Symmetry Propagation: Improved Dynamic Symmetry Breaking in SAT

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 For constraint programming, many well performing dynamic symmetry breaking techniques have been devised. For propositional satisfiability solving, dynamic symmetry breaking is still either slower or less general than static symmetry breaking. This paper presents Symmetry Propagation, which is an improvement to Lightweight Dynamic Symmetry Breaking, a dynamic symmetry breaking approach from CP. Symmetry Propagation uses any given symmetry as a propagator, and as a result is a general symmetry breaking technique. Experiments with an implementation in the SAT solver Minisat show that on many benchmarks, Symmetry Propagation outperforms the state-of-the-art static symmetry breaking method Shatter.

1 Introduction

Many difficult propositional satisfiability (SAT) and constraint satisfaction problems exhibit symmetry properties. Exploiting these symmetry properties can significantly reduce the time needed to solve such 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 solving method, but only the problem at hand, so 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, especially in SAT solving, where the static symmetry breaking approach Shatter [2] currently is the most effective way to deal with symmetry [18].

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 [16]. Another disadvantage is that the extra constraints can be too large for a solver to handle. Finally, static symmetry breaking is not always possible, for instance when symmetries are detected during search [4], or when a problem is symmetrical, but an optimization function over the solutions is not [3].

Dynamic symmetry breaking does not have these problems because it breaks symmetry by interacting with the solver during search, ensuring search paths symmetrical to an already inspected path are avoided. Both in constraint programming (CP) and SAT, many dynamic symmetry breaking methods have been proposed. Unlike in CP, dynamic symmetry breaking methods for SAT are either less performant than static symmetry breaking, or are limited to a specific class of symmetries [17,19].

In this paper, we present *Symmetry Propagation*, a dynamic symmetry breaking approach that speeds up search by propagating literals symmetrical to already propagated literals, for any given set of symmetries. To test its performance, we compared an implementation of SP in Minisat [9] to Shatter [2].

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, 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 in a SAT solver. Implementing SP in Minisat yielded the experimental results given in Sect. 4, while we relate SP to other symmetry breaking algorithms available in literature in Sect. 5. Further research opportunities are discussed in Sect. 6, to conclude the paper in Sect. 7.

2 Background

2.1 The SAT Problem

We define a *propositional 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 of 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 as many literals as there are variables in $\Sigma(T)$; if not complete, 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 literal 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, and a literal l is a logical consequence of T if the clause containing only l is. 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.

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

2.2 A Conflict Driven Clause Learning Solver

We briefly recall some of the concepts of a Conflict Driven Clause Learning (CDCL) SAT solver [21]. At each search step, a SAT solver chooses a value for an unassigned variable of the given theory, and adds the corresponding literal to the current assignment. This literal is called a *choice literal*, and may result in some clauses becoming unit clauses under the new assignment. This prompts a *unit propagation* phase, where all undefined literals occurring in a unit clause are added to the current assignment. These literals are *propagated literals*, with the unit clause they belong to referred to as their *explanation clause*. If no more unit clauses remain under the resulting assignment, the unit propagation phase ends, and a new search step starts by choosing a new choice literal.

If at some point a literal l would be propagated with $\neg l$ present in the current assignment, a *conflict* arises. At this moment, the CDCL solver will construct a *learned clause* by investigating the explanation clauses for the unit propagations leading to the conflict. This learned clause is a logical consequence of the theory. Adding it to the theory prevents the conflict from occurring again after a backjump. We refer to the collection of learned clauses of a CDCL SAT solver as the *learned clause store*.

Formally, we characterize the state of a CDCL solver solving a theory T by a triple $(\alpha, \delta, expl)$, where α is the current assignment, $\delta \subseteq \alpha$ the set of choice literals such that $T \cup \delta \models T \cup \alpha$, $\alpha \setminus \delta$ is the set of propagated literals, and $expl$ is a function from $\alpha \setminus \delta$ to T such that, for each propagated literal $l \in \alpha \setminus \delta$, $expl(l)$ is the explanation clause for l.

3 Symmetry Propagation

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 assume that this

symmetry also commutes with negation (i.e., it sends $\neg a$ to $\neg b$, d to $\neg e$,....) and sends all remaining literals to themselves.

Symmetries can be composed, so a set of symmetries $\mathcal{S}$ of a theory T generates a subgroup $G_{\mathcal{S}}$ of the group of all permutations of $\bar{\Sigma}(T)$. We call $G_{\mathcal{S}}$ a *symmetry group* of T. In general, the size of a symmetry 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. For a symmetry group $G_{\mathcal{S}}$, we define the *orbit* of a literal l as $\{\sigma(l)|\sigma \in G_{\mathcal{S}}\}$. Similarly, the orbit of a clause c is $\{\sigma(c)|\sigma \in G_{\mathcal{S}}\}$. The *order* of a symmetry σ is the smallest positive n such that σ^n is the identity.

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 Proposition 1 and add $\sigma(c)$ as a learned clause for every symmetry σ of some symmetry group of T. 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 orbit of this clause under some symmetry group will prevent the encounter of all symmetrical conflicts, resulting in complete symmetry breaking. However, since there are possibly exponentially many symmetries, this approach will in most cases add too many clauses to the theory to be of practical use. Several symmetry breaking methods based on Proposition 1 avoid this exponential number of clauses by limiting the number of symmetries applied to the learned clause. For example, the Symmetrical Learning Scheme presented by Benhamou e.a. only apply the elements of the generator $\mathcal{S}$ rather than the elements of the group $G_{\mathcal{S}}$ to the learned clause [5].

Symmetry Propagation (SP) on the other hand breaks symmetry by propagating symmetrical literals instead of adding symmetrical clauses. The following corollary of Proposition 1 is the foundation for the SP algorithm defined below.

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

Corollary 1 means that if a SAT solver has state $(\alpha, \delta, expl)$ where l can be propagated, then for every symmetry σ of $T \cup \alpha$, $\sigma(l)$ can also be propagated. To detect whether symmetries of T are symmetries of $T \cup \alpha$, Mears e.a. introduced the notion of *activity* for their Lightweight Dynamic Symmetry Breaking algorithm [14].

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

Which leads to the following proposition:

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, so for other propagated literals l', $\sigma(l')$ can again be propagated, which results in dynamic symmetry breaking. Many dynamic symmetry breaking methods in CP use this property [10,14].

We improve this approach, introducing 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 now 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 if enough symmetries are weakly active during the search, and if the group generated by those symmetries is big enough.

3.2 The SP-Algorithm

As mentioned before, we characterize the state of a CDCL solver 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, $\alpha = \emptyset$, $\mathcal{S}_{wa}^{\alpha,\delta} = \mathcal{S}$. 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 by keeping a counter per symmetry σ: the counter is increased whenever a literal l with $l \in \delta$ and $\sigma(l) \notin \alpha$ is added to α, and decreased whenever a literal l is added to α such that $\sigma^{-1}(l) \in \delta$. A symmetry is weakly active whenever its counter 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 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 executed alternately. More precisely: when no more unit propagations are 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, 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.

Example 1 presents a typical run of the SP algorithm:

Example 1. Consider theory $T = \{\neg f \vee a, \neg f \vee b, \neg a \vee d, \neg b \vee e \vee c, \neg c \vee \neg g, \neg c \vee g\}$ and its symmetry $\sigma = (ab)(de)$. Suppose the SAT algorithm chooses a, so $\alpha = \delta = \{a\}$. During the unit propagation phase, we can propagate d, so $\alpha = \{a, d\}$ and $\delta = \{a\}$. Since no more unit propagation is possible, a check for symmetry propagation is made. However, since $\sigma(\delta) = \{b\} \not\subseteq \alpha$, σ is not weakly active and no symmetry propagation occurs. Note that at this time, a is the first asymmetric literal for σ, since a is the first literal added to α and $\sigma(a) \notin \alpha$.

Since no more unit or symmetry propagation is possible, the algorithm chooses, say f, and propagates b during unit propagation. After this, $\alpha = \{a, d, f, b\}$, $\delta = \{a, f\}$, and symmetry propagation starts. σ is inactive since $\sigma(d) = e \notin \alpha$, but σ is weakly active since $\sigma(\delta) = \{b, f\} \subseteq \alpha$. Also, the first asymmetric literal for σ now is d, since $\sigma(a) \in \alpha$ and $\sigma(d) \notin \alpha$. As a result, $\sigma(d) = e$ can be propagated during symmetry propagation, so that $\alpha = \{a, d, f, b, e\}$. Now, unit propagation is immediately reactivated. Since symmetry propagation did not create new unit clauses, no further unit propagation happens, and symmetry propagation is continued. Even though σ now is (weakly) active, it has no first asymmetric literal, so it can no longer propagate, ending the unit and symmetry propagation. The search now continues by choosing a new choice literal, and so on.

During unit and symmetry propagation, the assignment α only grows, while the set of decision literals does not change. Hence, the set of weakly active symmetries $\mathcal{S}_{wa}^{\alpha,\delta}$ can only increase in this phase. 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 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 the moment $\sigma(l)$ is propagated, so when l is the first asymmetric literal of σ under assignment α. This means 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 $expl(\sigma(l))$ for the propagated literal $\sigma(l)$.

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.

3.3 Optimizations to SP

With the main ideas of SP being clear, we will discuss two optimizations to SP. In Sect. 4, we show that they really improve performance. The first optimization is based on the interaction of SP with *inverting* symmetries.

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

```
Data: theory T, set of symmetries S of T, assignment α
Result: modifies T and α
repeat
    ...// do unit propagation until fixpoint;
    foreach symmetry σ ∈ S do // symmetry propagation
        if σ is weakly active and a first asymmetric literal l for σ under α exists
        then
            expl(σ(l)) := σ(expl(l));
            push_back(T, expl(σ(l)));
            push_back(α, σ(l));
            if ¬σ(l) ∈ α then return; // conflict
            else break // go back to unit propagation;
until no more unit or symmetry propagation is possible;
```

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 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. 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 *inverting symmetry optimization* proved very effective for most 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 *inactive propagation optimization*, it still gave good results in our experiments.

It is worth noting that the inactive propagation optimization does not use 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. 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. Note also that this optimization requires a solver to incorporate an explanation clause mechanism, which is not required by plain SP.

4 Experiments

To test the algorithms outlined in the previous section, we implemented SP in the CDCL solver Minisat [9] released on GitHub on March 27th 2011 [8]. The implementation follows the algorithm described in Sect. 3.2, with the option to use the inverting symmetry optimization or inactive propagation optimization described in Sect. 3.3. We tested the algorithm with different combinations of these options, of which we will present two here. The first version has both optimizations deactivated. We refer to this regular version by *Minisat+SP*^{reg}. The second version has both optimizations activated. We refer to this optimized version by *Minisat+SP*^{opt}. We refer to both versions by *Minisat+SP*. The source code of Minisat+SP is available on GitHub as a branch of Niklas Sörensson's Minisat solver [8].

We compare the performance of our solvers with Shatter [2] as reference. Shatter is a static symmetry breaking tool which in a preprocessing step adds symmetry breaking clauses to a theory, on which we can then run Minisat. We will refer to this as the Minisat+Shatter approach. We use Shatter as reference because it is easy to use, freely available, and most importantly, it currently is the most effective approach for exploiting symmetry in SAT [18].

As benchmarks, we used SAT theories modeled in the standard DIMACS .cnf format. Since this format does not contain information about symmetries, we detect them by use of Shatter's builtin symmetry detection algorithm. For this, Shatter converts a theory T to a graph, and then uses Saucy 2.0 [7] to detect the graph's automorphism group. The generators of the graph automorphism group are then converted to $\mathcal{S}$, a set of generators of the detected symmetry group of T. We used these generators as input symmetries for all symmetry breaking algorithms. An improved version of Saucy has been developed [12], but is not freely available.

We also include measurements with plain Minisat in our statistics, to give an idea how important symmetry breaking is in each selected benchmark.

We constructed two benchmark sets to test the algorithms. The first benchmark set contains 96 problems from the SAT 2011 competition for which we could detect within a time limit of 1000 seconds that some symmetry was present. These results are summarized in Fig. 1.

The second benchmark set consists of classical SAT symmetry breaking problems: problems of wire routing in the channels of field-programmable integrated circuits (**chnl** and **fpga**) [15], pigeonhole problems (**hole**) and Urquhart's problems (**Urq**) [20]. We use a shuffled version of the pigeonhole problem, because we experienced that using the initial variable ordering resulted in very fast solving times for the symmetry breaking algorithms. Upon closer inspection, this behaviour was caused by the encoding of the literals: for each literal n representing

whether a pigeon p resides in hole h, if h was less than the number of holes, then literal $n+1$ represented that p resides in $h+1$. Minisat used this ordering as its decision literal heuristic, which resulted in a polynomial solving time for Minisat+SP. Minisat+Shatter also experienced this polynomial behaviour, because Shatter uses this ordering of the literals to construct its symmetry breaking clauses.

Problem information for the second benchmark set is given in Table 1 and detailed experimental results in Table 2. Two problem families from the SAT 2011 benchmark set (**x** and **battleship**) gave particularly interesting results, so we included their information in Table 1 and 2. For both benchmark sets, full experimental results are available online, as well as .cnf files for all problems in Table 1 [8]. For all problems of the SAT 2011 competition, .cnf files are available on the corresponding site [1].

The total solve time given to each algorithm on each problem was 5000 seconds, with symmetry detection time included for all algorithms except Minisat. When no answer was given in the desired time frame, a "-" is shown in Table 1 and 2. The problems were solved using an Intel Core i7-2600 @ 3.4GHz processor and 16 GiB of memory, with Ubuntu 10.04 64 bit as operating system.

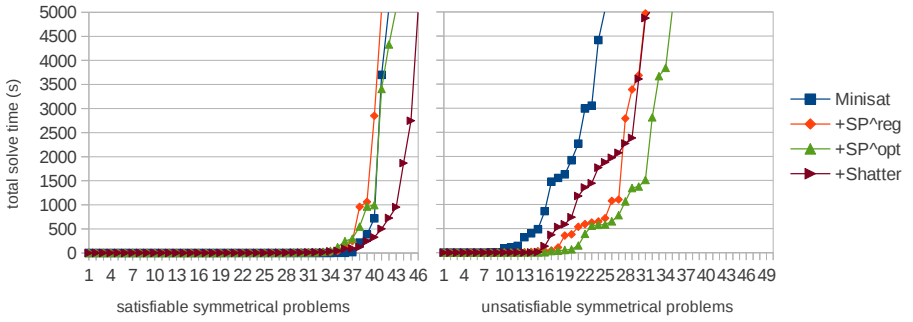

Figure 1. The performance of Minisat, Minisat+SP$^{\text{reg}}$, Minisat+SP$^{\text{opt}}$ and Minisat+Shatter on symmetry exhibiting problems of the SAT 2011 competition. For each algorithm, the problems are ordered on solve time.

The running times of the algorithms on the SAT 2011 competition problems in Fig. 1 show some interesting patterns. Firstly, on unsatisfiable symmetrical problems, adding symmetry information and exploiting it significantly improves performance of Minisat. We also learn that for unsatisfiable symmetrical problems, Minisat+Shatter and Minisat+SP$^{\text{reg}}$ solve about the same number of problems, with Minisat+SP$^{\text{reg}}$ being a bit faster on average. The best performing algorithm on the unsatisfiable instances clearly is Minisat+SP$^{\text{opt}}$.

The left part of Fig. 1 sketches another picture. Firstly, all algorithms are able to solve almost all satisfiable symmetric problems. This can be explained by the fact that when a problem is both satisfiable and symmetric, it often contains many (symmetric) solutions, which as a result are less hard to find. Secondly, Minisat+SP$^{\text{reg}}$ and Minisat+SP$^{\text{opt}}$ both perform similarly to Minisat,

Table 1. Detailed information of the problem families **fpga**, **chnl**, **hole**, **Urq**, **x**, **battleship**, and the time needed by Minisat to solve them.

Problem name	Status	Symmetry detection time (s)	# Symmetries Total	Inverting	Minisat Solve time (s)	Decisions	Unit propagations
fpga10_8_sat.cnf	SAT	0.0	22	0	0.0	405	3056
fpga10_9_sat.cnf	SAT	0.0	23	0	0.0	408	3364
fpga12_11_sat.cnf	SAT	0.0	29	0	0.0	467	3659
fpga12_8_sat.cnf	SAT	0.0	24	0	0.0	296	2264
fpga12_9_sat.cnf	SAT	0.0	25	0	0.0	451	3573
fpga13_10_sat.cnf	SAT	0.0	28	0	0.0	278	2223
fpga13_12_sat.cnf	SAT	0.0	32	0	0.0	316	2840
fpga13_9_sat.cnf	SAT	0.0	26	0	0.0	330	2569
fpga10_11_uns_rcr.cnf	UNSAT	0.0	38	0	71.5	5985130	67227297
fpga10_12_uns_rcr.cnf	UNSAT	0.0	41	0	209.8	12881772	150276769
fpga10_13_uns_rcr.cnf	UNSAT	0.0	42	0	955.2	42096627	519910130
fpga10_15_uns_rcr.cnf	UNSAT	0.1	46	0	1841.0	60680481	861526582
fpga10_20_uns_rcr.cnf	UNSAT	0.1	57	0	1271.8	29871337	578921696
fpga11_13_uns_rcr.cnf	UNSAT	0.1	44	0	-	-	-
fpga11_14_uns_rcr.cnf	UNSAT	0.1	46	0	-	-	-
fpga11_15_uns_rcr.cnf	UNSAT	0.1	49	0	-	-	-
fpga11_20_uns_rcr.cnf	UNSAT	0.2	59	0	-	-	-
chnl10_11.cnf	UNSAT	0.0	39	0	167.2	12437023	139066233
chnl10_12.cnf	UNSAT	0.0	41	0	85.0	6050506	75992394
chnl10_13.cnf	UNSAT	0.0	43	0	94.8	6006811	82625001
chnl11_12.cnf	UNSAT	0.0	43	0	3353.1	127407652	1441292940
chnl11_13.cnf	UNSAT	0.0	45	0	3868.6	126818865	1578025794
chnl11_20.cnf	UNSAT	0.1	59	0	3405.1	58302347	1173568158
hole010_shuffled.cnf	UNSAT	0.0	19	0	114.3	12675295	141859025
hole011_shuffled.cnf	UNSAT	0.0	21	0	3320.3	193488347	2237873772
hole012_shuffled.cnf	UNSAT	0.0	23	0	-	-	-
hole013_shuffled.cnf	UNSAT	0.0	25	0	-	-	-
hole014_shuffled.cnf	UNSAT	0.0	27	0	-	-	-
hole015_shuffled.cnf	UNSAT	0.0	29	0	-	-	-
hole016_shuffled.cnf	UNSAT	0.0	31	0	-	-	-
hole017_shuffled.cnf	UNSAT	0.0	33	0	-	-	-
hole018_shuffled.cnf	UNSAT	0.1	35	0	-	-	-
Urq3_5.cnf	UNSAT	0.0	29	29	139.7	73481538	301105603
Urq4_5.cnf	UNSAT	0.0	43	43	-	-	-
Urq5_5.cnf	UNSAT	0.2	72	72	-	-	-
Urq6_5.cnf	UNSAT	0.6	109	109	-	-	-
Urq7_5.cnf	UNSAT	1.3	143	143	-	-	-
Urq8_5.cnf	UNSAT	3.3	200	200	-	-	-
x1_40.shuffled.cnf	UNSAT	0.0	41	40	141.4	72930235	583733727
x1_80.shuffled.cnf	UNSAT	0.1	80	80	-	-	-
battleship-07-13-sat.cnf	SAT	0.4	12	0	0.0	414	3750
battleship-08-15-sat.cnf	SAT	0.0	14	0	0.0	277	1799
battleship-09-17-sat.cnf	SAT	0.1	15	0	0.0	1395	9720
battleship-10-17-sat.cnf	SAT	0.0	12	0	3.9	368958	4436583
battleship-10-18-sat.cnf	SAT	0.0	13	0	0.0	622	3491
battleship-10-19-sat.cnf	SAT	0.1	15	0	0.0	648	3914
battleship-12-23-sat.cnf	SAT	0.1	18	0	0.0	1252	7740
battleship-14-26-sat.cnf	SAT	0.1	17	0	718.2	29589334	380912396
battleship-15-29-sat.cnf	SAT	0.3	22	0	386.0	21961391	244436516
battleship-24-57-sat.cnf	SAT	1.8	41	0	16.5	1706712	10179138
battleship-05-08-unsat.cnf	UNSAT	0.0	8	0	0.0	9281	93196
battleship-06-09-unsat.cnf	UNSAT	0.0	7	0	0.1	45893	524592
battleship-07-12-unsat.cnf	UNSAT	0.0	10	0	485.1	61922751	694386783
battleship-10-10-unsat.cnf	UNSAT	0.0	8	0	1.6	199396	3463569
battleship-12-12-unsat.cnf	UNSAT	0.0	10	0	402.3	29000352	560429557
battleship-14-14-unsat.cnf	UNSAT	0.5	8	0	-	-	-
battleship-15-15-unsat.cnf	UNSAT	0.1	10	0	-	-	-
battleship-16-16-unsat.cnf	UNSAT	1.6	17	0	-	-	-

Table 2. Performance of Minisat+SP^{reg}, Minisat+SP^{opt} and Minisat+Shatter on the problem families **fpga**, **chnl**, **hole**, **Urq**, **x**, **battleship**. The best results are given in bold. The last two columns show the ratio of symmetry propagations to total propagations for Minisat+SP^{reg} and Minisat+SP^{opt}.

	Solve Time (s)			Decisions			Total propagations			Sym. prop.	
Problem name	+SP^{reg}	+SP^{opt}	+Shatter	+SP^{reg}	+SP^{opt}	+Shatter	+SP^{reg}	+SP^{opt}	+Shatter	+SP^{reg}	+SP^{opt}
fpga10_8_sat	**0.0**	**0.0**	**0.0**	352	**320**	503	2781	**2413**	3337	6.7%	**12.0%**
fpga10_9_sat	**0.0**	**0.0**	**0.0**	348	**316**	607	**2562**	2670	2872	6.2%	**9.4%**
fpga12_11_sat	**0.0**	**0.0**	**0.0**	**363**	524	474	**3001**	4441	3097	4.8%	**8.1%**
fpga12_8_sat	**0.0**	**0.0**	**0.0**	**324**	472	602	**2100**	3656	2838	7.0%	**8.8%**
fpga12_9_sat	**0.0**	**0.0**	**0.0**	**378**	492	713	**2388**	4159	6612	5.9%	**7.8%**
fpga13_10_sat	**0.0**	**0.0**	**0.0**	477	**396**	591	3452	3703	**2822**	5.0%	**6.1%**
fpga13_12_sat	**0.0**	**0.0**	**0.0**	**334**	365	1112	**2794**	4039	5243	2.5%	**6.1%**
fpga13_9_sat	**0.0**	**0.0**	**0.0**	858	696	**601**	7201	7705	**6654**	2.4%	**4.9%**
fpga10_11_uns_rcr	0.1	**0.1**	0.2	17093	**6888**	15152	184199	**84467**	406313	2.1%	**4.3%**
fpga10_12_uns_rcr	0.1	**0.1**	0.2	12154	**5530**	14446	141348	**77183**	478956	2.5%	**3.5%**
fpga10_13_uns_rcr	0.2	**0.1**	0.2	25671	**4938**	16921	353896	**72246**	668044	1.6%	**3.6%**
fpga10_15_uns_rcr	0.2	**0.2**	0.2	19039	**9183**	13249	269767	**139644**	509944	2.3%	**3.2%**
fpga10_20_uns_rcr	0.4	**0.2**	0.4	21404	**4498**	16306	438893	**94108**	842632	1.8%	**4.6%**
fpga11_13_uns_rcr	**0.3**	0.3	0.9	25790	**19236**	56623	295693	**268453**	2074116	2.8%	**4.2%**
fpga11_14_uns_rcr	0.6	**0.2**	0.8	51389	**10843**	48327	664593	**151860**	1705635	1.7%	**3.3%**
fpga11_15_uns_rcr	0.3	**0.2**	0.6	29983	**9836**	32194	424998	**151441**	1364837	1.9%	**3.4%**
fpga11_20_uns_rcr	0.8	**0.3**	1.3	45832	**9746**	54054	853502	**190844**	2930058	1.8%	**3.9%**
chnl10_11	**0.0**	**0.0**	**0.0**	**46**	**46**	1473	**239**	**239**	3620	**28.5%**	**28.5%**
chnl10_12	**0.0**	**0.0**	**0.0**	**46**	**46**	1682	**258**	**258**	3952	**25.9%**	**25.9%**
chnl10_13	**0.0**	**0.0**	**0.0**	**46**	**46**	1766	**277**	**277**	3998	**23.7%**	**23.7%**
chnl11_12	**0.0**	**0.0**	**0.0**	**56**	**56**	2017	**285**	**285**	7678	**29.0%**	**29.0%**
chnl11_13	**0.0**	**0.0**	**0.0**	**56**	**56**	2264	**306**	**306**	8206	**26.4%**	**26.4%**
chnl11_20	**0.1**	**0.1**	**0.1**	**56**	**56**	3954	**453**	**453**	8932	**16.5%**	**16.5%**
hole010_shuffled	0.3	**0.1**	0.1	33740	**10949**	17342	411875	**135380**	583292	1.7%	**4.2%**
hole011_shuffled	0.5	**0.3**	1.4	55047	**18662**	142711	587255	**357590**	5346166	1.8%	**3.3%**
hole012_shuffled	4.1	**0.3**	10.5	313497	**30992**	729083	3725964	**427055**	29411280	2.0%	**3.3%**
hole013_shuffled	31.0	**0.8**	105.2	1532124	**67184**	4531023	17615135	**847681**	207007682	1.8%	**3.3%**
hole014_shuffled	311.2	**13.3**	2821.2	9836194	**765810**	67375809	117694475	**9708228**	3065586388	1.6%	**3.5%**
hole015_shuffled	715.7	**201.5**	-	17048418	**7299563**	-	204519243	**99026025**	-	1.4%	**2.7%**
hole016_shuffled	-	**122.7**	-	-	**3884224**	-	-	**50052364**	-	-	**2.6%**
hole017_shuffled	-	**2863.2**	-	-	**54961134**	-	-	**754881930**	-	-	**2.1%**
hole018_shuffled	-	**994.5**	-	-	**18031344**	-	-	**243237829**	-	-	**2.1%**
Urq3_5	**0.0**	**0.0**	0.1	6124	**33**	91985	18479	**104**	453961	7.3%	**35.1%**
Urq4_5	**0.0**	**0.0**	39.0	1200	**43**	20323094	5182	**146**	115646665	4.4%	**40.4%**
Urq5_5	7.0	**0.2**	3810.3	2963855	**72**	1428323031	11279563	**240**	8379761695	2.4%	**42.0%**
Urq6_5	-	**0.6**	-	-	**139**	-	-	**499**	-	-	**27.3%**
Urq7_5	-	**1.3**	-	-	**145**	-	-	**517**	-	-	**37.5%**
Urq8_5	-	**3.3**	-	-	**205**	-	-	**694**	-	-	**39.9%**
x1_40.shuffled	0.0	**0.0**	3.1	29191	**100**	1686967	233505	**495**	18817376	0.9%	**7.6%**
x1_80.shuffled	29.8	**0.1**	1972.4	11256887	**84**	680826737	86139133	**525**	7609158516	0.6%	**17.4%**
battleship-07-13-sat	**0.4**	**0.4**	**0.4**	**237**	517	606	**1566**	4212	4671	0.4%	**1.0%**
battleship-08-15-sat	**0.0**	**0.0**	**0.0**	**268**	429	1078	**1692**	3329	7947	1.3%	**2.1%**
battleship-09-17-sat	**0.1**	0.1	0.1	**1773**	5148	4203	**14365**	44127	111235	1.1%	**1.6%**
battleship-10-17-sat	**1.4**	2.2	5.4	**142969**	143459	211317	**1632116**	1680128	7764867	1.1%	**1.2%**
battleship-10-18-sat	**0.0**	0.1	0.1	**659**	6360	6348	**5579**	53159	171868	0.6%	**1.4%**
battleship-10-19-sat	**0.1**	**0.1**	**0.1**	759	**462**	3293	4418	**2588**	57542	0.3%	**0.3%**
battleship-12-23-sat	0.1	**0.1**	0.1	3049	**1034**	4429	21642	**6437**	70647	**0.8%**	0.8%
battleship-14-26-sat	1060.2	546.1	**14.3**	37732810	17825596	**735680**	498425156	230079786	**31417543**	1.0%	**1.1%**
battleship-15-29-sat	**16.5**	296.6	88.1	**744001**	10058234	3373200	**8086762**	115729801	176748246	1.1%	**1.1%**
battleship-24-57-sat	**2.8**	21.9	34.3	**223013**	779295	3102966	**1001215**	3939231	81779875	1.0%	**1.2%**
battleship-05-08-uns	**0.0**	**0.0**	**0.0**	1406	463	**447**	13481	**4424**	5752	1.9%	**2.1%**
battleship-06-09-uns	0.0	**0.0**	**0.0**	7947	2358	**1105**	86815	24824	**15784**	1.0%	**1.6%**
battleship-07-12-uns	17.3	2.0	**1.4**	3040267	295311	**141734**	32192441	**3152124**	3507254	1.4%	**1.5%**
battleship-10-10-uns	1.1	0.2	**0.0**	107931	13673	**5017**	1950566	257212	**115266**	0.0%	**0.1%**
battleship-12-12-uns	45.6	**0.7**	1.3	2828621	**45112**	119304	55311246	**901134**	4549889	0.1%	**0.3%**
battleship-14-14-uns	-	1372.2	**736.6**	-	61447105	**18278571**	-	1491683531	**855847664**	-	**0.0%**
battleship-15-15-uns	-	**149.0**	-	-	**7252565**	-	-	**155851829**	-	-	**0.0%**
battleship-16-16-uns	-	**32.9**	-	-	**1476236**	-	-	**27953465**	-	-	**0.2%**

while Minisat+Shatter on the other hand is able to deliver improved performance. An explanation can be found in the difference between SP and Shatter: the symmetry breaking constraints generated by Shatter will always exclude some part of the search space, while SP can not guarantee that symmetry propagation (and corresponding search space reduction) will always happen. When symmetries are inactive or have no first asymmetric literal, they will not propagate. In this sense, Shatter is a more complete symmetry breaking tool, still breaking significant symmetry when solving relatively easy satisfiable symmetrical problems.

The results presented in Table 2 concerning the classical SAT symmetry breaking problems confirm the above observations: the satisfiable instances of **fpga** are easily solved, Minisat+SP$^{\text{reg}}$ has performance similar to Minisat+Shatter, and Minisat+SP$^{\text{opt}}$ performs best. Note that Table 1 shows again that Minisat is able to solve the satisfiable instances, but has great trouble with the unsatisfiable ones.

In relation to the effectiveness of the optimizations described in Sect. 3.3, we learn from Table 1 that the **Urq** and **x** problems contain only inverting symmetries. In Table 2 we see that Minisat+SP$^{\text{opt}}$ is able to solve the **Urq** and **x** problem families almost instantly, while Minisat+SP$^{\text{reg}}$ is not. Further testing revealed that this fast performance on **Urq** and **x** is solely due to the inverting symmetry optimization. Since this optimization does not influence the solving of problems without inverting symmetries, the performance difference between Minisat+SP$^{\text{reg}}$ and Minisat+SP$^{\text{opt}}$ on the unsatisfiable **fpga**, **hole** and **battleship** problems is due to the inactive propagation optimization. We can conclude that both optimizations significantly improve the performance of SP on many problems.

5 Related Work

We implemented SP in the SAT solver Minisat, but the propositions developed in Sect. 3 are equally applicable to CP solvers. This is not surprising, since SP has its roots in CP symmetry breaking: we borrowed and improved the notion of activity from Lightweight Dynamic Symmetry Breaking (LDSB) [14], which is a formalization of Shortcut Symmetry Breaking During Search (SBDS) [10].

SBDS is based on constraint generation in CP-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 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) [6]. 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, which will also learn $\sigma(c)$ if it is a unit clause whenever c is an explanation clause. The main difference however is that SP 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.

For SAT solvers, Benhamou e.a. proposed a general dynamic symmetry breaking approach based on Corollary 1 [4]: every time a SAT solver with theory T and assignment α backtracks from a certain choice literal l, a local symmetry group G of $T \cup \alpha$ is computed using Saucy, and the orbit of $\neg l$ under G is propagated. The drawback of this algorithm is that repeated computation of symmetries of $T \cup \alpha$ can be very expensive, especially if few local symmetries can be found. Even though SP also uses symmetries of $T \cup \alpha$ to propagate literals, it does not have this problem, since it only keeps track of the weakly active input symmetries.

Other general symmetry breaking approaches for SAT are based on Proposition 1, such as the already explained Symmetrical Learning Scheme (SLS) [5]. A similar method has been independently proposed by Keur e.a. [13], and used to break symmetry in the context of boolean optimization [3].

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 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 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.

6 Future Work

Besides achieving good performance, SP 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.

It has been shown that certain sets of generators are better than others for static symmetry breaking [11].The observation that composing two weakly inactive symmetries under an assignment α can result in a weakly active symmetry can be used to find such better sets of generators. Consider for example the

symmetry $\sigma = (ab)(cde)$. The group generated by σ is the same as the group generated by (ab) and (cde), but the latter symmetries would result in better symmetry propagation. A related possibility for improving propagation could be composing symmetries during search.

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

A last area of improvement is based on the fact that SP is not a complete symmetry breaking approach – a symmetry breaking approach is *complete* if it never traverses two symmetrical search paths. This results from the fact that Minisat+SP$^{\text{reg}}$ does not propagate symmetrical literals during periods where symmetries are weakly inactive, though this incompleteness problem is alleviated in part by the inactive propagation optimization. Nonetheless, when we asked Minisat+SP$^{\text{opt}}$ to give all solutions of pigeonhole problems with as many pigeons as holes, it returned more than one solution, so Minisat+SP$^{\text{opt}}$ also is not complete. Devising an improvement to SP which fully prunes the parts of the search tree symmetrical under the symmetry group generated by 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 on unsatisfiable symmetrical benchmarks, while the satisfiable symmetrical benchmarks appeared relatively easy to solve. Compared to other general dynamic symmetry breaking techniques for SAT, we believe that SP maintains a good balance between detecting local symmetry, not learning too many clauses, and assuring strong propagation. Furthermore, SP is not limited to SAT, but applicable to other problem solving techniques as well.

References

1. The international SAT Competitions web page. www.satcompetition.org
2. Aloul, F.A., Markov, I.L., Sakallah, K.A.: Shatter: efficient symmetry-breaking for boolean satisfiability. In: DAC'03. pp. 836–839 (2003)
3. 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)
4. Benhamou, B., Nabhani, T., Ostrowski, R., Saïdi, M.R.: Dynamic symmetry breaking in the satisfiability problem. In: Proceedings of the 16$^{\text{th}}$ international conference on Logic for Programming, Artificial intelligence, and Reasoning. LPAR-16, Dakar, Senegal (April 25 - may 1, 2010)

5. 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)
6. 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)
7. Darga, P.T., 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)
8. Devriendt, J.: An implementation of Symmetry Propagation in Minisat on Github. www.github.com/JoD/minisat-SPFS
9. 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)
10. Gent, I.P., Smith, B.M.: Symmetry breaking in constraint programming. In: Proceedings of ECAI-2000. pp. 599–603. IOS Press (2000)
11. 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)
12. Katebi, H., Sakallah, K., Markov, I.: Symmetry and satisfiability: An update. In: Strichman, O., Szeider, S. (eds.) Theory and Applications of Satisfiability Testing SAT 2010, Lecture Notes in Computer Science, vol. 6175, pp. 113–127. Springer Berlin / Heidelberg (2010), http://dx.doi.org/10.1007/978-3-642-14186-7_11, 10.1007/978-3-642-14186-7_11
13. Keur, A., Stevens, C., Voortman, M.: CNF Symmetry Breaking Options in Conflict Driven SAT Solving (2005)
14. 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)
15. 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)
16. Narodytska, N., Walsh, T.: Dynamic versus static value symmetry breaking. In: 11th International Workshop on Symmetry in Constraint Satisfaction Problems, SymCon'11 at CP'11 (2011)
17. Sabharwal, A.: SymChaff: exploiting symmetry in a structure-aware satisfiability solver. Constraints 14(4), 478–505 (Dec 2009)
18. 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)
19. Schaafsma, B., Heule, M., van Maaren, H.: Dynamic symmetry breaking by simulating zykov contraction. In: Kullmann, O. (ed.) Theory and Applications of Satisfiability Testing - SAT 2009, Lecture Notes in Computer Science, vol. 5584, pp. 223–236. Springer Berlin / Heidelberg (2009), 10.1007/978-3-642-02777-2_22
20. Urquhart, A.: Hard examples for resolution. J. ACM 34, 209–219 (January 1987)
21. Zhang, L., Madigan, C.F.: Efficient conflict driven learning in a boolean satisfiability solver. In: In ICCAD. pp. 279–285 (2001)