Symmetry Propagation (in SAT): A Novel Approach To Dynamic Symmetry Breaking

Jo Devriendt[1], Bart Bogaerts[1], Christopher Mears[2], Broes De Cat[1], and Marc Denecker[1]

[1] KU Leuven, Department of Computer Science
Celestijnenlaan 200A, 3001 Heverlee, Belgium
jo.devriendt@student.kuleuven.be, {bart.bogaerts, broes.decat, marc.denecker}@cs.kuleuven.be,
[2] Caulfield School of IT, Monash University, Australia
chris.mears@monash.edu

Abstract Many symmetry breaking methods for SAT and CSP have been proposed. In this paper, we present Symmetry Propagation (in SAT): a new dynamic symmetry breaking approach based on the notion of weakly active symmetries to detect symmetry propagations. We describe the algorithm for a conflict-driven clause learning SAT solver using unit propagation, and show with experimental data that on many benchmarks, our approach outperforms the state-of-the-art symmetry breaking method Shatter.

1 Introduction

Many difficult satisfiability (SAT) and constraint satisfaction problems (CSP) exhibit symmetry properties. Exploiting these symmetry properties can significantly reduce the time needed to solve the problems. The process of exploiting symmetries is called *symmetry breaking*, which aims to speed up search by only considering parts of the search space which are not symmetrical to already considered parts. Symmetry breaking can be subdivided into two categories: *static* and *dynamic* symmetry breaking.

Static symmetry breaking adds to an original problem extra constraints which exclude a large part of the search space. The goal is to let the solver only explore the remaining small part containing only solutions that are not symmetrical to each other. The advantages of static symmetry breaking are twofold. Firstly, one does not need to adjust the actual solver, but only the problem at hand, therefore static symmetry breaking can be done in a preprocessing step regardless of how the problem will be solved. Secondly, static symmetry breaking plainly performs well.

A disadvantage of static symmetry breaking is that it forces the solver to a particular part of the search space which might not be the most efficient part for the problem and solver at hand. Another disadvantage is that static symmetry breaking cannot always be applied. This is the case when symmetries are detected locally [3], when the extra constraints are too large for a solver to handle, or when a problem is symmetrical, but an optimization function over the solutions is not [2]. In these cases, dynamic symmetry breaking proves useful.

Dynamic symmetry breaking interacts with the solver during search, ensuring search paths symmetrical to an already inspected path are avoided. The main difficulty in dynamic symmetry breaking is how to detect when symmetry information such as symmetrical clauses or extra propagations can be added to the solver, without slowing down the solver too much.

In this paper, we present a dynamic symmetry breaking algorithm that — given a set of previously detected "input symmetries" — speeds up search by propagating literals symmetrical to already propagated literals. We christened our symmetry breaking approach *Symmetry Propagation (in SAT)* or SP(SAT). To test the performance of SP(SAT), we compared an implementation of SP(SAT) in Minisat [8] to the static symmetry breaking provided by Shatter [1]. Although we do not show this here, the presented method can also be applied in other contexts, such as constraint programming systems.

This paper is organized as follows: we start out by a brief overview of how modern SAT solvers work in Sect. 2. Section 3 delves into theoretical properties of symmetry in SAT, providing tools such as the notion of *weak activity* to detect logical consequences of a logical theory. This section also contains a detailed description of an implementation of SP(SAT) in a SAT solver. Implementing SP(SAT) in Minisat yielded the experimental results given in Sect. 4, while we compare SP(SAT) to other symmetry breaking algorithms available in literature in Sect. 5. Further research opportunities are discussed in Sect. 6, and we conclude the paper in Sect. 7.

2 Background

2.1 The SAT-Problem

We define a *boolean satisfiability problem* (SAT) as a *theory* for which we have to decide whether or not a *model* exists. We assume a theory T is a conjunction of *clauses*, a clause is a disjunction of *literals*, and a literal is an expression of the form x or $\neg x$, with x a boolean variable. The set of variables occurring in T is denoted by $\Sigma(T)$ and the set of all possible literals in T is $\bar{\Sigma}(T)$. An *assignment* for a SAT-problem is a set of literals $\alpha \subseteq \bar{\Sigma}(T)$ such that α contains at most one literal of every variable in $\Sigma(T)$. An assignment is *complete* if it contains exactly one literal for every variable in $\Sigma(T)$, if not, it is *partial.* A literal l is *true* under an assignment α if $l \in \alpha$, *false* if $\neg l \in \alpha$, and *undefined* otherwise. A clause c is a *conflict clause* under assignment α if it contains only false literals, c is *satisfied* under α if it contains at least one true literal, and c is a unit clause under α if all but one literals in c are false.

A complete assignment α is a *model* for T if all clauses of T are satisfied under α. A clause c is a logical consequence of T if c is satisfied in all models of T; we denote this by $T \models c$. A theory T' is a logical consequence of T if all of its clauses are. A theory T is *satisfiable* if there exists a model for T. The

SAT-problem consists of deciding whether or not a theory T is satisfiable. If T is satisfiable, the SAT-problem is usually solved by returning a model for T.

Finally, by $T \cup \alpha$, we denote a theory that consists of the clauses of T and for each literal $l \in \alpha$ the clause l consisting of one literal.

2.2 A Conflict Driven Clause Learning Solver

We briefly recall some of the concepts of a *Conflict Driven Clause Learning* (CDCL) SAT solver [17].

The state of a CDCL solver solving a theory T is characterized by a triple $(\alpha, \delta, \mathit{expl})$, where α is the current assignment, $\delta \subseteq \alpha$ the set of *choice literals* such that $T \cup \delta \models T \cup \alpha$, and expl a function from $\alpha \setminus \delta$ to T such that, for each $l \in \alpha \setminus \delta$, $\mathit{expl}(l)$ is the clause that propagated l. The literals in $\alpha \setminus \delta$ are *propagated literals*, and $\mathit{expl}(l)$ is the *explanation clause* for l, which is a unit clause under α.

A CDCL solver improves the backtracking mechanism of the DPLL algorithm [6]. Instead of backtracking over the last choice when a conflict occurs, CDCL analyzes the conflict, and adds a clause to the theory to ensure the conflict can never occur again. The added clause is called the *learned clause*, and expresses the reason why the conflict occurred. Learned clauses are constructed by applying resolution to the explanation clauses for certain propagated literals, thus learned clauses are logical consequences of the theory and are stored in the *learned clause store*.

3 Symmetry Propagation (in SAT)

3.1 Theoretical Properties of Symmetries

Given a theory T, let σ be a permutation of $\bar{\Sigma}(T)$. We can extend σ in a natural way to be a mapping of clauses, assignments and theories, and by slight abuse of notation, we will identify σ with those extensions. σ is said to *commute with negation* if $\sigma(\neg l) = \neg\sigma(l)$ for every literal l. σ is called a *symmetry* of T if it commutes with negation and if for every complete assignment α: α is a model of T if and only if $\sigma(\alpha)$ is. Equivalently, σ is a symmetry if it commutes with negation and if T and $\sigma(T)$ are logically equivalent. We will always denote our symmetries in disjoint cycle notation, i.e. $(ab)(c\neg de)$ denotes the symmetry that sends the literals a to b, b to a, c to $\neg d$, $\neg d$ to e and e to c. We implicitly assume that this symmetry sends all non-occurring literals to itself, and that it also sends $\neg a$ to $\neg b$, d to $\neg e$,... The *order* of a symmetry σ is the smallest n such that σ^n is the identity. For a set $\mathcal{S}$ of symmetries, we define the *orbit* of a literal l as $\{\sigma(l) | \sigma \in \mathcal{S}\}$.

The symmetries of a theory T form a subgroup of the group of all permutations of $\bar{\Sigma}(T)$. We call this subgroup the *symmetry group* of T. In general, the size of this group may be exponentially larger than a set of generators of this group. In practice, this means that if theories have symmetries, they often have very many of them.

A well known property of symmetries of a SAT problem is the following:

Proposition 1. *Given a SAT-problem with theory T, a symmetry σ for this problem and a clause c, if $T \models c$, then also $T \models \sigma(c)$.*

Since learned clauses are always logical consequences of the initial theory, every time a SAT solver learns a clause c, we may apply the Proposition 1 and add $\sigma(c)$ as a learned clause. In fact, this approach can be used as a symmetry breaking tool for SAT. Because every learned clause prevents the solver from encountering a certain conflict, the image of this clause under the symmetry will prevent the encounter of the symmetrical conflict, resulting in symmetry breaking. However, since there are possibly exponentially many symmetries, this approach will most likely add too many clauses to the theory to be of practical use. Nonetheless, several symmetry breaking methods are based on Proposition 1. To avoid the exponential number of learned clauses, all of them limit the number of learned clauses somehow (see for example [4], [12]).

SP(SAT) on the other hand bases its symmetry breaking on propagation of symmetrical literals instead of adding extra clauses. The following corollary of Proposition 1 is the foundation for the symmetry propagation algorithm defined below.

Corollary 1. *Let T be the theory of a SAT problem and α an assignment. If σ is a symmetry of $T \cup \alpha$ and $T \cup \alpha \models l$, then also $T \cup \alpha \models \sigma(l)$.*

Translated to the language of SAT algorithms, Corollary 1 means that if a solver has state $(\alpha, \delta, expl)$ and the solver propagates l, then for every symmetry σ of $T \cup \alpha$, $\sigma(l)$ can also be propagated. In fact, this approach has been taken by Benhamou e.a. [3]: when the solver backtracks from a certain choice literal l, a local symmetry group G of $T \cup \alpha$ is computed, and the orbit of $\neg l$ under G is propagated. The weakness of this algorithm is that repeated computation of symmetries of $T \cup \alpha$ is very expensive. In this paper, we develop an efficient detection algorithm that identifies a useful subset of the symmetries of $T \cup \alpha$.

Our algorithm is based on the notion of *activity*, introduced by Mears e.a. for their Lightweight Dynamic Symmetry Breaking algorithm [12].

Definition 1. *A symmetry σ is called* active *under assignment α if $\sigma(\alpha) = \alpha$.*

The following proposition shows where activity can be useful.

Proposition 2. *Let T be a theory and α an assignment. If σ is a symmetry of T that is active under α, then σ is also a symmetry of $T \cup \alpha$.*

Proposition 2 states that the symmetries of T active under assignment α form a subset of the symmetries of $T \cup \alpha$. By Corollary 1, we can conclude that if a symmetry σ of T is active under assignment α, and a literal l is propagated, we are also allowed to propagate $\sigma(l)$. Since the composition of two symmetries of $T \cup \alpha$ is again a symmetry of $T \cup \alpha$, we can also propagate $\sigma^2(l), \sigma^3(l), \ldots$ After doing so, σ will again be active.

In this paper, we introduce the notion of *weakly active* symmetries, a notion that generalizes activity.

Definition 2. *Given a theory T, let $(\alpha, \delta, expl)$ be the state of a solver. A symmetry σ of T is* weakly active *for assignment α and choice literals δ if $\sigma(\delta) \subseteq \alpha$.*

We will show that a literal $\sigma(l)$ is a logical consequence of a theory $T \cup \alpha$ if l is a logical consequence of $T \cup \alpha$ and σ a weakly active symmetry of T under α.

Proposition 3. *Let T be a theory and α an assignment. If there exists a subset $\delta \subseteq \alpha$ and a symmetry σ of T such that $\sigma(\delta) \subseteq \alpha$ and $T \cup \delta \models T \cup \alpha$, then σ is also a symmetry of $T \cup \alpha$.*

Proof. Since σ is a symmetry of T, we know that it commutes with negation, thus all we need to prove is that $T \cup \alpha$ and $\sigma(T) \cup \sigma(\alpha)$ are logically equivalent.

Because σ is a symmetry of T, $\sigma(T)$ and T are logically equivalent. Combining this with the fact that $T \cup \delta \models T \cup \alpha$ allows us to derive $T \cup \sigma(\delta) \models T \cup \sigma(\alpha)$, and because $\sigma(\delta) \subseteq \alpha$, $T \cup \alpha \models T \cup \sigma(\alpha)$. If we let $\alpha' = \sigma(\alpha)$ and $\delta' = \sigma(\delta)$, then α' and δ' satisfy the same conditions as α and δ respectively in the beginning of the proof. With the same reasoning, we find $T \cup \sigma(\alpha) \models T \cup \sigma^2(\alpha)$. Continuing this way, we find the chain

$$T \cup \alpha \models T \cup \sigma(\alpha) \models T \cup \sigma^2(\alpha) \models \cdots \models T \cup \sigma^{n-1}(\alpha) \models T \cup \alpha,$$

with n the order of σ. As a result, we see that $T \cup \alpha$ and $T \cup \sigma(\alpha)$ are logically equivalent. ☐

Note that if $(\alpha, \delta, expl)$ is the state of a solver, and σ is weakly active, the conditions in Proposition 3 are satisfied. Combined with Corollary 1, this proposition implies that for every propagated literal l, $\sigma(l)$ can also be propagated.

Weak activity has two advantages when compared to activity. The first is that weak activity is more general than activity, which allows for more propagations. The second is that keeping track of weakly active symmetries is easier than keeping track of active symmetries, since we only need to check for every choice literal l in δ whether $\sigma(l) \in \alpha$. We describe an efficient incremental approach to keep track of weakly active symmetries in Sect. 3.2.

To conclude this theoretical section, we derive from Proposition 3 a corollary telling us what set of literals is a logical consequence from a theory given a set of weakly active symmetries. It also shows another link between activity and weak activity.

Corollary 2. *Let T be a theory, and $(\alpha, \delta, expl)$ the state of a solver for T. Suppose $\mathcal{S}$ is a set consisting of weakly active symmetries under α and δ. Furthermore, let $G_{\mathcal{S}}$ be the group generated by $\mathcal{S}$, and β the assignment consisting of all elements of α and the orbits of all literals $l \in \alpha$ under $G_{\mathcal{S}}$. Then $T \cup \alpha \models T \cup \beta$ and all symmetries in $G_{\mathcal{S}}$ are active under β.*

Proof. We have proven in Proposition 3 that all $\sigma \in \mathcal{S}$ are symmetries of $T \cup \alpha$. The first statement then follows from the fact that all symmetries of $G_{\mathcal{S}}$ are symmetries of $T \cup \alpha$. The second statement follows from the construction of β. ☐

This property shows that potentially many literals can be propagated by SP(SAT) if enough symmetries are weakly active during the search, and if the group generated by those symmetries is big enough.

3.2 The SP(SAT)-Algorithm

As mentioned before, we characterize the state of a CDCL solver solving a theory T by a current assignment α, a set of choice literals δ, and an explanation clause function *expl*. Symmetry is represented in the solver by a set of *input symmetries* $\mathcal{S}$, given to the solver at the beginning of the search. $\mathcal{S}_{wa}^{\alpha,\delta} \subseteq \mathcal{S}$ denotes the set of weakly active input symmetries under α and δ .

At the start of the search, $\alpha = \emptyset$, so all input symmetries are weakly active and $\mathcal{S}_{wa}^{\alpha,\delta} = \mathcal{S}$. Every time a literal is added to α, all input symmetries containing that literal are notified using a watched literal scheme, so the symmetries can update their weak activity status accordingly. This updating is implemented using a literal counting scheme, increasing the count whenever a literal l with $l \in \delta$ and $\sigma(l) \notin \alpha$ is added to α and decreasing the count whenever a literal $\sigma(l)$ such that $l \in \delta$ is added. A symmetry is weakly active whenever its count is 0. This way, checking whether a symmetry becomes weakly active through the addition of a literal to the assignment is a constant time operation.

For every input symmetry σ, SP(SAT) also keeps track of the *first asymmetric literal* for σ under assignment α. If such a literal exists, the first asymmetric literal is the oldest literal $l \in \alpha$ for which $\sigma(l) \notin \alpha$. Whenever σ is weakly active, according to Proposition 3 and Corollary 1, $\sigma(l)$ can be propagated, with l the first asymmetric literal of σ under assignment α. We will refer to this type of propagation as *symmetry propagation*, as opposed to unit propagation by unit clauses.

During the propagation phase of the solver, unit propagation and symmetry propagation are alternately executed. More precise: when no more unit propagations is possible, the algorithm loops over the set of input symmetries in search of a weakly active symmetry σ which still has a first asymmetric literal l under the current assignment. If such a symmetry exists, the symmetry propagation of $\sigma(l)$ occurs, after which unit propagation is immediately reactivated. This process continues until no more unit or symmetry propagations can be made. Pseudocode for this propagation phase is given in Algorithm 1.

It is easy to see that during propagation, the set of weakly active symmetries $\mathcal{S}_{wa}^{\alpha,\delta}$ can only increase. Also, when $\sigma(l)$ is propagated by symmetry propagation, for each weakly active symmetry $\sigma' \in \mathcal{S}$, $\sigma'(\sigma(l))$ is propagated — either by unit propagation, or by symmetry propagation. As a consequence, all propagations predicted by Corollary 2 will take place.

A CDCL solver requests that for every propagated literal an explanation clause exists. SP(SAT) takes $\sigma(expl(l))$ as explanation for a literal $\sigma(l)$ propagated by symmetry propagation, and adds it to the learned clause store. By Proposition 1 we know the resulting theory $T \cup \sigma(expl(l))$ is logically equivalent to T. Note that $\sigma(expl(l))$ is only generated when $\sigma(l)$ is propagated. At that moment l is the first asymmetric literal of σ under assignment α. This means

Algorithm 1: Propagation phase in a typical SAT solver using SP(SAT)

```
Data: theory $T$, set of symmetries $\mathcal{S}$ of $T$, assignment $\alpha$
Result: modifies $T$ and $\alpha$
repeat
    ...// do unit propagation until fixpoint
    foreach symmetry $\sigma \in \mathcal{S}$ do // symmetry propagation
        if $\sigma$ is weakly active and a first asymmetric literal $l$ for $\sigma$ under $\alpha$ exists
        then
            $expl(\sigma(l)) := \sigma(expl(l))$
            $push\_back(T, expl(\sigma(l)))$
            $push\_back(\alpha, \sigma(l))$
            if $\neg\sigma(l) \in \alpha$ then return; // conflict
            else break // go back to unit propagation
until no more unit or symmetry propagation is possible
```

that for all false literals $l' \in expl(l)$, $\sigma(\neg l') \in \alpha$. Taking into account that all symmetries commute with negation, $\sigma(expl(l))$ is a unit clause under α, which makes $\sigma(expl(l))$ usable as the explanation clause $expl(\sigma(l))$.

Since the performance of CDCL solvers strongly correlates with the number of clauses it has to keep track of for unit propagation, adding many explanation clauses for symmetry propagations might harm performance. To keep the number of explanation clauses for symmetry propagations low, the solver returns to unit propagation — which uses clauses already in the learned clause store as explanation clause — after every single symmetry propagation. Another option would be to simply not use explanation clauses for symmetry propagations in the unit propagation mechanism. Otherwise put, to *construct* the explanation clauses, but not to *learn* them. We implemented this idea as a variant of SP(SAT), and give experimental results in Sect. 4 which show that the explanation clauses for symmetry propagations increase future propagations enough to justify their inclusion in the unit propagation mechanism.

3.3 Optimizations to SP(SAT)

With the main ideas of SP(SAT) being clear, we will discuss two optimizations which will improve performance of SP(SAT) in Sect. 4.

The first optimization is based on the interaction of SP(SAT) with *inverting* symmetries.

Definition 3. *A literal l is* inverting *for a symmetry σ if $\sigma(l) = \neg l$. A symmetry σ is* inverting *if at least one literal is inverting for σ.*

Whenever an inverting symmetry σ is weakly active for assignment α and choices δ, and one of its inverting literals l is propagated, SP(SAT) will propagate $\neg l$, resulting in a conflict and thus, a backjump in the search. However, if an inverting literal l for σ would become a choice literal, σ would become weakly inactive and remain so until the solver backtracks over l. So by choosing choice literals in such a way that they are inverting for as few symmetries as possible, we can keep as many inverting symmetries weakly active as long as possible. In our implementation, we simply ordered the variables by the number of symmetries the corresponding literals were inverting for, and used this as the initial variable ordering by which literals were selected to become choice literals. This simple measure proved very effective for problems with inverting symmetries.

A second optimization concerns symmetry propagation when a symmetry is not weakly active. Given a symmetry σ and a propagated literal l, if $\sigma(expl(l))$ is a unit clause for a given solver state $(\alpha, \delta, expl)$, we can propagate one literal $\sigma(l')$ with l' the non-false literal in $expl(l)$, even if σ is weakly inactive for α and δ. We implemented this idea using a simple loop over the weakly inactive symmetries at the end of the propagation phase, after unit propagation and weakly active symmetry propagation could propagate no more. During this loop, each weakly inactive symmetry σ runs over all explanation clauses $expl(l)$ for all literals $l \in \alpha \setminus \delta$ to detect whether $\sigma(expl(l))$ is a unit clause. If so, the appropriate propagation is made, after which unit propagation is again reactivated. Despite the incurred overhead of this simple implementation, it still gave good results in our experiments.

It is worth noting that this second optimization does not strictly need the notion of weak activity. It rather is a special case of dynamic symmetry breaking by Proposition 1, and a more general case of SP(SAT). However, the notion of weak activity is still useful, since if σ is weakly active, we can skip the check to determine whether $\sigma(expl(l))$ is a unit clause, because $\sigma(expl(l))$ will be unit after the propagation phase. Note also that this optimization requires a solver to incorporate an explanation clause mechanism, which is not required to implement SP(SAT) without this optimization.

4 Experiments

To test the algorithms outlined in the previous section, we have implemented three versions of SP(SAT) in the CDCL solver Minisat [8], as released on GitHub on April 7th 2011 [7]. The first version is the unoptimized version described in Sect. 3.2, which does not store the explanation clauses for symmetry propagations for future unit propagation. We refer to this non-storing version as *Minisat+SP(SAT)*^{ns}. The second version, *Minisat+SP(SAT)*^{s}, does store the explanation clauses for symmetry propagations as learned clauses for future unit propagation. The third version, *Minisat+SP(SAT)*^{opt}, is an optimization version of Minisat+SP(SAT)^{s}, implementing both optimizations described in Sect. 3.3. We refer to our implementations in general by *Minisat+SP(SAT)*. The source code of Minisat+SP(SAT) is available on GitHub [7] as a branch of Niklas Sörensson's Minisat solver.

We compare the execution time of our solvers with the static symmetry breaking tool Shatter [1] as reference, because compared to other both static and dynamic symmetry breaking approaches, Shatter currently is the most effective approach for exploiting symmetry in SAT [15]. Shatter first converts a theory T to a graph, and then uses Saucy [11] to detect its graph automorphism group. Saucy returns a set of generators of the graph automorphism group, which Shatter converts to a set, $\mathcal{S}$, of generators of the symmetry group of T. Next, Shatter constructs symmetry breaking clauses based on the symmetries in $\mathcal{S}$. Running Minisat on the original theory extended with Shatter's symmetry breaking clauses results in the *Minisat+Shatter* approach. To verify how important symmetry breaking is for a given problem, we also run plain *Minisat* on the unmodified problem.

We assume a set of symmetries representing the symmetry group of a problem is available to the solvers. These sets can be generated in different ways: they can for instance be detected by Shatter on a SAT theory, detected on a first-order logic theory before reduction to a SAT theory, or even explicitly given by the user. Since these approaches vary in the time needed to detect the symmetry sets, we will not take this time into account. To make the comparison between Minisat+SP(SAT) and Minisat+Shatter fair, we simply use the set of symmetries generated and used by Shatter as the set of input symmetries for Minisat+SP(SAT).

The benchmark set consists of classic symmetry-heavy problems plus the battleship problem, which is the problem exhibiting most symmetry amongst the problems of last year's SAT competition:

battleship is a symmetric 2011 SAT competition problem,

chnl and **fpga** are problems modeling instances of wire routing in the channels of field-programmable integrated circuits [13],

hole represents the classic pigeonhole problem, with the number in the name referring to the number of holes in which we try to fit one too many a pigeon,

Urq is a family of problems constructed by Urquhart that are hard for resolution, while also exhibiting inverting symmetries [16],

x is a problem with inverting symmetries from the 2002 SAT competition.

All instances are unsatisfiable, since satisfiable instances of those problems are solved very fast, both with and without symmetry breaking, and hence are of little value in a symmetry breaking comparison. The exact .cnf files used for these tests are available at the same location as the source code of Minisat+SP(SAT), as is a more detailed and complete description of the executed tests [7]. The time limit on each problem for each algorithm is 200 seconds. When no answer is given in the desired time frame, a "-" is added to the results. The hardware on which the problems were solved is a Dell Vostro 1510 with an Intel Core 2 Duo CPU T5670 @ 1.80GHz and 2 GiB of memory, using Ubuntu 11.10 32 bit as operating system. The results can be found in Table 1.

Analyzing Table 1, we notice that the original Minisat cannot solve these symmetry heavy problems in the given time frame, while the various symmetry breaking versions can.

Before comparing Minisat+Shatter to Minisat+SP(SAT), we observe that Minisat+SP(SAT)$^{\mathrm{s}}$ is uniformly better than Minisat+SP(SAT)$^{\mathrm{ns}}$: in all benchmark instances where the two methods do not have approximately equal performance, the former requires less time and fewer search choices than the latter.

Table 1. Performance of different symmetry breaking techniques on several symmetric, unsatisfiable SAT-problems. The best results are given in bold.

	Minisat		+Shatter		+SP(SAT)^{ns}		+SP(SAT)^{s}		+SP(SAT)^{opt}	
	Choices	Time (s)	Choices	Time (s)	Choices	Time (s)	Choices	Time (s)	Choices	Time (s)
battleship-05-08-unsat.cnf	9281	0.05	447	**0.00**	1790	0.01	1407	**0.00**	**434**	**0.00**
battleship-06-09-unsat.cnf	45893	0.28	**1105**	**0.00**	9919	0.08	7911	0.06	3657	0.04
battleship-07-12-unsat.cnf	-	-	**141734**	**2.02**	6064858	78.91	2877745	32.07	293105	4.12
battleship-10-10-unsat.cnf	199396	2.22	**5017**	**0.06**	245354	3.85	108109	1.64	26618	0.52
battleship-12-12-unsat.cnf	-	-	119304	2.85	3087992	67.30	2170693	45.50	**13850**	**0.28**
battleship-14-14-unsat.cnf	-	-	-	-	-	-	-	-	-	-
battleship-15-15-unsat.cnf	-	-	-	-	-	-	-	-	-	-
battleship-16-16-unsat.cnf	-	-	-	-	-	-	-	-	**2003119**	**99.78**
chnl10_11.cnf	-	-	1473	**0.00**	**46**	**0.00**	**46**	**0.00**	**46**	**0.00**
chnl10_12.cnf	6050506	163.10	1682	**0.00**	**46**	**0.00**	**46**	**0.00**	**46**	**0.00**
chnl10_13.cnf	6006811	174.74	1766	**0.00**	**46**	**0.00**	**46**	**0.00**	**46**	**0.00**
chnl11_12.cnf	-	-	2017	**0.00**	**56**	**0.00**	**56**	**0.00**	**56**	**0.00**
chnl11_13.cnf	-	-	2264	**0.00**	**56**	**0.00**	**56**	**0.00**	**56**	**0.00**
chnl11_20.cnf	-	-	3954	**0.00**	**56**	**0.00**	**56**	**0.00**	**56**	**0.00**
fpga10_11_uns_rcr.cnf	5985130	135.68	15152	0.28	68346	0.95	25422	0.36	**3266**	**0.06**
fpga10_12_uns_rcr.cnf	-	-	14446	0.32	72995	1.24	12348	0.18	**7043**	**0.17**
fpga10_13_uns_rcr.cnf	-	-	16921	0.41	121881	2.31	18079	0.28	**6009**	**0.14**
fpga10_15_uns_rcr.cnf	-	-	13249	0.32	38715	0.73	10526	**0.19**	**8570**	0.24
fpga10_20_uns_rcr.cnf	-	-	16306	0.53	79946	2.29	12717	0.29	**4654**	**0.18**
fpga11_12_uns_rcr.cnf	-	-	50614	1.33	369691	7.09	49363	0.81	**5669**	**0.11**
fpga11_13_uns_rcr.cnf	-	-	56623	1.60	164962	3.22	60510	1.14	**8419**	**0.20**
fpga11_14_uns_rcr.cnf	-	-	48327	1.38	353336	8.32	34548	0.64	**17147**	**0.48**
fpga11_15_uns_rcr.cnf	-	-	32194	1.01	169283	3.96	26451	**0.54**	**18296**	0.57
fpga11_20_uns_rcr.cnf	-	-	54054	2.27	320862	11.33	57198	1.67	**14556**	**0.54**
hole005.cnf	413	**0.00**	25	**0.00**	**11**	**0.00**	**11**	**0.00**	**11**	**0.00**
hole006.cnf	2357	**0.00**	48	**0.00**	**16**	**0.00**	**16**	**0.00**	**16**	**0.00**
hole007.cnf	13097	0.07	96	**0.00**	**22**	**0.00**	**22**	**0.00**	**22**	**0.00**
hole008.cnf	51046	0.33	124	**0.00**	**29**	**0.00**	**29**	**0.00**	**29**	**0.00**
hole009.cnf	363799	3.38	200	**0.00**	**37**	**0.00**	**37**	**0.00**	**37**	**0.00**
hole010.cnf	-	-	265	**0.00**	**46**	**0.00**	**46**	**0.00**	**46**	**0.00**
hole020.cnf	-	-	3380	**0.00**	**191**	**0.00**	**191**	**0.00**	**191**	**0.00**
hole030.cnf	-	-	12140	0.02	**436**	**0.00**	**436**	0.01	**436**	0.01
hole040.cnf	-	-	21688	0.04	**781**	**0.01**	**781**	0.02	**781**	0.04
hole050.cnf	-	-	76966	0.13	**1226**	**0.04**	**1226**	**0.04**	**1226**	0.07
hole100.cnf	-	-	2180380	2.37	**4951**	0.38	**4951**	**0.36**	**4951**	0.76
hole150.cnf	-	-	2757877	4.42	**11176**	1.57	**11176**	**1.53**	**11176**	4.53
Urq3_5.cnf	-	-	91985	0.24	6572	0.02	6124	0.02	**33**	**0.00**
Urq4_5.cnf	-	-	20323094	83.70	27809	0.10	1200	**0.00**	**43**	**0.00**
Urq5_5.cnf	-	-	-	-	4455744	23.47	2963855	15.06	**72**	**0.00**
Urq6_5.cnf	-	-	-	-	-	-	-	-	**139**	**0.01**
Urq7_5.cnf	-	-	-	-	-	-	-	-	**145**	**0.00**
Urq8_5.cnf	-	-	-	-	-	-	-	-	**205**	**0.01**
x1_40.shuffled.cnf	-	-	1686967	7.06	32840	0.15	29191	0.13	**100**	**0.00**
x1_80.shuffled.cnf	-	-	-	-	15402038	106.90	11256887	74.01	**84**	**0.00**

This means that using the explanation clauses of symmetry propagation for unit propagation can significantly reduce the search tree, and when it does not, it does not pose a significant overhead either. We therefore conclude that explanation clauses should be treated as regular learned clauses.

It is also clear that Minisat+SP(SAT)^{s} outperforms Minisat+Shatter on all problems but **battleship**, also solving two more problems in the given time limit. Minisat+SP(SAT)^{opt} even improves these results, being the best algorithm based on the number of choices, solving most instances within the time limit, and overall being the fastest algorithm.

Further experiments reveal that the first optimization concerning a variable reordering in the case of inverting symmetries made the problems **Urq** and **x**, which exhibit inverting symmetries, almost trivial to solve. The second optimization implementing weakly inactive symmetry propagation is responsible for the reduction of the search tree for Minisat+SP(SAT)^{opt} in the other problems. However, this improved pruning comes at a cost, which is visible in the **fpga** benchmarks, where Minisat+SP(SAT)^{s} sometimes is faster, even though Minisat+SP(SAT)^{opt} needs less choices. Also, on the **hole** problems, even though the number of choices is the same, runtime more than doubles when compared to Minisat+SP(SAT)^{s} and Minisat+SP(SAT)^{ns}. Given the fact that Minisat+SP(SAT)^{opt} still is the fastest solver on many problems, we think that in most cases, the search tree reduction is worth the extra overhead.

We do need to put a caveat next to Minisat+SP(SAT)'s performance on the pigeonhole problem: minor changes to the Minisat algorithm used in the experiments led to a significant decrease in performance on the **hole** problems. All three versions of Minisat+SP(SAT) were no longer able to solve hole020.cnf and greater in the given time limit, while Minisat+Shatter still was with the same changes. The performance for all other problems did not change significantly under these changes. This means that the **hole** problems are unstable for our Minisat+SP(SAT) algorithms, but that it still is possible to solve them quickly using SP(SAT). The instability in the **hole** problem was experienced in three separate algorithmic changes, namely activating clause simplification in Minisat, updating Minisat to the most recent version provided on GitHub, and changing the initial variable ordering of the problem so that the first variables were those occurring in the least symmetries.

5 Related Work

We implemented SP(SAT) in the SAT solver Minisat, but the propositions developed in Sect. 3 are equally applicable to Constraint Satisfaction Problem (CSP) solvers. This is not surprising, since SP(SAT) has its roots in CSP symmetry breaking: we borrowed and improved the notion of activity from Lightweight Dynamic Symmetry Breaking (LDSB) [12], which in turn is a formalization of Shortcut Symmetry Breaking During Search (SBDS) [9].

SBDS is based on constraint generation in CSP-solvers: if after an assignment α, the choice literal l gives rise to no solutions, SBDS imposes the extra constraint $\sigma(\alpha) \Rightarrow \sigma(\neg l)$ for each symmetry σ. This potentially leads to a large number of generated constraints since the number of symmetries can be exponential in the size of the problem. To avoid this constraint buildup, Shortcut SBDS and its formalization LDSB only add the constraint for active symmetries, i.e., only if the constraint simplifies to $\alpha \Rightarrow \sigma(\neg l)$, which makes it immediately applicable to the current assignment α. SP(SAT) improves LDSB by the notion of weak activity, which is a weaker condition than activity, but still sufficient to propagate $\sigma(\neg l)$.

Another variation on SBDS is SBDS adapted for a 1UIP solver (SBDS-1UIP) [5]. For every symmetry σ and every learned clause c, SBDS-1UIP also learns the clause $\sigma(c)$ if $\sigma(c)$ is a unit clause under the current assignment. This is similar to the optimized version of SP(SAT), which will also learn $\sigma(c)$ if it is a unit clause whenever c propagates a literal to the current assignment. The main difference however is that SP(SAT) looks for symmetry propagations and the corresponding unit clauses during the propagation phase, which provides much more opportunities to detect them, while using weak activity of symmetries to minimize overhead.

In the land of SAT, the dynamic symmetry breaking approaches most closely related to SP(SAT) are based on Proposition 1: they learn the symmetrical clauses $\sigma(c)$ for certain clauses c and symmetries σ. However, a mechanism must be devised to prevent learning too many symmetrical clauses since the number of symmetries can be huge. Benhamou e.a. apply each input symmetry only once to a learned clause [4], so each conflict results in at most as many symmetrical learned clauses as the amount of symmetries to be broken. This method is called the Symmetrical Learning Scheme (SLS). A similar method has been used to break symmetry in the context of Boolean optimization [2].

Even though good results have been achieved using SLS, a disadvantage is that not all learned clauses are guaranteed to contribute to the search by propagating a literal. This might result in lots of useless clauses being added to the solver, without breaking a significant amount of symmetry. In contrast, every learned clause added to the solver by SP(SAT) will always have propagated a literal at least once. SLS has another inefficiency: when learning clauses symmetrical to learned clauses, it is possible that an already known clause is added to the set of learned clauses. Since SP(SAT) only propagates literals which cannot be propagated by unit propagation, we are sure an explanation clause for a symmetry propagation was not yet present in the set of learned clauses; the explanation clause would have initiated unit propagation otherwise.

A different symmetry breaking approach for SAT is Sabharwal's structure-aware SAT solver SymChaff [14]. SymChaff detects sets of symmetrically interchangeable variables in a first order logic language, and adjusts the branching mechanism of the underlying SAT solver to break symmetry. This approach was very efficient on amongst others the pigeonhole problem, but the symmetries it can handle are less general than those broken by SP(SAT).

6 Future Work

Besides achieving good performance, SP(SAT) also opens up a number of research opportunities. A first one is to adjust the internal heuristics of a SAT solver to maximize the number of weakly active symmetries during search. For instance, when the solver decides upon the next choice literal, taking into account the amount of symmetries made (in)active by each candidate literal can improve the average number of weakly active symmetries. The variable reordering optimization from Sect. 3.3 is a simple step in this direction, resulting in significant speedups.

Our implementation of the weakly inactive symmetry propagation optimization can also be improved. One could devise a watched literal scheme to keep track of whether certain clauses symmetrical to unit clauses are also unit clauses during search. This approach should incur less overhead than the one described in Sect. 3.3.

The observation that composing two weakly inactive symmetries under an assignment α can result in a weakly active symmetry can be used for another potential improvement. The implementation of SP(SAT) tested here does not compose symmetries, which might result in missing key symmetry propagations. Consider for example the symmetry $\sigma = (abcd)$. If the state of a solver has assignment $\alpha = \{a, c, d\}$ and choices $\delta = \{a\}$, then σ is weakly inactive. However, σ^2 is weakly active; using this knowledge would result in the propagation of b.

A related possibility for future study lies in the effect of different sets of input symmetries representing the same symmetry group when breaking symmetry in SAT. In CSP, it has been shown that certain sets of generators are better than others for static symmetry breaking [10], which is true for SAT and dynamic symmetry breaking as well. Consider for example the symmetry $\sigma = (ab)(cde)$. The group generated by σ is the same as the group generated by (ab) and (cde). However, the latter symmetries would result in better symmetry propagation.

Since SP(SAT) at its core is not a clause generator, but a literal propagator in CSP style, it should be straightforward to construct a symmetry breaking approach SP(CSP) based on the ideas presented in this paper. The explanation clause generation mechanism for SP(SAT) also is applicable for lazy clause generation CSP solvers. Evaluating the performance of such a SP(CSP) implementation might yield interesting results.

A last area of improvement is based on the fact that SP(SAT) does not prune all parts of the search tree symmetrical under the input symmetries. This became clear when we adjusted our implementations of Minisat+SP(SAT) to return all models of a given theory: it occurred that Minisat+SP(SAT) returned two models M_1 and M_2 such that $\sigma(M_1) = M_2$, for some input symmetry σ. Devising an improvement to SP(SAT) which fully prunes the parts of the search tree symmetrical under the set of input symmetries is a challenge yet to be tackled.

7 Conclusion

In this paper, we presented a novel approach to dynamically break symmetries in SAT. Thanks to the notion of weak activity, it possesses promising theoretical properties. An implementation in Minisat was able to outperform the current state-of-the-art Shatter in our experiments. Compared to other dynamic symmetry breaking techniques for SAT, we believe that SP(SAT) maintains a good balance between not learning too many clauses and assuring strong propagation.

Furthermore, SP(SAT) is not limited to SAT, but applicable to other problem solving techniques as well.

References

1. Aloul, F.A., Markov, I.L., Sakallah, K.A.: Shatter: efficient symmetry-breaking for boolean satisfiability. In: DAC'03. pp. 836–839 (2003)
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. Benhamou, B., Nabhani, T., Ostrowski, R., Saïdi, M.R.: Détection et élimination dynamique de la symétrie dans le problème de satisfiabilité. In: Actes des $6^{\text{èmes}}$ Journées Francophones de Programmation par Contraintes. pp. 81–90. JFPC'2010, Caen, France (Juin 9-11, 2010)
4. Benhamou, B., Nabhani, T., Ostrowski, R., Sadi, M.R.: Enhancing clause learning by symmetry in SAT solvers. In: ICTAI (1). pp. 329–335. IEEE Computer Society (2010)
5. Chu, G., Stuckey, P.J., Garcia de la Banda, M., Mears, C.: Symmetries and lazy clause generation. In: International Joint Conference on Artificial Intelligence (IJCAI). pp. 516–521 (2011)
6. Davis, M., Logemann, G., Loveland, D.: A machine program for theorem-proving. Commun. ACM 5, 394–397 (July 1962)
7. Devriendt, J.: An implementation of Symmetry Propagation (in SAT) in Minisat on Github. www.github.com/JoD/minisat-SPFS
8. Eén, N., Sörensson, N.: An Extensible SAT-solver. In: Giunchiglia, E., Tacchella, A. (eds.) Theory and Applications of Satisfiability Testing, Lecture Notes in Computer Science, vol. 2919, chap. 37, pp. 333–336. Springer Berlin / Heidelberg, Berlin, Heidelberg (2004)
9. Gent, I.P., Smith, B.M.: Symmetry breaking in constraint programming. In: Proceedings of ECAI-2000. pp. 599–603. IOS Press (2000)
10. Jefferson, C., Petrie, K.: Automatic generation of constraints for partial symmetry breaking. In: Lee, J. (ed.) Principles and Practice of Constraint Programming CP 2011, Lecture Notes in Computer Science, vol. 6876, pp. 729–743. Springer Berlin / Heidelberg (2011)
11. Mark, P.D., Liffiton, M.H., Sakallah, K.A., Markov, I.L.: Exploiting Structure in Symmetry Detection for CNF. In: In Proceedings of the 41st Design Automation Conference. pp. 530–534 (2004)
12. Mears, C., Garcia de la Banda, M., Demoen, B., Wallace, M.: Lightweight dynamic symmetry breaking. In: Eighth International Workshop on Symmetry in Constraint Satisfaction Problems (SymCon'08) (2008)
13. Nam, G.J., Aloul, F., Sakallah, K.A., Rutenbar, R.A.: A comparative study of two boolean formulations of fpga detailed routing constraints. IEEE Trans. Comput. 53, 688–696 (June 2004)
14. Sabharwal, A.: SymChaff: exploiting symmetry in a structure-aware satisfiability solver. Constraints 14(4), 478–505 (Dec 2009)
15. Sakallah, K.A.: Symmetry and satisfiability. In: Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.) Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications, vol. 185, pp. 289–338. IOS Press (2009)
16. Urquhart, A.: Hard examples for resolution. J. ACM 34, 209–219 (January 1987)
17. Zhang, L., Madigan, C.F.: Efficient conflict driven learning in a boolean satisfiability solver. In: In ICCAD. pp. 279–285 (2001)