Journal of Automated Reasoning (2023) 67:31
https://doi.org/10.1007/s10817-023-09681-3

Preprocessing of Propagation Redundant Clauses

Joseph E. Reeves[1] · **Marijn J. H. Heule**[1] · **Randal E. Bryant**[1]

Received: 1 February 2023 / Accepted: 16 August 2023 / Published online: 14 September 2023

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. Lastly, we tested the robustness of PRELEARN by applying other forms of preprocessing as well as by randomly permuting variable names in the formula before running PRELEARN, and we found PRELEARN performed similarly with and without the changes to the formula.

Keywords Propagation redundant proof · SAT Solving · Preprocessing · Verification

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 inprocessing

✉ Joseph E. Reeves
jereeves@cs.cmu.edu

Marijn J. H. Heule
mheule@cs.cmu.edu

Randal E. Bryant
randy.bryant@cs.cmu.edu

[1] Computer Science Department, Carnegie Mellon University, 5000 Forbes Ave, Pittsburgh, PA 15213, USA

techniques captured by the *resolution asymmetric tautology* (RAT) proof system [22]. As examples, the well-studied pigeonhole and mutilated chessboard problems are challenging benchmarks with exponentially-sized resolution proofs [1, 13]. It is possible to construct small hand-crafted proofs for the pigeonhole problem using *extended resolution* (ER) [9], a proof system that allows the introduction of new variables [33]. ER can be expressed in RAT but has proved difficult to automate due to the large search space for introducing new variables. 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 [16, 18], 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 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. This can be viewed as a kind of *"without loss of generality"* reasoning where the branch that was pruned could be handled similarly (w.r.t. a proof of unsatisfiability) to the branch that is at least as satisfiable.

As an example, consider the perfect matching problem for a bipartite graph in Fig. 1. This problem involves finding a set of edges such that every vertex is incident to exactly one edge. It can be encoded naturally as a SAT problem, with at-least-one constraints for vertices on one side of the graph and at-most-one constraints for vertices on the other. Assume we are placing the at-most-one constraints on vertices from the bottom side of the graph, and edges are labelled by variables $x_{i,j}$ with i as the index of the top vertex. For any perfect matching solution that uses the two green edges in the (a), assigning those two edges to false (red) and instead using the edges in (b) will give another valid solution. It is easy to see that both choices for edges will be incident to vertices 1 and 2 on the top and bottom, so switching between then will not affect the matching on the remaining vertices. This means the partial assignment in (b) is at least as satisfiable as the partial assignment in (a). Therefore, one could carry out the proof after excluding the edge orientation in (a) without loss of generality. This is a powerful form of reasoning that can efficiently remove many symmetries.

The *satisfaction-driven clause learning* (SDCL) solver SADICAL [17] 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 [8]. 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.

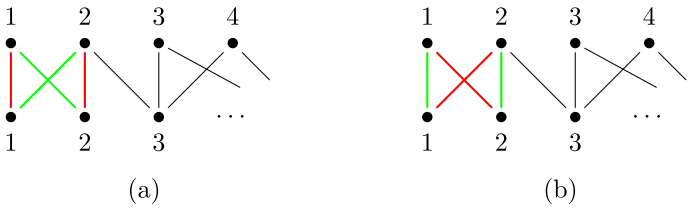

Fig. 1 Motivating example of a bipartite graph perfect matching problem. Green edges are assigned true (in the perfect matching), and red edges are assigned false (not in the perfect matching). (Color figure online)

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 [12] 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.

For some highly structured problems, the addition of short PR clauses can be viewed as a form of symmetry breaking, as shown with the bipartite matching example above. There exist tools for detecting symmetries and generating symmetry-breaking constraints, such as BREAKID [10]. They work by converting the SAT problem into a colored graph, then using an automorphism extractor such as SAUCY [23] to detect symmetries. Recent work has shown that some preprocessing techniques are harmful to graph-based symmetry breaking methods [2]. State-of-the-art SAT solvers incorporate many preprocessing techniques throughout solving [22], and the interaction between preprocessing techniques is not well understood. Therefore, it is important that new preprocessing techniques are robust if they are to be incorporated in a complex solver. PRELEARN performs symmetry-breaking in a much different way than the graph-based tools, finding PR clauses by solving small SAT problems that do not consider the formula's global structure. In our evaluation we show that randomly permuting variable names and applying preprocessors will not significantly alter the performance of PRELEARN.

Most SAT solvers support logging proofs of unsatisfiability for independent checking [15, 21, 34]. This has proved valuable for verifying solutions independent of a (potentially buggy) solver. Modern SAT solvers log proofs in the DRAT proof system (RAT [22] with deletions). DRAT captures all widely used pre- and inprocessing techniques including bounded variable elimination [11], bounded variable addition [26], and extended learning [5, 33]. DRAT can express the common symmetry-breaking techniques, but it is complicated [14]. PR can compactly express some symmetry-breaking techniques [16, 18], yielding short proofs that can be checked by the proof checker DPR-TRIM [17]. PR gives a framework for strong symmetry-breaking inferences and also maintains the highly desirable ability to independently verify proofs.

This journal article is an extension of a conference paper under the same name that appeared in IJCAR'22 [30]. The contributions of the original paper included: (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 improved the performance of the CDCL solver KISSAT on 22% of the satisfiable and unsatisfiable competition benchmarks we considered. The improvement was 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 produced refutation proofs for the mutilated chessboard problem containing only unit and binary PR clauses. We extend the results in this article by exploring the impact or permuting and preprocessing formulas before running PRELEARN. In an experimental evaluation we found that the effectiveness of PRELEARN was not significantly altered. We also present several examples and pseudocode for our main algorithm not appearing in the conference version.

2 Preliminaries

We consider propositional formulas in *conjunctive normal form* (CNF). A CNF formula F 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). The *polarity* of a literal is positive if it is a positive literal, and negative if it is a negative literal. Clauses are represented as a set of literals, but we often write them out directly, e.g., the clause $\{x_1, \overline{x}_2\}$ is written $(x_1 \vee \overline{x}_2)$. For a set of literals L the formula $F(L)$ is the clauses $\{C \in F \mid C \cap L \neq \emptyset\}$.

An *assignment* α is a mapping from variables to truth values 1 (*true*) and 0 (*false*). An assignment is *total* if it assigns every variable appearing in a finite formula to a value, and is *partial* if it assigns a subset of variables to values. For a literal ℓ, $\mathsf{var}(\ell)$ is the variable corresponding to ℓ, e.g., $\mathsf{var}(x_1) = \mathsf{var}(\overline{x}_1) = x_1$. The set of variables occurring in a formula, assignment, or clause is given by $\mathsf{var}(F)$, $\mathsf{var}(\alpha)$, or $\mathsf{var}(C)$. Assignment α *satisfies* a positive (negative) literal ℓ if α maps $\mathsf{var}(\ell)$ to true (α maps $\mathsf{var}(\ell)$ to false, respectively), and *falsifies* it if α maps $\mathsf{var}(\ell)$ to false (α maps $\mathsf{var}(\ell)$ to true, respectively). We write a partial assignment as the set of literals it satisfies, e.g., $x\,\overline{y}$ mapping $\alpha(x) = 1$ and $\alpha(y) = 0$. 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 formulas 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 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 F reduced by an assignment α is given by $F|_\alpha = \{C|_\alpha \mid C \in F \text{ and } C|_\alpha \neq \top\}$. Given an assignment $\alpha = \ell_1 \ldots \ell_n$, $C = (\overline{\ell}_1 \vee \cdots \vee \overline{\ell}_n)$ is the clause that *blocks* α. The assignment *blocked* by a clause is the negation of the literals in the clause. A *unit* is a clause containing a single literal. The *unit clause rule* takes the assignment α of all units in a formula F and generates $F|_\alpha$. 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 F *implies* a formula F', denoted $F \models F'$, if every assignment satisfying F satisfies F'. By $F \vdash_1 F'$ we denote that for every clause $C \in F'$, applying unit propagation to the assignment blocked by C in F derives a conflict. If unit propagation derives a conflict on the formula $F \cup \{\{\ell\}\}$, we say ℓ is a *failed literal* and the unit $\overline{\ell}$ is logically implied by the formula. Failed literal probing [12] 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 $\overline{\ell}$ 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, and 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 [7, 25], assigns a truth value to it, and performs unit propagation. The selected variables are *decision variables*, and the assignment including decision variables and propagated variables is called the *trail*. 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 F if F and $F \cup \{C\}$ are *satisfiability equivalent*. The clause sequence $F, C_1, C_2, \ldots, C_n$ is a clausal proof of C_n from F if each clause C_i $(1 \leq i \leq n)$ is redundant w.r.t. $F \cup \{C_1, C_2, \ldots, C_{i-1}\}$. The proof is a refutation of F 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. A clause has RUP if unit propagation on the falsified literals of the clause results in a conflict. 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, but requires witnesses for proof steps to be efficiently checkable.

Let C be a clause in the formula F and α the assignment blocked by C. Then C is PR w.r.t. F if and only if there exists an assignment ω such that $F|_\alpha \vdash_1 F|_\omega$ 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 F can be transformed into an assignment containing ω that satisfies F.

Running Example *As the running example, we consider the bipartite graph from Fig. 1. There may be more nodes in the graph but as long as there are no additional edges incident to the vertices* $1-3$ *on the top and bottom the additional edges will not affect the calculation of the positive reduct or any following examples. Variable* $x_{i,j}$ *denotes the edge from vertex* i *on the top to vertex* j *on the bottom set of nodes. The at-most-one constraints for the bottom vertices shown are:* $(\overline{x}_{1,1} \vee \overline{x}_{2,1}) \wedge (\overline{x}_{1,2} \vee \overline{x}_{2,2}) \wedge (\overline{x}_{2,3} \vee \overline{x}_{3,3}) \wedge (\overline{x}_{2,3} \vee \overline{x}_{4,3}) \wedge (\overline{x}_{3,3} \vee \overline{x}_{4,3})$. *The at-least-one constraints for the top vertices are:* $(x_{1,1} \vee x_{1,2}) \wedge (x_{2,1} \vee x_{2,2} \vee x_{2,3}) \wedge (x_{3,3} \vee \cdots) \wedge (x_{4,2} \vee x_{4,3} \vee \cdots)$. *In the examples below,* F *denotes the entire formula.*

Example 1 From Fig. 1, the clause $(\overline{x}_{1,1} \vee \overline{x}_{2,2})$ corresponds to the binary PR clause blocking the assignment of red edges in (a). One possible witness for this PR clause could be $\omega = x_{1,2}\, x_{2,1}\, \overline{x}_{1,1}$. The witness is saying that if $x_{1,1}\, x_{2,2}$ (red edges) were in a satisfying assignment to the problem, we could negate them then set $x_{1,2}\, x_{2,1}$ (green edges) to true in the assignment while keeping the rest of the assignment untouched, and it would still satisfy the formula.

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 NP-complete [19]. However, a PR proof is checkable in polynomial time if the witness assignments ω are included. A clausal proof with witnesses has the form $F, (C_1, \omega_1), (C_2, \omega_2), \ldots, (C_n, \omega_n)$. The proof checker DPR-TRIM can efficiently check PR proofs that include witnesses. Further, DPR-TRIM[1] can emit proofs in the LPR format. They can be validated by the formally-verified checker CAKE-LPR [32], which was used to validate results in recent SAT competitions.

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

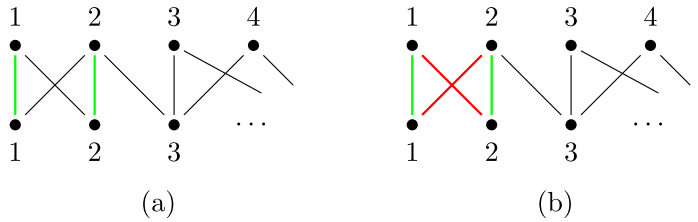

Fig. 2 Assignments and propagation used in Example 3

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 F, a *pruning predicate* is a formula such that if it is satisfiable, the clause C is redundant w.r.t. F. 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.

Definition 1 (*Definition* 3 *of* [19]) 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.

Definition 2 (*Definition* 6 *of* [18]) Let F be a formula, α an assignment, and C the clause that blocks α. The filtered positive reduct $f(F, \alpha)$ 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. Next, clauses are removed (filtered) if unit propagation on the formula $F|_\alpha$ conjuncted with negated literals from the clause that are not assigned by α derives a conflict. Finally, all literals that are not assigned by α are removed from the remaining clauses.

The filtered positive reduct is a subset of the positive reduct. The filtered positive reduct is more expensive to compute due to the filtering operation that applies unit propagation. In Example 2 the PR clause found by solving the positive reduct is larger than it needs to be. In fact, one can assign variables, perform unit propagation to get a new assignment α, solve the reduct based on α, then remove all propagated literals from the resulting clause. This is shown in Example 3, where the the positive reduct is SAT, and we can remove the propagated literals appearing in the PR clause from Example 2. For more information on the positive reduct and filtered positive reduct, we refer readers to [18].

Example 2 Given the assignment $\alpha = x_{1,1}\, x_{2,2}\, \overline{x}_{1,2}\, \overline{x}_{2,1}$, the positive reduct is the clause that blocks α, $(\overline{x}_{1,1} \vee \overline{x}_{2,2} \vee x_{1,2} \vee x_{2,1})$, conjuncted with the satisfied clauses with literals from α, $(\overline{x}_{1,1} \vee \overline{x}_{2,1}) \wedge (\overline{x}_{1,2} \vee \overline{x}_{2,2}) \wedge (x_{1,1} \vee x_{1,2}) \wedge (x_{2,1} \vee x_{2,2})$. The assignment $\omega = \overline{x}_{1,1}\, \overline{x}_{2,2}\, x_{1,2}\, x_{2,1}$ satisfies the positive reduct, so we can learn the clause $(\overline{x}_{1,1} \vee \overline{x}_{2,2} \vee x_{1,2} \vee x_{2,1})$ with witness ω added to the proof.

Example 3 In this example, (a) and (b) refer to Fig. 2. This example considers the positive reduct generated from an assignment after *unit propagation* and shows how propagated literals can be removed from the learned PR clause. After assigning $\alpha = x_{1,1}\, x_{2,2}$, shown in (a), then performing unit propagation, the new assignment is $\alpha = x_{1,1}\, x_{2,2}\, \overline{x}_{1,2}\, \overline{x}_{2,1}$, shown in

(b). The resulting positive reduct is satisfiable, with witness $\omega = \overline{x}_{1,1}\,\overline{x}_{2,2}\,x_{1,2}\,x_{2,1}$. Instead of learning $(\overline{x}_{1,1} \vee \overline{x}_{2,2} \vee x_{1,2} \vee x_{2,1})$, we can remove propagated literals and learn the clause $(\overline{x}_{1,1} \vee \overline{x}_{2,2})$ that blocks the assignment in (a), and this binary clause is far more effective at pruning the search space.

SADICAL [17] uses satisfaction-driven clause learning (SDCL) that extends CDCL by learning PR clauses [19] 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 vice versa. 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

PRELEARN is a preprocessor that adds short PR clauses to a formula. The new formula which now contains the additional PR clauses can be solved by an off-the-shelf SAT solver. In this section, we first discuss an overview of how PRELEARN fits in the general SAT solving tool-chain. Next, we describe the process for finding candidate short PR clauses. Then, we provide a method for determining if a candidate PR clause can be learned by solving the positive reduct. We present several additional configurations for finding and learning PR clauses that do not appear in the experimental evaluation. Finally, we detail the implementation of PRELEARN through pseudocode.

5.1 Overview of PRELEARN

The goal of PRELEARN is to find useful PR clauses that improve the performance of CDCL solvers on both satisfiable and unsatisfiable instances. Figure 3 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. When the preprocessing stage ends, the new formula that includes learned PR clauses is solved by a CDCL solver. If the 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.

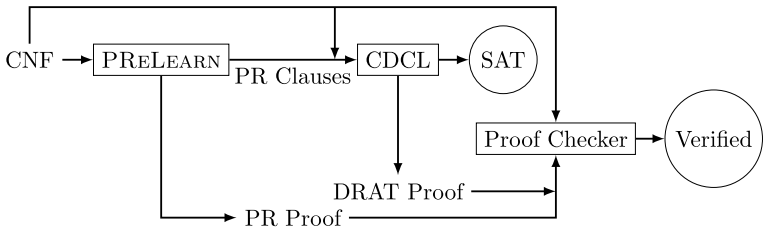

Fig. 3 Solving a formula with PReLearn and a CDCL solver

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. We use unit propagation on the original formula to construct the positive reduct. When unit propagation after assigning a single literal derives the empty clause, the negation of the failed literal is RUP and is learned as a unit. This is in contrast to a PR unit found by solving the positive reduct, which requires storing a witness in the proof. 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 nonconflicting. The selection may maximize the number of PR clauses learned or optimize for some other metric. Adding PR clauses and failed literals may cause new clauses to become PR, so the entire process is iterated multiple times.

5.2 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 formula F and variable x, $\mathsf{neighbors}(F, x)$ denotes the set of variables occurring in clauses containing literal x or $\overline{x}$, excluding variable x. For a partial assignment α, $\mathsf{neighbors}(F, \alpha)$ denotes $\bigcup_{x \in \mathsf{var}(\alpha)} \mathsf{neighbors}(F, x) \backslash \mathsf{var}(\alpha)$. 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 $\{(\overline{\ell} \vee y), (\overline{\ell} \vee \overline{y}) \mid y \in \mathsf{neighbors}(F, \alpha)\}$.

Example 4 shows how candidate binary PR clauses are constructed using a single polarity of an initial variable x. This can also be done by assigning x to false and proceeding with candidate generation in a similar way. If x was assigned false, candidate PR clauses would be of the form $(x \vee y)$, $(x \vee \overline{y})$ for some variable y. The example also describes how larger sets of candidate clauses can be created with recursive applications of neighbors. The is important for solving the mutilated chess problem from Sect. 5.5.

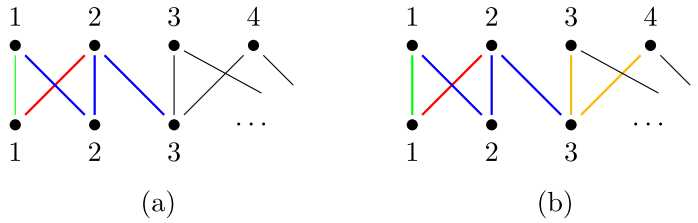

Fig. 4 Neighbors visualizations used in Example 4

Example 4 In this example, (a) and (b) refer to Fig. 4. To find the neighbors for a candidate variable $x_{1,1}$ we first assign $x_{1,1}$ to true (green), then perform unit propagation assigning $x_{2,1}$ to false (red). With $\alpha = x_{1,1}\,\overline{x}_{2,1}$, $\mathsf{neighbors}(F, \alpha) = \{x_{1,2}, x_{2,2}, x_{2,3}\}$ (blue), shown in (a). Using these variables, we can construct the following candidate PR clauses, $(\overline{x}_{1,1} \vee x_{1,2})$, $(\overline{x}_{1,1} \vee \overline{x}_{1,2}), \ldots, (\overline{x}_{1,1} \vee x_{2,3}), (\overline{x}_{1,1} \vee \overline{x}_{2,3})$.

We can extend the candidates by taking $\mathsf{neighbors}(F, \mathsf{neighbors}(F, \alpha)) = \{x_{3,3}, x_{4,3}\}$, shown as yellow in (b). This will increase the number of candidate PR clauses to test. In general, we can continue to iteratively extend the the candidates by calculating the neighbors of the previous iteration.

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 $z \in \mathsf{neighbors}(F, \alpha)$ 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. Given a formula F and assignment α, $\mathsf{reduced}(F, \alpha) = \mathsf{var}(F|_\alpha \setminus F)$. The idea behind this heuristic is to direct the assignment towards conditional autarkies that imply a satisfiable positive reduct [19]. The neighbors approach generalizes this to variables in all reduced clauses whether or not they are unsatisfied. A comparison can be found in Sect. 5.4.

5.3 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 5 below 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 clause 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 partial assignment that generated the pruning predicate for D satisfies C and C is not satisfied by the witness of D, then C conflicts with D. Intuitively, if the partial assignment α that generated the pruning predicate for D satisfies C then C will appear in the positive reduct generated for D after removing literals from C not assigned by α. If this clause is not satisfied by the previous witness for D, then the previous witness for D no longer satisfies the positive reduct, making the witness invalid. In some cases reordering the two PR clauses may avoid a conflict. In Example 5 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 5 Assume the following clause witness pairs are valid in a formula F: $\{(x_1 \vee x_2 \vee x_3), x_1\,\overline{x}_2\,\overline{x}_3\}$, and $\{(x_1 \vee \overline{x}_2 \vee x_4), \overline{x}_1\,\overline{x}_2\,x_4\}$. The first clause conflicts with the second. If the first clause is added to F, the clause $(x_1 \vee x_2)$ would be in the positive reduct for the second clause, but it is not satisfied by the witness of the second clause.

5.4 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.

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 4. We also implement the reduced heuristic, shown in Example 6. 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. Also, increasing the timeout for PRELEARN by 5×'s would increase the number of PR clauses learned when using larger values of i in $\mathsf{neighbors}_i$, but these additional PR clauses did not improve solver performance. More work needs to be done to determine when and how to apply these additional configurations.

Example 6 Given the assignment $\alpha = x_{1,1}\,\overline{x}_{2,1}$ from Example 4, the only clause reduced but not satisfied is $(x_{2,1} \vee x_{2,2} \vee x_{2,3})$, yielding the candidate variables $\{x_{2,2}, x_{2,3}\}$. This is a

Algorithm 1 PREL EARN (F : CNF)

```
1: for ℓ in Both(1, 2, ..., NumVariables) do
2:     F ← SimplifyUnits(F)                          ▷ Propagate top-level units, simplify formula
3:     if IsAssigned(ℓ) then
4:         Continue                                  ▷ Skip already assigned literals
5:     end if
6:     α ← [ℓ]
7:     if UnitProp(F, α) is UNSAT then
8:         F ← Learn(F, ℓ̄, ())                       ▷ Learn failed literal
9:         Continue
10:    end if
11:    if ω ← Solve(GetReduct(F, α)) is SAT then
12:        F ← Learn(F, ℓ̄, ω)                        ▷ Learn unit PR
13:        Continue
14:    end if
15:    for k in Both(neighbors(F, α)) do
16:        α′ ← α + [k]
17:        if UnitProp(F, α′) is not UNSAT then
18:            if ω ← Solve(GetReduct(F, α′)) is SAT then
19:                F ← Learn(F, (ℓ̄ ∨ k̄), ω)          ▷ Learn binary PR
20:                Break
21:            end if
22:        end if
23:    end for
24: end for
```

smaller set than $\mathsf{neighbors}(F, \alpha) = \{x_{1,2}, x_{2,2}, x_{2,3}\}$, because neighbors does not consider if the clause is satisfied, and will include $x_{1,2}$ from the satisfied clause $(x_{1,1} \vee x_{1,2})$.

5.5 Implementation

PREL EARN was implemented using the inner/outer-solver framework in SADICAL, and the general algorithm for the outer solver is presented in Algorithm 1. The outer solver takes a formula F as input, and learns clauses by adding them to the formula while writing clauses and witnesses to a proof. Instead of performing a complete search procedure like CDCL, the outer solver searches for short clauses to learn. The outer loop ranges over both polarities of each variable in the formula, selecting candidate literals ℓ. Given a set of variables V, the function *Both* returns set of literals $\bigcup_{x \in V} \{x, \overline{x}\}$. In line 2, it propagates all unit clauses and simplifies the formula (i.e., removes falsified literals from clauses and removes satisfied clauses from the formula). This ensures that as PREL EARN learns unit clauses they are removed from the formula and do not affect candidate generation (which depends on unit propagation). If the candidate literal ℓ is assigned, the candidate is skipped and the next candidate literal is considered. The assignment of ℓ in α on line 6 is achieved by deciding on ℓ within the outer solver at the top decision level. The unit propagation algorithm propagates an input assignment α on an input formula, adding propagated literals to α during the procedure, and returns UNSAT if a conflict is found. α is represented internally as the trail of the outer solver. The function call in line 7 checks if ℓ is a failed literal, and if so learns $\overline{\ell}$. The *Learn* function adds a clause to the input formula and writes a clause and witness to the proof (the witness is empty for a failed literal since it is RUP). In line 11, the positive reduct is generated for the assignment α after propagating ℓ. If the inner solver returns SAT, then the satisfying assignment to the positive reduct (ω) is used as the witness for the unit PR clause $\overline{\ell}$. In the

case of a failed literal or unit PR clause, the procedure continues with the next candidate literal.

Candidate binary PR clauses are tested in lines 15–24. Line 15 ranges over both polarities of all of the neighbors of α, and this can be substituted with any procedure that finds candidate variables (e.g., the reduced heuristic). The assignment of k in α' is achieved by deciding on k within the outer solver at the first decision level. Again, unit propagation is necessary to obtain the new assignment α in order to generate the positive reduct. If the reduct is SAT, then the satisfying assignment (ω) to the positive reduct is used as the witness for the binary PR clause $(\bar{\ell} \vee \bar{k})$. Upon each subsequent iteration of this inner loop, the assignment in line 16 is achieved by backtracking one decision level, (unassigning the previous k), then deciding on the new literal k.

In this pseudocode, clauses are learned immediately and added to the formula, so in subsequent calls to unit propagation, all learned units will be propagated and learned binary PR clauses will be included in the formula. For this reason, after learning a binary PR clause we move on to the next candidate variable in the outer for loop, instead of finding more PR clauses with the same candidate. Alternatively, the *Learn* function could store found binary PR clauses, and every so often select some of the non-conflicting binary PR clauses to learn and add to the formula in a batch.

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 domino placements on the board. At-most-one constraints say only one domino can cover each square, and at-least-one constraints 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 6 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 Fig. 5 the PR clauses that can be learned involve blocking one domino orientation that can be replaced by a symmetric

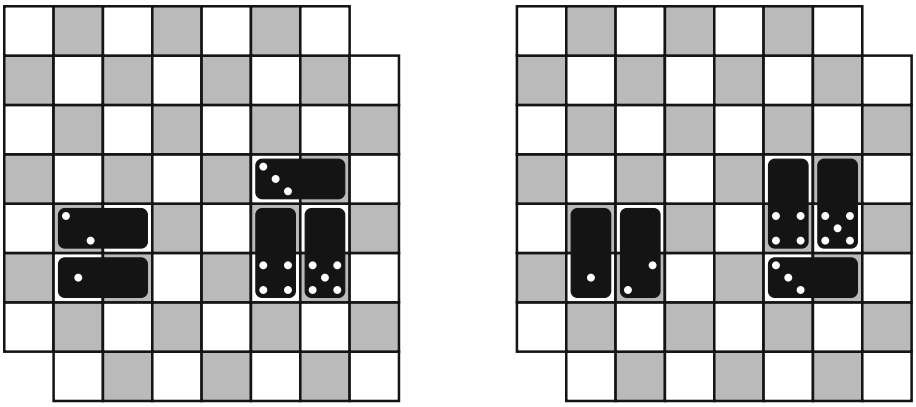

Fig. 5 Occurrences of two horizontal dominoes may be replaced by two vertical dominos in a solution. Similarly, occurrences of a horizontal domino atop two vertical dominos can be replaced by shifting the horizontal domino down

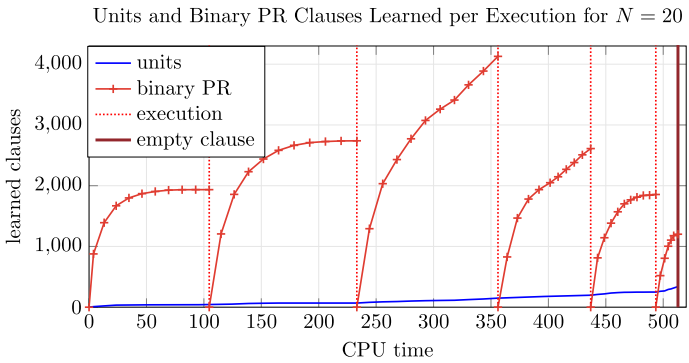

Fig. 6 Unit and binary PR clauses learned each execution (red-dotted line) until the empty clause was learned. Markers on binary PR lines represent an iteration within an execution. (Color figure online)

Table 1 Statistics running multiple executions of PRELEARN on the mutilated chessboard problem with the configurations described below

N	Time (s)	# Exe.	Avg. (s)	Units	Bin.	Avg. units	Avg. bin.
8	0.14	1	0.14	30	164	30.00	164.00
12	4.94	1	4.94	103	1045	103.00	1045.00
16	62.47	2	31.23	195	3988	97.50	1994.00
20	513.12	6	85.52	339	14,470	56.50	2411.67
24	4941.38	26	190.05	512	64,038	19.69	2463.00

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

orientation. To optimize for these types of PR clauses, we only 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 5000 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 Sect. 5.4) to find the best results (presented above). The reduced heuristic (a subset of neighbors) did not yield complete proofs. The same was true for neighbors when the depth parameter was not large enough. The configurations that worked best were those that tested many candidate PR clauses, allowing units to be found in each successive iteration and eventually completing the proof. In fact, we found that increasing the depth of neighbors was necessary for larger boards including $N = 24$.

We know of PR proofs using only unit and binary PR clauses that are shorter than the proofs PRELEARN produced. One way to achieve a more optimal proof size is through batch learning – systematically adding only the PR clauses that are most effective in the proof – instead of adding PR clauses immediately after they are found. The current heuristics for batch learning, random and sorted, are not clever enough to achieve this result.

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 **10–50k** contains 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://zenodo.org/record/7882172.

We ran our experiments on StarExec [31]. The specs for the compute nodes can be found online.[2] 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 5000 seconds 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 [6] 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.

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

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)	Gen. reducts	Sat. reducts	% Sat.	Failed lits
0–10k	174/323	22.68	101,459,481	531,482	0.52	3412
10–50k	115/348	69.98	156,983,732	762,363	0.49	6095

Table 3 Number of total solved instances and exclusive solved instances running KISSAT with and without PRELEARN

	0–10k SAT	0–10k UNSAT	10–50k SAT	10–50k UNSAT
Total w/ PRELEARN	85	148	139	89
Total w/o PRELEARN	80	140	142	89
Exclusively w/ PRELEARN	5	10	1	1
Exclusively w/o PRELEARN	0	2	4	1
Improved w/ PRELEARN	22	42	21	13

Number of improved instances running KISSAT with PRELEARN. PRELEARN execution times were included in total execution times. An instance was improved if it was solved at least a second faster and the difference in solving time was at least one percent

In the satisfiable case, the derivation for the learned PR clauses was verified using a forward check in DPR-TRIM, and the satisfying assignment found by KISSAT was verified by the StarExec post-processing tool. In the unsatisfiable case, we verified complete proofs with a formally-verified proof checker. To do this, the derivation for the learned PR clauses was concatenated to the proof traced by KISSAT. The proof was then passed through DPR-TRIM to produce a proof with hints, which was then checked by CAKE-LPR. Only two proofs were not verified on the cluster because of resource limitations (timeouts), so we verified them locally.

Table 2 shows the cumulative statistics for running PRELEARN on the benchmark sets. PR clauses are found in about one third of the formulas (174 of the 323 small formulas and 115 of the 348 larger 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. 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 10–50k formulas the average runtime more than triples, but the number of generated reducts less than doubles.

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 PR clauses are well-suited for. There were 5 additional SAT formulas solved with PRELEARN that KISSAT alone could not solve. This shows that PRELEARN impacts not only hard unsatisfiable problems but satisfiable problems as well. On the other hand, the addition of PR clauses makes some problems more difficult. This is clear with the 10–50k results, where 2 benchmarks are solved exclusively with PRELEARN and 5 are solved exclusively without. Additionally, PRELEARN improved KISSAT's

performance on 98 of 671 or approximately 15% of benchmarks. This is a large portion of benchmarks, both SAT and UNSAT, for which PREPLEARN is helpful.

Figure 7 gives a more detailed picture on the impact of PREPLEARN 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 PREPLEARN execution time, and so the right-hand end gives the total time for PREPLEARN plus KISSAT. Lines that cross the diagonal indicate that the preprocessing improved KISSAT's performance but ran for longer than the improvement. PREPLEARN 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 PREPLEARN. 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 PREPLEARN.

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 PREPLEARN. Many of these formulas are satisfiable, unlike for the smaller benchmarks. Most of the formulas solved within 100 seconds barely cross the diagonal, indicating that the addition of PR clauses provided a negligible benefit.

The results in Fig. 7 are encouraging, with many formulas significantly benefitting from PREPLEARN. PREPLEARN 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 PREPLEARN alone would produce more improved instances. For future work, it would be beneficial to classify formulas before running PREPLEARN. There may exist general properties of a formula that signal when PREPLEARN will be useful and when PREPLEARN will be harmful to a CDCL solver. For instance, a formula's community structure [3] 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 PREPLEARN had the greatest positive (negative) effect on, found in Table 4. Studying the formulas PREPLEARN 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 [16, 18]. The perfect matching benchmarks (randomG) [8] 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 exist 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. PREPLEARN 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 benchmark family that benefited from PREPLEARN, 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

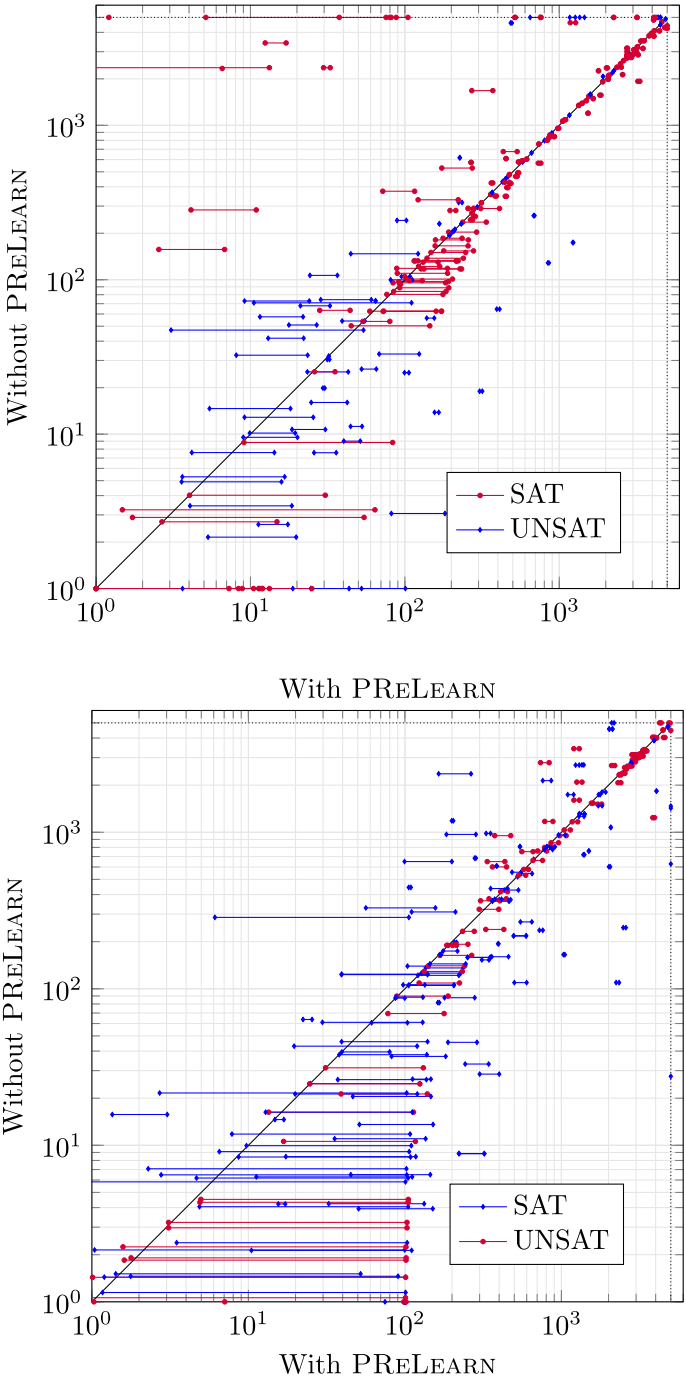

Fig. 7 Execution times w/ and w/o PreLearn on 0–10k (top) and 10–50k (bottom) benchmarks. The left-hand point of each segment shows the time for the SAT solver alone; the right-hand point indicates the combined time for preprocessing and solving

Table 4 Some formulas solved by KISSAT exclusively *with* PRELEARN (top) and some formulas solved exclusively *without* PRELEARN (bottom)

Set	Value	With	Without	Clauses	Formula/year
0–10k	UNSAT	1.21	–	2033	ph12* '13
0–10k	UNSAT	37.58	–	20,179	Pb-chnl15-16_c18* '19
0–10k	UNSAT	104.72	–	44,373	Pb-chnl20-21_c18 '19
0–10k	UNSAT	87.83	–	1,480	randomG-Mix-n17-d05 '21
0–10k	UNSAT	81.48	–	1320	randomG-n17-d05 '21
0–10k	UNSAT	519.24	–	1469	randomG-n18-d05 '21
0–10k	UNSAT	762.22	–	1535	randomG-Mix-n18-d05 '21
0–10k	SAT	1269.13	–	9650	fsf-300-354-2-2-3−2.23.opt '13
0–10k	SAT	1456.33	–	10,058	fsf-300-354-2-2-3−2.46.opt '13
10–50k	SAT	–	22.99	254	Ptn-7824-b13 '16
10–50k	SAT	–	549.27	133	Ptn-7824-b09 '16
10–50k	SAT	–	1246.42	39	Ptn-7824-b02 '16
10–50k	SAT	–	1290.49	121	Ptn-7824-b08 '16
10–50k	UNSAT	–	3650.21	30,975	rphp4_110_shuffled '16
10–50k	UNSAT	–	4273.88	30,434	rphp4_115_shuffled '16

(*) solved without KISSAT. Clauses include PR clauses and failed literals learned

the Plaisted-Greenbaum transformation, introducing circuit-like structure to the problem. It cannot be determined without a deeper knowledge of the benchmark how PR clauses are improving the solving time.

The relativized pigeonhole problem (RPHP) [4] 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 [10] 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), where variable names are permuted within a formula, or clause ordering is shuffled (neither operation changes the meaning of the formula but may affect solver heuristics). 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. These benchmarks motivated a further exploration of the robustness of PRELEARN in Sect. 8.

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) [20] 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 bN 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. For example, we found that adding the found PR clauses to the beginning of the formula instead of the end, or removing blocked clauses from the set of found PR clauses, improved the solving time of KISSAT. 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 Robustness of PRELEARN

Preprocessing is necessary for state-of-the-art SAT solvers to achieve good performance on a diverse benchmark set. In general, preprocessing can significantly change the structure of a formula by removing literals from clauses, removing clauses, and propagating literals. There are a multitude of preprocessing techniques, many of which are run repeatedly during a solver's execution. Therefore, new preprocessing techniques must be robust against changes to the formula in order to be effective in relation to other important forms of preprocessing. Recent work has shown that some preprocessing techniques are harmful to graph-based symmetry breaking methods [2]. The work presented a restricted set of preprocessing techniques that still allowed for, and aided, syntactic-based symmetry breaking. In addition to preprocessing, it is important that solving techniques are robust under clause order permutations (shuffling) and variable name permutations (renaming), as there is no guarantee of how a user will translate some problem into a formula. Graph-based symmetry breaking is resilient to such renamings and shuffling, as they will not affect the graphical structure of the formula.

PR clauses are found by solving the positive reduct. Preprocessing can significantly affect the structure of a positive reduct on certain literals, potentially turning a satisfiable reduct unsatisfiable. Furthermore, preprocessing can remove variables themselves either through propagation or elimination, nullifying any PR clause that may have contained those variables. It is unclear whether the combination of preprocessing techniques will prevent many PR clauses from being found, or whether new PR clauses will appear. With respect to clause shuffling and variable renamings, if PRELEARN were to loop over every variable in the formula it should *find* all of the same PR clauses. However, learning PR clauses in a different order might block other PR clauses from being found and learned, and this could occur under a variable renaming.

8.1 Evaluation with Preprocessing

For this evaluation we perform preprocessing using the SAT solver CADICAL with flags `-o inproc.cnf -c 10000`. With this configuration, the solver returns the irredundant formula clauses after 10,000 conflicts have been found. Before 10,000 conflicts, CADICAL will have applied all of the standard preprocessing techniques to the formula. This approach mimics the preprocessing that would occur to a real problem before it was passed to another tool like PRELEARN. There are four configurations we consider: preprocessing, PRELEARN

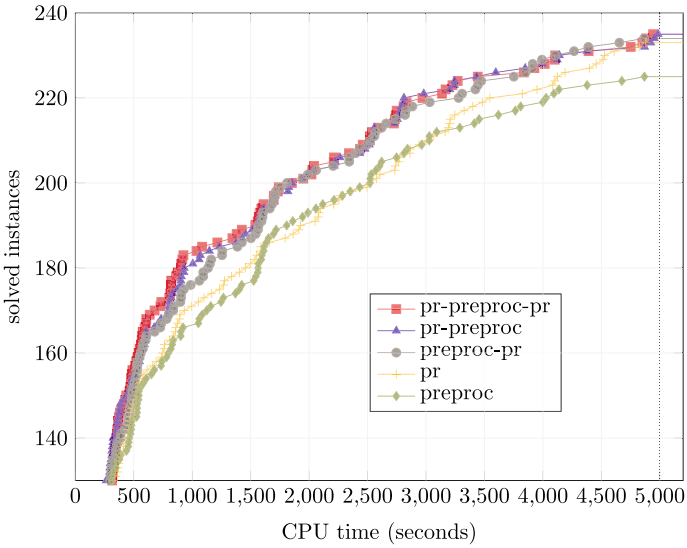

Fig. 8 Cumulative number of solved instances with different preprocessing configurations. In the legend, pr is a call to PRELEARN, and preproc is a call to the preprocessor. After running some combination of the two, KISSAT is called on the resulting formula

then preprocessing, preprocessing then PRELEARN, and PRELEARN then preprocessing then PRELEARN. In each case PRELEARN was run with the same configuration as the main evaluation. Experiments were run on the 0-10k benchmark set for 5000 seconds as in the main evaluation.

Figure 8 shows the cumulative instances solved for the four configurations including preprocessing and the fifth configuration with only PRELEARN. The runtimes consist of the entire toolchain for each configuration, including preprocessing with CADICAL, preprocessing with PRELEARN, and solving with KISSAT. Only using preprocessing solves the fewest instances. Note that many of the inprocessing techniques from CADICAL are included in KISSAT. This result shows that applying preprocessing before PRELEARN will not be worse than only applying PRELEARN. In other words, the PR clauses learned by PRELEARN will still benefit solving even after preprocessing. However, preprocessing before PRELEARN does affect the number of solved instances, producing slightly worse performance than applying PRELEARN before preprocessing. The three configurations with PRELEARN applied first show similar performance, suggesting that PRELEARN is most effective when applied to the original formula before preprocessing, and nearly as effective after preprocessing.

The scatter plots in Fig. 9 show the number of PR clauses learned (top) and the execution times (bottom) for each formula. From the top plot it is clear that fewer PR clauses are learned after applying preprocessing, and in many formulas no PR clauses are learned after preprocessing. The bottom plot shows that execution times are varied, and fewer PR clauses does not necessarily mean worse performance. In fact, there are several problems solved exclusively when preprocessing is applied before PRELEARN, and vice versa. However, this does not lead to a large difference in overall performance, as seen by the cumulative solved instances in Fig. 8.

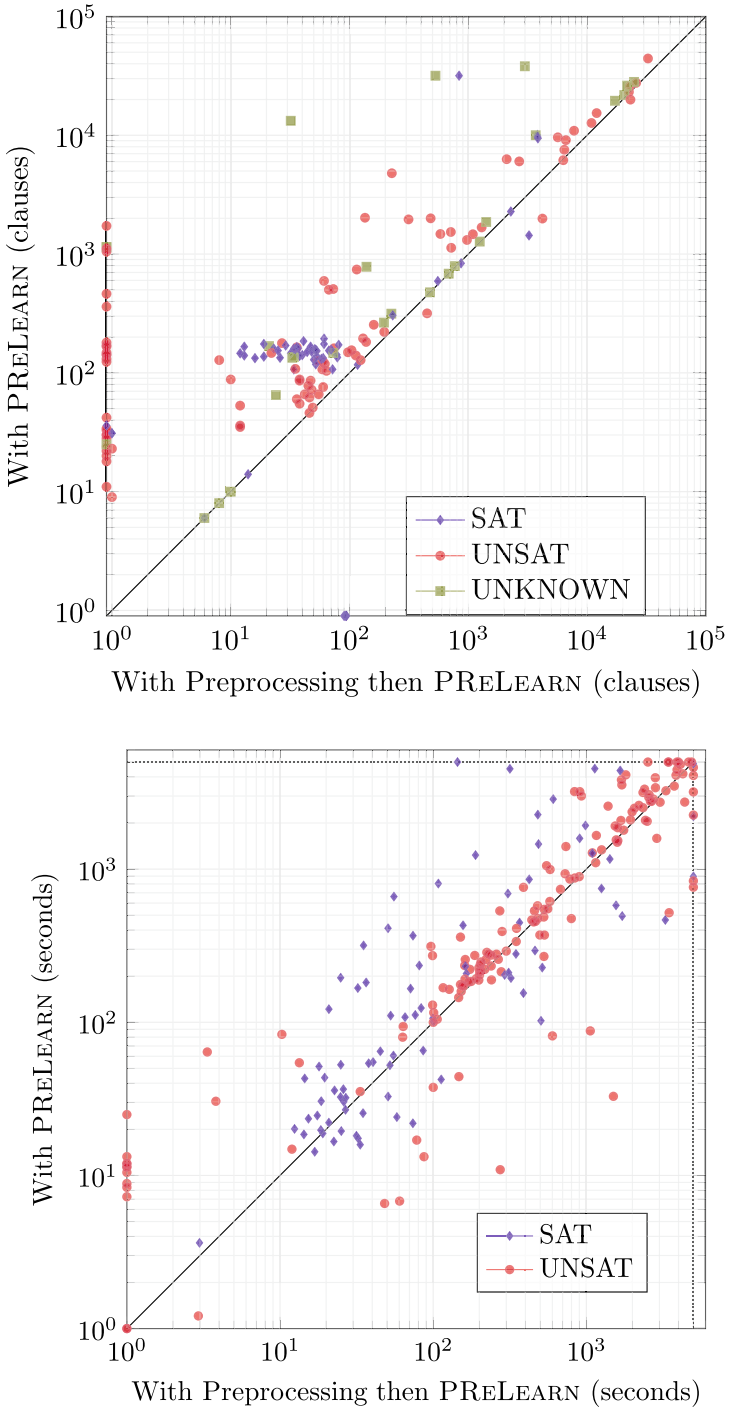

Fig. 9 Number of unit and binary PR clauses learned (top) and execution times (bottom) with PRELEARN after preprocessing vs. without preprocessing

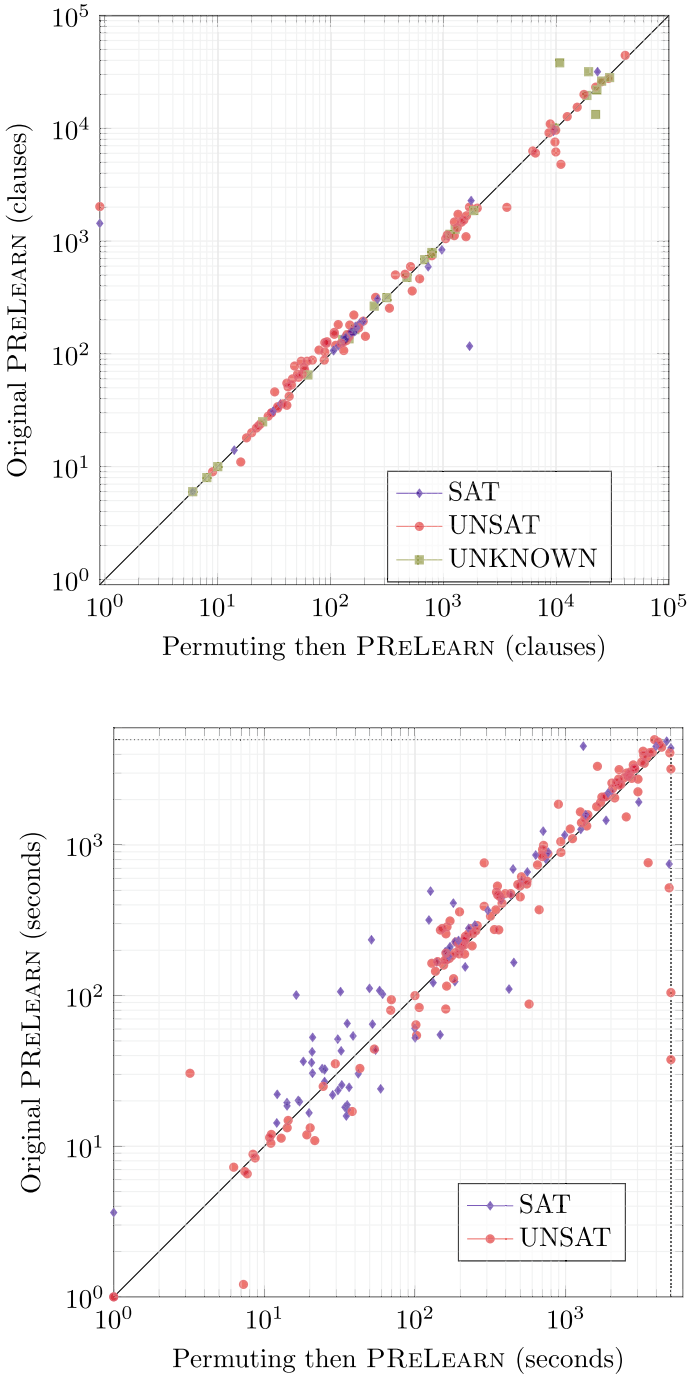

Fig. 10 Unit and binary PR clauses learned (top) execution times (bottom) with PReLearn on the original formulas vs. formulas after permuting variables

8.2 Evaluation with Variable Permutating

For this evaluation, we randomly permuted the variable names in the formula then ran the default PRELEARN configuration on the modified formulas. The learned PR clauses were then permuted back to the original variable naming and appended to the original formula. With this method we could pinpoint the effect of variable permuting on PRELEARN without including noise from variable permuting on solving with KISSAT. We did not consider clause permuting because PRELEARN sorts candidates based on variable naming so that it works independently of clause ordering; while the renaming of variables can impact the order variables are expanded and the choice of candidates. The runtimes consist of the preprocessing with PRELEARN and solving with KISSAT, but do not include the script for permuting variables before and after executing PRELEARN.

The top plot of Fig. 10 shows the number of unit and binary PR clauses learned on the original formula versus the formula with permuted variables when applying PRELEARN. For most instances, permuting variables within the formula had little impact on the number of PR clauses learned. There is an outlier, the SAT point below the diagonal with 100 learned PR clauses on the original formulas and 1000 learned PR clauses on the permuted formulas. There was one instance, BATTLESHIP-13-25-SAT, where in the original formula, the variable with name 1 is the first variable expanded by PRELEARN. After propagating this variable, most other variables in the formula are neighbors of the assignment, so the candidate list is large, and the pruning predicates require nearly all clauses in the formula. Thus, PRELEARN spends the entire 100 seconds looking for candidates that include the variable 1, finding only 100 PR clauses. After permuting variable names, a different variable is selected first. This new variable will not generate as many candidates, and so PRELEARN can explore many more variables and find more PR clauses. This type of problem can arise in any formula where propagating a few variables touches all or most other variables in the formula. To avoid this, a limit can be set on the number of candidates each variable can generate, or on the size of the pruning predicate produced. This would allow PRELEARN to search across the entire formula without getting stuck on candidates involving a small set of variables.

The bottom plot of Fig. 10 shows the execution time on the original formula versus the formula with permuted variables when applying PRELEARN. There are a few UNSAT formulas that are solved without permutation, due to a difference of a few thousand PR clauses not learned after variable permuting. In one of these cases, PRELEARN timed out with and without variable permuting, so an increase in the timeout may have brought the execution times closer together. In general, a small variation in the number of PR clauses added to a formula will negatively impact some formulas and positively impact others, but in most cases will not make a large difference.

9 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 improved the performance of the CDCL solver KISSAT on many benchmarks from past SAT competitions. In addition, we showed that the performance of PRELEARN does not decline after preprocessing a formula or permuting variables names.

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. However, we found that for some problems where PR clauses made the solving time worse, many of the PR clauses were blocked and could be removed by an additional preprocessing tool. Finding a way to filter certain PR clauses as to not harm CDCL heuristics may be a general way to improve the effectiveness of PRELEARN. 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 inprocessing 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. This work was supported by the U.S. National Science Foundation under grant CCF-2108521, and supported in part by a fellowship award under contract FA9550-21-F-0003 through the National Defense Science and Engineering Graduate (NDSEG) Fellowship Program, sponsored by the Air Force Research Laboratory (AFRL), the Office of Naval Research (ONR) and the Army Research Office (ARO).

Author Contributions JER wrote the main manuscript text, implemented the tool, and performed all experiments. MJHH and REB gave advice and provided comments on earlier versions. All authors reviewed the manuscript.

Funding Open Access funding provided by Carnegie Mellon University

Declarations

Conflict of interest The authors declare no competing interests.

References

1. Alekhnovich, M.: Mutilated chessboard problem is exponentially hard for resolution. Theoret. Comput. Sci. **310**(1), 513–525 (2004)
2. Anders, M.: SAT preprocessors and symmetry. In: Theory and Applications of Satisfiability Testing (SAT). Leibniz International Proceedings in Informatics (LIPIcs), vol. 236, pp. 1–1120. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2022)
3. Ansótegui, C., Bonet, M.L., Giráldez-Cru, J., Levy, J., Simon, L.: Community structure in industrial SAT instances. J. Artif. Intell. Res. **66**, 443–472 (2019)
4. Atserias, A., Lauria, M., Nordström, J.: Narrow proofs may be maximally long. ACM Trans. Comput. Logic **17**(3), 1–30 (2016)
5. Audemard, G., Katsirelos, G., Simon, L.: A restriction of extended resolution for clause learning sat solvers. In: Proceedings of the AAAI Conference on Artificial Intelligence, pp. 15–20 (2010)

6. Biere, A., Fazekas, K., Fleury, M., Heisinger, M.: CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling Entering the SAT Competition 2020. unpublished (2020)
7. Biere, A., Fröhlich, A.: Evaluating CDCL variable scoring schemes. In: Theory and Applications of Satisfiability Testing (SAT). LNCS, vol. 9340, pp. 405–422 (2015)
8. Codel, C.R., Reeves, J.E., Heule, M.J.H., Bryant, R.E.: Bipartite perfect matching benchmarks. In: Pragmatics of SAT (2021)
9. Cook, S.A.: A short proof of the pigeon hole principle using extended resolution. SIGACT News **8**(4), 28–32 (1976)
10. 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, Cham (2016)
11. 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, Berlin (2005)
12. Freeman, J.W.: Improvements to propositional satisfiability search algorithms. PhD thesis, University of Pennsylvania, United States of America (1995)
13. Haken, A.: The intractability of resolution. Theoret. Comput. Sci. **39**, 297–308 (1985)
14. 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, Cham (2015)
15. Heule, M.J.H., Hunt, W.A., Wetzler, N.: Trimming while checking clausal proofs. In: Formal Methods in Computer-Aided Design (FMCAD), pp. 181–188 (2013)
16. Heule, M.J.H., Kiesl, B., Biere, A.: Clausal proofs of mutilated chessboards. In: NASA Formal Methods. LNCS, vol. 11460, pp. 204–210. Cham (2019)
17. 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, Cham (2019)
18. 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, Cham (2017)
19. 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)
20. 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, Cham (2016)
21. Heule, M.J.H., Kiesl, B., Biere, A.: Strong extension free proof systems. J. Autom. Reason. **64**, 533–544 (2020)
22. 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, Berlin (2012)
23. Katebi, H., Sakallah, K.A., Markov, I.L.: Symmetry and satisfiability: an update. In: Strichman, O., Szeider, S. (eds.) Theory and Applications of Satisfiability Testing (SAT), pp. 113–127. Springer, Berlin (2010)
24. Lecoutre, C., Roussel, O.: Proceedings of the 2018 XCSP3 Competition, 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.P., Lynce, I., Malik, S.: Conflict-driven clause learning SAT solvers. In: Handbook of Satisfiability, pp. 131–153. IOS Press, Amsterdam (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, pp. 530–535. ACM, New York, NY, USA (2001)
29. Pérez, J.A.N., Voronkov, A.: Generation of hard non-clausal random satisfiability problems. In: Veloso, M.M., Kambhampati, S. (eds.) AAAI Conference on Artificial Intelligence, pp. 436–442. AAAI Press, Palo Alto (2005)
30. Reeves, J.E., Heule, M.J.H., Bryant, R.E.: Preprocessing of Propagation Redundant Clauses, pp. 106–124. Springer, Berlin (2022)
31. 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, Cham (2014)
32. 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)

33. Tseitin, G.S.: On the Complexity of Derivation in Propositional Calculus, pp. 466–483. Springer, Berlin (1983)
34. 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. Springer, Cham (2014)