Formal Methods in Computer-Aided Design 2025

Learning Short Clauses via Conditional Autarkies

Amar Shah, Twain Byrnes, Joseph Reeves, Marijn J. H. Heule
Carnegie Mellon University, Pittsburgh, PA, USA
amarshah@cmu.edu, binarynewts@cmu.edu, jereeves@andrew.cmu.edu, marijn@cmu.edu

***Abstract*—State-of-the-art Boolean satisfiability (SAT) solvers increasingly use techniques that go beyond resolution. These admit short proofs for benchmark families with exponentially large resolution proofs, such as the pigeonhole principle and mutilated chessboard. One of the strongest such techniques is Propagation Redundancy (PR) learning. However, existing PR clause learning techniques require an NP-hard check. Hence they are computationally expensive and difficult to integrate into existing tools.**

We propose a new method for learning PR clauses based on conditional autarkies and implement it in the SAT solver CaDiCaL. Our method is modular and learns PR clauses in linear time. Additionally, we introduce a number of heuristics, including a clause-shrinking technique, to ensure that the learned clauses are useful. We show that we are competitive with state-of-the-art PR clause learning techniques, improving the performance on a large portion of SAT competition benchmarks.

I. Introduction

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 use expands, so has the need for more powerful and specialized solving techniques.

One such class of techniques is propagation redundant (PR) clause learning, where new, satisfiability-preserving clauses are added to the formula, shrinking the space of potential solutions. These learned clauses act like lemmas for that solver can leverage. Crucially, they must preserve satisfiability: the original formula is satisfiable if and only if the formula with the additional clause is satisfiable. For instance, in the pigeonhole problem, one may learn the clause stating *pigeon 1 is not in hole 1*. This is useful since it restricts the search space for pigeon 1, but it preserves satisfiability as hole 1 is symmetrically the same as any other hole.

The power of clause learning is tied to the strength of the underlying proof system. Most SAT solvers rely on resolution-based proof systems, which are ineffective for many hard instances. In particular, problems like the pigeonhole principle and the mutilated chessboard are known to require exponentially large resolution proofs [12, 13]. Proof systems based on PR clauses have short proofs for such problems.

Despite their theoretical power, learning PR clauses remains 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]. 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. As of today, the most popular SAT solvers such as CaDiCaL [15], Kissat [16], CryptoMiniSat [7], and Lingeling [17] do not support PR clause learning.

In this work, we propose a new, more efficient approach to learn PR clauses based on conditional autarkies. Given a partial assignment to a SAT formula, our method divides the assignment into a "conditional part" and an "autarky part." Using this distinction, we can learn a globally blocked clause (a type of PR clause) in linear time.

While this solves the theoretical gap in identifying PR clauses, it is not immediately useful in real SAT solving applications. Indeed, there are two major practical limitations namely: (1) globally blocked clauses can be too large to meaningfully shrink the search space, and (2) it is difficult to select useful clauses. We solve (1) by introducing a shrinking procedure to extract compact, useful PR clauses. We solve (2) by introducing a number of heuristics.

To summarize, we make the following contributions:

1) We introduce a method for learning PR clauses in linear time in the size of the formula.
2) We develop a number of heuristics, including a shrinking technique to extract concise and useful PR clauses from globally blocked clauses.
3) We implement these techniques in a solver, CAutiCal (a fork of the popular SAT solver CaDiCaL), and evaluate it on a suite of benchmarks, demonstrating substantial performance improvements ...

II. Background

We begin with some SAT preliminaries. Variables $x_1, x_2, \dots$ can take values *true* ($\top$) or *false* ($\bot$). A literal l can be a variable x or its negation $\overline{x}$. A clause C is a disjunction of literals, e.g. $x_1 \vee \overline{x_4} \vee x_6$. Occasionally, we will use set notation and may represent $C = \{x_1, \overline{x_4}, x_6\}$ and denote that a literal l is present in clause C with $l \in C$. A conjunctive normal form (CNF) formula ϕ is a conjunction of clauses. We will assume that any formula ϕ is in CNF for the rest of this paper.

We notate $var(\phi)$ as the set of variables that occur in formula ϕ. For some $V \subseteq var(\phi)$, a partial assignment $\alpha : V \rightarrow \{\top, \bot\}$ maps some variables of a formula ϕ to true or false. If $V = var(\phi)$, then α is a (total) assignment. We abuse notation by denoting $\alpha(l) = \alpha(x)$ if $l = x$ and $\alpha(l) = \neg\alpha(x)$ if $l = \overline{x}$.

A formula restricted to a partial assignment $\phi|_\alpha$ is the formula where every variable x in the domain of α is assigned to $\alpha(x)$. We can simplify such a formula by removing all literals that are assigned to false and all clauses with a

https://doi.org/

literal assigned to true. For example, with formula $\phi = (x_1 \vee \overline{x_4} \vee x_6) \wedge (x_2 \vee x_3) \wedge \overline{x_5}$ and assignment α with $\alpha(x_1) = \bot$ and $\alpha(x_2) = \top$. Then, $\phi|_\alpha = (\overline{x_4} \vee x_6) \wedge \overline{x_5}$.

A clause $C = l_1 \vee ... \vee l_m$ is blocked by the partial assignment α that assigns each l_i to false ($\bot$). An assignment α *touches* a clause C if there is a variable x in the domain of α such that $x \in C$ or $\overline{x} \in C$. We say α *satisfies* C if $\alpha(l) = \top$ and $l \in C$.

An assignment α *satisfies* formula ϕ if it satisfies every clause $C \in \phi$. If there is such a satisfying assignment, then ϕ is *satisfiable*. The boolean satisfiability problem asks whether a given formula ϕ is satisfiable.

A. Redundant Clauses

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

In a *clausal proof system*, each step will add or remove a redundant clause C. The step may contain extra information, such as a boolean witness, justifying why C is redundant. A list of redundant clauses ending with the empty clause $\bot$ is a proof of unsatisfiability for formula ϕ.

Adding redundant clauses may be helpful in certain cases, since adding a clause will constrain the set of possible solutions. However, it could be harmful as new clauses may negatively interact with solver heuristics.

Resolution is the main proof step in a clausal proof system. It states that two clauses $C \vee x$ and $\overline{x} \vee D$ and returns a new clause $C \vee D$.

$$\frac{C \vee x \qquad \overline{x} \vee D}{C \vee D}\ \textsc{Resolution}$$

Unit propagation is a core reasoning techniques in a SAT solver. Starting with empty partial assignment α, if a formula ϕ, contains a *unit clause* l, i.e. a clause with only a single literal, then we set $\alpha(l) = \top$. Then, this unit is propagated, i.e. we consider the formula $\phi|_\alpha$ and continue this process until there are no unit clauses remaining. When unit propagation terminates with $\phi|_\alpha = \bot$, we say that it derives a conflict.

Unit propagation can be thought of as a proof step. Indeed, it can be thought of as a mechanized application of resolution.

$$\frac{C \vee l \qquad \bar{l}}{C}\ \textsc{Unit-Prop}$$

Given a formula ϕ and clause $C = l_1 \vee ... \vee l_k$, we can say $\phi \vdash_1 C$, i.e. "ϕ implies C via unit propagation," if $\phi \wedge \overline{l_1} \wedge ... \wedge \overline{l_k}$ derives a conflict. Here, C is a simple example of a redundant clause w.r.t. ϕ.

For two formulas we say $\phi \vdash_1 \psi$ if for every clause $C \in \psi$, $\phi \vdash_1 C$. For literals l_1 and l_2, we sometimes notate $l_1 \vdash_1^\phi l_2$ to mean $\phi \vdash_1 \overline{l_1} \wedge l_2$. Informally, we can think of this as saying "l_1 implies l_2 via unit propagation on ϕ," which means that .

B. Propagation Redundant

While resolution is complete for propositional logic, more powerful proof steps can yield shorter proofs and faster runtimes. One such step is based on PR clauses.

Definition 1 (Propagation Redundant (PR) clauses)**.** *For formula ϕ, we say that clause C (blocked by β) is propagation redundant (PR) w.r.t. ϕ if there exists an assignment ω known as the witness such that $F|_\beta \vdash_1 F|_\omega$ and ω satisfies C*

Such a clause C must be redundant. Say there is a satisfying assignment α for ϕ. Then if α is not a satisfying assignment for $\phi \wedge C$, then there it must be that $\beta \subseteq \alpha$, i.e. α extends β. However, since $F|_\beta \vdash_1 F|_\omega$, it must be that any assignment that satisfies $F|_\beta$, will satisfy $F|_\omega$.

Then we can define $\alpha'(x) = \omega(x)$ if $x \in \omega$ and $\alpha'(x) = \alpha(x)$ otherwise. Thus, α' satisfies ϕ and α' satisfies C.

However, checking if a clause is PR is NP-complete [**?**], so witnesses must be provided for proof checking. PR clauses subsume many classes of redundant clauses including resolution asymmetric tautologies (RATs) [18], blocked clauses [19], set-blocked clauses [20], and globally-blocked clauses [21].

C. Autarkies and Globally Blocked Clauses

Definition 2 (Autarky)**.** *A nonempty assignment α is an autarky for a formula ϕ if every clause $C \in \phi$ touched by α is satisfied.*

In plain words, an autarky is an assignment that satisfies every clause that it touches. For example, if α was a satisfying assignment, it would be an autarky since it satisfies every clauses.

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

Specifically, we can think about searching for conditional autarkies by first looking for a partial assignment α_c, and then finding an autarky α_a on the reduced formula $\phi|_{\alpha_c}$.

Conditional autarkies can be very useful for learning equisatisfiable clauses, for instance if $\alpha_c = c_1, ..., c_n$ and $\alpha_a = a_1, ..., a_m$, we add clauses of the form:

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

This results in m different clauses as in the following theorem from Kiesel et al [21]:

Theorem 1. *Formula ϕ and $\phi \wedge \bigwedge_{1 \le i \le m} (\overline{c_1} \vee ...\overline{c_n} \vee a_i)$ are equisatisfiable.*

This means that ϕ is satisfiable if and only if $\phi \wedge \bigwedge_{1 \le i \le m} (\overline{c_1} \vee ...\overline{c_n} \vee a_i)$ is satisfiable. Thus, the solver can add any of the m clauses $\overline{c_1} \vee ...\overline{c_n} \vee a_i$ and preserve satisfiability. Each of these clauses is a PR clause with witness α_a. Each of these clauses is a *globally blocked clause*, however, this is not too important for our purposes, so we will not discuss it further.

D. Related Work

The pigeonhole problem can ask if $n+1$ pigeons can fit into n holes. The mutilated chessboard problem asks if a $2n \times 2n$ board with two diagonally opposite corners removed can be tiled with 2×1 rectangular dominoes. Both are unsatisfiable and it is easy for a human to see why. However, it has been shown there are no polynomial-sized resolution proofs for either problem [12, 13].

While, proof systems like extended resolution (ER) provided $O(n^4)$ for the pigeonhole problems, the proof system involved introducing new variables [22]. In general, the search space for new variables is infinite and thus tools like GLUCOSER [23] based on ER did not scale well.

The PR proof system remedies this by producing $O(n^3)$ proofs for the pigeonhole formula [24] without learning any new clauses. Later, this was shown to produce short proofs for mutilated chessboard [25].

The satisfaction-driven clause learning (SDCL) framework [26], extends the conflict-driven clause learning (CDCL) framework [?] with PR clause learning. After propagating an assignment, they would check if the clause C that was blocked by this assignment was PR. This was done by creating a new SAT formula called the *positive reduct*. If the positive reduct was satisfiable, then C is a PR clause and it was 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 Tseitin formulas [27] and mutilated chessboard. This was implemented in a new SDCL solver called SADICAL.

Additionally, this was extended by PRELEARN, which is a preprocessing technique for PR clauses [28]. This was done by initially considering many possible clauses and querying SADICAL to see which were PR.

Our work differs as we do not use a *positive reduct* to test if a clause is PR. Instead, our clauses PR by construction as they come from a conditional autarky. This has the potential downside that our clause may be very large. To remedy this we apply a shrinking technique to reduce 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 tank performance. We provide two *symmetry hardening* techniques to handle reordering. We compare our implementation CAUTICAL to SADICAL and PRELEARN in Section V.

Finally, Kiesel et al [21] first introduced conditional autarkies to identify globally blocked clauses. Their aim was to eliminate globally blocked clauses from a formula. This technique allowed them to simulate circuit-simplification techniques. Our work differs from this as we add clauses instead of removing them and we target a different class of benchmarks.

III. METHODOLOGY

Our methodology is based on four steps:

Algorithm 1: Our Methodology

```
Function LearnClause($\psi$, $\alpha$):
    for $i \in vars(\psi)$ :
        for $j \in vars(\psi)$ :
            propagate($i$) propagate($j$) $\alpha_c, \alpha_a :=$ LCP($\alpha$)
             clause := shrink($\alpha_c, \alpha_a$) $\psi := \psi \wedge clause$
```

A. Learning Clauses

As described in Subsection II-C, we can split a partial assignment $\alpha = \alpha_c \sqcup \alpha_a$ and use this to learn a PR clause. Since all of α_c appears in such a clause, we must try to keep in as small as possible.

We present this algorihtm from Kiesel et al [21] to find the smallest possible α_c:

Algorithm 2: Minimiazing α_c in $\alpha = \alpha_c \sqcup \alpha_a$

```
Function LeastConditional($\psi$, $\alpha$):
    $\alpha_c := \emptyset$;
    for $C \in \phi$ :
        if $\alpha$ touches $C$ without satisfying $C$ :
            $\alpha_c := \alpha_c \cup (\alpha \cap \overline{C})$;
    return $\alpha_c$;
```

We can then compute $\alpha_a := \alpha \backslash \alpha_c$. It is pretty easy to see why this is a conditional autarky, since every clause that α_a touches, must be satisfied by some literal in α.

We can see that α_c is minimal since for each clause that is touched, we add in those literals from the assignment that touch and do not satisfy the clause. They must be in α_c, since otherwise α_a will touch a clause that is not satisfied in α_a and α_c, violating the conditional autarky property.

B. Shrinking Clauses

While we minimize α_c, this still leads to a long clause in many cases. However, we can shrink the clause using some clever techniques. Notice that if $\alpha_c = c_1, ..., c_n$ and $\alpha_a = a_1, ..., a_m$, we could add clauses of the form $\overline{c_1} \vee ... \vee \overline{c_n} \vee a_i$ where $1 \leq i \leq m$.

We consider the set $C_0 = \{c_j \in C | \exists a_i \in \alpha_a \; \overline{a_i} \vdash_1^\phi c_j\}$. This is essentially the set of literals in α_c that can be implied by unit propagation by negating some literal in α_a.

We also define $A_0 \subseteq \alpha_a$ as a set such that for each $c \in C_0$, there is some $a \in A_0$ such that $\overline{a} \vdash_1^\phi c$. By definition, there must be some A_0, but there could be many possible A_0. We will discuss how we pick A_0 in Subsection III-C as it will be very important for having a technique resistant to different encodings.

We want learn the clause $\bigvee_{c \in C \setminus C_0} \overline{c} \vee \bigvee_{a \in A_0} a$

Theorem 2. *The formula ϕ is satisfiable if and only if $\phi \wedge (\bigvee_{c \in C \setminus C_0} \overline{c} \vee \bigvee_{a \in A_0} a)$ is satisfiable.*

Proof. $\underline{\Leftarrow}$: This is immediate

$\underline{\Rightarrow}$: As a corollary to Theorem 1, ϕ is satisfiable if and only if $\phi \wedge (\bigvee_{c\in C} \overline{c} \vee \bigvee_{a\in A_0} a)$ is satisfiable.

Thus we can assume $\phi \wedge (\bigvee_{c\in C} \overline{c} \vee \bigvee_{a\in A_0} a)$ is satisfiable by some satisfying assignment β.

We claim β is a satisfying assignment for $\phi \wedge (\bigvee_{c\in C\setminus C_0} \overline{c} \vee \bigvee_{a\in A_0} a)$. This could only not be the case if (1) for all $a \in A_0$ $\overline{a} \in \beta$ and (2) there is some $c \in C_0$ such that $\overline{c} \in \beta$.

But by definition there is some $a \in A_0$ such that $\overline{a} \vdash_1^\phi c$. Thus $\phi \wedge \overline{a} \wedge \overline{c}$ is unsatisfiable via unit propagation. However, this cannot be the case since $\overline{a}, \overline{c} \in \beta$ and β is a satisfying assignment for ϕ. □

C. Resilience to Encoding

As mentioned in Subsection III-B, our choice for A_0 can matter a lot for which clause we learn.

Greedy Set Cover: In order to minimize the size of the clause, we may want the smallest possible A_0 that maximizes C_0. Say for each $a \in \alpha_a$, we define $\alpha_a^{SETS}(a) = \{c \in \alpha_c \mid \overline{a} \vdash_1^\phi c\}$.

In the case that $C_0 = \alpha_C$, finding the smallest A_0 is exactly the set cover problem with α_a^{SETS}. This is NP-hard and we do not want to spend time figuring this out. Instead, we can use the greedy algorithm to solve, which is a $1 + \ln|\alpha_c|$ approximation algorithm. Specifically, the greedy algorithm will find the largest set in $\{\alpha_a^{SETS} \cap (\alpha_c \backslash C_0) | a \in \alpha_a\}$ at each step and add it to C_0. It stops when each of the $\alpha_a^{SETS} \cap (\alpha_c \backslash C_0)$ are empty.

Algorithm 3: Algorithm finding A_0

```
Function FindA0($\psi$, $\alpha_a$, $\alpha_c$):
    $\alpha_a$ := sort($\alpha_a$);
    $\alpha_a^{SETS}$ := init Array[len($\alpha_a$)];
    for $i \in range(\alpha_a)$ :
        propagate($-\alpha_a[i]$);
        implied := {};
        for $c \in \alpha_c$ :
            propagate(-c);
            if unsat :
                implied := implied $\cup$ {$c$};
        $\alpha_a^{SETS}[i]$ := implied;
    return greedySetCover($\alpha_a^{SETS}$);
```

In Algorithm 3, we describe our process for calculating A_0. We initially pre-sort $alpha_a$ by implication, which we will later discuss in more detail. Then we intialize α_a^{SETS} as an array, we iterate the counter i through α_a, populating $\alpha_a^{SETS}[i]$ with the set of literals in α_c that are implied by $-\alpha_a^{SETS}[i]$. Finally, we apply `greedySetCover`

Sorting α_a by Implication: The greedy set cover algorithm described above is almost sufficient for being insensitive to heuristics, but there is one other optimization that we made. Algorithm 3 describes sorting α_a as an initial step. This is important for cases where there are more than one cover of the same size. The most common occurence of this is for $a_1, a_2 \in \alpha_a$, we may have that $\overline{a_2} \vdash_1^\phi \overline{a_1} \vdash_1^\phi c_1, ..., c_n$.

Thus, greedy set cover could pick $\{a_1\}$ or $\{a_2\}$ as singleton sets. However, since $a_1 \vdash_1^\phi a_2$, we really want to learn the unit clause a_1 since it is much more powerful.

D. Other Heuristics

We employ several other heuristics:

Checking if trivial: Introducing globally blocked clauses can affect how we divide $\alpha = \alpha_c \sqcup \alpha_a$ as done in Algorithm 2. Thus, it is better to avoid introducing globally blocked clauses if they are not useful. We do this by checking if $\phi \vdash_1 C$ for each potential clause C. If it is implied by unit propagation, we do not learn it since we consider it easy to learn.

IV. Motivating Example

We discuss the pigeonhole principle as a motivating example. The problem asks whether we can put m 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 easily encoded as a SAT problem where variable $x_{i,j}$ represents putting the i-th pigeon into the j-th hole. We can translate constraint (1) as $\bigvee_{1\le j\le n} x_{i,j}$ for each $1 \le i \le m$. We can also translate constraint (2) as $\overline{x_{i,j}} \vee \overline{x_{k,j}}$ for each $1 \le i \ne k \le m$ and $1 \le j \le n$.

We represent this diagrammatically as in **??**, where the rows are the pigeons and the columns are the holes. The cell in row i and column j represents the literal $x_{i,j}$, i.e. whether the i-th pigeon is in the j-th hole. If the (i, j)-th cell is green this represents $x_{i,j}$ is set to true and if it is red this represents $x_{i,j}$ is set to false. Thus, constraint (1) asks that each row has at least one green cell and constraint (2) asks that no two green cells share a column.

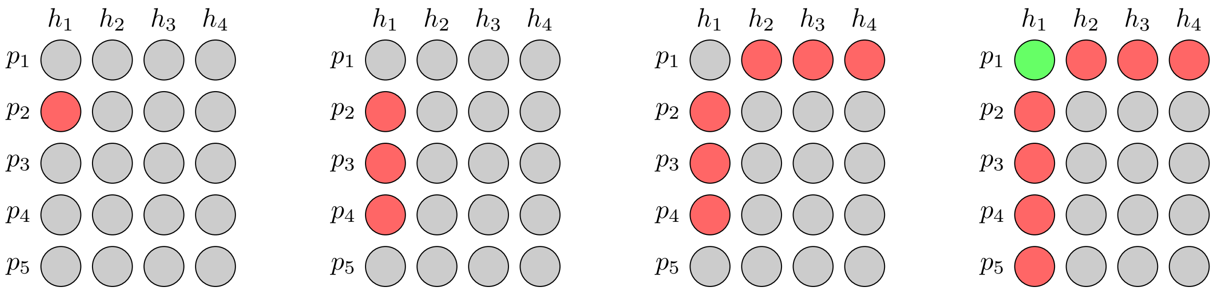
We will only consider the case where $m = n + 1$ which we denote $\text{PHP}(n)$. This is clearly unsatisfiable, however there are no polynomial-sized resolution proofs of $\text{PHP}(n)$ [12]. There are $O(n^3)$ PR proofs of the pigeonhole principle, but this is difficult to achieve in practice and requires specific techniques [24]. We can learn these proofs, with a very low constant factor and little sensitivity to the encoding.

A. Pigeonhole Setup

Our technique makes a pair of decisions, propagates and then divides this assignment into a conditional and an autarky part, and finally shrinks the clause to derive a useful PR clause.

While our technique does allow for decisions on negative literals, this is not useful for the Pigeonhole Principle. We first decide on some literal i. For ease, we assume that $i = x_{1,1}$. This does not matter because of symmetry.

Notice that the set of literals we consider for j are the *touched literals*, i.e. those that are in a clause with $x_{1,1}$ or any of the variables it propagates. By constraint (2), $x_{1,1}$ propagates $\overline{x_{k,1}}$ for $2 \le k \le n + 1$. But these literals will touch every literal in the formula via the clauses from constraint (1).



(a) The first unit learned is of the form $x_{2,1}$

(b) In a similar fashion we then learn the units $\overline{x_{3,1}}$ and $\overline{x_{4,1}}$

(c) By the symmetry argument, we learn units $\overline{x_{1,2}}$, $\overline{x_{1,3}}$, and $\overline{x_{1,4}}$

(d) Constraint (1) gives $x_{1,1}$, then constraint (2) gives $\overline{x_{5,1}}$, completing the reduction to PHP(3)

Fig. 1: Process for reducing PHP(4) to PHP(3)

B. *Solving the Pigeonhole Principle*

Below we describe three stages at which we learn useful clauses:

1) Useful Binary Clauses: When learning a second variable $j = x_{2,2}$, propagation gives us the literals $\overline{x_{2,1}}, ...\overline{x_{n+1,1}}$ and $\overline{x_{1,2}}, \overline{x_{3,2}}, ...\overline{x_{n+1,2}}$. This can be divided into a autarky part $\alpha_a = x_{1,1}, x_{2,2}, \overline{x_{1,2}}, \overline{x_{2,1}}$ and a conditional part $\alpha_c = \overline{x_{3,1}}, \ldots, \overline{x_{n+1,1}}, \overline{x_{3,2}}, \ldots, \overline{x_{n+1,2}}$.

This can be shrunk to the clause $\overline{x_{1,2}} \vee \overline{x_{2,1}}$.

Then when considering the next assumption $j = x_{2,3}$, we end up learning the clause $\overline{x_{1,3}} \vee \overline{x_{2,1}}$. This continues until we learn $\overline{x_{1,n}} \vee \overline{x_{2,1}}$. Notice that have the clause $\overline{x_{1,n}} \vee \overline{x_{2,1}}$ from constraint (2) and the clause $\bigwedge_{1 \leq j \leq n} x_{1,j}$ from constraint (1). These combined will yield the unit clause $\overline{x_{2,1}}$ as shown in Figure 1a.

Similary, we can learn the unit clauses $\overline{x_{3,1}}, \overline{x_{4,1}}, \ldots, \overline{x_{n,1}}$ as shown in Figure 1b.

2) Useful Units: In the last row of learning, we get something even more useful. Starting with the assumption $j = x_{n+1,2}$, we propagate the literals $\overline{x_{n+1,1}}$ and $\overline{x_{1,2}}, \overline{x_{2,2}}, \ldots, \overline{x_{n,2}}$. This can be divided into a autarky part $\alpha_a = x_{1,1}, x_{n+1,2}, \overline{x_{1,2}}, \overline{x_{n+1,1}}$ and a conditional part $\alpha_c = \overline{x_{1,2}}, \ldots, \overline{x_{n,2}}$.

Shrinking gives us just the unit clause $\overline{x_{1,2}}$. Notice that if we had does this at the beginning, we would have also had $\overline{x_{2,1}}, \ldots, \overline{x_{n,1}}$ and thus shrinking would have given a binary clause.

In a similar manner, we also learn the units $\overline{x_{2,1}}, \ldots, \overline{x_{n,1}}$ as shown in Figure 1c.

3) Final Simplifications: Finally, constraint (1) will allow us to learn the unit $x_{1,1}$ and constraint (2) will imply the unit $x_{n+1,1}$ as shown in Figure 1d. Thus, we have successfully reduced the PHP(n) to PHP($n-1$). Then we pick some other variable for i (for instance $i = x_{2,2}$) and repeat the process.

C. *Complexity Analysis*

The complexity of this technique is $O(n^3)$ for the proof size. The time it takes to execute a proof step is linear in the size of a formula. Since the formula for pigeonhole has size $O(n^2)$, the total runtime is $O(n^5)$.

We make a brief remark on the constant factors in the proof size as they compare favorably to similar techniques. First, in the useful binary clause phase, for each unit we learn, we must learn $n-1$ binary clauses. We end learning $n-1$ units, for a total of $(n-1)^2$ clauses (note the final binary clause and the unit clause can be learned together in a single step). Second, in the useful unit phase, we learn $n-1$ units, for a total of $n-1$ clauses. Third, in the final simplification phase, we learn 3 clauses. This gives a total of $n(n-1)+3$ clauses to reduce PHP(n) to PHP($n-1$).

Thus, there needs to be

V. Evaluation

In this section, we aim to answer the following research questions:

1) Does our technique outperform other SAT solvers on certain benchmark families?
2) Does our technique underperform compared to other SAT solvers on certain benchmark families?
3) Is our technique less sensitive to encoding choices compared to other PR learning techniques?

We do so by implementing our technique in a tool CAutiCal (a fork of CaDiCal commit f13d74439a5b5c963ac5b02d05ce93a8098018b8). In Subsection V-A, we compare CAutiCal to standard SAT solvers as well as PR learning techniques. We evaluate based on time taken, the length of the proof, and the sensitivity to renaming variables and reordering clauses.

Second, in Subsection V-B, we compare CAutiCal to other SAT solvers on the benchmarks from the annual Satisfiability competition from the years 2022, 2023, and 2024.

Third, we analyze the use of specific heuristic choices in CAutiCal by turning heuristics off one by one and observing the effect on the performance of CAutiCal. This is described in Subsection V-C.

Finally, we conclude, in Subsection V-D, by discussing the benchmarks families upon which our technique performs well and poorly.

A. Pigeonhole results

B. SATCOMP results

C. Analysis of heuristics

D. Discussion of Benchmark Families

VI. Conclusion and Future Work

References

[1] N. Rungta, "A billion smt queries a day," *Computer Aided Verification*, pp. 3–18, 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*, M. Bernardo and A. Cimatti, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 108–143.

[3] A. R. Bradley, "Understanding ic3," in *Theory and Applications of Satisfiability Testing – SAT 2012*, A. Cimatti and R. Sebastiani, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 1–14.

[4] E. Clarke, A. Biere, R. Raimi, and Y. Zhu, "Bounded Model Checking Using Satisfiability Solving," vol. 19, no. 1, pp. 7–34. [Online]. Available: https://doi.org/10.1023/A:1011276507260

[5] J. Rintanen, "Heuristics for planning with sat," in *Principles and Practice of Constraint Programming – CP 2010*, D. Cohen, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 414–428.

[6] H. Kautz and B. Selman, "Planning as satisfiability," in *Proceedings of the 10th European Conference on Artificial Intelligence*, ser. ECAI '92. USA: John Wiley & Sons, Inc., 1992, p. 359–363.

[7] M. Soos, K. Nohl, and C. Castelluccia, "Extending sat solvers to cryptographic problems," in *Proceedings of the 12th International Conference on Theory and Applications of Satisfiability Testing*, ser. SAT '09. Berlin, Heidelberg: Springer-Verlag, 2009, p. 244–257. [Online]. Available: https://doi.org/10.1007/978-3-642-02777-2_24

[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*, S. Sankaranarayanan and N. Sharygina, Eds. Cham: Springer Nature Switzerland, 2023, pp. 389–406.

[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*, N. Creignou and D. Le Berre, Eds. Cham: Springer International Publishing, 2016, pp. 228–245.

[10] J. Brakensiek, M. Heule, J. Mackey, and D. Narváez, "The Resolution of Keller's Conjecture," vol. 66, no. 3, pp. 277–300. [Online]. Available: https://doi.org/10.1007/s10817-022-09623-5

[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*, B. Finkbeiner and L. Kovács, Eds. Cham: Springer Nature Switzerland, 2024, pp. 61–80.

[12] A. Haken, "The intractability of resolution," *Theoretical Computer Science*, vol. 39, pp. 297–308, 1985, third Conference on Foundations of Software Technology and Theoretical Computer Science. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0304397585901446

[13] M. Alekhnovich, "Mutilated chessboard problem is exponentially hard for resolution," *Theoretical Computer Science*, vol. 310, no. 1, pp. 513–525, 2004. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0304397503003955

[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*, T. Vojnar and L. Zhang, Eds. Cham: Springer International Publishing, 2019, pp. 41–58.

[15] A. Biere, T. Faller, K. Fazekas, M. Fleury, N. Froleyks, and F. Pollitt, "Cadical 2.0," in *Computer Aided Verification: 36th International Conference, CAV 2024, Montreal, QC, Canada, July 24–27, 2024, Proceedings, Part I*. Berlin, Heidelberg: Springer-Verlag, 2024, p. 133–152. [Online]. Available: https://doi.org/10.1007/978-3-031-65627-9_7

[16] A. Fleury and M. Heisinger, "Cadical, kissat, paracooba, plingeling and treengeling entering the sat competition 2020," *Sat Competition*, vol. 2020, p. 50, 2020.

[17] A. Biere, "Lingeling, plingeling, picosat and precosat at sat race 2010," 2010. [Online]. Available: https://api.semanticscholar.org/CorpusID:1087401

[18] O. Kullmann, "On a generalization of extended resolution," *Discrete Applied Mathematics*, vol. 96-97, pp. 149–176, 1999. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0166218X99000372

[19] M. Järvisalo, M. J. H. Heule, and A. Biere, "Inprocessing rules," in *Automated Reasoning*, B. Gramlich, D. Miller, and U. Sattler, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 355–370.

[20] B. Kiesl, M. Seidl, H. Tompits, and A. Biere, "Super-blocked clauses," in *Proceedings of the 8th International Joint Conference on Automated Reasoning - Volume 9706*. Berlin, Heidelberg: Springer-Verlag, 2016, p. 45–61. [Online]. Available: https://doi.org/10.1007/978-3-319-40229-1_5

[21] B. Kiesl, M. J. H. Heule, and A. Biere, "Truth assignments as conditional autarkies," in *Automated Technology for Verification and Analysis: 17th International Symposium, ATVA 2019, Taipei, Taiwan, October 28–31, 2019, Proceedings*. Berlin, Heidelberg: Springer-Verlag, 2019, p. 48–64. [Online]. Available: https://doi.org/10.1007/978-3-030-31784-3_3

[22] G. S. Tseitin, *On the Complexity of Derivation in Propositional Calculus*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1983, pp. 466–483. [Online]. Available: https://doi.org/10.1007/978-3-642-81955-1_28

[23] G. Audemard, G. Katsirelos, and L. Simon, "A restriction of extended resolution for clause learning sat solvers," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 24, no. 1, pp. 15–20, Jul. 2010. [Online]. Available: https://ojs.aaai.org/index.php/AAAI/article/view/7553

[24] M. J. H. Heule, B. Kiesl, and A. Biere, "Short proofs

without new variables," in *Automated Deduction – CADE 26*, L. de Moura, Ed. Cham: Springer International Publishing, 2017, pp. 130–147.

[25] ——, "Clausal proofs of mutilated chessboards," in *NASA Formal Methods*, J. M. Badger and K. Y. Rozier, Eds. Cham: Springer International Publishing, 2019, pp. 204–210.

[26] M. J. H. Heule, B. Kiesl, M. Seidl, and A. Biere, "Pruning through satisfaction," in *Hardware and Software: Verification and Testing*, O. Strichman and R. Tzoref-Brill, Eds. Cham: Springer International Publishing, 2017, pp. 179–194.

[27] A. Urquhart, "Hard examples for resolution," *J. ACM*, vol. 34, no. 1, p. 209–219, Jan. 1987. [Online]. Available: https://doi.org/10.1145/7531.8928

[28] J. E. Reeves, M. J. H. Heule, and R. E. Bryant, "Preprocessing of Propagation Redundant Clauses," *Journal of Automated Reasoning*, vol. 67, no. 3, p. 31, Sep. 2023.