


# Learning Short Clauses via Conditional Autarkies

Amar Shah , Twain Byrnes , Joseph Reeves , Marijn J. H. Heule 

Carnegie Mellon University, Pittsburgh, PA, USA

{amarshah, binarynews, jereeves, marijn}@cmu.edu

**Abstract**—State-of-the-art Boolean satisfiability (SAT) solvers increasingly use techniques beyond resolution. One of the strongest such techniques is Propagation Redundant (PR) clause learning. Solvers utilizing PR clause learning may admit short proofs for benchmark families with exponentially large resolution proofs, including pigeonhole and mutilated chessboard. However, existing PR clause learning techniques require an NP-hard check; hence, they are computationally expensive and difficult to add to existing tools.

We propose a new technique for learning PR clauses based on conditional autarkies and implement it in the SAT solver CADICAL. Our method is modular, allowing for cross-solver compatibility, and learns PR clauses in linear time. Additionally, we introduce a number of heuristics, including a clause-shrinking technique and filtering to avoid trivial PR clauses, ensuring that our method learns useful clauses. We show that this is competitive with state-of-the-art PR clause learning techniques, and improves performance on a portion of SAT competition benchmarks.

## I. INTRODUCTION

Boolean satisfiability (SAT) solving is a core tool in computer science with applications in program verification [1–4], planning [5, 6], cryptography [7], and mathematics [8–11]. As its usage expands, so too does the need for more powerful and specialized solving techniques.

One such class of techniques is propagation redundant (PR) clause learning [12]. In this technique, a solver generates and adds clauses which are satisfiability-preserving; that is, clauses whose conjunction with the formula is satisfiable if and only if the original formula is satisfiable. The solver aims to learn clauses that drastically shrink the potential solution space, such as clauses that break symmetries.

The power of this technique is tied to the strength of the underlying proof system: propagation redundancy (PR). A more powerful proof system can admit shorter proofs, which can be found and checked faster. Most SAT solvers rely on resolution-based proof systems, which are ineffective for many hard instances. Consider the pigeonhole principle (PHP), which asks if it is possible to fit  $n + 1$  pigeons into  $n$  holes with at most one pigeon per hole. Using PR reasoning, one may learn a PR clause equivalent to the lemma: *pigeon 1 is not in hole 1*. Adding this clause restricts the search space for the first pigeon (and thus the problem), but it preserves satisfiability, as hole 1 is symmetric with all holes.

It is difficult to perform similar reasoning compactly with resolution, and in fact, PHP requires exponentially large resolution proofs [13]. The PR proof system, however, has short proofs for such problems, including a cubic proof (in the number of pigeons) for PHP [12]. PHP frequently occurs as a

subproblem for SAT benchmarks, so it is important to be able to solve it efficiently.

The theoretical power of PR learning comes at a cost: efficiently learning useful PR clauses is a challenge. State-of-the-art techniques such as Satisfiability Driven Clause Learning (SDCL) rely on calling another SAT solver to verify that a candidate clause is PR [14]. Thus, in the worst case, the solver takes exponential time to learn a single PR clause. This makes integration into high-performance solvers difficult and limits their practical impact. To the best of our knowledge, none of the most popular SAT solvers such as CADICAL [15], KISSAT [16], CRYPTOMINISAT [7], or LINGELING [17] support PR clause learning in their main branch.

In this work, we propose a new, efficient approach to learn PR clauses based on conditional autarkies. Intuitively, a conditional autarky provides a way to force the value of certain variables given a set of assumptions, e.g., in PHP if pigeons 3 to  $n + 1$  are not in hole 1 or hole 2 (see Figure 1c for a visual representation), then pigeon 1 can be placed in hole 1 and pigeon 2 can be placed in hole 2. Kiesl et. al. proposed an algorithm for finding conditional autarkies in linear time [18], using them to delete clauses. In our work, we will use conditional autarkies to construct and add PR clauses.

While conditional autarkies present a means for bridging the theoretical gap in identifying PR clauses, the PR clauses they produce are often not immediately useful in real SAT solving applications. There are two major practical limitations: (1) larger PR clauses may not meaningfully constrain the search space, and (2) some smaller PR clauses may be trivial and distract the solver. We solve (1) by introducing a shrinking procedure to extract compact, useful PR clauses, and (2) by introducing a number of heuristics for filtering away trivial PR clauses. Returning to PHP, instead of the conditional autarky described above which produces a clause with at least  $2n$  literals, it is possible through shrinking to learn a binary PR clause either forbidding pigeon 1 in hole 2 or pigeon 2 in hole 1.

We make the following contributions:

- 1) We introduce a method for learning PR clauses in linear time relative to the size of the formula.
- 2) We develop a number of shrinking and filtering heuristics to learn concise and useful PR clauses.
- 3) We implement these techniques in a solver, CAUTICAL<sup>1</sup> (a fork of the state-of-the-art SAT solver CADICAL),

<sup>1</sup>CAUTICAL’s code is available at <https://github.com/amarshah10/cautical>.

and evaluate it on both PHP benchmarks and a suite of benchmarks from the annual SAT competitions.

## II. BACKGROUND

We begin with some SAT preliminaries. Variables  $x_1, x_2, \dots$  take values *true* ( $\top$ ) or *false* ( $\perp$ ). A literal  $l$  is a variable  $x$  or its negation  $\bar{x}$ . A clause  $C$  is a disjunction of literals, e.g.,  $C = x_1 \vee \bar{x}_4 \vee x_6$ . A clause may also be represented by the set of its literals: e.g., the prior clause  $C = \{x_1, \bar{x}_4, x_6\}$ . Hence, we denote that a literal  $l$  occurs in clause  $C$  via  $l \in C$  and the negation of clause is the set of negations of its literals,  $\bar{C} = \{\bar{x}_1, x_4, \bar{x}_6\}$ . A conjunctive normal form (CNF) formula  $\Gamma$  is a conjunction of (disjunctive) clauses. In this paper, all formulas  $\Gamma$  are in CNF.

We use  $\text{var}(\Gamma)$  to represent the set of variables occurring in formula  $\Gamma$ . An *assignment*  $\alpha : V \rightarrow \{\top, \perp\}$  on  $\Gamma$  maps variables  $V \subseteq \text{var}(\Gamma)$  to  $\top$  or  $\perp$ . If  $V = \text{var}(\Gamma)$ , then  $\alpha$  is a *total assignment*. An assignment that is not total is called a *partial assignment*. For this paper, assignment refers to both total and partial assignments. We abuse notation to extend an assignment  $\alpha$  to literals, by denoting  $\alpha(l) = \alpha(x)$  if  $l = x$  and  $\alpha(l) = \neg\alpha(x)$  if  $l = \bar{x}$ .

A formula restricted to an assignment  $\Gamma|_\alpha$  is the formula resulting from mapping variables in the domain of  $\alpha$  to their assignments. We can simplify such a formula by removing *literals* assigned  $\perp$  and *clauses* containing a literal assigned  $\top$  (which are thus satisfied). For example, with formula  $\Gamma = (x_1 \vee \bar{x}_4 \vee x_6) \wedge (x_2 \vee x_3) \wedge \bar{x}_5$  and assignment  $\alpha$  mapping  $x_1$  to  $\perp$  and  $x_2$  to  $\top$ , we have  $\Gamma|_\alpha = (\bar{x}_4 \vee x_6) \wedge \bar{x}_5$ .

A clause  $C = l_1 \vee \dots \vee l_m$  *blocks* an assignment mapping each  $l_i \in C$  to false ( $\perp$ ). An assignment  $\alpha$  *touches* a clause  $C$  if there is a variable  $x$  assigned by  $\alpha$  such that  $x \in C$  or  $\bar{x} \in C$ . We say  $\alpha$  *satisfies*  $C$  if there is some  $l_i \in C$  with  $\alpha(l_i) = \top$ .

Assignment  $\alpha$  *satisfies* formula  $\Gamma$  if it satisfies every clause  $C \in \Gamma$ . If such a satisfying assignment exists, then  $\Gamma$  is *satisfiable*.

**Redundant Clauses:** A clause  $C$  is *redundant* (or *satisfiability-preserving*) with respect to a formula  $\Gamma$  if the formulas  $\Gamma$  and  $\Gamma \wedge C$  are *equisatisfiable*, i.e.,  $\Gamma$  is satisfiable if and only if  $\Gamma \wedge C$  is satisfiable.

In a *clausal proof system*, each step adds or removes a redundant clause  $C$ . The step may contain extra information, such as a boolean witness justifying  $C$ 's redundancy. A list of redundant clauses ending with the empty clause  $\perp$  is a proof of unsatisfiability for formula  $\Gamma$ .

Adding a redundant clause may help a solver by greatly constraining the set of possible solutions, which gives the solver a smaller solution space to search; however, the addition of these clauses could also negatively interact with solver heuristics, increasing solve time.

Clausal proof systems range in complexity, with one of the simplest derivation rules being resolution. Given two clauses,  $C \vee x$  and  $\bar{x} \vee D$ , *resolution* produces the logically implied clause  $C \vee D$ .

*Unit propagation* is a core reasoning technique in a SAT solver. Starting with empty assignment  $\alpha$ , if a formula  $\Gamma$  contains a *unit clause*  $l$ , i.e., a clause with only a single literal, we set  $\alpha(l) = \top$ . Then, this unit is propagated: we consider the unit clauses of the formula  $\Gamma|_\alpha$  and continue this process until no unit clauses remain. If unit propagation terminates with  $\Gamma|_\alpha = \emptyset$ , we say that it derives a conflict.

Given a formula  $\Gamma$  and clause  $C = l_1 \vee \dots \vee l_k$ , we can say  $\Gamma \vdash_1 C$ , read as “ $\Gamma$  implies  $C$  via unit propagation,” if  $\Gamma \wedge \bar{C} \equiv \Gamma \wedge \bar{l}_1 \wedge \dots \wedge \bar{l}_k$  derives a conflict after applying unit propagation. Here,  $C$  is a *reverse unit propagation (RUP)* clause, a simple example of a redundant clause with respect to  $\Gamma$ . For some formula  $\Gamma'$ , we say  $\Gamma \vdash_1 \Gamma'$  if  $\Gamma \vdash_1 C$  for every clause  $C \in \Gamma'$ .

**Definition 1.** [*Propagation Redundant (PR) clauses* [12]] For formula  $\Gamma$ , clause  $C$ , and assignment  $\alpha$  blocked by  $C$ , we say that  $C$  is *propagation redundant (PR)* if there exists a witness assignment  $\omega$  such that  $\omega$  satisfies  $C$  and

$$\Gamma|_\alpha \vdash_1 \Gamma|_\omega.$$

Informally, a clause  $C$  is PR if every assignment that satisfies  $\Gamma$  but falsifies  $C$  can be turned into an assignment that satisfies  $\Gamma \wedge C$ . While this property can be validated in polynomial time (by unit propagation), checking if a clause is PR without hints is NP-complete [12]. Thus, the propagation redundancy (PR) proof system introduces witnesses, which must be provided for proof checking.

PR clauses subsume many classes of redundant clauses, including resolution asymmetric tautologies (RATs) [19], blocked clauses [20], set-blocked clauses [21], and globally-blocked clauses [18].

### A. Conditional Autarkies

**Definition 2** (Autarky [18]). A nonempty assignment  $\alpha$  is an *autarky* for a formula  $\Gamma$  if every clause  $C \in \Gamma$  touched by  $\alpha$  is satisfied.

Simply, an autarky is an assignment that satisfies every clause it touches. Unfortunately, formulas often do not contain autarkies, and they are difficult to find even when they exist. Instead, we use the following weakening of an autarky:

**Definition 3** (Conditional Autarky [18]). A nonempty assignment  $\alpha = \alpha_c \sqcup \alpha_a$  (disjoint union) is a *conditional autarky* for a formula  $\Gamma$  if  $\alpha_a$  is an autarky for  $\Gamma|_{\alpha_c}$ .

Consider, as an example, the formula with aptly named variables  $\Gamma = (c \vee \bar{a}) \wedge (a \vee x) \wedge (\bar{c} \vee \bar{y})$ . We have a conditional autarky  $\alpha = \alpha_c \sqcup \alpha_a$ , with  $\alpha_c = c$  and  $\alpha_a = a$ . Since  $\Gamma|_{\alpha_c} = (a \vee x) \wedge (\bar{y})$  and  $a$  occurs here only positively,  $a$  is an autarky for  $\Gamma|_{\alpha_c}$ . Thus,  $\alpha$  is a conditional autarky.

Conditional autarkies can be very useful for learning satisfiability-preserving clauses as in the following theorem from Kiesl et al. [18]:

**Theorem 1.** Let  $\Gamma$  be a formula and  $\alpha = \alpha_c \sqcup \alpha_a$  be a conditional autarky on  $\Gamma$ , with  $\alpha_c = c_1, \dots, c_n$  and  $\alpha_a = a_1, \dots, a_m$ . Formula  $\Gamma$  and  $\Gamma \wedge \bigwedge_{1 \leq i \leq m} (\bar{c}_1 \vee \dots \vee \bar{c}_n \vee a_i)$  are equisatisfiable.

A solver may thus add any of the  $m$  clauses  $\bar{c}_1 \vee \dots \vee \bar{c}_n \vee a_i$  and preserve satisfiability.

Intuitively, Theorem 1 states that adding this implication preserves satisfiability:

$$[c_1 \wedge \dots \wedge c_n] \rightarrow [a_1 \wedge \dots \wedge a_m]$$

When converted to clausal form, this results in the  $m$  different clauses from the theorem. Indeed, each of these is a PR clause with witness  $\alpha_a$ .

### B. Related Work

Solvers that implement the PR proof system typically use the satisfaction-driven clause learning (SDCL) framework [22], which extends conflict-driven clause learning (CDCL) [23]. After propagating an assignment, they check if the clause  $C$  blocking this assignment is PR by creating a new SAT formula called the *positive reduct*. If the positive reduct is satisfiable, then  $C$  is a PR clause and is added. This was implemented in an extension of the solver LINGELING and was shown to scale well on pigeonhole benchmarks.

Later, two new variants of the positive reduct were proposed for more aggressive pruning of the search space [14]. This allowed SDCL to solve other difficult problems such as mutilated chessboard [24] and Tseitin formulas over expander graphs [25]. This is implemented in a new SDCL solver SADICAL.

PRELEARN is a preprocessing technique for PR clauses [26]. It initially considers many possible clauses and queries SADICAL to see which are PR.

Our work differs as we do not use a *positive reduct* to test if a clause is PR. Instead, our clauses are PR by construction, as they come from a conditional autarky. This has the potential downside that our clause may be large, and thus weak. To remedy this, we apply a shrinking technique, reducing the size of a clause. Additionally, prior techniques are sensitive to the encoding of the problem. Minor changes such as literal and clause reordering can have a large effect on performance. We compare our implementation CAUTICAL to SADICAL and PRELEARN in Section V.

Kiesl et al. [18] first introduced conditional autarkies to identify a class of PR clauses known as globally blocked clauses. They aimed to eliminate globally blocked clauses from a formula to simulate circuit-simplification techniques. Our work adds clauses instead of removing them.

## III. METHODOLOGY

### A. Motivating Example

We use the pigeonhole principle as a motivating example. The problem PHP( $n$ ) asks whether we can put  $n + 1$  pigeons in  $n$  holes such that (1) every pigeon is in a hole, and (2) no hole contains more than one pigeon. This can be encoded as a SAT problem where variable  $x_{i,j}$  represents putting the  $i$ -th pigeon into the  $j$ -th hole. Constraint (1) is encoded as  $\bigvee_{1 \leq j \leq n} x_{i,j}$  for each  $1 \leq i \leq n + 1$  and (2) as  $\bar{x}_{i,j} \vee \bar{x}_{k,j}$  for each  $1 \leq i < k \leq n + 1$  and  $1 \leq j \leq n$ .

Figure 1 provides a visualization where the rows represent the pigeons and the columns represent the holes. The cell in row  $i$  and column  $j$  represents the literal  $x_{i,j}$ . A  $+$  in the  $(i, j)$ -th cell indicates that  $x_{i,j}$  is set to  $\top$ , and a  $-$  symbol indicates  $\perp$ . Thus, constraint (1) asks that each row has at least one  $+$  and constraint (2) asks that each column has at most one  $+$ .

We achieve an  $O(n^3)$  PR proof for PHP( $n$ ), matching the best known result [12]. In Figure 1 we learn the clause  $\bar{x}_{1,2} \vee \bar{x}_{2,1}$ . After learning  $n$  such clauses, we learn the clause  $\bar{x}_{1,2}$ , i.e. “pigeon 1 is not in hole 2.” Since there are  $n+1$  pigeons and  $n$  holes, we must rule out  $O(n^2)$  pigeon-hole pairs, so the proof has size  $O(n^3)$ . We learn these proofs with a very low constant factor and robustness against encoding perturbations (see Subsection V-A).

### B. PR Clause Learning Framework

---

#### Algorithm 1: Learning PR clauses

---

```

1 Function LearnClause ( $\Gamma, \alpha$ ):
2   for  $i \in \text{vars}(\Gamma)$  :
3     for  $j \in \text{vars}(\Gamma)$  :
4       Propagate ( $i$ );
5       Propagate ( $j$ );
6        $\alpha_c, \alpha_a := \text{LeastConditional}(\Gamma, \alpha)$ ;
7        $C := \text{Shrink}(\Gamma, \alpha_c, \alpha_a)$ ;
8       if not Filter ( $C, \Gamma$ ) :
9          $\Gamma := \Gamma \wedge C$ ;
10      Backtrack ();
```

---

We present the general framework for using conditional autarkies to learn PR clauses in Algorithm 1. A conditional autarky is computed based on an assignment, so the algorithm first selects a set of literals to propagate (lines 4, 5), creating the assignment  $\alpha$ . We propagate two literals at a time using nested **for** loops, and undo the propagations after each iteration (backtracking in line 10). In line 6 Algorithm 2 makes a single pass over the formula to generate a conditional autarky, with conditional part  $\alpha_c$  and autarky part  $\alpha_a$ . The PR clause  $C$  is generated from the conditional autarky and shrunk in line 7, and if  $C$  does not pass the usefulness heuristics it is filtered away in line 8. Details regarding design choices and heuristics are found in Section IV, including a discussion of Filter.

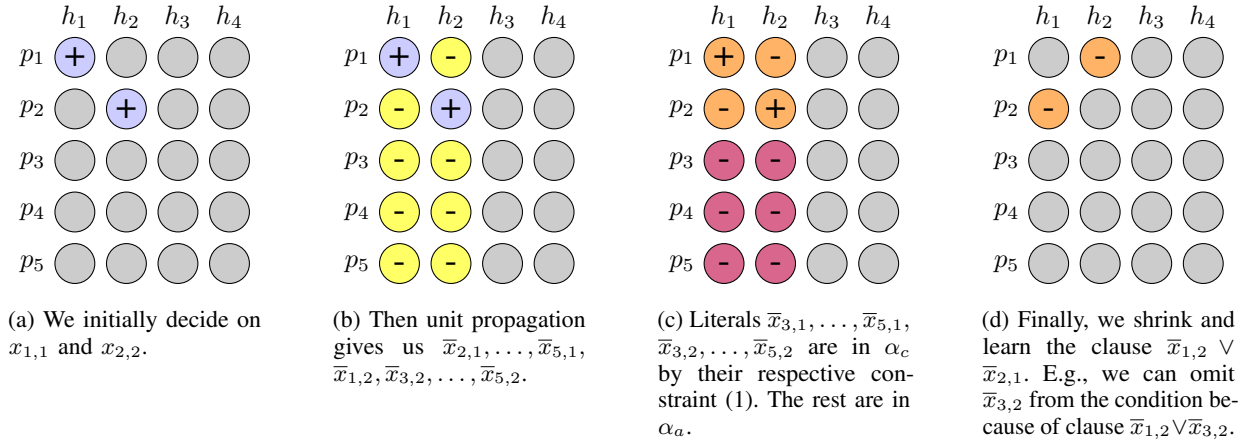


Fig. 1: Learning the clause  $\bar{x}_{1,2} \vee \bar{x}_{2,1}$  for PHP(4)

In the following sections, we will describe PR clause learning and shrinking in the context of our running example PHP(4). We will consider the decisions  $i = x_{1,1}$  and  $j = x_{2,2}$  shown in Figure 1a, with unit propagation shown in Figure 1b.

### C. Learning PR Clauses

As described in Subsection II-A, any assignment can be split into a conditional and a (potentially empty) autarky part,  $\alpha = \alpha_c \sqcup \alpha_a$ , and the following clauses are PR:  $\bigvee_{c \in \alpha_c} \bar{c} \vee a$  for any  $a \in \alpha_a$ . In order to produce smaller PR clauses, we use Algorithm 2 from Kiesel et al. [18] to find the unique smallest possible  $\alpha_c$ .

---

**Algorithm 2:** Unique minimal  $\alpha_c$  in  $\alpha = \alpha_c \sqcup \alpha_a$

---

```

1 Function LeastConditional( $\Gamma, \alpha$ ):
2    $\alpha_c := \emptyset$ ;
3   for  $C \in \Gamma$  :
4     if  $\alpha$  touches  $C$  without satisfying  $C$  :
5        $\alpha_c := \alpha_c \cup (\alpha \cap \bar{C})$ ;
6   return  $\alpha_c, \alpha \setminus \alpha_c$ ;
```

---

Algorithm 2 returns  $\alpha_c \sqcup \alpha_a$ , which is a conditional autarky, as every clause that  $\alpha_a$  touches is satisfied by a literal in  $\alpha$ .

Additionally,  $\alpha_c$  is minimal: for any other conditional autarky  $\alpha = \alpha'_c \sqcup \alpha'_a$ , it must be the case that  $\alpha_c \subseteq \alpha'_c$  since for each clause that is touched but not satisfied, we add to  $\alpha_c$  all literals from the assignment that touch and do not satisfy this clause. These literals must be in  $\alpha'_c$ , since otherwise  $\alpha'_a$  will touch a clause that is not satisfied by  $\alpha$ , violating the conditional autarky property.

Running Algorithm 2 on the assignment from Figure 1b gives the conditional part (red) and autarky part (orange) in Figure 1c. The assigned literals from pigeons 3, 4, and 5 appear in  $\alpha_c$  because they touch but do not satisfy the constraint (1) clauses for the respective pigeons stating that every pigeon is

in a hole. However, the constraint (1) clauses are satisfied for pigeons 1 and 2, along with the touched constraint (2) clauses, placing the pigeons 1 and 2 literals in  $\alpha_a$ . Intuitively, if pigeons 3, 4, and 5 are not in holes 1 or 2, then pigeon 1 can be placed in hole 1:  $x_{3,1} \vee x_{3,2} \vee x_{4,1} \vee x_{4,2} \vee x_{5,1} \vee x_{5,2} \vee x_{1,1}$  and pigeon 1 can be kept out of hole 2:  $x_{3,1} \vee x_{3,2} \vee x_{4,1} \vee x_{4,2} \vee x_{5,1} \vee x_{5,2} \vee \bar{x}_{1,2}$  (likewise for pigeon 2 but with the holes swapped). The shrinking technique in the following section will help reduce the size of these large PR clauses.

### D. Shrinking PR Clauses

The PR clause derived from a conditional autarky can be too weak to effectively reduce the search space. Shrinking the PR clause, i.e., removing literals from the clause via resolution, will strengthen its pruning power.

At a high-level, we will generate two sets:  $C_0 \subseteq \alpha_c$  and  $A_0 \subseteq \alpha_a$ , and show that we can learn the PR clause  $\bigvee_{c \in C_0} \bar{c} \vee \bigvee_{a \in A_0} a$ .

First we start with  $C_0$ , the set of literals in the conditional part that are inconsistent with any literal in the autarky part. Two literals  $l_i$  and  $l_j$  are inconsistent in a formula  $\Gamma$  if  $\Gamma \wedge l_i \vdash \bar{l}_j$ , meaning the clause  $\bar{l}_i \vee \bar{l}_j$  is RUP. Formally,  $C_0 = \{c_j \in \alpha_c \mid \exists a_i \in \alpha_a \text{ s.t. } \Gamma \wedge \bar{a}_i \vdash c_j\}$ , with  $a_i$  appearing negated in the inconsistency check so that we can perform a resolution between the derived binary clause  $a_i \vee c_j$  and the original PR clause  $C$ .

Next we generate  $A_0$  which is a subset of the autarky literals that are together inconsistent with the literals in  $C_0$ . So, for each  $c \in C_0$  there exists an  $a \in A_0$  such that  $\Gamma \wedge \bar{a} \vdash c$ . There always exists at least one  $A_0$ , namely  $\alpha_a$  which is used to define  $C_0$ , but if a smaller  $A_0$  exists it can be used to shrink the size of the PR clause.

**Theorem 2.** *Let  $\Gamma$  be a formula and  $\alpha = \alpha_c \sqcup \alpha_a$  be a conditional autarky on  $\Gamma$ , with  $\alpha_c = c_1, \dots, c_n$  and  $\alpha_a = a_1, \dots, a_m$ . Let  $A_0 \subseteq \alpha_a$  be non-empty. Let  $C_0 = \{c_j \in \alpha_c \mid$*

$\exists a_i \in A_0$  s.t.  $\Gamma \wedge \bar{a}_i \vdash_1 c_j$ . Then formula  $\Gamma$  is satisfiable if and only if  $\Gamma \wedge (\bigvee_{c \in C \setminus C_0} \bar{c} \vee \bigvee_{a \in A_0} a)$  is satisfiable.

*Proof.*  $\Leftarrow$ : This is immediate

$\Rightarrow$ : From Theorem 1,  $\Gamma$  is satisfiable implies  $\Gamma \wedge (\bigvee_{c \in C} \bar{c} \vee a)$  is satisfiable for any  $a \in \alpha_a$ . We will pick an  $a \in A_0$ . Hence,  $\Gamma \wedge (\bigvee_{c \in C} \bar{c} \vee \bigvee_{a \in A_0} a)$  is satisfiable since we are weakening a clause.

By the definition of inconsistency, for each  $c \in C_0$  there exists an  $a \in A_0$  such that  $\Gamma \wedge \bar{a} \vdash_1 c$ , allowing us to derive the RUP clause  $a \vee c$ . Each binary clause can be resolved with  $(\bigvee_{c \in C} \bar{c} \vee \bigvee_{a \in A_0} a)$ , producing  $(\bigvee_{c \in C \setminus C_0} \bar{c} \vee \bigvee_{a \in A_0} a)$ .  $\square$

In fact,  $\bigvee_{c \in C \setminus C_0} \bar{c} \vee \bigvee_{a \in A_0} a$  is a PR clause with witness  $\alpha_a$ .

**Greedy Set Cover.** We can interpret the problem of finding the smallest possible  $A_0$  as a set cover problem where each literal  $a \in \alpha_a$  defines the set  $SETS(a) = \{c \in \alpha_c \mid \Gamma \wedge \bar{a} \vdash_1 c\}$ . Finding the minimum set cover is NP-hard, so we instead approximate using a greedy algorithm, returning a set cover at most roughly  $(\ln |\alpha_c| + 1) \times$  the size of the smallest set cover [27].

The greedy algorithm initializes  $C_0$  as empty, and in each iteration finds the  $a \in \alpha_a$  that generates the largest set  $SETS(a) \cap (\alpha_c \setminus C_0)$ , adding  $a$  to  $A_0$  and adding  $SETS(a)$  to  $C_0$ . The algorithm terminates once all sets in  $\{SETS(a) \cap (\alpha_c \setminus C_0) \mid a \in \alpha_a\}$  are empty. Notice that oftentimes,  $C_0 \subsetneq \alpha_c$ , i.e. we do not achieve a complete cover of  $\alpha_c$ .

---

**Algorithm 3:** Algorithm finding  $A_0$

---

```

1 Function Shrink( $\Gamma, \alpha_a, \alpha_c$ ):
2    $SETS := \text{init Array}[\text{len}(\alpha_a)]$ ;
3   for  $i \in \text{range}(\alpha_a)$  :
4     Propagate( $\alpha_a[i]$ );
5      $\text{implied} := \{\}$ ;
6     for  $c \in \alpha_c$  :
7       Propagate( $\bar{c}$ );
8       if  $\text{unsat}$  :
9          $\text{implied} := \text{implied} \cup \{c\}$ ;
10      Backtrack(1);
11       $SETS[i] := \text{implied}$ ;
12 return GreedySetCover( $SETS$ );
```

---

In Algorithm 3, we describe our process for calculating  $A_0$ . We initialize  $SETS$  as an array (line 2), iterate the counter  $i$  through  $\alpha_a$  (line 3), populating  $SETS[i]$  with the set of literals  $c \in \alpha_c$  such that  $\Gamma \wedge \bar{a}_i \vdash_1 c$  (lines 4-11). Finally, we apply GreedySetCover to get a small set  $A_0$  that covers as much of  $C$  as possible.

Returning to the pigeonhole example,  $C_0$  contains all of the literals in  $\alpha_c$  because of the binary clauses in constraint (2). The smallest possible  $A_0$  includes  $\bar{x}_{2,1}$  and  $\bar{x}_{1,2}$ , whose sets cover  $\bar{x}_{3,1}, \bar{x}_{4,1}, \dots, \bar{x}_{5,1}$  and  $\bar{x}_{3,2}, \bar{x}_{4,2}, \dots, \bar{x}_{5,2}$  respectively. Whereas, the sets from  $x_{1,1}$  and  $x_{2,2}$  are empty (they are not

inconsistent with the other literals). Therefore, we can learn the clause  $\bar{x}_{2,1} \vee \bar{x}_{1,2}$ , shown in Figure 1d.

#### IV. IMPLEMENTATION

We implement our technique in CAUTICAL (a fork of CADICAL). We choose CADICAL as it has shown strong performance in the SAT Competition. For instance, a fork of CaDiCaL won in 2023 [28]. Our techniques can be implemented in any CDCL SAT solver, but we leave this as future work.

By default, CAUTICAL spends 30 seconds searching for PR clauses in a preprocessing step. After this time limit, the solver exits and commences normal solving, regardless of whether or not it has found any PR clauses. This adds  $\sim 800$  lines of C++ code to CADICAL and is implementable within any CDCL SAT solver.

We discuss three important design decisions in CAUTICAL:

- 1) **shrink**: As discussed in Subsection III-D, we may shrink a clause using the techniques inspired by greedy set cover.
- 2) **filter-triv**: We filter out clauses that are trivial, i.e. if  $\Gamma \vdash_1 C$  for some clause  $C$ .
- 3) **filter-long**: We filter out clauses that are longer than some set length  $l$ . We pick  $l = 2$  as a default and thus only learn binary and unit clauses.

The filter function in Algorithm 1 combines **filter-triv** and **filter-long**, disallowing trivial clauses or clauses with length greater than 2.

Additionally, we describe three natural design decisions that we did not make:

- 1) **longer-preprocess**: We choose 30 seconds as the default preprocessing time. However, we could choose a longer time limit, for instance 100 seconds.
- 2) **order-i**: On Algorithm 1 line 2, by default we pick  $i$  randomly. However, we could choose  $i$  based on some ordering. For instance, we could order literals  $i$  by how frequently they occur in the original formula.
- 3) **select-j**: On Algorithm 1 line 3, we iterate through all possible literals  $j$ . However, we could choose some subset of possible literals  $j$ . One such example is to pick  $j$  from the neighbors of  $i$ . This is implemented in PRELEARN. The neighbors of  $i$  include any literal  $j$  that belongs to the same clause as a literal fixed by unit propagation on  $i$  (including  $i$  itself).

We evaluate these heuristics in Subsection V-D and find that **shrink**, **filter-triv**, and **filter-long** are all beneficial. On the other hand **longer-preprocess**, **order-i**, and **select-j** are not beneficial and can sometimes be harmful.

#### V. EVALUATION

In this section, we empirically evaluate our technique against other PR clause learning techniques. In doing so, we aim to answer the following research questions:

- RQ1 Can our approach provide a speedup on certain benchmark families?



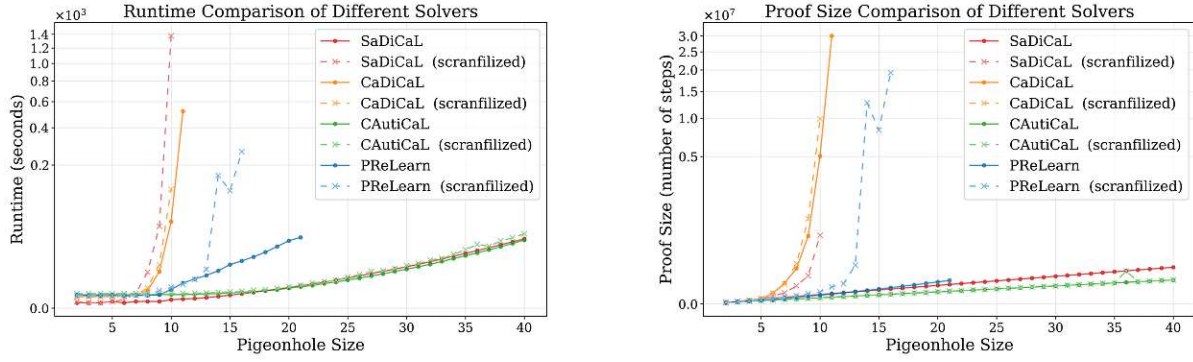


Fig. 2: Comparison of CAUTICAL, CADICAL, SADICAL, and PRELEARN on pigeonhole principle benchmarks up to size 40. The y-axis is on a cube root scale. The performance of a solver on the original benchmark is shown with a solid line. The median of 5 scanfilized queries is shown with a dashed line. If a solver times out on a query in 5000s, it is not shown.

RQ2 Is our approach less sensitive to encoding choices compared to other PR learning techniques?

We compare the two main tools learning PR clauses: SADICAL (based on SDCL) and PRELEARN (a preprocessing technique that calls SADICAL). To be consistent with our approach, we run PRELEARN with its default settings for 30 seconds, then solve the preprocessed formula with CADICAL. We also compare to CADICAL as a baseline with no PR clause learning. We check all proofs of unsatisfiability from CAUTICAL using `dpr-trim` [29].

All experiments were performed in the Anvil Supercomputing Center on nodes with 128 cores and 2 GB RAM per core [30]. We ran 64 experiments in parallel per node with a 5,000 second timeout, the default timeout for the SAT competition.

In Subsection V-A, we compare all approaches on the pigeonhole principle, evaluating runtime, proof length, and sensitivity to the encoding of the formula. In Subsection V-B, we evaluate the solvers on benchmarks from the '22, '23, and '24 SAT competition's main tracks [28, 31, 32]. In Subsection V-C, we highlight certain benchmark families that benefit from PR clause learning. In Subsection V-D, we evaluate the benefit of different heuristic choices in CAUTICAL. Finally, we conclude in Subsection V-E with a discussion of how performed on RQ1 and RQ2.

#### A. Pigeonhole results

Approaches based on SDCL, such as SADICAL, are successful for learning  $O(n^3)$  proofs for the pigeonhole principle, but are very sensitive to the encoding of the formula. We compare the solvers on pigeonhole principle from PHP(2) to PHP(40) and plot these results in Figure 2. As the expected best-behavior is cubic, we use a cube root scale for the y-axis.

As expected, CADICAL grows exponentially, while SADICAL and CAUTICAL scale cubically in both runtime and proof size. Significantly, CAUTICAL is able to learn  $3.59\text{--}3.64\times$  shorter proofs compared to SADICAL. PRELEARN

scales cubically on small formulas, but for PHP(22) and larger, will not learn enough useful PR clauses in the preprocessing step and will timeout after spending the rest of its time running CADICAL.

Additionally, we evaluate all solvers on scanfilized variations of the pigeonhole principle. Scanfilization is a technique for generating an satisfiability-equivalent formula [33]. We use the tool `scanfilize` [33] with the options permuting variables, permuting clauses, and flipping literals (with probability 0.5) all turned on. We run each solver on 5 scanfilized variations for each benchmark and take the median runtime and proof size. This is shown in Figure 2 with dashed lines.

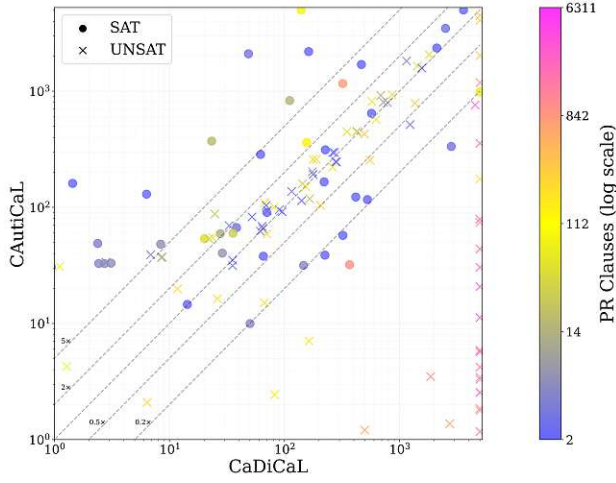
SADICAL and PRELEARN exhibit an exponential trend for runtime and proof size on the scanfilized benchmarks. SADICAL will spend all its time in the main SDCL loop not learning enough useful clauses. PRELEARN will learn some useful PR clauses in preprocessing, but not enough to sufficiently shrink the search space for formulas larger than PHP(16).

On the other hand, CAUTICAL almost matches its non-scanfilized performance, demonstrating that it learns useful PR clauses regardless of the encoding.

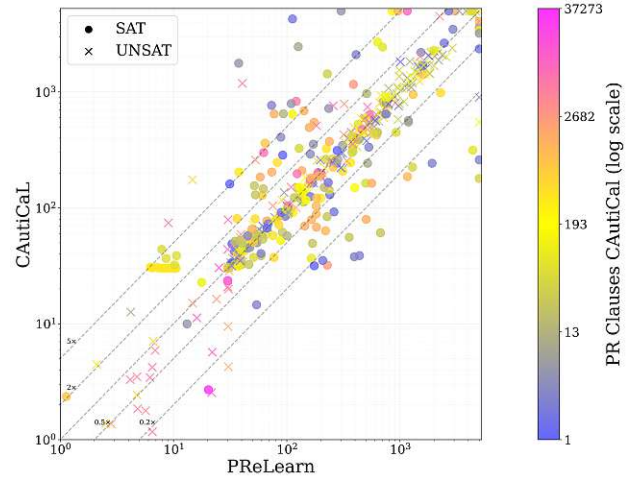
#### B. SAT competition results

We compare the performance of CAUTICAL with CADICAL and PRELEARN on the benchmarks from the '22, '23, and '24 SAT competition's main tracks [28, 31, 32]. We remove duplicates and exclude all benchmarks with more than twenty million clauses as these are out of scope for our technique. This gives us a total of 1,089 benchmarks. We exclude SADICAL from our evaluation as it only solves 22 of these benchmarks.

Table I shows the number of instances solved by each solver. Out of the total number solved, it shows the number of formulas for which PRELEARN and CAUTICAL learn additional PR clauses, improve upon CADICAL by at least 5%, and solve a formula that CADICAL does not solve.



(a) Comparing CAUTiCaL to CaDiCaL.



(b) Comparing CAUTiCaL to PRELearn.

Fig. 3: Performance comparison of CAUTiCaL with PRELearn and CaDiCaL on SAT competition benchmarks. On each graph, we filter out all benchmarks where neither solver learns any PR clauses. The color indicates the number of PR clauses learnt by CAUTiCaL.

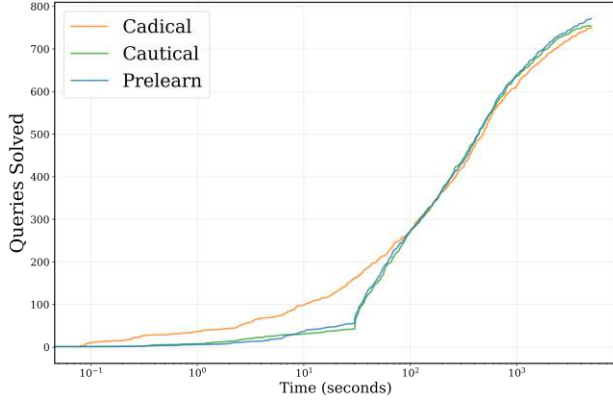


Fig. 4: The number of formulas solved by CAUTiCaL, PRELearn, and CaDiCaL.

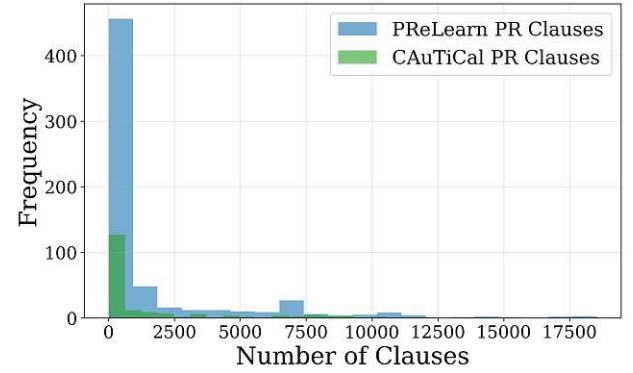


Fig. 5: The size of PR clauses learnt by CAUTiCaL and PRELearn

We divide the benchmarks based on number of clauses (0-10k or 10k-20M) and status (SAT or UNSAT). The number of clauses is a good indicator of the type of benchmark, with many hard combinatorial problems containing fewer than ten thousand clauses.

Figure 3 shows CAUTiCaL and CaDiCaL’s performance relative to CaDiCaL. Figure 3a shows that CAUTiCaL learns a large number of PR clauses for formulas which it solves quickly and CaDiCaL times out on. Figure 3b shows that PRELearn also solves a number of formulas that CAUTiCaL cannot, but CAUTiCaL typically performs better on formulas where it learns many PR clauses.

We find that CAUTiCaL can show a performance improvement or degradation on formulas where it does not learn any PR clauses. This is because as CAUTiCaL runs, it updates internal data structures such as watched literals, clause occurrence lists, and variable phases.

For instance, during the initial phase of searching for PR

clauses, the solver will decide on a literal. Unit propagation of this literal may lead to a conflict, and the solver can learn the unit clause without any autarky/PR reasoning. Unit clauses are not stored or treated as learned clauses inside CaDiCaL, but instead the literal becomes “fixed,” i.e., the solver treats the literal as always true. For instance, this happens on the three satcoin benchmarks where CAUTiCaL shows the most improvement (see Figure 6).

PAR-2 score is a standard metric used to evaluate the performance of solvers. It is evaluated as the sum of the runtimes of solved instances and twice the timeout of unsolved instances. On this dataset, CaDiCaL has a PAR-2 score of 3522 seconds, PRELearn has a PAR-2 score of 3331 seconds, and CAUTiCaL has a PAR-2 score of 3442 seconds.

Figure 4 shows the distribution of benchmarks solved by each solver. Both PRELearn and CAUTiCaL lag behind CaDiCaL during the 30 second preprocessing stage, but eventually solve

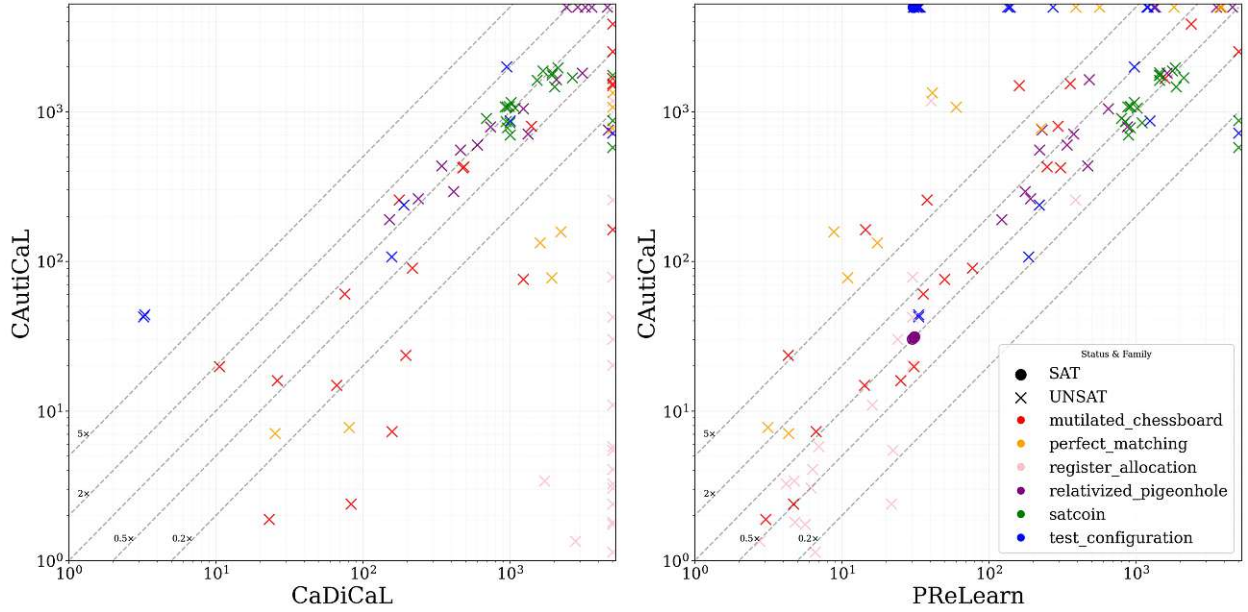


Fig. 6: Comparing CAUTICAL with CADICAL and PRELEARN on various benchmark families.

TABLE I: Number of formulas that CADICAL, PRELEARN and CAUTICAL solve, divided between 0-10k clause and 10k-20M clause formulas, and SAT and UNSAT formulas. For PRELEARN and CAUTICAL we include the number of benchmarks with PR clauses learnt, more than 50 PR clauses learnt, improved on CADICAL by 5% or more; and solved that CADICAL did not solve.

	0–10k clause		10k–20M clause		Total
	SAT	UNSAT	SAT	UNSAT	
CADICAL Solved	54	73	319	303	749
PRELEARN					
Total	52	90	322	307	771
Learn PR clause	40	73	179	145	431
Learn > 50 PR clauses	22	51	104	79	256
Improve on CADICAL	11	42	57	36	146
Unique from CADICAL	1	17	6	6	30
CAUTICAL					
Total	52	87	317	298	754
Learn PR clause	16	58	30	35	139
Learn >50 PR clauses	1	39	6	11	57
Improve on CADICAL	23	48	89	59	219
Unique from CADICAL	0	18	9	9	36

more benchmarks. In the end, PRELEARN solves the most formulas at 771, followed by CAUTICAL at 754, and CADICAL at 749.

CAUTICAL exercises more selective PR clause learning techniques, only learning PR clauses for 139 formulas, while PRELEARN learns PR clauses for 431 formulas (see Figure 5). On those formulas, CAUTICAL improves CADICAL’s runtime by 5% or more on 29.9% of benchmarks, while PRELEARN improves it on 24.7% of benchmarks.

### C. Discussion of Benchmark Families

We identify six benchmark families for which PR clauses perform well. We choose them based on prior work [26] and our experiments on SAT competition benchmarks:

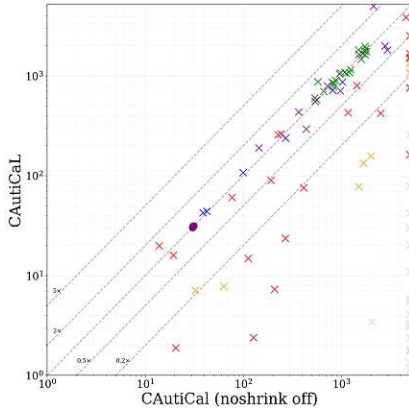
- 1) *mutilated-chessboard*: famous problem asking if one can use 2-by-1 tiles to cover an  $2n$ -by- $2n$  chessboard with opposite corners removed. This is difficult for resolution [34], but there exists  $O(n^3)$  PR proofs [24].
- 2) *perfect\_matching*: generalization of the pigeonhole principle and mutilated chessboard problems with various at-most-one constraints [35].
- 3) *register\_allocation*: the graph coloring problem generated by simulating register allocation on individual Python functions [36].
- 4) *relativized\_pigeonhole*: generalization of the pigeonhole principle where we place  $n + 1$  pigeons in  $n$  holes with  $k$  nesting places.
- 5) *satcoin*: variant of a bitcoin mining problem [37].
- 6) *test\_configuration*: is there a list of configurations of size  $k$  that covers every pairwise combination of configurations of a SAT solver [38].

TABLE II: Overview of benchmark families used in evaluation

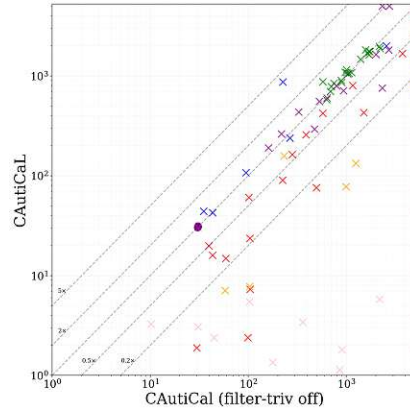
Benchmark Family	Number of Clauses	CAUTICAL # PR
mutilated-chessboard	900-3K	115-351
perfect_matching	300-1K	183-447
register_allocation	1K-23K	671-3322
relativized_pigeonhole	3K-2M	469-9091
satcoin	600K-600K	0-0
test_configuration	31K-64K	0-0

Figure 6 compares the performance of CAUTICAL with

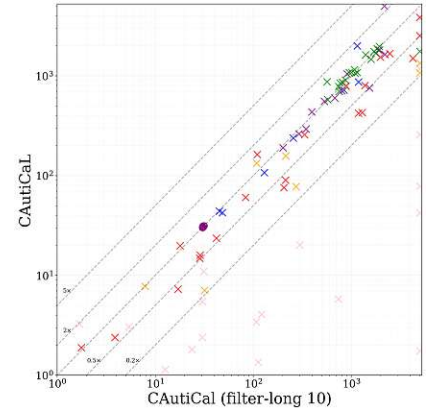




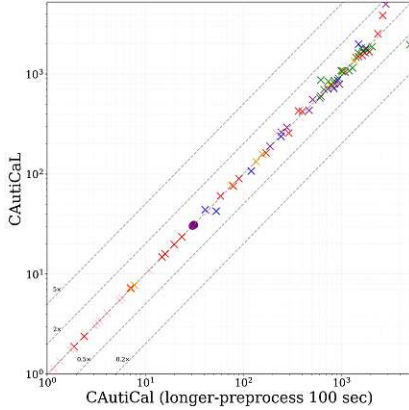
(a) CAUTICAL compared to CAUTICAL with shrink turned off



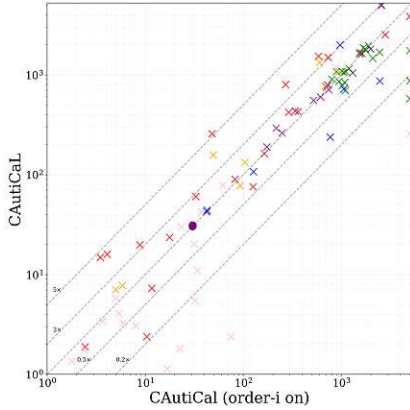
(b) CAUTICAL compared to CAUTICAL with filter-triv turned off



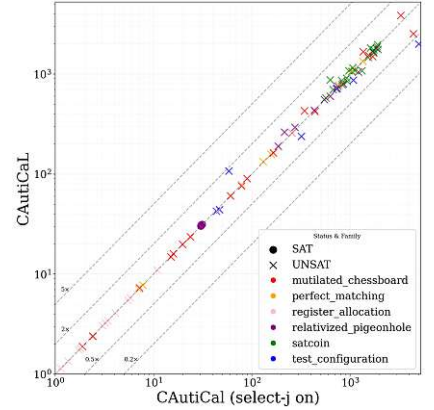
(c) CAUTICAL compared to CAUTICAL with filter-long set to 10



(d) CAUTICAL compared to CAUTICAL with longer-preprocess set to 100 seconds



(e) CAUTICAL compared to CAUTICAL with order-i turned on



(f) CAUTICAL compared to CAUTICAL with select-j turned on

Fig. 7: Performance comparison of CAUTICAL with various heuristics turned on and off

CADICAL and PRELEARN on these benchmark families. We do not evaluate SADICAL as it does very poorly on SAT competition benchmarks. Table II provides a brief overview of the families: the number of clauses in the original formulas (up to 1 significant digit) and the number of PR clauses learnt by CAUTICAL.

CAUTICAL compares favorably to CADICAL on all benchmark families, showing especially large speedups on *register\_allocation*, *mutilated\_chessboard* and *perfect\_matching*.

PRELEARN exhibits large speedups over CAUTICAL on *test\_configuration* and *perfect\_matching*. CAUTICAL performs better on smaller instances of *register\_allocation* and *mutilated\_chessboard*, while PRELEARN performs better on larger instances. CAUTICAL can also solve instances of *satcoin* that PRELEARN cannot.

#### D. Heuristics

Figure 7 shows the performance of CAUTICAL with different heuristics (discussed in Section IV) turned on and off. We evaluate on the different benchmark families discussed in Subsection V-C.

First we consider turning three optimizations off. Figure 7a compares CAUTICAL to CAUTICAL with *shrink* turned off, i.e. we do not shrink PR clauses. Figure 7b compares CAUTICAL to CAUTICAL with *filter-triv* off, i.e. we do not filter trivial clauses. Figure 7c compares CAUTICAL to CAUTICAL with *filter-long* set to 10, i.e. we filter out clauses of size  $> 10$  (as opposed to 2 by default).

When we disable any of three optimizations, the performance is significantly worse, especially on *perfect\_matching*, *mutilated-chessboard*, and *register-allocation* benchmarks.

Next, we consider turning three potential “optimizations” on. Figure 7d sets *longer-preprocess* to 100 seconds (as opposed to 30 seconds by default). Figure 7e turns on *order-i*, propagating the first literal  $i$  ordered by which literals occurs most frequently in the original formula. Finally, Figure 7 turns

on `select-j`, picking  $j$  the second literal by whether it “touches” the first literal  $i$ .

Enabling any of these three “optimizations” does not improve performance and in the case of `order-i`, it slightly hurts performance.

#### E. Research Questions

To conclude, we discuss the two research questions.

For RQ1, CAUTICAL decisively outperforms all solvers except SADICAL on the pigeonhole benchmarks. On SAT competition benchmarks, CAUTICAL improves CADICAL’s PAR-2 score by 2.2%. PRELEARN performs even better, improving CADICAL’s PAR-2 score by 5.4%. However, on formulas where CAUTICAL learns a PR clause, it improves CADICAL’s runtime by 5% or more on 29.9% of benchmarks. On formulas where PRELEARN learns a PR clause, it improves CADICAL’s runtime by 5% or more on 24.7% of benchmarks.

For RQ2, CAUTICAL is the only solver to solve all pigeonhole formulas after scanfilization. Indeed, scanfilization has a negligible effect on CAUTICAL’s performance.

We experienced some success with robustness on the SAT competition benchmarks, but not as significant as for pigeonhole. For instance, in the initial stage when propagating first on literal  $i$  (see Algorithm 1), we randomly pick the order for literal  $i$ .

Prior PR learning approaches such as PRELEARN used specific orderings of literals to their advantage. We evaluated such an ordering in Figure 7e and found that it did not make a difference for CAUTICAL.

#### VI. FUTURE WORK

We believe conditional autarkies provide a route to incorporate PR learning as a regular part of state-of-the-art SAT solvers such as CADICAL. However, there are a few important limitations that need to be addressed first.

- 1) **Improving the algorithms:** The heuristics in CAUTICAL are designed to be modular and easy to modify. However, they can be directly incorporated into the main algorithm. For instance, we could embed the binary clause filter by modifying the greedy set cover to discard clauses with more than 2 literals.
- 2) **Shortest proofs for specific families:** Prior work has shown  $O(n^3)$  PR proofs of mutilated chessboard [24]. While CAUTICAL learns useful clauses to speed up solving for mutilated chessboard, it is unknown whether conditional autarkies can match the  $O(n^3)$  complexity. Similar questions exist for other well-known families such as Tseitin formulas over expander graphs [25, 39].
- 3) **Learning PR clauses with an inprocessing step:** We experimented with learning PR clauses via inprocessing (instead of preprocessing). However, the current PR proof system defines a PR clause as in Definition 1 where  $\Gamma$  is the set of all clauses in the formula, including the learnt, redundant clauses. However, not all redundant clauses

must be considered. This could produce shorter clauses via conditional autarky. However, to realize this, we must modify the PR proof checker.

#### VII. CONCLUSION

PR clause learning is effective for learning short proofs for difficult problems. However, current PR clause learning techniques require an NP-hard check. We solve this by providing a technique that learns PR clauses in a linear time preprocessing step. We provide clause shrinking and filtering techniques to ensure that we learn useful PR clauses.

Our implementation, CAUTICAL, provides short PR proofs for the pigeonhole principle matching the best possible results. While prior work is only effective on specific pigeonhole encodings, CAUTICAL finds these proofs even when the formula is scanfilized.

Additionally, CAUTICAL is effective on a number of benchmarks from the SAT competition, including most of the families identified in Subsection V-C.

In the future, we hope that PR clause learning can find its way into the main branch of a popular SAT solver. We believe this work is an effective step in this direction.

#### VIII. ACKNOWLEDGEMENTS

We thank the anonymous reviewers for their helpful comments and suggestions. This work was supported by funding from AFRL and DARPA under Agreement FA8750-24-9-1000. Joseph Reeves was 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).

#### REFERENCES

- [1] N. Rungta, “A billion SMT queries a day,” 2022.
- [2] A. Gupta, M. K. Ganai, and C. Wang, “SAT-based verification methods and applications in hardware verification,” in *Formal Methods for Hardware Verification*, 2006.
- [3] A. R. Bradley, “Understanding IC3,” in *Theory and Applications of Satisfiability Testing (SAT)*, 2012.
- [4] E. Clarke, A. Biere, R. Raimi, and Y. Zhu, “Bounded model checking using satisfiability solving,” *Formal Methods in System Design*, vol. 19, pp. 7–34, 2001.
- [5] J. Rintanen, “Heuristics for planning with SAT,” in *Principles and Practice of Constraint Programming (CP)*, D. Cohen, Ed., 2010.
- [6] H. Kautz and B. Selman, “Planning as satisfiability,” in *Proceedings of the 10th European Conference on Artificial Intelligence (ECAI)*, 1992.
- [7] M. Soos, K. Nohl, and C. Castelluccia, “Extending SAT solvers to cryptographic problems,” in *Theory and Applications of Satisfiability Testing (SAT)*, 2009.
- [8] B. Subercaseaux and M. J. H. Heule, “The packing chromatic number of the infinite square grid is 15,” in

*Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2023.

- [9] M. J. H. Heule, O. Kullmann, and V. W. Marek, “Solving and verifying the Boolean Pythagorean triples problem via cube-and-conquer,” in *Theory and Applications of Satisfiability Testing (SAT)*, 2016.
- [10] J. Brakensiek, M. Heule, J. Mackey, and D. Narváez, “The resolution of Keller’s conjecture,” *Journal of Automated Reasoning (JAR)*, vol. 66, no. 3, pp. 277–300.
- [11] M. J. H. Heule and M. Scheucher, “Happy ending: An empty hexagon in every set of 30 points,” in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2024.
- [12] M. J. H. Heule, B. Kiesl, and A. Biere, “Short proofs without new variables,” in *Conference on Automated Deduction (CADE)*, 2017.
- [13] A. Haken, “The intractability of resolution,” *Theoretical Computer Science*, vol. 39, pp. 297–308, 1985.
- [14] M. J. H. Heule, B. Kiesl, and A. Biere, “Encoding redundancy for satisfaction-driven clause learning,” in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2019.
- [15] A. Biere, T. Faller, K. Fazekas, M. Fleury, N. Froleyks, and F. Pollitt, “Cadical 2.0,” in *Computer Aided Verification (CAV)*, 2024.
- [16] A. Fleury and M. Heisinger, “Cadical, kissat, paracooba, plingeling and treengeling entering the SAT competition 2020,” *SAT Competition*, 2020.
- [17] A. Biere, “Lingeling, plingeling, picosat and precosat at SAT race 2010,” 2010.
- [18] B. Kiesl, M. J. H. Heule, and A. Biere, “Truth assignments as conditional autarkies,” in *Automated Technology for Verification and Analysis (ATVA)*, 2019.
- [19] O. Kullmann, “On a generalization of extended resolution,” *Discrete Applied Mathematics*, vol. 96–97, pp. 149–176, 1999.
- [20] M. Järvisalo, M. J. H. Heule, and A. Biere, “Inprocessing rules,” in *International Joint Conference on Automated Reasoning (IJCAR)*, 2012.
- [21] B. Kiesl, M. Seidl, H. Tompits, and A. Biere, “Super-blocked clauses,” in *International Joint Conference on Automated Reasoning (IJCAR)*, 2016.
- [22] M. J. H. Heule, B. Kiesl, M. Seidl, and A. Biere, “Pruning through satisfaction,” in *Hardware and Software: Verification and Testing (HVC)*, 2017.
- [23] A. Biere, M. Heule, H. van Maaren, and T. Walsh, *Handbook of Satisfiability: Volume 185 Frontiers in Artificial Intelligence and Applications*. IOS Press, 2009.
- [24] M. J. H. Heule, B. Kiesl, and A. Biere, “Clausal proofs of mutilated chessboards,” in *NASA Formal Methods (NFM)*, 2019.
- [25] A. Urquhart, “Hard examples for resolution,” *Journal of the Association for Computing Machinery (JACM)*, vol. 34, no. 1, p. 209–219, 1987.
- [26] J. E. Reeves, M. J. H. Heule, and R. E. Bryant, “Preprocessing of Propagation Redundant Clauses,” *Journal of Automated Reasoning (JAR)*, vol. 67, no. 3, p. 31, 2023.
- [27] P. Slavík, “A tight analysis of the greedy algorithm for set cover,” in *Symposium on Theory of Computing (STOC)*, 1996.
- [28] T. Balyo, M. Heule, M. Iser, M. Järvisalo, and M. Suda, Eds., *Proceedings of SAT Competition 2023: Solver, Benchmark and Proof Checker Descriptions*, 2023.
- [29] M. J. H. Heule, “dpr-trim: Checker of dpr proof files,” <https://github.com/marijnheule/dpr-trim>, accessed: May 1, 2025.
- [30] X. C. Song, P. Smith, R. Kalyanam, X. Zhu, E. Adams, K. Colby, P. Finnegan, E. Gough, E. Hillery, R. Irvine, A. Maji, and J. St. John, “Anvil - system architecture and experiences from deployment and early user operations,” in *Practice and Experience in Advanced Research Computing (PEARC)*, 2022.
- [31] T. Balyo, M. Heule, M. Iser, M. Järvisalo, and M. Suda, Eds., *Proceedings of SAT Competition 2022: Solver and Benchmark Descriptions*, 2022.
- [32] M. Heule, M. Iser, M. Järvisalo, and M. Suda, Eds., *Proceedings of SAT Competition 2024: Solver, Benchmark and Proof Checker Descriptions*, 2024.
- [33] A. Biere and M. Heule, “The effect of scrambling cnfs,” in *Pragmatics of SAT (PoS)*, 2019.
- [34] M. Alekhovich, “Mutilated chessboard problem is exponentially hard for resolution,” *Theoretical Computer Science*, vol. 310, no. 1, pp. 513–525, 2004.
- [35] C. R. Codel, J. Reeves, M. J. H. Heule, and R. E. Bryant, “Bipartite perfect matching benchmarks,” in *Pragmatics of SAT (PoS)*, 2021.
- [36] A. Haberlandt and H. Green, “Python function register allocation benchmarks,” in *Proceedings of SAT Competition 2023: Solver, Benchmark and Proof Checker Descriptions*, 2023, p. 56.
- [37] J. Chung, S. Buss, and V. Ganesh, “Unsatcoin,” in *Proceedings of SAT Competition 2023: Solver, Benchmark and Proof Checker Descriptions*, 2023, p. 52.
- [38] A. Biere, “CNF encodings of complete pairwise combinatorial testing of our SAT solver SATCATCH,” in *Proceedings of SAT Competition 2021: Solver, Benchmark and Proof Checker Descriptions*, 2023, p. 46.
- [39] G. S. Tseitin, “On the complexity of derivation in propositional calculus,” *Automation of Reasoning 2: Classical Papers on Computational Logic 1967–1970*, pp. 466–483, 1983.