Learning Propagation Redundant SAT clauses via Conditional Autarkies

TODO: add authors and orcid
Carnegie Mellon University, Pittsburgh, PA, USA
{email1,email2,email3,email4}@cmu.edu

***Abstract*—TODO: write abstract**

I. Introduction

Satisfiability (SAT) solving is a core tool in computer science, with applications in program verification [**?**], planning [**?**], and resolving long-standing mathematical conjectures [**?**]. As its use grows, so has the need for more powerful and specialized solving techniques.

One such class of techniques is clause learning, where new, satisfiability-preserving clauses are added to the formula, shrinking the space of potential solutions. These learned clauses can act like lemmas that the 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.

However, 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. To overcome this limitation, researchers have proposed more expressive systems based on propagation redundant (PR) clauses, which can admit 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 expensive SAT solver calls to verify that a candidate clause is PR [**?**]. In the worst case, we may have to spend exponential time learning 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 [**?**], Kissat [**?**], and Lingeling [**?**] do not support PR clause learning.

In this work, we propose a new, more efficient approach to learning 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 divide, we can learn a class of identifies globally blocked clauses, which are guaranteed to preserve satisfiability. This technique gives a linear time algorithm for identifying a PR clause.

While this technique solves the theoretical gap in identifying PR clauses, it is not clear that it learns clauses that are actually useful. Indeed, there are two major practical limitations namely: (1) globally blocked clauses can be too large and hence not shrink the search space in any meaningful way, and (2) there are many potential globally blocked clauses and it is not clear which are the most useful. 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 $x_1, x_2, ...$ 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 ask if a literal l is present in clause C with $l \in C$. A formula ϕ is a conjunction of clauses.

We notate $var(\phi)$ as the set of variables that occur in formula ϕ. For some $V \subseteq var(\phi)$, an assignment $\alpha : V \to \{\top, \bot\}$ maps some variables from a formula ϕ to true or false. If $V = var(\phi)$, we say α is a total assignment. We denote a formula restricted to an assignment $\phi|_\alpha$ which 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 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 $\alpha(x_1) = \bot, \alpha(x_2) = \top$. We get that $\phi|_\alpha = (\overline{x_4} \vee x_6) \wedge \overline{x_5}$.

We say that clause $C = l_1 \vee ... \vee l_m$ is blocked by the partial assignment α that assigns each l_i to false ($\bot$). We say that 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 $\alpha(x) = \top$ and $x \in C$ or $\alpha(x) = \bot$ and $\overline{x} \in C$.

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

https://doi.org/

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. This may seem silly: if ϕ is unsatisfiable, then any clause C will be redundant. However, when solving, we do not know whether ϕ is satisfiable a priori. If we did, there would be no need to solve. Instead we must "justify" that a clause is unsatisfiable in some proof system. This can be helpful in certain cases, since adding a clause will constrain the set of possible solutions, but it can be unhelpful since it may negatively interact with solver heuristics.

Unit propagation is one of the core reasoning techniques of a SAT solver. If a formula ϕ, contains a unit clause l, i.e. a clause with only a single literal, then any satisfying assignment must set $\alpha(l) = \top$. Thus, this unit can be propagated, i.e. we instead look at the formula $\phi|_\alpha$ and thus every clause with l can be removed and every literal $\bar{l}$ can be removed. We repeat this process until there are no unit clauses remaining.

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 $\phi \wedge \overline{l_1} \wedge ... \wedge \overline{l_k}$ can be shown to be unsatisfiable via unit propagation. 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 $l_1 \vdash_1^\phi l_2$, i.e. "l_1 implies l_2 via unit propagation on ϕ," which means that $\phi \wedge l_1 \wedge \overline{l_2}$ can be shown to be unsatisfiable only using unit propagation. However, this is not very useful as it is quiet easy for the solver to figure this out. Instead, we must appeal to a more complicated notion of redundancy.

B. Propagation Redundant

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

Intuitively, we can think of adding C as the constraint that prunes all assignments that extend α. Since $F|_\alpha \vdash_1 F|_\omega$, it must be that any assignment that satisfies F_α, will satisfy F_ω. Additionally, since ω satisfies C, removing the assignments that extend α, will not affect satisfiability.

If C is a PR clause w.r.t ϕ, then ϕ and $\phi \wedge C$ are equisatisfiable. Indeed, the clause defined in Theorem 1 is PR with witness α_a. Checking if a clause is PR is NP-complete [**?**], so witnesses must be provided in proof checking.

PR clauses subsume many classes of redundant clauses including resolution asymmetric tautologies (RATs) [**?**], blocked clauses [**?**], set-blocked clauses![**?**], and globally-blocked clauses [**?**].

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 clasues as in the following theorem from Kiesel et al [**?**]:

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

This means that ϕ is satisfiable if and only if $\phi \wedge \bigwedge_{1 \leq i \leq m} (\overline{c_1} \vee ... \overline{c_n} \vee a_i)$ 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.

D. Related Work

PR clause learning was first introduced in an extension of the solver LINGELING [**?**]. 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 approach was shown to scale well on pigeonhole benchmarks.

This was generalized by SADICAL which provided two new variants of the positive reduct for more aggressive pruning of the search space [**?**]. Finally, this was extended by PRELEARN, which added unary and binary PR clauses in a preprocessing step [**?**]. This was done by initially considering all possible clauses and querying SADICAL to see which were PR.

Our work differs as we do not use a *positive reduct* to test is a clause is PR. Instead, our clauses PR by construction since they come from a conditional autarkies. 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.

III. METHODOLOGY

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 [**?**] to find the smallest possible α_c:

Algorithm 1: Minimiazing α_c in $\alpha = \alpha_c \sqcup \alpha_a$

```
Function LeastConditional(ψ, α):
    α_c := ∅;
    for C ∈ φ :
        if α touches C without satisfying C :
            α_c := α_c ∪ (α ∩ C̄);
    return α_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 $\bigvee_{c \in C \backslash C_0} \overline{c} \vee \bigvee_{a \in A_0} a$

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

Proof. $\Leftarrow$: This is immediate

$\Rightarrow$: 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 $\phi \wedge (\bigvee_{c \in C \backslash C_0} \overline{c} \vee \bigvee_{a \in A_0} a)$. 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 2: Algorithm finding A_0

```
Function FindA0(ψ, α_a, α_c):
    α_a := sort(α_a);
    α_a^SETS := init Array[len(α_a)];
    for i ∈ range(α_a) :
        propagate(−α_a[i]);
        implied := {};
        for c ∈ α_c :
            propagate(-c);
            if unsat :
                implied := implied ∪ {c};
        α_a^SETS[i] := implied;
    return greedySetCover(α_a^SETS);
```

In Algorithm 2, 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 2 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 1. 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

In this section, we will walk through the classic example of the Pigeonhole Principle as a SAT problem [], and how our techniques can be used to solve it.

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 with variable $x_{i,j}$ represents putting the i-th pigeon into the j-th hole. We can represent constraint (1) as $\overline{x_{i,j}} \vee \overline{x_{k,j}}$ for each $1 \leq i \neq k \leq m$ and constraint (2) as $\bigvee_{1 \leq j \leq n} x_{i,j}$ for each $1 \leq i \leq m$

We will only consider the case where $m = n + 1$ and this is always unsatisfiable. We are able to give an $O(n^3)$ size proof for the unsatisfiability of the pigeonhole principle.

The key idea is learning binary clauses such as either pigeon 1 is not in hole 2 or pigeon 2 is not in hole 1. Then we learn the clause that either pigeon 1 is not in hole 3 or pigeon 2 is not in hole 1, and so on. These clauses may seem strange, but we can prove (in the PR system) that their addition preserves satisfiability. We keep going until we eventually get that if pigeon 2 is in hole 1, then pigeon 1 is not in any hole. This violates constraint (1) and thus, we learn that pigeon 2 is not in hole 1.

This results in an $O(n^3)$ size pigeon hole proof for the pigeonhole benchmarks. Additionally, since discovering globally blocked clauses is linear in the size of the formula and the formula is $O(n^2)$ for the pigeonhole benchmarks. The total runtime of our algorithm on pigeonhole benchmarks is $O(n^5)$

To better motivate this, we will discuss an example based on the pigeonhole problem with 5 pigeons and 4 holes. First, we include a visual and then explain what we have done.

Essentially, we learn the following sets of clauses:

We make a set of alternating decisions $x_{1,1}, x_{2,j}$ and then $x_{1,j}x_{2,1}$ for $j = 2, ..., 4$.

The first pair (when $j = 2$ will allow us to lead us to learn the two globally blocked clauses:

$$\overline{x_{3,1}} \vee \overline{x_{4,1}} \vee \overline{x_{5,1}} \vee \overline{x_{3,2}} \vee \overline{x_{4,2}} \vee \overline{x_{5,2}} \vee \overline{x_{1,2}}$$
$$\overline{x_{3,1}} \vee \overline{x_{4,1}} \vee \overline{x_{5,1}} \vee \overline{x_{3,2}} \vee \overline{x_{4,2}} \vee \overline{x_{5,2}} \vee \overline{x_{2,1}}$$

Then when we introduce the decision $x_{1,2}, x_{2,1}$, we would reach a conflict by unit propagation, and thus introduce the conflict clause $\overline{x_{1,2}} \vee \overline{x_{2,1}}$.

We repeat this to also obtain the conflict clauses $\overline{x_{1,2}} \vee \overline{x_{2,1}}$, $\overline{x_{1,3}} \vee \overline{x_{2,1}}$, $\overline{x_{1,4}} \vee \overline{x_{2,1}}$.

The globally blocked clauses combined with these conflict clauses is enough to learn the clause $\overline{x_{2,1}}$. We repeat to learn $\overline{x_{3,1}}, \overline{x_{4,1}}$ and so on until we reach a contradiction.

V. Evaluation