section 5.4 for the presentation of our main results.

In batch learning a set of PR clauses are found in batches then learned. Learning as many nonconflicting clauses as possible coincides with the maximum independent set problem. This problem is NP-Hard. We approximate the solution by adding the clause causing the fewest conflicts with unblocked clauses. When a clause is added, the clauses it blocks are removed from the batch and conflict counts are recalculated Alternatively, clauses can be added in a random order. Random ordering requires less computation at the cost of potentially fewer learned PR clauses.

The neighbors heuristic for constructing candidate clauses can be modified to include a depth parameter. $\mathsf{neighbors}(i)$ indicates the number of iterations expanding the variables. For example, $\mathsf{neighbors}(2)$ expands on the variables in $\mathsf{neighbors}(1)$, seen in Example 3. We also implement the reduced heuristic, shown in Example 5. Detailed evaluations and comparisons can be found in our repository. In general, we found that the additional configurations did not improve on our main configuration. More work needs to be done to determine when and how to apply these additional configurations.

Example 5. Given the set of clauses $(x_1 \lor x_2 \lor x_3) \land (\overline{x}_1 \lor x_3 \lor x_4) \land (x_3 \lor x_5)$, and initial assignment $\alpha = x_1$, only the second clause is reduced and not satisfied, giving $\mathsf{reduced}(\alpha) = \{x_3, x_4\}$ and candidate clauses $(\overline{x}_1 \lor x_3)$, $(\overline{x}_1 \lor x_4)$, $(\overline{x}_1 \lor \overline{x}_3)$, $(\overline{x}_1 \lor \overline{x}_4)$.

5.4 Implementation

PRELEARN was implemented using the inner/outer-solver framework in SADICAL. The inner solver acts the same as in SADICAL, solving pruning predicates using CDCL.

The outer solver is not used for SDCL, but the SDCL data-structures are used to find and learn PR clauses. The outer solver is initialized with the original formula and maintains the list of variables, clauses, and watch pointers. By default, the outer solver has no variables assigned other than learned units. When finding candidates, the variables in the partial clause are assigned in the outer solver. Unit propagation makes it possible to find all reduced clauses in the formula with a single pass. This is necessary for constructing the positive reduct. After a candidate clause has been assigned and the positive reduct solved, the variables are unassigned. This returns the outer solver to the top-level before examining the next candidate. When a PR clause is learned, it is added to the formula along with its watch pointers. Additionally, failed literals are found if assigning a variable at the top-level causes a conflict through unit propagation. The negation of a failed literal is a unit that can be added to the formula.

In a single iteration each variable in the formula is processed in a breadth-first search (BFS) starting from the first variable in the numbering. When a variable is encountered it is first checked whether either assignment of the variable is a failed literal or a unit PR clause. If not, binary candidates are generated based on the selected heuristic and PR clauses are learned based on the learning configuration. Variables are added to the frontier of the BFS as they are encountered during candidate clause generation, but they are not repeated. Optionally, after all variables have been encountered the BFS restarts, now constructing ternary candidates. The repetition continues to the desired clause length. Then another iteration begins again with binary clauses. Running PRELEARN multiple times is important because adding PR clauses in one iteration may allow additional clauses to be added in the next.

6 Mutilated Chessboard

The *mutilated chessboard* is an $n \times n$ grid of alternating black and white squares with two opposite corners removed. The problem is whether or not the the board can be covered with 2×1 dominoes. This can be encoded in CNF by using variables to represent

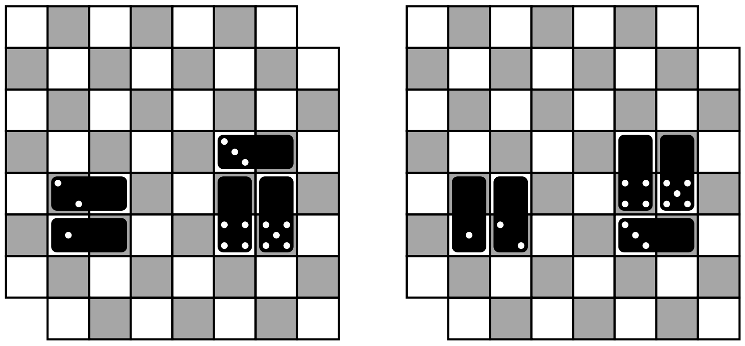


Fig. 2. Occurrences of two horizontal dominoes may be replaced by two vertical dominos in a solution. Similarly, occurrences of a horizontal domino atop two vertical dominos can be replaced by shifting the horizontal domino down.

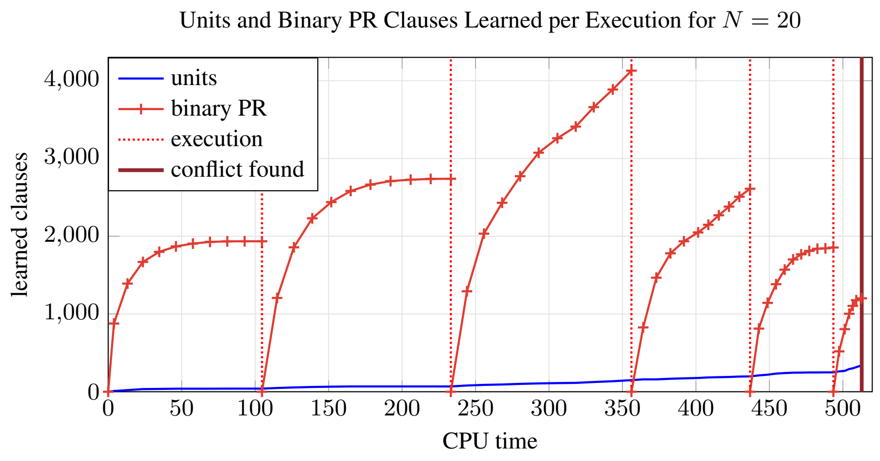


Fig. 3. Unit and binary PR clauses learned each execution (red-dotted line) until a contradiction was found. Markers on binary PR lines represent an iteration within an execution.

domino placements on the board. At-most-one constraints (using the pairwise encodings) say only one domino can cover each square, and at-least-one constraints (using a disjunction) say some domino must cover each square.

In recent SAT competitions, no proof-generating SAT solver could deal with instances larger than $N = 18$. In ongoing work, we found refutation proofs that contain only units and binary PR clauses for some boards of size $N \leq 30$. PRELEARN can be modified to automatically find proofs of this type. Running iterations of PRELEARN until *saturation*, meaning no new binary PR clauses or units can be found, yields some set of units and binary PR clauses. Removing the binary PR clauses from the formula and rerunning PRELEARN will yield additional units and a new set of binary PR clauses. Repeating the process of removing binary PR clauses and keeping units will eventually derive the empty clause for this problem. Figure 3 gives detailed values for $N = 20$. Within each execution (red dotted lines) there are at most 10 iterations (red tick markers), and each iteration learns some set of binary PR clauses (red). Some executions saturate binary PR clauses before the tenth iteration and exit early. At the end of each execution the binary PR clauses are deleted, but the units (blue) are kept for the following execution. A complete DPR proof (PR with deletion) can be constructed by adding deletion information for the binary PR clauses removed between each execution when concatenating the PRELEARN proofs. The approach works for mutilated chess because in each execution there are many binary PR clauses that can be learned and will lead to units, but they are mutually exclusive and cannot be learned simultaneously. Further, adding units allows new binary PR clauses to be learned in following executions.

Table 1 shows the statistics for PRELEARN. Achieving these results required some modifications to the configuration of PRELEARN. First, notice in Figure 2 the PR clauses that can be learned involve blocking one domino orientation that can be replaced by a symmetric orientation. To optimize for these types of PR clauses, we only

Table 1. Statistics running multiple executions of PRELEARN on the mutilated chessboard problem with the configurations described below. Total units includes failed literals and learned PR units. The average units and average binary PR clauses learned during each execution (Exe.) are shown as well.

N	Time (s)	# Exe.	Avg. (s)	Total Units	Total Bin.	Avg. Units	Avg. Bin.
8	0.14	1	0.14	30	164	30.00	164.00
12	4.94	1	4.94	103	1,045	103.00	1,045.00
16	62.47	2	31.23	195	3,988	97.50	1,994.00
20	513.12	6	85.52	339	1,4470	56.50	2,411.67
24	4,941.38	26	190.05	512	64,038	19.69	2,463.00

constructed candidates where the first literal was negative. The neighbors heuristic had to be increased to a depth of 6, meaning more candidates were generated for each variable. Intuitively, the proof is constructed by adding binary PR clauses in order to find negative units (dominos that cannot be placed) around the borders of the board. Following iterations build more units inwards, until a point is reached where units cover almost the entire board. This forces an impossible domino placement leading to a contradiction. Complete proofs using only units and binary PR clauses were found for boards up to size $N = 24$ within $5{,}000$ seconds. We verified all proofs using DPR-TRIM. The mutilated chessboard has a high degree of symmetry and structure, making it suitable for this approach. For most problems it is not expected that multiple executions while keeping learned units will find new PR clauses.

Experiments were done with several configurations (see Section 5.3) to find the best results. We found that increasing the depth of neighbors was necessary for larger boards including $N = 24$. Increasing the depth allows more binary PR clauses to be found, at the cost of generating more reducts. This is necessary to find units. The reduced heuristic (a subset of neighbors) did not yield complete proofs. We also tried incrementing the depth after each execution starting with 1 and reseting at 9. In this approach, the execution times for depth greater than 6 were larger but did not yield more unit clauses on average. We attempted batch learning on every 500 found clauses using either random or the sorted heuristic. In each batch many of the 500 PR clauses blocked each other because many conflicting PR clauses can be found on a small set of variables in mutilated chess. The PR clauses that were blocked would be found again in following iterations, leading to more reducts generated and solved. This caused much longer execution times. Adding PR clauses instantly is a good configuration for reducing execution time when there are many conflicting clauses. However, for some less symmetric problems it may be worth the tradeoff to learn the clauses in batches, because learning a few bad PR clauses may disrupt the subsequent iterations.

7 SAT Competition Benchmarks

We evaluated PRELEARN on previous SAT competition formulas. Formulas from the '13, '15, '16, '19, '20, and '21 SAT competitions' main tracks were grouped by size. **0-10k** contains the 323 formulas with less than $10{,}000$ clauses and **10k-50k** contains

Table 2. Fraction of benchmarks where PR clauses were learned, average runtime of PReLearn, generated positive reducts and satisfiable positive reducts (PR clauses learned), and number of failed literals found.

Set	Benches	Avg. (s)	Generated Reducts	Sat. Reducts	% Sat.	Failed Lits
0-10k	221/323	22.36	104,850,011	548,417	0.52%	3,416
10k-50k	237/348	71.08	163,014,068	789,281	0.48%	6,290

the 348 formulas with between 10,000 and 50,000 clauses. In general, short PR proofs have been found for hard combinatorial problems typically having few clauses (0-10k). These include the pigeonhole and mutilated chessboard problems, some of which appear in 0-10k benchmarks. The PR clauses that can be derived for these formulas are intuitive and almost always beneficial to solvers. Less is known about the impact of PR clauses on larger formulas, motivating our separation of test sets by size. The repository containing the preprocessing tool, experiment configurations, and experiment data can be found at https://github.com/jreeves3/PReLearn.

We ran our experiments on StarExec [30]. The specs for the compute nodes can be found online.[1] The compute nodes that ran our experiments were Intel Xeon E5 cores with 2.4 GHz, and all experiments ran with 64 GB of memory and a 5,000 second timeout. We run PReLearn for 50 iterations over 100 seconds, exiting early if no new PR clauses were found in an iteration.

PReLearn was executed as a stand-alone program, producing a derivation proof and a modified CNF. For experiments, the CDCL solver Kissat [5] was called once on the original formula and once on the modified CNF. Kissat was selected because of its high-rankings in previous SAT competitions, but we expect the results to generalize to other CDCL SAT solvers.

Derivation proofs from PReLearn were verified in all solved instances using the independent proof checker DPR-trim using a forward check. This can be extended to complete proofs in the following way. In the unsatisfiable case the proof for the learned PR clauses is concatenated to the proof traced by Kissat, and the complete proof is verified against the original formula. In the satisfiable case the partial proof for the learned PR clauses is verified using a forward check in DPR-trim, and the satisfying assignment found by Kissat is verified by the StarExec post-processing tool. Due to resource limitations, we verified a subset of complete proofs in DPR-trim. This is more costly because it involves running Kissat with proof logging, then running DPR-trim on the complete proof.

Table 2 shows the cumulative statistics for running PReLearn on the benchmark sets. Note the number of satisfiable reducts is the number of learned PR clauses, because PR clauses are learned immediately after the reduct is solved. These include both unit and binary PR clauses. A very small percentage of generated reducts is satisfiable, and subsequently learned. This is less important for small formulas when reducts can be computed quickly and there are fewer candidates to consider. However, for the 10k-50k formulas the average runtime more than triples but the number of generated reducts

[1] https://starexec.org/starexec/public/about.jsp

Table 3. Number of total solved instances and exclusive solved instances running KISSAT with and without PRELEARN. Number of improved instances running KISSAT with PRELEARN. PRELEARN execution times were included in total execution times.

	0-10k SAT	0-10k UNSAT	10k-50k SAT	10k-50k UNSAT
Total w/ PRELEARN	84	149	143	89
Total w/o PRELEARN	80	141	143	91
Exclusively w/ PRELEARN	4	10	4	1
Exclusively w/o PRELEARN	0	2	4	3
Improved w/ PRELEARN	20	44	25	13

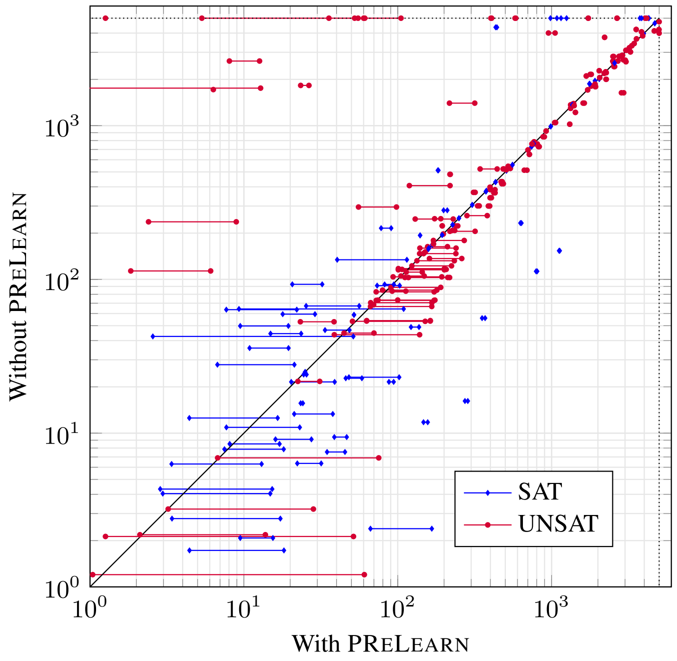
less than doubles. PR clauses are found in about two thirds of the formulas, showing our approach generalizes beyond the canonical problems for which we knew PR clauses existed. Expanding the exploration and increasing the time limit did not help to find PR clauses in the remaining one third.

Table 3 gives a high-level picture of PRELEARN's impact on KISSAT. PRELEARN significantly improves performance on 0-10k SAT and UNSAT benchmarks. These contain the hard combinatorial problems including pigeonhole that PRELEARN was expected to perform well on. There were 4 additional SAT formulas solved with PRELEARN that KISSAT alone could not solve. This shows that PRELEARN impacts not only hard unsatisfiable problems but satifsiable problems as well. On the other hand, the addition of PR clauses makes some problems more difficult. This is clear with the 10k-50k results, where 5 benchmarks are solved exclusively with PRELEARN and 7 are solved exclusively without. Additionally, PRELEARN improved KISSAT's performance on 102 of 671 or approx. 15% of benchmarks. This is a large portion of benchmarks, both SAT and UNSAT, for which PRELEARN is helpful.

Figure 4 gives a more detailed picture on the impact of PRELEARN per benchmark. In the scatter plot the left-hand end of each line indicates the KISSAT execution time, while the length of the line indicates the PRELEARN execution time, and so the right-hand end gives the total time for PRELEARN plus KISSAT. Lines that cross the diagonal indicate that the preprocessing improved KISSAT's performance but ran for longer than the improvement. PRELEARN improved performance for points above the diagonal. Points on the dotted-lines (timeout) are solved by one configuration and not the other.

The top plot gives the results for the 0-10k formulas, with many points on the top timeout line as expected. These are the hard combinatorial problems that can only be solved with PRELEARN. In general, the unsatisfiable formulas benefit more than the satisfiable formulas. PR clauses can reduce the number of solutions in a formula and this may explain the negative impact on many satisfiable formulas. However, there are still some satisfiable formulas that are only solved with PRELEARN.

In the bottom plot, formulas that take a long time to solve (above the diagonal in the upper right-hand corner) are helped more by PRELEARN. In the bottom half of the plot, many lines cross the diagonal meaning the addition of PR clauses provided a negligible benefit. For this set there are more satisfiable formulas for which PRELEARN is helpful.



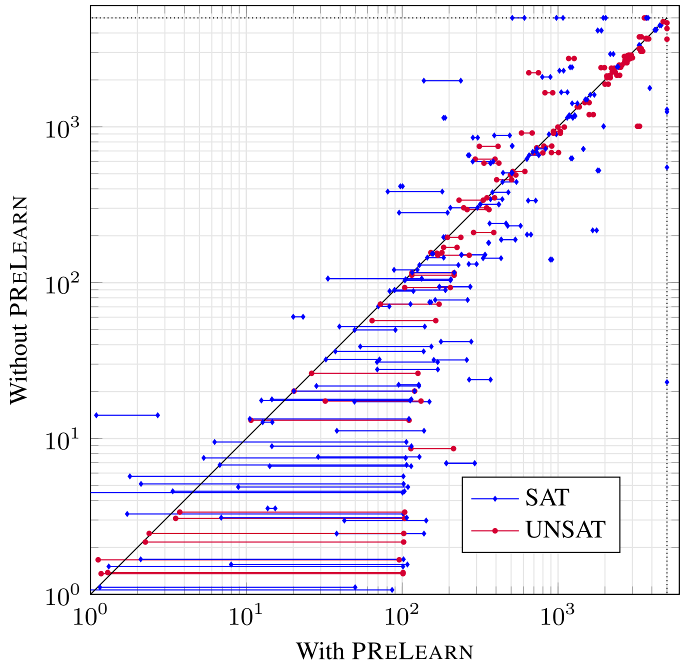


Fig. 4. Execution times w/ and w/o PRELEARN on 0-10k (top) and 10k-50k (bottom) benchmarks. The left-hand point of each segment shows the time for the SAT solver alone; the right-hand point indicates the combined time for preprocessing and solving.

Table 4. Some formulas solved by KISSAT exclusively *with* PRELEARN (top) and some formulas solved exclusively *without* PRELEARN (bottom). (*) solved without KISSAT. Clauses include PR clauses and failed literals learned.

Set	Value	With	Without	Clauses	Formula	Year
0-10k	UNSAT	1.26	–	2,033	ph12*	2013
0-10k	UNSAT	35.69	–	20,179	Pb-chnl15-16_c18*	2019
0-10k	UNSAT	105.01	–	46,759	Pb-chnl20-21_c18	2019
0-10k	UNSAT	59.99	–	1,633	randomG-Mix-n17-d05	2021
0-10k	UNSAT	61.08	–	1,472	randomG-n17-d05	2021
0-10k	UNSAT	407.51	–	1,640	randomG-n18-d05	2021
0-10k	UNSAT	584.95	–	1,706	randomG-Mix-n18-d05	2021
0-10k	SAT	1,082.62	–	9,650	fsf-300-354-2-2-3-2.23.opt	2013
0-10k	SAT	1,250.82	–	10,058	fsf-300-354-2-2-3-2.46.opt	2013
10k-50k	SAT	1,076.34	–	804	sp5-26-19-bin-stri-flat-noid	2021
10k-50k	SAT	608.48	–	901	sp5-26-19-una-nons-tree-noid	2021
10k-50k	SAT	–	22.99	254	Ptn-7824-b13	2016
10k-50k	SAT	–	549.27	133	Ptn-7824-b09	2016
10k-50k	SAT	–	1,246.42	39	Ptn-7824-b02	2016
10k-50k	SAT	–	1,290.49	121	Ptn-7824-b08	2016
10k-50k	UNSAT	–	3,650.21	31,860	rphp4_110_shuffled	2016
10k-50k	UNSAT	–	4,273.88	31,531	rphp4_115_shuffled	2016

The results in Figure 4 are encouraging, with many formulas significantly benefitting from PRELEARN. PRELEARN improves the performance on both SAT and UNSAT formulas of varying size and difficulty. In addition, lines that cross the diagonal imply that improving the runtime efficiency of PRELEARN alone would produce more improved instances. For future work, it would be beneficial to classify formulas before running PRELEARN. There may exist general properties of a formula that signal when PRELEARN will be useful and when PRELEARN will be harmful to a CDCL solver. For instance, a formula's community structure [2] may help focus the search to parts of the formula where PR clauses are beneficial.

7.1 Benchmark Families

In this section we analyze benchmark families that PRELEARN had the greatest positive (negative) effect on, found in Table 4. Studying the formulas PRELEARN works well on may reveal better heuristics for finding good PR clauses.

It has been shown that PR works well for hard combinatorial problems based on perfect matchings [14, 15]. The perfect matching benchmarks (randomG) [7] are a generalization of the pigeonhole (php) and mutilated chessboard problems with varying at-most-one encodings and edge densities. The binary PR clauses can be intuitively understood as blocking two edges from the perfect matching if there exists two other edges that match the same nodes. These benchmarks are relatively small but extremely hard for CDCL solvers. Symmetry-breaking with PR clauses greatly reduces the search space and leads KISSAT to a short proof of unsatisfiability. PRELEARN also benefits other hard combinatorial problems that use pseudo-Boolean constraints. The pseudo-Boolean (Pb-chnl) [24] benchmarks are based on at-most-one constraints (using the pairwise encoding) and at-least-one constraints. These formulas have a similar graphical structure to the perfect matching benchmarks. Binary PR clauses block two edges when another set of edges exists that are incident to the same nodes.

For the other two benchmark families that benefited from PRELEARN, the intuition behind PR learning is less clear. The fixed-shape random formulas (fsf) [29] are parameterized non-clausal random formulas built from hyper-clauses. The SAT encoding makes use of the Plaisted-Greenbaum transformation, introducing circuit-like structure to the problem. The superpermutation problem (sp) [22] asks whether a sequence of digits 1–n of length l can contain every permutation of $[1, n]$ as a subsequence, and the optimization variant asks for the smallest such l given n. The sequence of l digits is encoded directly and passed through a multi-layered circuit that checks for the existence of each individual permutation. Digits use the binary (*bin*) or unary (*una*) encoding, are strict *stri* if clauses constrain digit bits to valid encodings and nonstrict *nons* otherwise, and *flat* if the circuit is a large AND or *tree* for prefix recognizing nested circuits. The formulas given ask to find a prefix of a superpermutation for $n = 5$ or length 26 with 19 permutations. The check for 19 permutations was encoded as cardinality constraints in a pseudo-Boolean instance, then converted back to SAT. Each individual permutation is checked by duplicating circuits at each possible starting position of the permutation in l. PR clauses may be pruning certain starting positions for some permutations or affecting the pseudo-Boolean constraints. This cannot be determined without a deeper knowledge of the benchmark generator.

The relativized pigeonhole problem (rphp) [3] involves placing k pigeons in $k - 1$ holes with n nesting places. This problem has polynomial hardness for resolution, unlike the exponential hardness of the classical pigeonhole problem. The symmetry-breaking preprocessor BREAKID [9] generates symmetry-breaking formulas for rphp that are easy for a CDCL solver. PRELEARN can learn many PR clauses but the formula does not become easier. Note PRELEARN can solve the php with $n = 12$ in a second.

One problem is clause and variable permuting (a.k.a. shuffling). The mutilated chessboard problem can still be solved by PRELEARN after permuting variables and clauses. The pigeonhole problem can be solved after permuting clauses but not after permuting variable names. In PRELEARN, PR candidates are sorted by variable name independent of clause ordering, but when the variable names change the order of learned clauses changes. In the mutilated chessboard problem there is local structure, so similar PR clauses are learned under variable renaming. In the pigeonhole problem there is global structure, so a variable renaming can significantly change the binary PR clauses learned and cause earlier saturation with far fewer units.

Another problem is that the addition of PR clauses can change the existing structure of a formula and negatively affect CDCL heuristics. The Pythagorean Triples Problem (Ptn) [19] asks whether monochromatic solutions of the equation $a^2 + b^2 = c^2$ can be avoided. The formulas encode numbers $\{1, \ldots, 7824\}$, for which a valid 2-coloring is possible. In the namings, the *N* in b*N* denotes the number of backbone literals added to the formula. A backbone literal is a literal assigned true in every solution. Adding more than 20 backbone literals makes the problem easy. For each formula KISSAT can find a satisfying assignment, but timeouts with the addition of PR clauses. For one instance, adding only 39 PR clauses will lead to a timeout. In some hard SAT and UNSAT problems solvers require some amount of luck and adding a few clauses or shuffling a formula can cause a CDCL solver's performance to sharply decrease. The Pythagorean Triples problem was originally solved with a local search solver, and local search still performs well after adding PR clauses.

In a straight-forward way, one can avoid the negative effects of adding harmful PR clauses by running two solvers in parallel: one with PRELEARN and one without. This fits with the portfolio approach for solving SAT problems.

8 Conclusion and Future Work

In this paper we presented PRELEARN, a tool built from the SADICAL framework that learns PR clauses in a preprocessing stage. We developed several heuristics for finding PR clauses and multiple configurations for clause learning. In the evaluation we found that PRELEARN improves the performance of the CDCL solver KISSAT on many benchmarks from past SAT competitions.

For future work, quantifying the usefulness of each PR clause in relation to guiding the CDCL solver may lead to better learning heuristics. This is a difficult task that likely requires problem specific information. Separately, failed clause caching can improve performance by remembering and avoiding candidate clauses that fail with unsatisfiable reducts in multiple iterations. This would be most beneficial for problems like the mutilated chessboard that have many conflicting PR clauses. Lastly, incorporating PRELEARN during in-processing may allow for more PR clauses to be learned. This could be implemented with the inner/outer solver framework but would require a significantly narrowed search. CDCL learns many clauses during execution and it would be infeasible to examine binary PR clauses across the entire formula.

Acknowledgements We thank the community at StarExec for providing computational resources.

References

1. Alekhnovich, M.: Mutilated chessboard problem is exponentially hard for resolution. Theoretical Computer Science **310**(1), 513–525 (2004)
2. Ansótegui, C., Bonet, M.L., Giráldez-Cru, J., Levy, J., Simon, L.: Community structure in industrial SAT instances. Journal of Artificial Intelligence Research (JAR) **66**, 443–472 (2019)
3. Atserias, A., Lauria, M., Nordström, J.: Narrow proofs may be maximally long. ACM Transactions on Computational Logic **17**(3) (2016)
4. Audemard, G., Katsirelos, G., Simon, L.: A restriction of extended resolution for clause learning SAT solvers. In: AAAI Conference on Artificial Intelligence. pp. 15–20. AAAI Press (2010)
5. Biere, A., Fazekas, K., Fleury, M., Heisinger, M.: CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT competition 2020 (2020), unpublished
6. Biere, A., Fröhlich, A.: Evaluating CDCL variable scoring schemes. In: Theory and Applications of Satisfiability Testing (SAT). LNCS, vol. 9340, pp. 405–422 (2015)
7. Codel, C.R., Reeves, J.E., Heule, M.J.H., Bryant, R.E.: Bipartite perfect matching benchmarks. In: Pragmatics of SAT (2021)
8. Cook, S.A.: A short proof of the pigeon hole principle using extended resolution. SIGACT News **8**(4), 28–32 (1976)
9. Devriendt, J., Bogaerts, B., Bruynooghe, M., Denecker, M.: Improved static symmetry breaking for SAT. In: Theory and Applications of Satisfiability Testing (SAT). LNCS, vol. 9710, pp. 104–122. Springer (2016)
10. Eén, N., Biere, A.: Effective preprocessing in SAT through variable and clause elimination. In: Theory and Applications of Satisfiability Testing (SAT). LNCS, vol. 3569, pp. 61–75. Springer (2005)
11. Freeman, J.W.: Improvements to Propositional Satisfiability Search Algorithms. Ph.D. thesis, USA (1995)
12. Haken, A.: The intractability of resolution. Theoretical Computer Science **39**, 297–308 (1985)
13. Heule, M.J.H., Hunt, W.A., Wetzler, N.: Expressing symmetry breaking in DRAT proofs. In: Conference on Automated Deduction (CADE). LNCS, vol. 9195, pp. 591–606. Springer (2015)
14. Heule, M.J.H., Kiesl, B., Biere, A.: Short proofs without new variables. In: Conference on Automated Deduction (CADE). LNCS, vol. 10395, pp. 130–147. Springer (2017)
15. Heule, M.J.H., Kiesl, B., Biere, A.: Clausal proofs of mutilated chessboards. In: NASA Formal Methods. LNCS, vol. 11460, pp. 204–210 (2019)
16. Heule, M.J.H., Kiesl, B., Biere, A.: Encoding redundancy for satisfaction-driven clause learning. In: Tools and Algorithms for the Construction and Analysis of Systems (TACAS). LNCS, vol. 11427, pp. 41–58. Springer (2019)
17. Heule, M.J.H., Kiesl, B., Biere, A.: Strong extension free proof systems. In: Journal of Automated Reasoning. vol. 64, pp. 533–544 (2020)
18. Heule, M.J.H., Kiesl, B., Seidl, M., Biere, A.: PRuning through satisfaction. In: Haifa Verification Conference (HVC). LNCS, vol. 10629, pp. 179–194 (2017)
19. Heule, M.J.H., Kullmann, O., Marek, V.W.: Solving and verifying the boolean pythagorean triples problem via cube-and-conquer. In: Theory and Applications of Satisfiability Testing (SAT). LNCS, vol. 9710, pp. 228–245. Springer (2016)
20. Heule, M.J., Hunt, W.A., Wetzler, N.: Trimming while checking clausal proofs. In: Formal Methods in Computer-Aided Design (FMCAD). pp. 181–188 (2013)
21. Järvisalo, M., Heule, M.J.H., Biere, A.: Inprocessing rules. In: International Joint Conference on Automated Reasoning (IJCAR). LNCS, vol. 7364, pp. 355–370. Springer (2012)

22. Johnston, N.: Non-uniqueness of minimal superpermutations. Discrete Mathematics **313**(14), 1553–1557 (2013)
23. Kiesl, B., Seidl, M., Tompits, H., Biere, A.: Super-blocked clauses. In: International Joint Conference on Automated Reasoning (IJCAR). LNCS, vol. 9706, pp. 45–61 (2016)
24. Lecoutre, C., Roussel, O.: XCSP3 competition 2018 proceedings. pp. 40–41 (2018)
25. Liang, J., Ganesh, V., Poupart, P., Czarnecki, K.: Learning rate based branching heuristic for SAT solvers. In: Theory and Applications of Satisfiability Testing (SAT). LNCS, vol. 9710, pp. 123–140 (2016)
26. Manthey, N., Heule, M.J.H., Biere, A.: Automated reencoding of Boolean formulas. In: Haifa Verification Conference (HVC). LNCS, vol. 7857, pp. 102–117 (2013)
27. Marques-Silva, J., Lynce, I., Malik, S.: Conflict-driven clause learning SAT solvers. In: Handbook of Satisfiability, pp. 131–153. IOS Press (2009)
28. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient sat solver. In: Proceedings of the 38th Annual Design Automation Conference. p. 530–535. ACM (2001)
29. Navarro, J.A., Voronkov, A.: Generation of hard non-clausal random satisfiability problems. In: AAAI Conference on Artificial Intelligence. pp. 436–442. The MIT Press (2005)
30. Stump, A., Sutcliffe, G., Tinelli, C.: StarExec: A cross-community infrastructure for logic solving. In: International Joint Conference on Automated Reasoning (IJCAR). LNCS, vol. 8562, pp. 367–373. Springer (2014)
31. Tan, Y.K., Heule, M.J.H., Myreen, M.O.: cake_lpr: Verified propagation redundancy checking in CakeML. In: Tools and Algorithms for the Construction and Analysis of Systems (TACAS), Part II. LNCS, vol. 12652, pp. 223–241 (2021)
32. Tseitin, G.S.: On the Complexity of Derivation in Propositional Calculus, pp. 466–483. Springer (1983)
33. Wetzler, N., Heule, M.J.H., Hunt, W.A.: DRAT-trim: Efficient checking and trimming using expressive clausal proofs. In: Theory and Applications of Satisfiability Testing (SAT). LNCS, vol. 8561, pp. 422–429 (2014)

A Additional Configurations

Table 5. Number of benchmarks where PR clauses were learned, average runtime of PRELEARN, generated positive reducts and solved reducts, number of failed literals found, and total number of PR clauses (unit, binary, or ternary) found on the 0-10k formulas for each configuration. Configurations run for 100 or (*) 500 seconds. 323 formulas total (top). 10k-50k 348 formulas total (bottom).

Config.	Bench	Avg. (s)	Generated Red.	Solved Red.	Failed Lits	PR Clauses
0-10k						
inst-reduced-b	207	15.48	65,356,152	314,056	2,462	31,4056
inst-reduced-ter-b	213	64.08	433,723,302	541,150	2,023	541,150
inst-neighbors(1)-b	221	22.36	104,850,011	548,417	3,416	548,417
rand-neighbors(1)-p	221	20.27	90,816,726	519,514	2,061	494,647
sort-neighbors(1)-p	221	18.42	73,953,349	629,412	2,776	611,411
inst-neighbors(1)-p	221	18.08	67,232,188	659,720	2,845	659,720
inst-neighbors(1)-p*	221	34.73	88,853,680	847,071	3,197	847,071
inst-neighbors(4)-p	217	47.54	407,899,345	366,982	2,799	366,982
inst-neighbors(4)-p*	221	153.21	125,0826,691	511,531	3,580	511,531
10k-50k						
inst-reduced-b	240	59.9	105,886,666	710,993	11,939	710,993
inst-reduced-ter-b	243	96.65	315,454,569	756,729	11,406	756,729
inst-neighbors(1)-b	237	71.08	163,014,068	789,281	6,290	789,281
rand-neighbors(1)-p	240	64.66	133,622,697	731,766	8,721	706,021
sort-neighbors(1)-p	240	63.6	123,508,679	1,080,110	8,893	1,056,112
inst-neighbors(1)-p	242	63.23	119,637,859	950,338	9,292	950,336
inst-neighbors(1)-p*	247	188.52	253,495,888	1,765,086	14,051	1,765,086
inst-neighbors(4)-p	202	78.21	210,218,248	649,343	1,878	649,343
inst-neighbors(4)-p*	220	346.44	841,737,739	1,211,844	5,459	1,211,844

For the evaluation we use the following naming conventions to describe PRELEARN configurations. *inst* / *rand* / *sort* for the three learning configurations with *inst* adding PR clauses instantly as they are found, and *rand* / *sort* adding clauses in batches of size 50. *reduced* / *neighbors(i)* for finding candidate clauses. All configurations use failed literal probing and learn unit and binary PR clauses, with the addition of *-ter* if ternary PR clauses are learned. And *-b* (both) if the first variable in a PR clause is assigned both true or false, or *-p* (positive branching) if the first variable in a PR clause is only assigned true when finding candidates (see Section 5.4). All configurations use the positive reduct. We did not see improvements when using the filtered positive reduct so excluded the option from the results.

Table **??** shows the statistics for various configurations. Note PR clauses includes units and binary PR clauses, plus ternary PR clauses for -ter. The difference in PR clauses learned between the reduced and neighbors(i) heuristics show how restricting the search space can cause many PR clauses to be missed. Also, adding more time (500)

Table 6. Number of solved instances. Configurations run for 100 or (*) 500 seconds.

Config.	0-10k SAT	0-10k UNSAT	10k-50k SAT	10k-50k UNSAT
KISSAT	80	141	143	91
inst-reduced-b	82	147	143	86
inst-reduced-ter-b	83	149	143	87
inst-neighbors(1)-b	84	149	143	89
rand-neighbors(1)-p	82	143	141	88
sort-neighbors(1)-p	83	147	136	86
inst-neighbors(1)-p	80	147	141	87
inst-neighbors(1)-p*	80	147	142	87
inst-neighbors(4)-p	83	148	142	88
inst-neighbors(4)-p*	79	148	141	89

seconds will increase the number of PR clauses for the neighbors heuristic because it finds more candidates. In the second half of the table almost all values are larger. The additional PR clauses per benchmark do not correlate with an overall improved solver performance. Notably the cost per reduct generated is larger because the entire formula is looped through for each reduct. Additionally, the reduced heuristic would require looping over the entire formula to find reduced clauses. In our experiments this was not a bottleneck, but larger formulas may require additional data structures to make the computation feasible.

In Table **??** many of the configurations improve KISSAT's performance on the 0-10k formulas. The inst-neighbors(1)-b configuration solves a total of 12 more formulas than KISSAT alone for these formulas. PRELEARN is less helpful on the 10k-50k formulas, with only one configuration solving more SAT formulas and all configurations solving at least 2 less UNSAT formulas. There is a tradeoff between time spent searching for and learning PR clauses and the effect on solving. In many of the configurations with 500 seconds, the additional PR clauses did not benefit KISSAT. Likewise, the ternary clauses improved the inst-reduced configuration but did not beat the inst-neighbors(1) configuration that only learns binary PR clauses.

With the mutilated chess problem increasing the depth of neighbors was necessary to find new binary PR clauses, and positive branching made the process more efficient. For other problems different configuration setting were more beneficial. A key focus moving forward is to maximize exploration in an attempt to find more candidates while minimizing the number of unsatisfiable reducts generated. This may require problem specific or dynamic configuration control.