Symmetric Explanation Learning: Effective Dynamic Symmetry Handling for SAT

Jo Devriendt(✉), Bart Bogaerts(✉), and Maurice Bruynooghe

Department of Computer Science, KU Leuven,
Celestijnenlaan 200A, 3001 Heverlee, Belgium
{jo.devriendt,bart.bogaerts,maurice.bruynooghe}@cs.kuleuven.be

Abstract. The presence of symmetry in Boolean satisfiability (SAT) problem instances often poses challenges to solvers. Currently, the most effective approach to handle symmetry is by *static symmetry breaking*, which generates asymmetric constraints to add to the instance. An alternative way is to handle symmetry *dynamically* during solving. As modern SAT solvers can be viewed as propositional proof generators, adding a symmetry rule in a solver's proof system would be a straightforward technique to handle symmetry dynamically. However, none of these proposed *symmetrical learning* techniques are competitive to static symmetry breaking. In this paper, we present *symmetric explanation learning*, a form of symmetrical learning based on learning symmetric images of explanation clauses for unit propagations performed during search. A key idea is that these symmetric clauses are only learned when they would restrict the current search state, i.e., when they are unit or conflicting. We further provide a theoretical discussion on symmetric explanation learning and a working implementation in a state-of-the-art SAT solver. We also present extensive experimental results indicating that symmetric explanation learning is the first symmetrical learning scheme competitive with static symmetry breaking.

Keywords: Boolean satisfiability · Symmetry · Proof theory · Symmetric learning · Dynamic symmetry breaking

1 Introduction

Hard combinatorial problems often exhibit symmetry. When these symmetries are not taken into account, solvers are often needlessly exploring isomorphic parts of a search space. Hence, we need methods to handle symmetries that improve solver performance on symmetric instances.

One common method to eliminate symmetries is to add symmetry breaking formulas to the problem specification [1,10], which is called *static symmetry breaking*. For the Boolean satisfiability problem (SAT), the tools Shatter [3] and BreakID [14] implement this technique; they function as a preprocessor that can be used with any SAT solver.

DOI: 10.1007/978-3-319-66263-3_6

Dynamic symmetry handling, on the other hand, interferes in the search process itself. For SAT, dynamic symmetry handling has taken on many forms. Early work on this topic dynamically detects symmetry after failing a search branch to avoid failing symmetrical search branches, using an incomplete symmetry detection strategy [7]. Next, *dynamic symmetry breaking* posts and retracts symmetry breaking formulas during search, dynamically detecting symmetry with graph automorphism techniques [5].

A more principled approach is implemented by SymChaff, a *structure-aware* SAT solver [29]. Next to a conjunctive normal form (CNF) theory, SymChaff assumes as input a special type of symmetry, structuring the Boolean variables from the theory in so-called *k-complete m-classes.* This structure is then used to branch over a subset of variables from the same class instead of over a single variable, allowing the solver to avoid assignments symmetric to these variables.

Arguably, the most studied dynamic symmetry handling approach is *symmetrical learning*, which allows a SAT solver to learn symmetrical clauses when constructing an unsatisfiability proof. The idea is that SAT solvers do not only *search* for a satisfying assignment, but simultaneously try to *prove* that none exists. For this, their theoretical underpinning is the propositional resolution proof system [28], which lets a SAT solver learn only those clauses that are resolvents of given or previously learned clauses. A SAT solver's proof system provides upper bounds on the effectiveness of SAT solvers when solving unsatisfiable instances. For instance, for encodings of the pigeonhole principle, no polynomial resolution proofs exist [19], and hence, a SAT solver cannot solve such encodings efficiently.

However, if one were to add a rule that under a symmetry argument, a symmetrical clause may be learned, then short proofs for problems such as the pigeonhole encoding exist [23]. As with the resolution rule, the central question for systems that allow the symmetry argument rule then becomes what selection of symmetrical clauses to learn, as learning all of them is infeasible [20] (nonetheless, some have experimented with learning all symmetrical clauses in a SAT solver [30]). The *symmetrical learning scheme* (SLS) only learns the symmetrical images of clauses learned by resolution, under some small set of generators of a given symmetry group [6]. Alternatively, *symmetry propagation* (SP) learns a symmetrical clause if it is guaranteed to propagate a new literal immediately [15]. Finally, for the graph coloring problem, symmetry-handling clauses can be learned based on *Zykov contraction* [20]. Unfortunately, none of these are competitive to state-of-the-art static symmetry breaking for SAT, as we will show with extensive experiments.

Symmetrical learning, as discussed in the previous paragraph differs significantly from the other methods discussed. These other methods all prune the search tree in a satisfiability-preserving way, but possibly also prune out models, for instance by adding symmetry breaking clauses or by not considering all possible choices at a given choice point. As such, they change the set of models of the theory, hence why we call them *symmetry breaking.* Symmetrical learning exploits symmetry in another way: if unsatisfiabilty of a certain branch of the search tree is concluded, it manages to learn that symmetrical parts of the search tree are also unsatisfiable; all clauses learned by symmetrical learning are consequences of the original specification. Hence, it never eliminates any models: symmetries are not broken, but merely exploited.

In this paper, we propose a new approach to symmetrical learning – *symmetric explanation learning* (SEL) – that improves upon our earlier work on symmetry propagation [15]. SEL's central idea is to learn a symmetric image of a clause only if (i) the clause is an explanation for a unit propagated literal and if (ii) the symmetric image itself is either unit or conflicting. In short, (i) limits the number of symmetric images under investigation to a manageable set, and (ii) guarantees that any learned symmetric clause is useful – it restricts the search state – at least once.

We experimentally validate this algorithm and conclude that SEL is the first dynamic symmetry exploitation approach to successfully implement a symmetric learning scheme. It performs on-par with the award winning static symmetry breaking tool BreakID [14] and outperforms previous symmetrical learning algorithms such as SLS [6] and SP [15].

The rest of this paper is structured as follows. In Sect. 2 we recall some preliminaries on symmetry and satisfiability solving. Afterwards, we introduce our new algorithm in Sect. 3 and compare it to related work in Sect. 4. We present experimental results in Sect. 5 and conclude in Sect. 6.

2 Preliminaries

Satisfiability problem. Let Σ be a set of Boolean variables and $\mathbb{B} = \{\mathbf{t}, \mathbf{f}\}$ the set of Boolean values denoting true and false respectively. For each $x \in \Sigma$, there exist two *literals*; the *positive* literal denoted by x and the *negative* literal denoted by $\neg x$. The negation $\neg(\neg x)$ of a negative literal $\neg x$ is the positive literal x, and vice versa. The set of all literals over Σ is denoted $\overline{\Sigma}$. A *clause* is a finite disjunction of literals $(l_1 \vee \ldots \vee l_n)$ and a *formula* is a finite conjunction of clauses $(c_1 \wedge \ldots \wedge c_m)$. By this definition, we implicitly assume a formula is an expression in *conjunctive normal form* (CNF).

A *(partial) assignment* is a set of literals ($\alpha \subset \overline{\Sigma}$) such that α contains at most one literal over each variable in Σ. Under assignment α, a literal l is said to be *true* if $l \in \alpha$, *false* if $\neg l \in \alpha$, and *unknown* otherwise. An assignment α *satisfies* a clause c if it contains at least one true literal under α. An assignment α *satisfies* a formula φ, denoted $\alpha \models \varphi$, if α satisfies each clause in φ. If $\alpha \models \varphi$, we also say that φ *holds* in α. A formula is *satisfiable* if an assignment exists that satisfies it, and is *unsatisfiable* otherwise. The Boolean satisfiability (SAT) problem consists of deciding whether a formula is satisfiable. Two formulas are *equisatisfiable* if both are satisfiable or both are unsatisfiable.

An assignment α is *complete* if it contains exactly one literal over each variable in Σ. A formula ψ (resp. clause c) is a *logical consequence* of a formula φ, denoted $\varphi \models \psi$ (resp. $\varphi \models c$), if for all complete assignments α satisfying φ, α satisfies ψ (resp. α satisfies c). Two formulas are *logically equivalent* if each is a logical consequence of the other.

A clause c is a *unit clause* under assignment α if all but one literals in c are false. A clause c is a *conflict clause* (or *conflicting*) under α if all literals in c are false.

We often consider a formula φ in the context of some assignment α. For this, we introduce the notion of φ *under* α, denoted as $\varphi \downarrow \alpha$, which is the formula obtained by conjoining φ with a unit clause (l) for each literal $l \in \alpha$. Formally, $\varphi \downarrow \alpha$ is the formula

$$\varphi \wedge \bigwedge_{l \in \alpha} l$$

Symmetry in SAT. Let π be a permutation of a set of literals $\overline{\Sigma}$. We extend π to clauses: $\pi(l_1 \vee \ldots \vee l_n) = \pi(l_1) \vee \ldots \vee \pi(l_n)$, to formulas: $\pi(c_1 \wedge \ldots \wedge c_n) = \pi(c_1) \wedge \ldots \wedge \pi(c_n)$, and to assignments: $\pi(\alpha) = \{\pi(l) \mid l \in \alpha\}$. We write permutations in *cycle notation*. For example, $(a\ b\ c)(\neg a\ \neg b\ \neg c)(\neg d\ d)$ is the permutation that maps a to b, b to c, c to a, $\neg a$ to $\neg b$, $\neg b$ to $\neg c$, $\neg c$ to $\neg a$, swaps d with $\neg d$, and maps any other literals to themselves.

Permutations form algebraic groups under the composition relation $(\circ)$. A set of permutations $\mathcal{P}$ is a set of *generators* for a permutation group $\mathbb{G}$ if each permutation in $\mathbb{G}$ is a composition of permutations from $\mathcal{P}$. The group $Grp(\mathcal{P})$ is the permutation group *generated* by all compositions of permutations in $\mathcal{P}$. The *orbit* $Orb_{\mathbb{G}}(x)$ of a literal or clause x under a permutation group $\mathbb{G}$ is the set $\{\pi(x) \mid \pi \in \mathbb{G}\}$.

A *symmetry* π of a propositional formula φ over Σ is a permutation over $\overline{\Sigma}$ that *preserves satisfaction to* φ; i.e., $\alpha \models \varphi$ iff $\pi(\alpha) \models \varphi$.

A permutation π of $\overline{\Sigma}$ is a symmetry of a propositional formula φ over Σ if the following sufficient syntactic condition is met:

– π commutes with negation: $\pi(\neg l) = \neg\pi(l)$ for all $l \in \overline{\Sigma}$, and
– π fixes the formula: $\pi(\varphi) = \varphi$.

It is easy to see that these two conditions guarantee that π maps assignments to assignments, preserving satisfaction to φ.

Typically, only this syntactical type of symmetry is exploited, since it can be detected with relative ease. One first converts a formula φ over variables Σ to a colored graph such that any automorphism – a permutation of a graph's nodes that maps the graph onto itself – of the graph corresponds to a permutation of $\overline{\Sigma}$ that commutes with negation and that fixes the formula. Next, the graph's automorphism group is detected by tools such as NAUTY [25], SAUCY [22] or BLISS [21], and is translated back to a symmetry group of φ.

The technique we present in this paper works for all kinds of symmetries, syntactical and others. However, our implementations use BREAKID for symmetry detection, which only detects a syntactical symmetry group by the method described above.

2.1 Conflict Driven Clause Learning SAT Solvers

We briefly recall some of the characteristics of modern conflict driven clause learning SAT (CDCL) solvers [24].

A CDCL solver takes as input a formula φ over a set of Boolean variables Σ. As output, it returns an (often complete) assignment satisfying φ, or reports that none exists.

Internally, a CDCL solver keeps track of a partial assignment α – called the *current assignment* – which initially is empty. At each search step, the solver chooses a variable x for which the current assignment α does not yet contain a literal, and adds either the positive literal x or the negative literal $\neg x$ to α. The added literal is now a *choice literal*, and may result in some clauses becoming unit clauses under the *refined* current assignment. This prompts a *unit propagation* phase, where for all unknown literals l occurring in a unit clause, the current assignment is extended with l. Such literals are *propagated literals*; we refer to the unit clause that initiated l's unit propagation as l's *explanation clause.* If no more unit clauses remain under the resulting assignment, the unit propagation phase ends, and a new search step starts by deciding on a next choice literal.

During unit propagation, a clause c can become conflicting when another clause propagates the last unknown literal l of c to false. At this moment, a CDCL solver will construct a *learned clause* by investigating the explanation clauses for the unit propagations leading to the conflict clause. This learned clause c is a logical consequence of the input formula, and using c in unit propagation prevents the conflict from occurring again after a *backjump*.[1] We refer to the set of learned clauses of a CDCL solver as the *learned clause store* Δ.

Formally, we characterize the *state* of a CDCL solver solving a formula φ by a quadruple $(\alpha, \gamma, \Delta, \mathcal{E})$, where

- α is the current assignment,
- $\gamma \subseteq \alpha$ is the set of choice literals – the set of literals $\alpha \setminus \gamma$ are known as *propagated* literals,
- Δ is the learned clause store,
- $\mathcal{E}$ is a function mapping the propagated literals $l \in \alpha \setminus \gamma$ to their explanation clause $\mathcal{E}(l)$, which can be either a clause from the input formula φ or from the learned clause store Δ.

During the search process, the invariant holds that the current assignment is a logical consequence of the decision literals, given the input formula. Formally:

$$\varphi \downarrow \gamma \models \varphi \downarrow \alpha.$$

Secondly, the learned clauses are logical consequences of the input formula:

$$\varphi \models c \text{ for each } c \in \Delta.$$

3 Symmetric Explanation Learning

From the definition of symmetry, the following proposition easily follows:

1 *Backjumping* is a generalization of the more classical *backtracking* over choices in combinatorial solvers.

Proposition 1. *Let φ be a propositional formula, π a symmetry of φ, and c a clause. If $\varphi \models c$, then also $\varphi \models \pi(c)$.*

Proof. If $\varphi \models c$ then $\pi(\varphi) \models \pi(c)$, as π renames the literals in formulas and clauses. Symmetries preserve models, hence $\pi(\varphi)$ is logically equivalent to φ, hence $\varphi \models \pi(c)$.

Since learned clauses are always logical consequences of the input formula, every time a CDCL solver learns a clause c, one may apply Proposition 1 and add $\pi(c)$ as a learned clause for every symmetry π of some symmetry group of $\mathbb{G}$. This is called *symmetrical learning*, which extends the resolution proof system underpinning a SAT solver's learning technology with a symmetry rule.

Symmetrical learning can be used as a symmetry handling tool for SAT: because every learned clause prevents the solver from encountering a certain conflict, the orbit of this clause under the symmetry group will prevent the encounter of all symmetrical conflicts, resulting in a solver never visiting two symmetrical parts of the search space.

However, since the size of permutation groups can grow exponentially in the number of permuted elements, learning all possible symmetrical clauses will in most cases add too many symmetrical clauses to the formula to be of practical use. Symmetrical learning approaches need to limit the amount of symmetrical learned clauses [20].

Given a set of input symmetries $\mathcal{P}$, the idea behind symmetric explanation learning (SEL) is to aim at learning symmetrical variants of learned clauses on the moment these variants propagate. A naive way to obtain this behaviour would be to check at each propagation phase for each clause $c \in \Delta$ and each symmetry $\pi \in \mathcal{P}$ whether $\pi(c)$ is a unit clause. Such an approach would have an unsurmountable overhead. Therefore, we implemented SEL using two optimizations.

The first is that we make a selection of "interesting" clauses: the symmetrical variants of clauses in Δ that are explanation clauses of some propagation in the current search state. The intuition is that an explanation clause c contains mostly false literals, so, assuming that the number of literals permuted by some symmetry π is much smaller than the total number of literals in the formula, $\pi(c)$ has a good chance of containing the same mostly false literals.

Secondly, we store those promising symmetrical variants in a separate *symmetrical learned clause store* Θ. Clauses in this store are handled similar to clauses in Δ, with the following differences:

1. propagation with Δ is always prioritized over propagation with Θ,
2. whenever a clause in Θ propagates, it is added to Δ,
3. whenever the solver backjumps over a propagation of a literal l, all symmetrical clauses $\pi(\mathcal{E}(l))$ are removed from Θ.

The first two points ensure that no duplicate clauses will ever be added to Δ without the need for checking for duplicates. Indeed, by prioritizing propagation with Δ, a clause in Θ can only propagate if it is not a part of Δ yet. The third point guarantees that Θ contains only symmetrical variants of clauses that have shown to be relevant in the current branch of the search tree.

On a technical note, Θ contains clauses $\pi(\mathcal{E}(l))$ from the moment l is propagated until a backjump unassigns l. As a result, it is useless to add $\pi(\mathcal{E}(l))$ to Θ if it is satisfied at the moment l is propagated, as it will never become an unsatisfied unit clause before backjumping over l. Similarly, it is not necessary to store any literals of $\pi(\mathcal{E}(l))$ that are false at the moment l is propagated, as these will not change status before backjumping over l. To combat this, Θ contains an approximation $\pi(\mathcal{E}(l))^*$ of $\pi(\mathcal{E}(l))$, which excludes any literals that are false at the moment l is propagated. If $\pi(\mathcal{E}(l))^*$ ever becomes unit, so does $\pi(\mathcal{E}(l))$. At this point, we recover the original clause $\pi(\mathcal{E}(l))$ from some stored reference to π and l, by simply applying π to $\mathcal{E}(l)$ again. Additionally, before adding a unit $\pi(\mathcal{E}(l))$ as a learned clause to Δ, our implementation performs a self-subsumption clause simplification step, as this is a simple optimization leading to stronger learned clauses [31].

Finally, keeping track of unit clauses in Θ during refinement of the current assignment is efficiently done by the well-known two-watched literal scheme [27].

We give pseudocode for SEL's behavior during a CDCL solver's propagation phase in Algorithm 1.

```
data: a formula $\varphi$, a set of symmetries $\mathcal{P}$ of $\varphi$, a partial assignment $\alpha$, a set of
      learned clauses $\Delta$, an explanation function $\mathcal{E}$, a set of symmetrical
      explanation clauses $\Theta$
repeat
    foreach unsatisfied unit clause $c$ in $\varphi$ or $\Delta$ do
        let $l$ be the unassigned literal in $c$;
        add $l$ to $\alpha$;
        set $c$ as $\mathcal{E}(l)$;
        foreach symmetry $\pi$ in $\mathcal{P}$ do
            if $\pi(\mathcal{E}(l))$ is not yet satisfied by $\alpha$ then
                add the approximation $\pi(\mathcal{E}(l))^*$ to $\Theta$;
            end
        end
    end
    if an unsatisfied unit clause $\pi(\mathcal{E}(l))^*$ in $\Theta$ exists then
        add the self-subsumed simplification of $\pi(\mathcal{E}(l))$ to $\Delta$;
    end
until no new literals have been propagated or a conflict has occurred;
```

Algorithm 1. propagation phase of a CDCL solver using SEL

Example 1 presents a unit propagation phase with the SEL technique.

Example 1. Let a CDCL solver have a state $(\alpha, \gamma, \Delta, \mathcal{E})$ with current assignment $\alpha = \emptyset$, choice $\gamma = \emptyset$, learned clause store $\Delta = \{(a \vee b), (\neg c \vee d \vee e)\}$ and explanation function $\mathcal{E}$ the empty function. Let $\pi = (a\ c)(\neg a\ \neg c)(b\ d)(\neg b\ \neg d)$ be a syntactical symmetry of the input formula φ, and assume for the following exposition that no propagation happens from clauses in φ. As the current assignment is currently empty, the symmetrical learned clause store Θ is empty as well.

Suppose the CDCL algorithm chooses $\neg a$, so $\alpha = \gamma = \{\neg a\}$. During unit propagation, the CDCL algorithm propagates b, so $\alpha = \{\neg a, b\}$, $\gamma = \{\neg a\}$ and $\mathcal{E}(b) = a \vee b$. By Algorithm 1, SEL adds $\pi(\mathcal{E}(b)) = c \vee d$ to Θ, so $\Theta = \{c \vee d\}$. No further unit propagation is possible, and $c \vee d$ is not unit or conflicting, so the solver enters a new decision phase.

We let the solver choose $\neg d$, so $\alpha = \{\neg a, b, \neg d\}$, $\gamma = \{\neg a, \neg d\}$. Still, no unit propagation on clauses from φ or from the learned clause store Δ is possible. However, $c \vee d$ in Θ is unit, so SEL adds $c \vee d$ to Δ.

Now unit propagation is reinitiated, leading to the propagation of c with reason $\mathcal{E}(c) = c \vee d$ and e with reason $\mathcal{E}(e) = \neg c \vee d \vee e$, so $\alpha = \{\neg a, b, \neg d, c, e\}$. As both $\pi(\mathcal{E}(c)) = a \vee b$ and $\pi(\mathcal{E}(e)) = \neg a \vee d \vee e$ are satisfied by α, they are not added to Θ. No further propagation is possible, ending the propagation loop. ▲

Note that if a symmetry π is a syntactic symmetry of a formula φ, SEL will never learn a symmetrical clause $\pi(c)$ from a clause $c \in \varphi$, as $\pi(c) \in \varphi$ already, and has propagation priority on any other $\pi(c)$ constructed by SEL. Moreover, due to technical optimizations, $\pi(c)$ will not even be constructed by SEL, as it is satisfied due to unit propagation from φ's clauses. From another perspective, any clause learned by SEL is the symmetrical image of some previously learned clause.

Also note that SEL is able to learn symmetrical clauses of symmetry compositions $\pi' \circ \pi$, with π and π' two symmetries of the input formula. This happens when at a certain point, c is an explanation clause and $\pi(c)$ an unsatisfied unit clause, and at some later moment during search, $\pi(c)$ is an explanation clause and $\pi'(\pi(c))$ an unsatisfied unit clause.

3.1 Complexity of SEL

Assuming a two-watched literal implementation for checking the symmetrical clause store Θ on conflict or unit clauses, the computationally most intensive step for SEL is filling Θ with symmetrical explanation clauses during unit propagation. Worst case, for each propagated literal l, SEL constructs $\pi(\mathcal{E}(l))$ for each π in the set of input symmetries $\mathcal{P}$. Assuming k to be the size of the largest clause in φ or Δ, this incurs a polynomial $O(|\mathcal{P}|k)$ time overhead at each propagation. As for memory overhead, SEL must maintain a symmetrical clause store containing $O(|\mathcal{P}||\alpha|)$ clauses, with α the solver's current assignment.

Of course, as with any symmetrical learning approach, SEL might flood the learned clause store with many symmetrical clauses. In effect, as only symmetrical explanation clauses are added to the learned clause store if they propagate or are conflicting, an upper bound on the number of symmetrical clauses added is the number of propagations performed by the solver, which can be huge. Aggressive learned clause store cleaning strategies might be required to maintain efficiency.

4 Related Work

In this section, we describe the relation of SEL and symmetric learning to other SAT solving techniques from literature.

4.1 SEL and SLS

One proposed way to restrict the number of clauses generated by symmetrical learning is the *symmetrical learning scheme* (SLS) [6]. Given an input set of symmetries $\mathcal{P}$, SLS only learns $\pi(c)$ for each $\pi \in \mathcal{P}$, and for each clause c learned by resolution after a conflict. If c contains only one literal, the set of symmetrical learned clauses from c is extended to the orbit of c under the group generated by $\mathcal{P}$.

A disadvantage of SLS is that not all symmetrical learned clauses are guaranteed to contribute to the search by propagating a literal at least once. This might result in lots of useless clauses being learned, which do not actively avoid a symmetrical part of the search space. It also is possible that some clauses learned by this scheme already belong to the set of learned clauses, since most SAT solvers do not perform an expensive check for duplicate learned clauses.

In Sect. 5, we give experimental results with an implementation of SLS.

4.2 SEL and SP

Another way to restrict the number of learned symmetrical clauses is given by *symmetry propagation* (SP) [15]. SP also learns symmetrical clauses only when they are unit or conflicting, but it uses the notion of *weak activity* to derive which symmetrical clauses it will learn.

Definition 1. Let φ be a formula and $(\alpha, \gamma, \Delta, \mathcal{E})$ the state of a CDCL solver. A symmetry π of φ is *weakly active* for assignment α and choice literals γ if $\pi(\gamma) \subseteq \alpha$.

Weak activity is a is a refinement of *activity*; the latter is a technique used in dynamic symmetry handling approaches for constraint programming [18,26].

Now, if a symmetry π of a formula φ is weakly active in the current solver state $(\alpha, \gamma, \Delta, \mathcal{E})$, then SP's implementation guarantees that for propagated literals $l \in \alpha \setminus \gamma$, $\pi(\mathcal{E}(l))$ is unit [15]. Then, SP adds any unsatisfied unit clauses $\pi(\mathcal{E}(l))$ to the learned clause store, and uses these to propagate $\pi(l)$.[2]

As SEL checks whether $\pi(\mathcal{E}(l))$ is unit for any input symmetry π, regardless of whether π is weakly active or not, SEL detects at least as many symmetrical clauses that are unit and unsatisfied as SP. Note that in Example 1, after making the choice $\neg a$, π is not weakly active, as $\neg a \in \alpha$ but $\pi(\neg a) = \neg c \notin \alpha$.

[2] SP focuses its presentation on propagating symmetrical literals $\pi(l)$ for weakly active symmetries, hence the name symmetry *propagation*. We present SP from a symmetrical learning point of view, using the fact that SP employs $\pi(\mathcal{E}(l))$ as a valid explanation clause for $\pi(l)$'s propagation.

Furthermore, after propagation of b and making the choice $\neg d$, SEL does learn the symmetrical explanation clause $\pi(\mathcal{E}(b)) = (c \vee d)$, propagating $\pi(a) = c$ in the process. This shows that SEL learns *strictly* more symmetrical clauses, performs more propagation than SP, and closes an increasing number of symmetrical search branches over time.

4.3 Compatibility of Symmetrical Learning and Preprocessing Techniques

As modern SAT solvers employ several preprocessing techniques [8] to transform an input formula φ to a smaller, hopefully easier, equisatisfiable formula φ', we should argue the soundness of SEL combined with those techniques. We do this by giving a sufficient condition of the preprocessed formula for which symmetrical learning remains a sound extension of a SAT solver's proof system.

Theorem 1. *Let φ and φ' be two formulas over vocabulary Σ, and let π be a symmetry of φ. Also, let φ' be*

1. *a logical consequence of φ and*
2. *equisatisfiable to φ.*

If clause c is a logical consequence of φ then $\varphi' \wedge \pi(c)$ is

1. *a logical consequence of φ and*
2. *equisatisfiable to φ.*

Proof. As c is a logical consequence of φ, $\pi(c)$ is as well, by Proposition 1. Hence, $\varphi' \wedge \pi(c)$ remains a logical consequence of φ, since both φ' and $\pi(c)$ hold in all models of φ, proving 1.

This also means that any satisfying assignment to φ is a satisfying assignment to $\varphi' \wedge \pi(c)$, so if φ is satisfiable, $\varphi' \wedge \pi(c)$ is satisfiable too. As the addition of an extra clause to a formula only reduces the number of satisfying assignments, if φ' is unsatisfiable, $\varphi' \wedge \pi(c)$ is unsatisfiable too. Since φ' is equisatisfiable with φ, $\varphi' \wedge \pi(c)$ is unsatisfiable if φ is unsatisfiable. This proves 2.

Corollary 1. *Let φ be a formula and π be a symmetry of φ. Symmetrical learning with symmetry π is sound for CDCL SAT solvers over a preprocessed formula φ' if φ' is a logical consequence of φ and if φ' is equisatisfiable to φ.*

Proof. Any clause c learned by resolution or symmetry application on clauses from φ' or logical consequences of φ is a logical consequence of φ. Hence, by Theorem 1, it is sound to learn the symmetrical clause $\pi(c)$ when solving for φ'.

In other words, if a preprocessing technique satisfies the conditions from Theorem 1, it is sound to symmetrically learn clauses in a CDCL SAT solver, as is done by the SEL algorithm.

This is not a trivial, but also not a strict requirement. For instance, common variable and clause elimination techniques based pioneered by SatELite [16] and still employed by i.a. Glucose satisfy this requirement. One exception, ironically, is static symmetry breaking, as the added symmetry breaking clauses are not logical consequences of the original formula. Also, a preprocessing technique that introduces new variables does not satisfy the above requirements, and risks to combine unsoundly with symmetrical learning.

4.4 Symmetrical Learning Does Not Break Symmetry

The earliest techniques to handle symmetry constructed formulas that removed symmetrical solutions from a problem specification, a process that *breaks* the symmetry in the original problem specification. Ever since, *handling* symmetry seems to have become eponymous with *breaking* it, even though a symmetrical learning based technique such as SEL only infers logical consequences of a formula, and hence does not a priori remove any solutions. In this paper, we tried to consistently use the term *symmetry handling* where appropriate.

An advantage of non-breaking symmetry handling approaches is that it remains possible for a solver to obtain any solution to the original formula. For instance, non-breaking symmetry handling approaches such as SEL can be used to generate solutions to a symmetric formula, which are evaluated under an asymmetric objective function [2]. Similarly, approaches such as SEL can be used in a #SAT solver [9].

5 Experiments

In this section, we present experiments gauging SEL's performance. We implemented SEL in the state-of-the-art SAT solver Glucose 4.0 [4] and made our implementation available online [12]. Symmetry was detected by running BreakID, which internally uses Saucy as graph automorphism detector.

All experiments have a 5000 s time limit and a 16 GB memory limit. The hardware was an Intel Core i5-3570 CPU with 32 GiB of RAM and Ubuntu 14.04 Linux as operating system. Detailed results and benchmark instances have been made available online[3].

We first present a preliminary experiment on *row interchangeability* – a particular form of symmetry detected by BreakID – in Subsect. 5.1, and give our main result in Subsect. 5.2.

5.1 Row Interchangeability

Row interchangeability is a particular form of symmetry where a subset of the literals are arranged as a matrix with k rows and m columns, and any permutation of the rows induces a symmetry [14]. This type of symmetry is common, occurring often when objects in some problem domain are interchangeable. For example, the interchangeability of pigeons in a pigeonhole problem, or the interchangeability of colors in a graph coloring problem lead to row interchangeability

[3] bitbucket.org/krr/sat_symmetry_experiments.

at the level of a propositional specification. This type of symmetry can be broken completely by static symmetry breaking formulas of polynomial size, resulting in exponential search space reduction in e.g. pigeonhole problem specifications [14].

While the dynamic symmetry breaking solver SymChaff specializes in row interchangeability, it is unclear how SEL (or symmetrical learning in general) can efficiently handle this type of symmetry. As SEL only supports an input set of simple symmetries as defined in Sect. 2, we currently use a set of generators as a representation for a row interchangeability symmetry group. We investigate two possible representations for a row interchangeability group with k rows:

– a *linear* representation, containing $k-1$ symmetries that swap consecutive rows, as well as the one symmetry swapping the first and last row.
– a *quadratic* representation, containing $k(k-1)/2$ symmetries that swap any two rows, including non-consecutive ones.

To experimentally verify the effectiveness of both approaches, we generated two benchmark sets. The first consists of unsatisfiable **pigeonhole** formulas, which assign k pigeons to $k-1$ holes. We only provided the row interchangeability symmetry group stemming from interchangeable pigeons to the symmetry handling routines. We shuffled the variable order of the formulas, to minimize lucky guesses by Glucose's heuristic. The number of pigeons in the instances ranges between 10 and 100.

The second benchmark set consists of 110 both satisfiable and unsatisfiable graph **coloring** problems, where we try to color graphs with k or $k-1$ colors, where k is the input graph's chromatic number[4]. We only provided the row interchangeability symmetry group stemming from interchangeable colors to the symmetry handling routines, ignoring any potential symmetry in the input graph. Input graphs are taken from Michael Trick's web page [32].

As a baseline, we use Glucose 4.0 coupled with the static symmetry breaking preprocessor BreakID, whose symmetry detection routine is disabled and only gets to break the row interchangeability groups mentioned previously. The results for **pigeonhole** are given in Table 1, and for **coloring** are given in Table 2. Solving time needed by BreakID's symmetry detection is ignored, as this was always less than 2.1 s, and the same for any approach.

On **pigeonhole**, the quadratic approach outperforms the linear approach, solving instances with up to 30 pigeons opposed to only 16 pigeons. However, even the quadratic approach does not reach the speedup exhibited by BreakID, easily handling instances with 100 pigeons. On **coloring**, the quadratic approach does manage to outperform both the linear approach and BreakID.

Based on this experiment, we default SEL to use a quadratic amount of row-swapping symmetry generators to represent a row interchangeability symmetry group.

[4] For the few instances where the chromatic number was not known, we made an educated guess based on the graph's name.

Table 1. Solving time in seconds of BreakID and SEL with different generator sets for **pigeonhole** interchangeability. A "-" means the time limit of 5000 s was reached.

# pigeons	BreakID	SEL–quadratic	SEL–linear
10	0	0	0.04
11	0	0.06	0.3
12	0	0.15	0.08
13	0	0.93	0.81
14	0	0.01	4.97
15	0	0.03	791.26
16	0	0.04	4766.31
17	0	0.53	-
18	0	0.25	-
19	0	3.73	-
20	0	42.85	-
25	0.01	0.9	-
30	0.02	277.57	-
40	0.05	-	-
50	0.13	-	-
70	0.41	-	-
100	1.47	-	-

Table 2. Total number of graph coloring instances solved within 5000 s for BreakID and SEL with different generator sets for graph **coloring** interchangeability.

# instances	BreakID	SEL–quadratic	SEL–linear
110	70	**87**	74

5.2 Evaluation of SEL

We evaluate SEL by comparing the following five solver configurations:

– Glucose: pure Glucose 4.0, without any symmetry detection or handling routines.
– BreakID: Glucose 4.0 coupled with the BreakID symmetry breaking preprocessor in its default settings.
– SEL: our implementation of SEL in Glucose 4.0, taking as input the symmetries detected by BreakID in its default settings. This includes any row interchangeability symmetry detected by BreakID, which is interpreted as a quadratic set of row swapping symmetries.
– SP: the existing implementation of Symmetry propagation [13] in the classic SAT solver MiniSAT [17], using its optimal configuration [15]. We slightly extended this implementation to take BreakID's symmetry output as input,

and interpreted any row interchangeability symmetry as a quadratic set of symmetries.
- SLS: our implementation of SLS in Glucose 4.0, taking as input the symmetries detected by BreakID in its default settings. This includes any row interchangeability symmetry detected by BreakID. Contrary to the other solvers, we interpret interchangeability here as the *linear* amount of generators. The reason for this is that we noticed in preliminary testing that SLS simply couldn't handle a quadratic number of generators: with this it almost always ran out of memory.

Our benchmark instances are partitioned in five benchmark sets:

- **app14**: the application benchmarks of the 2014 SAT competition. 300 instances; BreakID detected some symmetry for 160 of these.
- **hard14**: the hard-combinatorial benchmarks of the 2014 SAT competition. 300 instances; BreakID detected some symmetry for 107 of these.
- **app16**: the application benchmarks of the 2016 SAT competition. 299 instances; BreakID detected some symmetry for 131 of these.
- **hard16**: the hard-combinatorial benchmarks of the 2016 SAT competition. 200 instances; BreakID detected some symmetry for 131 of these.
- **highly**: an eclectic set of highly symmetric instances collected over the years. 204 instances; BreakID detected some symmetry for 202 of these. Instance families include graph coloring, pigeonhole, Ramsey theorem, channel routing, planning, counting, logistics and Urquhart's problems.

We reiterate that detailed experimental results, benchmark instances and source code of the employed systems have been made available online [11–13].

Table 3 lists the number of successfully solved instances for each of the five solver configurations and each of the five benchmark sets. Except for Glucose, BreakID's symmetry detection and breaking time are accounted in the total solving time.

Table 3. Total number of successfully solved instances for each of the five solver configurations and each of the five benchmark sets.

Benchmark set	Glucose	BreakID	SEL	SP	SLS
app14 (300/160)	**222**	220	215	179	187
hard14 (300/107)	174	**189**	188	172	175
app16 (299/131)	150	**151**	**151**	127	141
hard16 (200/131)	52	**93**	80	48	70
highly (204/202)	106	151	**152**	151	113

First and foremost, BreakID jumps out of the pack as the best all-around configuration: performing well on the strongly symmetric **highly** instances, as well as on the more challenging SAT competition instances, of which especially **app14** and **app16** feature very large instances.

Second, SEL seems quite competitive. In **hard14**, **app16** and **highly**, SEL and BreakID trade blows. Only on **hard16**, SEL's performance seems significantly inferior to BreakID.

Third, Glucose performs badly on **highly**, which can be expected given the strong symmetry properties of those instances. **hard16**'s low success rate is due to 35 pigeonhole instances and 38 highly symmetric *tseitingrid* instances.

Fourth, SP performs very well on **highly**, but is dead last in all other benchmark sets. This might be due to its embedding in the older MiniSAT, or due to the overhead of keeping track of weakly inactive symmetries.

Finally, SLS is not able to clinch the lead in any of the benchmark sets, and especially the bad results on **highly** are surprising for a symmetry exploiting technique. We conjecture that highly symmetric instances lead to an uncontrolled symmetrical clause generation, choking SLS's learned clause store.

For this, it is worth looking at the number of instances where a solver exceeded the 16 GB memory limit (a *memout*). These results are given in Table 4.

Table 4. Total number of instances that exceeded the 16 GB memory limit.

Benchmark set	Glucose	BreakID	SEL	SP	SLS
app14 (300/160)	**0**	**0**	18	23	41
hard14 (300/107)	**0**	**0**	1	1	13
app16 (299/131)	**0**	**0**	2	21	47
hard16 (200/131)	**0**	**0**	14	**0**	49
highly (204/202)	**0**	**0**	5	3	50

From Table 4, it is clear that all symmetrical learning approaches (SEL, SP, SLS) struggle with heavy memory consumption. SLS in particular is unable to solve many symmetrical instances due to memory constraints. SEL on the other hand, has relatively few memouts, concentrated mainly in the benchmark sets **app14** and **hard16** – the same as those were it had to give BreakID the lead.

We conclude that SEL is a viable symmetrical learning, and by extension, dynamic symmetry handling approach. However, care must be taken that not too many symmetrical clauses are learned, filling up all available memory.

6 Conclusion

In this paper, we presented symmetric explanation learning (SEL), a form of symmetrical learning based on learning symmetric images of explanation clauses for unit propagations performed during search. A key idea is that these symmetric clauses are only learned when they would restrict the current search state, i.e., when they are unit or conflicting.

We related SEL to symmetry propagation (SP) and the symmetrical learning scheme (SLS), and gave a sufficient condition on when symmetrical learning can be combined with common SAT preprocessing techniques.

We further provided a working implementation of SEL and SLS embedded in Glucose, and experimentally evaluated SEL, SLS, SP, Glucose and the symmetry breaking preprocessor BreakID on more than 1300 benchmark instances. Our conclusion is that SEL outperforms other symmetrical learning approaches, and functions as an effective general purpose dynamic symmetry handling technique, almost closing the gap with static symmetry breaking.

For future work, we expect that the efficiency of our implementation can still be improved. Specifically, investigating how to reduce SEL's memory overhead, perhaps by aggressive learned clause deletion techniques, has definite potential.

On the theoretical front of symmetrical learning, much work remains to be done on effectiveness guarantees similar to those provided by *complete* static symmetry breaking. Informally, a symmetry breaking formula ψ is complete for a given symmetry group if no two symmetric solutions satisfy ψ [33]. For instance, BreakID guarantees that its symmetry breaking formulas are complete for row interchangeability symmetry, resulting in very fast pigeonhole solving times. Maybe a similar guarantee can be given for some form of symmetrical learning?

References

1. Aloul, F., Ramani, A., Markov, I., Sakallah, K.: Solving difficult SAT instances in the presence of symmetry. In: 39th Design Automation Conference, 2002 Proceedings, pp. 731–736 (2002)
2. Aloul, F.A., Ramani, A., Markov, I.L., Sakallah, K.A.: Dynamic symmetry-breaking for boolean satisfiability. Ann. Math. Artif. Intell. **57**(1), 59–73 (2009)
3. Aloul, F.A., Sakallah, K.A., Markov, I.L.: Efficient symmetry breaking for boolean satisfiability. IEEE Trans. Comput. **55**(5), 549–558 (2006)
4. Audemard, G., Simon, L.: Predicting learnt clauses quality in modern SAT solvers. In: Boutilier, C. (ed.) IJCAI, pp. 399–404 (2009)
5. Benhamou, B., Nabhani, T., Ostrowski, R., Saïdi, M.R.: Dynamic symmetry breaking in the satisfiability problem. In: Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning, LPAR-16, Dakar, Senegal, 25 April –1 May 2010
6. Benhamou, B., Nabhani, T., Ostrowski, R., Saïdi, M.R.: Enhancing clause learning by symmetry in SAT solvers. In: Proceedings of the 2010 22nd IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2010, vol. 01, pp. 329–335 (2010). http://dx.doi.org/10.1109/ICTAI.2010.55
7. Benhamou, B., Saïs, L.: Tractability through symmetries in propositional calculus. J. Autom. Reason. **12**(1), 89–102 (1994). http://dx.doi.org/10.1007/BF00881844
8. Biere, A.: Preprocessing and inprocessing techniques in SAT. In: Eder, K., Lourenço, J., Shehory, O. (eds.) HVC 2011. LNCS, vol. 7261, p. 1. Springer, Heidelberg (2012). doi:10.1007/978-3-642-34188-5_1
9. Birnbaum, E., Lozinskii, E.L.: The good old Davis-Putnam procedure helps counting models. J. Artif. Intell. Res. **10**, 457–477 (1999)

10. Crawford, J.M., Ginsberg, M.L., Luks, E.M., Roy, A.: Symmetry-breaking predicates for search problems. In: Principles of Knowledge Representation and Reasoning, pp. 148–159. Morgan Kaufmann (1996)
11. Devriendt, J.: Binaries, experimental results and benchmark instances for "symmetric explanation learning: effective dynamic symmetry handling for SAT". bitbucket.org/krr/sat_symmetry_experiments
12. Devriendt, J.: An implementation of symmetric explanation learning in Glucose on Bitbucket. bitbucket.org/krr/glucose-sel
13. Devriendt, J.: An implementation of symmetry propagation in MiniSat on Github. github.com/JoD/minisat-SPFS
14. Devriendt, J., Bogaerts, B., Bruynooghe, M., Denecker, M.: Improved static symmetry breaking for SAT. In: Creignou, N., Le Berre, D. (eds.) SAT 2016. LNCS, vol. 9710, pp. 104–122. Springer, Cham (2016). doi:10.1007/978-3-319-40970-2_8
15. Devriendt, J., Bogaerts, B., De Cat, B., Denecker, M., Mears, C.: Symmetry propagation: improved dynamic symmetry breaking in SAT. In: IEEE 24th International Conference on Tools with Artificial Intelligence, ICTAI 2012, Athens, Greece, pp. 49–56. IEEE Computer Society, 7–9 November 2012. http://dx.doi.org/10.1109/ICTAI.2012.16
16. Eén, N., Biere, A.: Effective preprocessing in SAT through variable and clause elimination. In: Bacchus, F., Walsh, T. (eds.) SAT 2005. LNCS, vol. 3569, pp. 61–75. Springer, Heidelberg (2005). doi:10.1007/11499107_5
17. Eén, N., Sörensson, N.: An extensible SAT-solver. In: Giunchiglia, E., Tacchella, A. (eds.) SAT 2003. LNCS, vol. 2919, pp. 502–518. Springer, Heidelberg (2004). doi:10.1007/978-3-540-24605-3_37
18. Gent, I.P., Smith, B.M.: Symmetry breaking in constraint programming. In: Proceedings of ECAI-2000, pp. 599–603. IOS Press (2000)
19. Haken, A.: The intractability of resolution. Theor. Comput. Sci. **39**, 297–308 (1985). Third Conference on Foundations of Software Technology and Theoretical Computer Science. http://www.sciencedirect.com/science/article/pii/0304397585901446
20. Heule, M., Keur, A., Maaren, H.V., Stevens, C., Voortman, M.: CNF symmetry breaking options in conflict driven SAT solving (2005)
21. Junttila, T., Kaski, P.: Engineering an efficient canonical labeling tool for large and sparse graphs. In: Applegate, D., Brodal, G.S., Panario, D., Sedgewick, R. (eds.) Proceedings of the Ninth Workshop on Algorithm Engineering and Experiments and the Fourth Workshop on Analytic Algorithms and Combinatorics, pp. 135–149. SIAM (2007)
22. Katebi, H., Sakallah, K.A., Markov, I.L.: Symmetry and satisfiability: an update. In: Strichman, O., Szeider, S. (eds.) SAT 2010. LNCS, vol. 6175, pp. 113–127. Springer, Heidelberg (2010). doi:10.1007/978-3-642-14186-7_11
23. Krishnamurthy, B.: Short proofs for tricky formulas. Acta Inf. **22**(3), 253–275 (1985). http://dl.acm.org/citation.cfm?id=4336.4338
24. Marques-Silva, J.P., Sakallah, K.A.: GRASP: a search algorithm for propositional satisfiability. IEEE Trans. Comput. **48**(5), 506–521 (1999)
25. McKay, B.D., Piperno, A.: Practical graph isomorphism, II. J. Symbolic Comput. **60**, 94–112 (2014). http://www.sciencedirect.com/science/article/pii/S0747717113001193
26. Mears, C., García de la Banda, M., Demoen, B., Wallace, M.: Lightweight dynamic symmetry breaking. Constraints **19**(3), 195–242 (2014). http://dx.doi.org/10.1007/s10601-013-9154-2

27. Moskewicz, M., Madigan, C., Zhao, Y., Zhang, L., Malik, S.: Chaff: engineering an efficient SAT solver. In: DAC 2001, pp. 530–535. ACM (2001)
28. Pipatsrisawat, K., Darwiche, A.: On the power of clause-learning SAT solvers as resolution engines. Artif. Intell. **175**(2), 512–525 (2011). http://dx.doi.org/10.1016/j.artint.2010.10.002
29. Sabharwal, A.: SymChaff: exploiting symmetry in a structure-aware satisfiability solver. Constraints **14**(4), 478–505 (2009). http://dx.doi.org/10.1007/s10601-008-9060-1
30. Schaafsma, B., van Heule, M.J.H., Maaren, H.: Dynamic symmetry breaking by simulating zykov contraction. In: Kullmann, O. (ed.) SAT 2009. LNCS, vol. 5584, pp. 223–236. Springer, Heidelberg (2009). doi:10.1007/978-3-642-02777-2_22
31. Sörensson, N., Eén, N.: MiniSat v1.13 - a Sat solver with conflict-clause minimization. 2005. sat-2005 poster. Technical report (2005)
32. Trick, M.: Network resources for coloring a graph (1994). mat.gsia.cmu.edu/COLOR/color.html
33. Walsh, T.: Symmetry breaking constraints: Recent results. CoRR abs/1204.3348 (2012)