Constraints manuscript No.
(will be inserted by the editor)

Symmetries, Almost Symmetries, and Lazy Clause Generation

Geoffrey Chu · Maria Garcia de la Banda · Christopher Mears · Peter J. Stuckey

Received: - / Accepted: -

Abstract Lazy Clause Generation is a powerful approach for reducing search in Constraint Programming. This is achieved by recording sets of domain restrictions that previously led to failure as new clausal propagators. Symmetry breaking approaches are also powerful methods for reducing search by avoiding the exploration of symmetric parts of the search space. In this paper, we show how we can successfully combine Symmetry Breaking During Search and Lazy Clause Generation to create a new symmetry breaking method which we call SBDS-1UIP. We show that the more precise nogoods generated by a lazy clause solver allow our combined approach to exploit symmetries that cannot be exploited by any previous symmetry breaking method. We also show that SBDS-1UIP can easily be modified to exploit *almost symmetries* very effectively.

Keywords Symmetry Breaking · Almost Symmetry · Lazy Clause Generation

1 Introduction

Lazy Clause Generation (LCG) [28,9] is a powerful approach for reducing search in Constraint Programming. It works by instrumenting finite domain propagation to record the reasons for each propagation step, thus creating an implication graph like the ones built by a SAT solver [26]. This graph is then used to derive nogoods (i.e., reasons for failure) which can be propagated efficiently using SAT unit propagation technology, and can lead to exponential reductions in search space on structured problems. LCG provides state of the art solutions to a number of combinatorial optimization problems such as the Resource Constrained Project Scheduling Problems [32] and the Carpet Cutting Problem [33].

An earlier version of this paper appeared as [4]

Geoffrey Chu · Peter J. Stuckey National ICT Australia, Victoria Laboratory,
Department of Computing and Information Systems,
University of Melbourne, Australia E-mail: gchu,pjs@cis.unimelb.edu.au · Maria Garcia de la Banda · Christopher Mears Faculty of Information Technology,
Monash University, Australia E-mail: mbanda,cmears@infotech.monash.edu.au

Symmetry breaking is also a powerful method for reducing search. Symmetry breaking refers to techniques that, given a constraint problem with a known set of symmetries, avoid the exploration of symmetric parts of the search space when solving the problem. There are two main approaches to symmetry breaking: *static* and *dynamic*. Static symmetry breaking (see, for example, [6,19]) alters the original problem by adding new constraints that ensure the search will find only a single representative of each group of symmetric solutions. In contrast, dynamic symmetry breaking (see, for example, [1,14,8,30]) leaves the original problem unaltered and, instead, alters the search procedure itself to exclude symmetric regions.

Combining static symmetry breaking with LCG is straightforward and quite successful. However, choosing symmetry breaking constraints that are both efficient and correct can be non-trivial, particularly when the problem has different kinds of symmetries. Further, static methods sometimes disagree with the search strategy [11], that is, they might select a representative that is not among the solutions that the search would have found first, or might add constraints with poor propagation for the particular search strategy used. In such cases, it may be more effective to combine LCG with dynamic methods, such as Symmetry Breaking During Search (SBDS) [14,1] or Symmetry Breaking by Dominance Detection (SBDD) [8], since they always agree with the search strategy.

Both SBDS and SBDD can be seen as using symmetric versions of a particular kind of nogood called *decision nogoods* to prune symmetric parts of the search space. Suppose we have a search that fixes a variable to a value at each decision level. Suppose also that the search node reached via the search decisions $d_1, \ldots, d_k$ has no solutions. Then, we can derive the decision nogood $\neg(d_1 \wedge \cdots \wedge d_k)$. If σ is a symmetry of the problem, then the symmetric version of the decision nogood is also valid, that is, $\neg(\sigma(d_1) \wedge \cdots \wedge \sigma(d_k))$ holds. SBDS and SBDD use this knowledge to eliminate from the search any other search node that can later be reached by the set of symmetric decisions $\{\sigma(d_1), \ldots, \sigma(d_k)\}$.

Example 1 Consider a simple constraint problem with variables x_1, x_2, x_3, x_4, all with initial domain $\{1, 2, 3, 4\}$, and two constraints: $x_1 + x_2 + x_3 + x_4 \leq 8$ and *alldiff* $(\{x_1, x_2, x_3, x_4\})$. All variables in this problem are interchangeable, that is, every permutation of the variables is a symmetry. Assume that during the search we make the following two decisions: $x_1 = 1, x_2 = 2$. Propagating the *alldiff* constraint forces $x_3 \geq 3, x_4 \geq 3$, which violates the linear constraint. This conflict leads to the decision nogood $\neg(x_1 = 1 \wedge x_2 = 2)$. We can apply any of the variable symmetries in the problem to this nogood to get other valid nogoods, such as $\neg(x_1 = 1 \wedge x_3 = 2)$ and $\neg(x_4 = 1 \wedge x_1 = 2)$. □

While SBDS and SBDD only use decision nogoods, LCG and Boolean Satisfiability (SAT) solvers use what are known as First Unique Implication Point (1UIP) nogoods. 1UIP nogoods have empirically been found to be much stronger than decision nogoods in terms of their pruning power in the non-symmetric case [34]. Clearly, it would be interesting to see if the extra pruning power of 1UIP nogoods carries over to the symmetric case.

In this paper, we show how to combine an LCG solver with dynamic symmetry breaking methods that, like SBDS, work by posting symmetric nogoods. This combination can be understood either as extending the LCG solver to post symmetric versions of its 1UIP nogoods, or as extending SBDS to make use of 1UIP nogoods. We call our new method SBDS-1UIP. The strength of our method comes from the fact that SBDS only uses simple decision nogoods involving equality literals on the decision variables, whereas the 1UIP nogoods derived by LCG solvers may involve equality, disequality and inequality literals on any problem variable (including intermediate variables introduced by the solver). Furthermore, we formally show that SBDS-1UIP is at least as strong as SBDS, and that it can be strictly stronger on some problems. In fact, SBDS-1UIP allows us to exploit types of symmetries that no other general symmetry breaking method we are aware of can exploit.

Finally, we show that, with a slight modification, SBDS-1UIP can be adapted to exploit *almost symmetries* [20], that is, symmetries that would appear in the problem if a small set of constraints was removed from it. Almost symmetries are important because they appear in many real world problems and can often be exploited to yield significant speedups. Unfortunately, they are not well behaved mathematically and it is difficult to adapt traditional symmetry breaking methods such as lex-leader static symmetry breaking constraints [6], SBDS or SBDD to exploit them. While there are a few theoretical works in this area [16,17], there exists only one implementation of a general method for exploiting almost symmetries [20]. Our new method for exploiting almost symmetries using SBDS-1UIP is, therefore, an important contribution to this area.

The rest of the paper is organized as follows. Section 2 provides our definitions and describes LCG and SBDS. Section 3 describes the SBDS-1UIP method, while Section 4 describes its implementation. Section 5 proves that symmetric 1UIPs are at least as strong as their associated symmetric decision nogoods, while Section 6 shows that symmetric 1UIPs can be strictly stronger. Section 7, shows how SBDS-1UIP can be modified to exploit almost symmetries. Section 8 discussed related work. Section 9 presents experimental results. And finally, Section 10 concludes.

2 Background

Let $vars(O)$ denote the set of variables of any syntactic object O, $|S|$ denote the cardinality of set S, $\equiv$ denote syntactic identity, and $\models$ denote logical implication. A Constraint Satisfaction Problem (CSP) is a triple (V, D, C), where V is a set of variables, D is a set of unary constraints (the *domain*), and C is a set of constraints. For each variable $x \in V$, we can see the domain D as containing exactly one unary constraint of the form $x \in D(x)$, where $D(x)$ is a finite set of values defining the values that variable x can take. Domain D is said to be stronger than domain D' if $D \models D'$.

A *pair* of $P = (V, D, C)$ is of the form $x \mapsto d$ where $x \in V$ and $d \in D(x)$. We denote the set of all pairs of P by $pairs(P)$. A *pairset* of P is a subset of $pairs(P)$, and a *valuation of P over $X \subseteq V$* is a pairset that contains exactly one pair $x \mapsto d$ for each variable $x \in X$. Where the identity of P is clear, we omit the "of P" part. A constraint $c \in C$ over a set $X \subseteq V$ of variables is a set of valuations over X, and we say that $scope(c) = X$. Valuation θ over $scope(c)$ is *allowed* by c if $\theta \in c$. Valuation θ over $X \subseteq V$ *satisfies* c if $scope(c) \subseteq X$ and the projection of θ over $scope(c)$ — defined as $\{x \mapsto d \mid x \mapsto d \in \theta, x \in scope(c)\}$ — is allowed by c. If valuation θ over $X \subseteq V$ does not satisfy constraint c and $scope(c) \subseteq X$, we say that θ *violates* c. For brevity, constraints are usually written *intensionally* as formulas (e.g., $x_1 + x_2 = x_3$) from which the allowed valuations can be determined. A *solution* of P is a valuation over V that satisfies every $c \in C$.

In an abuse of notation, if a symbol A refers to a set of constraints $\{c_1, \ldots, c_k\}$, we will often also use A to refer to constraint $c_1 \wedge \cdots \wedge c_k$. This allows us to avoid repetitive use of conjunction symbols. It should be clear from the context which meaning we take: if we apply set operators to A like $\in, \cup, \cap$, we are treating A as a set of constraints, while if we apply logical operators to A like $\wedge, \vee, \models$, we are treating A as a constraint.

Constraint propagation infers new information about the possible values of variables V from CSP $P = (V, D, C)$ by using the domain D and the constraints $c \in C$ of the problem. The propagator of a constraint c, denoted $\mathit{prop}(c)$, is a function that maps a domain D to a set U of unary constraints, each of which is either *false* or a constraint over a single variable in $\mathit{scope}(c)$, such that each constraint in U is a logical consequence of c and D. We can use the propagation of c to create a new stronger domain D' given by $D' = D \wedge \mathit{prop}(c)(D)$. Clearly the two problems P and (V, D', C) are equivalent. A propagator $\mathit{prop}(c)$ is a *bounds propagator* if U is always a conjunction of inequality literals or $\{\mathit{false}\}$.

Propagation is the process or repeatedly applying propagation on all constraints in the problem until no new information is created. Let $\mathit{propfix}(C, D) = D'$ characterize the propagation engine of our solver, i.e., *propfix* takes the constraints C and the current domain D and returns a new stronger domain D' obtained by repeatedly applying propagation for every constraint $c \in C$ until the domain no longer changes.

Constraint programming solvers solve a CSP by interleaving search with propagation as follows. We begin with the original problem (V, D, C) at the root of the search tree. At each node in the search tree, we apply the propagation engine to the current domain D to determine a new domain $D' = \mathit{propfix}(C, D)$. If D' is equivalent to false (i.e., $\exists x \in V, |D'(x)| = 0$), the subtree has no solution and the solver backtracks. If all the variables are assigned (i.e., $\forall\ x \in V, |D'(x)| = 1$) and no constraint is violated, then a solution has been found and the solver can terminate. If neither of these cases hold, then the solver splits the problem into a number of more constrained subproblems and searches each of those in turn. The problem is split by making different *decisions* d_i down each branch, e.g., $x = 1$ down one branch and $x \neq 1$ down the other, or $x \leq 3$ down one and $x > 3$ down the other, etc. We assume that all decisions are unary constraints of the form $x = v, x \neq v, x \geq v, x \leq v$. While this is not a strong restriction, it does rule out some kinds of search. Given a node s in the search tree, we will denote by $DS(s)$ the sequence $[d_1, \ldots, d_k]$ of decisions taken to reach s. Note that s implicitly represents the CSP $(V, D, C \cup \{d_1, \ldots, d_k\})$.

2.1 Lazy Clause Generation

Lazy Clause Generation (LCG) extends a constraint propagation solver by recording the effects of propagation in terms of literals, and using this to derive nogoods during search which are then used as propagators to reduce the search space. A *nogood* n is a constraint of the form $\neg(l_1 \wedge \cdots \wedge l_k)$, where each *literal* l_i is a unary constraint, and n is implied by the problem constraints, i.e., $D \wedge C \models n$. In other words, the problem constraints imply that there is no solution in which all l_i literals are satisfied. In practice and due to the way propagation works by modifying the domain of the variables, the literals in a nogood are always either *equality* literals ($x = d$ for $d \in D(x)$), *disequality* literals ($x \neq d$ for $d \in D(x)$), or *inequality* literals ($x \geq d$ or $x \leq d$ for $d \in D(x)$). Given the nogood $\neg(l_1 \wedge \cdots \wedge l_k)$, we shall sometimes use its equivalent Horn clause representation $l_1 \wedge \cdots \wedge l_{k-1} \rightarrow \neg l_k$. Given two nogoods with complementary terms a and $\neg a$, we can *resolve* them together to get a new nogood, i.e., $\neg(a \wedge l_1 \wedge \cdots \wedge l_k) \wedge \neg(\neg a \wedge l'_1 \wedge \cdots \wedge l'_m) \models \neg(l_1 \wedge \cdots \wedge l_n \wedge l'_1 \wedge \cdots \wedge l'_m)$.

LCG solvers derive nogoods by instrumenting their propagators to *explain* each of their propagations using a nogood. In particular, suppose the propagator for constraint c with current domain D makes a (unary) inference m, i.e., $m \in prop(c, D)$ and, therefore, $c \wedge D \models m$. An *explanation* for this inference $expl(m)$ is a nogood $l_1 \wedge \cdots \wedge l_k \rightarrow m$ such that $c \models expl(m)$ and $D \models l_1 \wedge \cdots \wedge l_k$. Intuitively, the nogood $expl(m)$ explains why m has to hold given c and the current domain D. We can consider $expl(m)$ as the fragment of the constraint c from which we inferred that m has to hold. When the propagation infers *false* with explanation $l_1 \wedge \cdots \wedge l_k \rightarrow false$, the associated nogood $\neg(l_1 \wedge \cdots \wedge l_k)$ is referred to as the *conflicting* nogood.

Example 2 Given constraint $x \leq y$ and current domain $x \in \{3, 4, 5\}$, the propagator may infer $y \geq 3$ with explanation $x \geq 3 \rightarrow y \geq 3$. If y were known to have domain $\{1, 2\}$, then the propagator might infer *false* with explanation $x \geq 3 \wedge y \leq 2 \rightarrow false$, where $\neg(x \geq 3 \wedge y \leq 2)$ is the conflicting nogood. □

These explanations form an acyclic implication graph where each node corresponds to a newly inferred literal m obtained by $prop(c, D)$, and there are edges from each node $l_i, 1 \leq i \leq k$ occurring on the left hand side of $expl(m)$ to m. Acyclicity follows since we add a new node in the graph m corresponding to a new inference $m \in prop(c, D)$ and, hence, all edges to m are from literals that were already true in D and hence already exist in the graph. Since we only add edges from nodes already in the implication graph to new nodes just added to the implication graph, the graph is always acyclic. Each literal has a *decision level*, defined as the depth in the search tree at which the literal was first inferred to be true. Since the root is defined as depth 0, literals that are true at the root have decision level 0.

Whenever a conflict is found by an LCG solver, the implication graph can be analyzed in order to derive a 1UIP nogood which is asserting (always asserts new information upon backtracking), quite general (still close to the cause of the conflict) and fast to compute. This is done by repeatedly resolving the conflicting nogood with explanation nogoods, resulting in a new nogood for the problem, until it contains at most one literal from the last decision level. Therefore, every time we have a nogood $\neg(a \wedge l_1 \wedge \cdots \wedge l_k)$ we look for the explanation of the last inferred literal a, $l'_1 \wedge \cdots \wedge l'_m \rightarrow a$, and resolve the two. In terms of the implication graph, this is the same as eliminating a from the nogood and adding to it the conjunction of the literals whose edges end in a. Let $resolve(n_1, n_2)$ denote the resolution of two nogoods n_1 and n_2. The algorithm for deriving the 1UIP nogood is as follows:

```
derive1UIP()
    $n$ := conflicting nogood
    while ($n$ has more than one literal in the last decision level)
        $a$ := last literal to be inferred among literals in $n$
        $n := resolve(n, expl(a))$
    return $n$
```

The one remaining literal from the last decision level in n is called the *asserting literal.* Note that we can eliminate from n any literal that is true at the root level (i.e., any literal with level 0), as such literals are globally true and can be safely removed.

Example 3 Consider again the simple constraint problem from Example 1 with variables x_1, x_2, x_3, x_4, all with initial domain $\{1, 2, 3, 4\}$, and two constraints: $x_1+x_2+x_3+x_4 \leq 8$ and $alldiff(\{x_1, x_2, x_3, x_4\})$. Figure 1 shows the relevant part of the implication graph already computed when propagation infers *false* after the sequence of decisions $[x_1 = 1, x_2 = 2]$. The double boxes indicate decision literals while the dashed lines partition literals into decision levels. Note that literals which are true at the 0th level, e.g., $x_1 \geq 1$ are technically part of explanations, but can be ignored for the purpose of conflict analysis.

To obtain the 1UIP nogood using the derive1UIP algorithm, we start with the conflict nogood $\neg(x_2 \geq 2 \land x_3 \geq 3 \land x_4 \geq 3)$, which contains every literal directly connected to the *false* conclusion. We have two literals from the last decision level ($x_3 \geq 3$ and $x_4 \geq 3$). Since $x_4 \geq 3$ was the last literal to be inferred of those two, we resolve the current nogood with $expl(x_4 \geq 3) \equiv x_4 \geq 2 \land x_4 \neq 2 \rightarrow x_4 \geq 3$, yielding the new nogood: $\neg(x_2 \geq 2 \land x_3 \geq 3 \land x_4 \geq 2 \land x_4 \neq 2)$. In other words, we remove $x_4 \geq 3$ from the nogood and add to it the conjunction of the literals whose edges end in $x_4 \geq 3$, according to the implication graph. Since we still have two literals from the last decision level ($x_3 \geq 3$ and $x_4 \neq 2$), we choose to resolve with $expl(x_3 \geq 3) \equiv x_3 \geq 2 \land x_3 \neq 2 \rightarrow x_3 \geq 3$, yielding: $\neg(x_2 \geq 2 \land x_3 \geq 2 \land x_3 \neq 2 \land x_4 \geq 2 \land x_4 \neq 2)$. Since we still have two literals from the last decision level ($x_3 \neq 2$ and $x_4 \neq 2$), we choose to resolve with $expl(x_4 \neq 2) \equiv x_2 = 2 \rightarrow x_4 \neq 2$, yielding: $\neg(x_2 \geq 2 \land x_3 \geq 2 \land x_3 \neq 2 \land x_4 \geq 2 \land x_2 = 2)$. Since we still have two literals from the last decision level ($x_3 \neq 2$ and $x_2 = 2$), we choose to resolve with $expl(x_3 \neq 2) \equiv x_2 = 2 \rightarrow x_3 \neq 2$, yielding: $\neg(x_2 \geq 2 \land x_3 \geq 2 \land x_4 \geq 2 \land x_2 = 2)$. At this point, there is only one literal from the last decision level left ($x_2 = 2$) and we can terminate. The asserting literal is $x_2 = 2$ and we write the 1UIP nogood as a Horn clause with this literal as the head, i.e., as $x_2 \geq 2 \land x_3 \geq 2 \land x_4 \geq 2 \rightarrow x_2 \neq 2$.

Contrast the 1UIP nogood with the decision nogood which just collects all decision literals that are antecedents of the failure, in this case $x_1 = 1 \land x_2 = 2 \rightarrow$ *false* or equivalently $x_1 = 1 \rightarrow x_2 \neq 2$. It propagates that $x_2 \neq 2$ strictly less often than the 1UIP nogood, since $x_1 = 1$ will always ensure that $x_2 \geq 2 \land x_3 \geq 2 \land x_4 \geq 2$. The propagation in the reverse direction is incomparable, but in practice this is not the important direction of propagation of the nogood.

2.2 Symmetries

Given a permutation σ on $pairs(P)$, we extend σ to map a subset θ of $pairs(P)$ in the obvious manner: $\sigma(\theta) = \{\sigma(x \mapsto v) \mid x \mapsto v \in \theta\}$. Note that it is possible for σ to map valuations to non-valuations and vice versa.

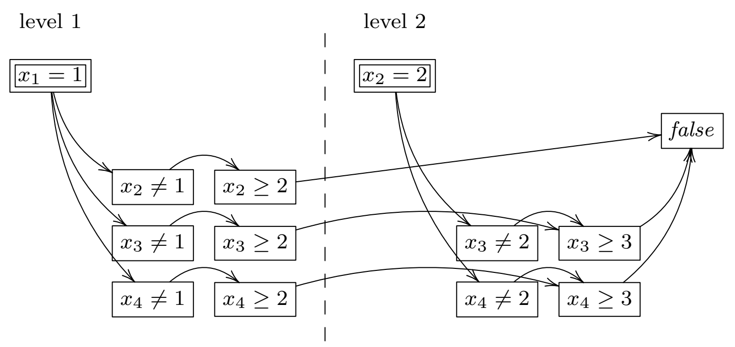

Fig. 1 Implication graph for Example 3. Decision literals are double boxed. Decision levels are separated by dashed lines.

A *solution symmetry* of problem $P \equiv (V, D, C)$ is a permutation σ on *pairs*(P) that preserves the set of solutions of P [5], that is, σ maps solutions of P to solutions of P and non-solutions of P to non-solutions of P.

A *symmetry group* of problem P is a set S of solution symmetries of P, s.t. S is a group under the operation of function composition, that is, $\forall \sigma_1, \sigma_2 \in S.\ \sigma_1 \cdot \sigma_2 \in S$, $\forall \sigma \in S.\ \sigma^{-1} \in S$, and the identity symmetry is in S. The *orbit* of pairset θ under the symmetry group S is the set: $\{\sigma(\theta) \mid \sigma \in S\}$. Set S' is a *generating set* of symmetry group S if it *generates* S, that is, if its closure under composition and inverse is equal to S.

Two important kinds of solution symmetry are induced by permuting either the variables or the values in P. In particular, a permutation s on the variables in V induces a permutation σ_s on *pairs*(P) by defining $\sigma_s(x \mapsto d) = s(x) \mapsto d$. A *variable symmetry* is a permutation of the variables in P whose induced pair permutation is a solution symmetry [29]. A set S of value permutations s_x, one for the values in each $D(x), x \in V$, induces a permutation σ_S on *pairs*(P) by defining $\sigma_S(x \mapsto d) = x \mapsto s_x(d)$. A *value symmetry* is a set of value permutations whose induced pair permutation is a solution symmetry [29]. Like most other papers in the area, we only discuss value symmetries where the values are treated equivalently across variables, i.e. where $s_i = s_j$ for all $s_i, s_j \in S$ and, therefore, a single s_i serves as a representative of S.

Variable (value) symmetries often appear in the form of *interchangeable variables (values)* [19], that is, a set of variables (values) W s.t. any permutation of W is a variable (value) symmetry of the CSP.

Example 4 Consider problem $P \equiv (V, D, C)$, where $V \equiv \{x_1, x_2, x_3\}$, $D \equiv \{x_i \in \{1, \ldots, 5\} \mid i \in \{1, 2, 3\}\}$ and $C \equiv \{\mathit{alldiff}(x_1, x_2, x_3), x_1 + x_2 + x_3 \geq 10\}$. Variables x_1, x_2 and x_3 are interchangeable since any permutation of V is a variable symmetry of P. Let S be the set of all permutations of V. Then, S induces a symmetry group of P and the orbit of valuation $\theta \equiv \{x_1 \mapsto 1, x_2 \mapsto 2, x_3 \mapsto 3\}$ under S is $\{\{x_1 \mapsto 1, x_2 \mapsto 2, x_3 \mapsto 3\}, \{x_1 \mapsto 1, x_2 \mapsto 3, x_3 \mapsto 2\}, \{x_1 \mapsto 2, x_2 \mapsto 1, x_3 \mapsto 3\}, \{x_1 \mapsto 2, x_2 \mapsto 3, x_3 \mapsto 1\}, \{x_1 \mapsto 3, x_2 \mapsto 1, x_3 \mapsto 2\}, \{x_1 \mapsto 3, x_2 \mapsto 2, x_3 \mapsto 1\}\}$.

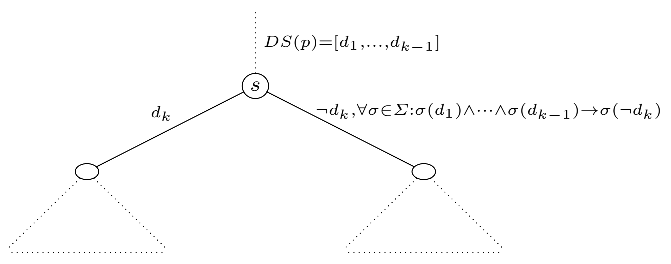
$DS(p)=[d_1,\ldots,d_{k-1}]$
s
d_k
$\neg d_k, \forall \sigma \in \Sigma : \sigma(d_1) \wedge \cdots \wedge \sigma(d_{k-1}) \rightarrow \sigma(\neg d_k)$

Fig. 2 Overview of SBDS decision point.

Consider now the problem $P \equiv (V, D, C)$, $V \equiv \{x_1, x_2, x_3\}$, $D \equiv \{x_i \in \{1, \ldots, 5\} \mid i \in \{1, 2, 3\}\}$, $C \equiv \{\mathit{alldiff}(x_1, x_2, x_3), x_1 \neq x_2, x_2 \neq 2, x_3 \neq 4\}$. Values $\{1, 3, 5\}$ on $\{x_1, x_2, x_3\}$ are interchangeable, since any permutation of $\{1, 2, 3, 4, 5\}$ that fixes 2 and 4 is a value symmetry of P. Let S be the set of all such value symmetries. Then, S induces a symmetry group of P and the orbit of valuation $\theta \equiv \{x_1 \mapsto 1, x_2 \mapsto 2, x_3 \mapsto 3\}$ under S is $\{\{x_1 \mapsto 1, x_2 \mapsto 2, x_3 \mapsto 3\}, \{x_1 \mapsto 1, x_2 \mapsto 2, x_3 \mapsto 5\}, \{x_1 \mapsto 3, x_2 \mapsto 2, x_3 \mapsto 1\}, \{x_1 \mapsto 3, x_2 \mapsto 2, x_3 \mapsto 5\}, \{x_1 \mapsto 5, x_2 \mapsto 2, x_3 \mapsto 1\}, \{x_1 \mapsto 5, x_2 \mapsto 2, x_3 \mapsto 3\}\}$. □

A special case of solution symmetry, which we will call a *valuation* symmetry (originally defined by [25]), occurs when σ maps all the literals of each variable to the literals of one other variable only, i.e., there exists a variable permutation π s.t. for all x and v, $\sigma(x \mapsto v) = \pi(x) \mapsto v'$ for some v'. This subclass of symmetries have the property that σ always maps a valuation to another valuation. It includes many common symmetries such as interchangeable variables, interchangeable values, or interchangeable rows or columns in matrix problems. The advantage of valuation symmetries is that it is easy to extend a valuation symmetry to map constraints to constraints, since the mapping on variables is uniform. Given a valuation symmetry σ, then $\sigma(c)$ is the constraint with solutions $\{\sigma(\theta) \mid \theta \in c\}$ and scope $\pi(\mathit{scope}(c))$. For non-valuation symmetries this definition may not be meaningful since $\sigma(\theta)$ may not be a valuation!

Example 5 Consider the n-queens problem modelled with variables $x_1, \ldots, x_n$ taking values from $\{1, \ldots, n\}$, where $x_i = j$ represents the fact that the queen in the i^{th} row occurs in the j^{th} column. The solution symmetry $\sigma(x_i \mapsto j) = x_j \mapsto i$, which describes a diagonal flip, is clearly not a valuation symmetry since different pairs for the same variable are mapped to pairs on different variables. When this symmetry is applied to one of the constraints of the n-queens problem $x_1 \neq x_2 + 1$, it maps the solution $\theta = \{x_1 \mapsto 1, x_2 \mapsto 1\}$ of this constraint to the non-valuation $\sigma(\theta) = \{x_1 \mapsto 1, x_1 \mapsto 2\}$. □

2.3 Symmetry Breaking During Search

SBDS can be seen as an instance of the $\mathcal{S}$*-excluding* search tree method of Backofen and Will [1], but was independently developed by Gent and Smith [14]. SBDS solves $P \equiv (V, D, C)$ with set of solution symmetries S by performing a depth-first search of a tree whose nodes have either zero or two children, with the left and right children (if any) labelled by search decisions d_k (an equality literal) and $\neg d_k$, respectively.[1] The search proceeds as usual until it backtracks to the right child of a node s. Let $DS(s) = [d_1, \ldots, d_{k-1}]$, and assume the left child of s is made with decision d_k, as shown in Figure 2. Once the region under the left child has been explored, every possible assignment that includes $d_1 \wedge \cdots \wedge d_k$ has been examined and, therefore, we can exclude from the search any symmetric assignment. SBDS achieves this by posting on the right child of s, for each solution symmetry $\sigma \in S$, the constraint $\sigma(d_1) \wedge \cdots \wedge \sigma(d_{k-1}) \rightarrow \sigma(\neg d_k)$. Note that since each d_i is known to be an equality literal, computing $\sigma(d_i)$ is very easy: let d_i be of the form $x = d$ and $\sigma(x \mapsto d)$ be $x' \mapsto d'$, then $\sigma(d_i) = (x' = d')$. Further, $\sigma(\neg d_i) = \neg\sigma(d_i) = \neg(x' = d')$.

SBDS posts the constraint $\sigma(d_1) \wedge \cdots \wedge \sigma(d_{k-1}) \rightarrow \sigma(\neg d_k)$ only locally, that is, on backtracking the constraint is removed. This is not only easy to implement in a backtracking solver, it is also efficient because once we fail the right branch we know that the parent node $DS(s) = [d_1, \ldots, d_{k-1}]$ is failed, and its parent will post the *stronger constraint* $\sigma(d_1) \wedge \cdots \wedge \sigma(d_{k-2}) \rightarrow \sigma(\neg d_{k-1})$.

3 SBDS-1UIP

Our idea is to extend LCG by posting not only 1UIP nogoods but also their symmetric versions. For this we need to be able to apply symmetries to nogoods in such a way that we can efficiently obtain another nogood with the same form. To do so, we define the application $\sigma(n)$ of solution symmetry σ of $P \equiv (V, C, D)$ to nogood n of the form $\neg(l_1 \wedge \cdots \wedge l_k)$ as $\neg(\sigma(l_1) \wedge \cdots \wedge \sigma(l_k))$, and prove that the result is also a nogood if each l_i is either an equality or a disequality literal (inequality literals will be discussed in Section 4.1).

Theorem 1 *Let n be of the form $\neg(l_1 \wedge \cdots \wedge l_k)$, where each l_i is either an equality or a disequality literal. Given a problem $P \equiv (V, D, C)$ and a solution symmetry σ of P, if n is a nogood of P, then so is $\sigma(n) \equiv \neg(\sigma(l_1) \wedge \cdots \wedge \sigma(l_k))$.*

Proof The theorem holds if no solution of P can satisfy $\sigma(n)$. Let us reason by contradiction and assume there is a solution θ of P such that θ satisfies $\sigma(n)$. By the definition of symmetry, $\sigma^{-1}(\theta)$ must also be a solution of P.

For any equality literal $x = d$ in $l_1 \wedge \cdots \wedge l_k$, $\sigma(x \mapsto d)$ must be in θ and, therefore, $x \mapsto d$ must be in $\sigma^{-1}(\theta)$. Thus, $\sigma^{-1}(\theta)$ satisfies every equality literal in $l_1 \wedge \cdots \wedge l_k$.

For any disequality literal $x \neq d$ in $l_1 \wedge \cdots \wedge l_k$, $\sigma(x \mapsto d)$ cannot be in θ and, therefore $x \mapsto d$ cannot be in $\sigma^{-1}(\theta)$. Thus, $\sigma^{-1}(\theta)$ also satisfies every disequality literal in $l_1 \wedge \cdots \wedge l_k$. Since by assumption of nogood, P cannot have a solution $\sigma^{-1}(\theta)$ that satisfies every literal in $l_1 \wedge \cdots \wedge l_k$, θ cannot exist. □

Using the above result, SBDS can be explained in terms of nogoods: it simply detects that $d_1 \wedge \cdots \wedge d_{k-1} \rightarrow \neg d_k$ is a nogood and uses it to post the symmetric

[1] Note that while SBDS requires d_k to be an equality literal, that formulation of [1] allows d_k to be any constraint.

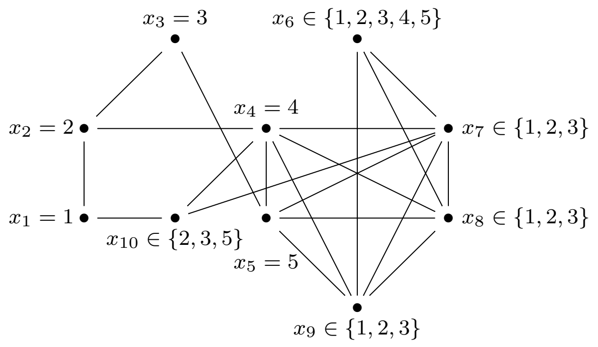

Fig. 3 A graph coloring problem where we can exploit additional symmetries.

nogood $\sigma(d_1) \wedge \cdots \wedge \sigma(d_{k-1}) \rightarrow \neg\sigma(d_k)$ as a local constraint under the right child of node s.

Extending LCG to use 1UIP nogoods is therefore very simple. Upon backtracking of node s, the LCG solver will derive some 1UIP nogood $n \equiv (l_1 \wedge \cdots \wedge l_{k-1} \rightarrow \neg l_k)$. Theorem 1 tells us that every symmetric version of this nogood $\sigma(n)$ is also a nogood. Therefore, if $\sigma(n)$ is not globally satisfied already, we can post $\sigma(n)$ as a nogood.

Example 6 Consider the graph coloring problem where we aim to color the nodes of a graph using a finite number m of colors with no adjacent nodes having the same color. This can be modeled as CSP $P \equiv (V, D, C)$ where the nodes are the variables and the colors are the values, that is, where V has one integer variable for each node, D assigns domain $\{1, \ldots, m\}$ to every variable, and C ensures that adjacent nodes have different colors. For the graph of Figure 3, $V \equiv \{x_1, \ldots, x_{10}\}$, $D(x_i) = \{1, \ldots, 5\}, i \in \{1, \ldots, 10\}$, and for each edge in the graph between node i and node j, C includes constraint $x_i \neq x_j$. Clearly, all 5 colors are interchangeable, that is, every permutation of the values is a value symmetry.

Assume we have made decisions $DS \equiv x_1 = 1 \wedge x_2 = 2 \wedge x_3 = 3 \wedge x_4 = 4 \wedge x_5 = 5$, and propagation has produced the domains shown in Figure 3. Suppose we decide to try $x_6 = 1$ next, obtaining by propagation $x_7 \in \{2,3\}, x_8 \in \{2,3\}, x_9 \in \{2,3\}$ and we then decide to try $x_7 = 2$. This forces $x_8 = 3, x_9 = 3$, which conflicts. The decision nogood from this conflict is: $DS \wedge x_6 = 1 \rightarrow x_7 \neq 2$. The 1UIP nogood obtained from the implication graph shown in Figure 4 by applying the derive1UIP algorithm to conflict nogood $x_8 = 3 \wedge x_9 = 3 \rightarrow \mathit{false}$ is: $x_8 \neq 1 \wedge x_8 \neq 4 \wedge x_8 \neq 5 \wedge x_9 \neq 1 \wedge x_9 \neq 4 \wedge x_9 \neq 5 \rightarrow x_7 \neq 2$.

Upon backtracking from decision $x_7 = 2$, SBDS will not be able to prune anything, since for every symmetry σ except the identity, $\sigma(DS \wedge x_6 = 1)$ is inconsistent with $DS \wedge x_6 = 1$ (at least two variables will be assigned to different values), so the LHS is false and the symmetric nogood is trivially satisfied. In

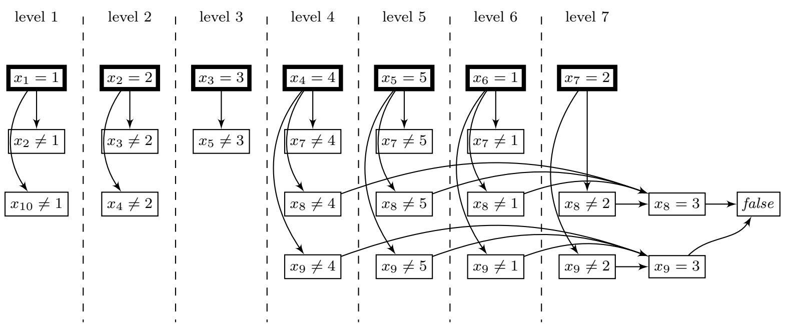

Fig. 4 Relevant part of the implication graph for the graph coloring problem.

contrast, SBDS-1UIP will be able to prune. For example, if we apply the value symmetry between 2 and 3 to the 1UIP nogood, we get: $x_8 \neq 1 \wedge x_8 \neq 4 \wedge x_8 \neq 5 \wedge x_9 \neq 1 \wedge x_9 \neq 4 \wedge x_9 \neq 5 \rightarrow x_7 \neq 3$, which immediately prunes the value of 3 from x_7. Similarly, upon backtracking from decision $x_6 = 1$, SBDS can prune nothing while SBDS-1UIP can: the 1UIP nogood is $x_7 \neq 4 \wedge x_7 \neq 5 \wedge x_8 \neq 4 \wedge x_8 \neq 5 \wedge x_9 \neq 4 \wedge x_9 \neq 5 \rightarrow x_6 \neq 1$, and we can apply the value symmetries between 1 and 2, and between 1 and 3 to prune values 2 and 3 from x_6.

Interestingly, the other commonly used symmetry breaking techniques are also powerless here. In particular, as SBDD is equivalent in power to SBDS, it cannot exploit this symmetry either. Also, the standard lex-leader symmetry breaking constraint for a value symmetry is: $min\{i|x_i = 1\} < \cdots < min\{i|x_i = 5\}$ which simply states that value 1 has to be used by a "smaller" variable than value 2, etc. Since all 5 values have already been used in $x_1, \ldots, x_5$ and in the right order, the symmetry breaking constraint is already satisfied and can prune nothing further.

Even the traditional conditional symmetry breaking constraints [13] cannot exploit the symmetries that SBDS-1UIP exploits here. In a graph coloring problem, we can post conditional symmetry breaking constraints to exploit more symmetries than a static lex-leader symmetry breaking constraint. In particular, since only the values of the nodes in the current labelled frontier have any effect on the remaining variables, there are sometimes conditional symmetries in the subproblems. For example, given the current partial assignment, the values of x_2 and x_3 no longer affect those of $x_6, x_7, x_8, x_9, x_{10}$. Thus, we can add the following conditional symmetry: given condition $x_1 = 1 \wedge x_4 = 4 \wedge x_5 = 5$, the values 2 and 3 are interchangeable on $x_6, x_7, x_8, x_9, x_{10}$. However, the symmetry exploited by SBDS-1UIP is the one between values 1 and 2, and between values 1 and 3, which are not traditional[2] value symmetries of this subproblem. □

We will discuss the difference between the power of these methods in more detail in Section 6. Note that, in contrast to SBDS, it can be useful in SBDS-1UIP to post $\sigma(n)$ as a global rather than as a local constraint. This is because in SBDS, the decision nogood derived at search node q is always subsumed by that derived for its parent node and, therefore, it was sufficient to post $\sigma(n)$ locally. However,

[2] They are not value symmetries for *all* variables in the subproblem: they are for variables x_6, x_7 and x_8, but not for x_9 and x_{10},

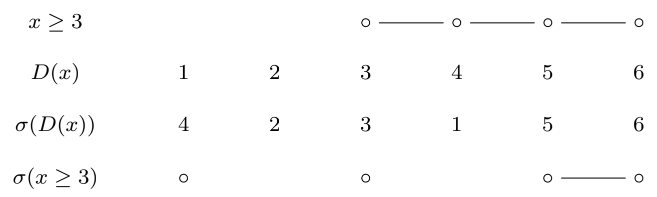

Fig. 5 Illustration of the mapping of inequality literal $x \geq 3$ under the value symmetry σ which swaps values 1 and 4.

the 1UIP nogood at the parent does not generally subsume the one at q and, therefore, posting $\sigma(n)$ globally might yield additional pruning.

4 Implementing SBDS-1UIP

There are three main issues encountered when implementing SBDS-1UIP. The first two arise from the need to calculate the symmetric versions of 1UIP nogoods that might involve inequality literals and might be defined not only on decision variables but also on intermediate variables. The third issue arises from the need to give up completeness in order to speedup the search. This section discusses these three issues in more detail.

4.1 Inequality literals

Propagators often infer inequality constraints (in fact, that is the only constraint inferred by bounds propagators). However, Theorem 1 assumes the nogoods contain only equality and disequality literals. This is because it is easy to see (and thus prove) that the application of any symmetry to an equality or disequality literal is also an equality or disequality literal. This is, however, not so clear when the symmetry is applied to an inequality literal $x_i \geq v$. If the symmetry is a valuation symmetry, the meaning is well defined even if it is not necessarily a literal. But we can extend the notion to an arbitrary solution symmetry σ by noting that if l is a lower bound for x_i's original domain, then $x_i \geq v$ is equivalent to $x_i \neq l \wedge x_i \neq l+1 \wedge \cdots \wedge x_i \neq v-1$, and we can apply σ to each inequality literal separately.

Example 7 Let x be a variable with domain $D(x) = \{1,2,3,4,5,6\}$, and σ a value symmetry swapping values 1 and 4. As shown in Figure 5, the set of solutions of inequality $x \geq 3$ (marked by circles) clearly forms a continuous domain, as represented by the lines connecting every possible solution at the top. In contrast, the set of solutions of $\sigma(x \geq 3)$ does not (solutions $x = 1$ and $x = 3$ are disconnected from each other and from the rest). If $x \geq 3$ appears in a nogood we can first convert it into $x \neq 1 \wedge x \neq 2$ and then apply the symmetry to each disequality obtaining $x \neq 4 \wedge x \neq 2$. □

Given the above discussion for any σ and variable x with initial domain $\{l, \ldots, u\}$ we define

$$\begin{array}{l}\sigma(x \geq v) = \bigwedge_{v' \in \{l,\ldots,v-1\}} \sigma(x \neq l) \\ \sigma(x \leq v) = \bigwedge_{v' \in \{v+1,\ldots,u\}} \sigma(x \neq l)\end{array}$$

With this extension we have the following corollary to Theorem 1.

Corollary 1 *Given a problem $P \equiv (V, D, C)$ and a solution symmetry σ of P, if n is a nogood of P, then so is $\sigma(n)$.* □

Note that for variable symmetries we can directly map inequality literals to inequality literals. For other symmetries, each inequality literal may potentially map to quite a large number of disequality literals. While this can significantly increase the length of $\sigma(n)$ compared to that of nogood n, the impact on performance is not high. This is because the cost of propagating nogoods in LCG (and SAT) solvers is not linear in the length of the nogood [26]. In practice, increasing the length of the nogood typically does not increase its propagation cost by much. The only real cost is the memory required to store the longer nogood, which is rarely a problem on modern computers.

4.2 Intermediate variables

In practice CSPs are not quite as straightforward as the definitions provided in Section 2 make them appear. It is common to model complex constraints using higher order constructs, which in practice are decomposed into many smaller constraints by introducing *intermediate variables*, that is, variables that are not present in the original model. These variables appear in the constraints that actually are sent to the solver. For example, a high level model written in the modeling language MiniZinc [27] is first flattened into primitive constraints using the instance data, with intermediate variables introduced as necessary. This flattened form is then given to a solver which may, for example, introduce global propagators implemented by decomposition.

Example 8 Consider the Concert Hall problem where we have k identical halls and n orders for renting these halls, each order with an associated start time, end time, and price. We can choose to accept each order or not but, if we accept the order, we must schedule it in one of the k halls. Two accepted orders that overlap in time cannot be scheduled in the same hall. The aim is to choose the orders (and assign them a hall) that maximize the profit we can make. A MiniZinc [27] model for the Concert Hall problem is as follows:

```
% ============ Parameters ============
int: n; % number of orders
int: k; % number of halls

set of int: Orders = 1..n;    % set of possible orders
array[Orders] of int: start; % start[i]: start time for order i
array[Orders] of int: end;   % end[i]:   end time for order i
array[Orders] of int: price; % price[i]: price for order i
```

```
% o[i,j] = true iff orders i and j overlap
array[Orders,Orders] of bool: o = array2d(Orders, Orders,
 [start[i] <= end[j] /\ start[j] <= end[i] | i in Orders, j in Orders]);

% ============ Variables ============
array[Orders] of var 1..k+1: x; % x[i] = j if order i is assigned hall j
var int: total;                 % objective var: total profit

% ============ Constraints ============
% orders that overlap cannot be assigned to the same hall
constraint forall (i, j in Orders where i < j /\ o[i,j]) (
  forall (v in 1..k) (not(x[i] = v) \/ not(x[j] = v))
);

% objective function
constraint sum (i in Orders)
  (bool2int(not(x[i] = k+1)) * price[i]) >= total;

solve maximize total;
```

When flattening high level constraints into primitive constraints, intermediate variables might be added to represent the value of some expressions. For example, for the above model the flattening process will add:

```
array[Orders,1..k+1] of var bool: neq;
constraint forall (i in Orders, v in 1..k+1)
                  (neq[i,v] <-> not(x[i] = v));

array[Orders] of var 0..1: b;
constraint forall (i in Orders) (b[i] = bool2int(neq[i,k+1]));
```

thus introducing *intermediate* Boolean variables $neq[i, v]$ to represent $x[i] \neq v$ and *intermediate* 0-1 integer variables $b[i]$ to represent $x[i] \neq k + 1$.

Since intermediate variables are not present in the original model, they are not likely to be covered by any declarations about symmetries. This leads to an important problem when manipulating 1UIP nogoods. 1UIP nogoods often contain literals from such intermediate variables. However, if the symmetry was defined only for the variables in the original model, the solver will be unable to derive the symmetric versions of the 1UIP nogoods, since the symmetries do not apply to such literals.

There are several ways to handle this problem. First, we can modify the model to add any intermediate variables introduced by flattening or by the solver and, then, either ask the user or use an automatic inference system (such as [23]) to provide symmetry declarations that include those intermediate variables. Alternatively, we can modify the solver's conflict analysis algorithm to ensure all literals in the nogoods it derives only contain variables for which the symmetries are defined.

Example 9 Consider the model of the Concert Hall problem provided in Example 8. In this model, the values in the set $\{1, \ldots, k\}$ (but not $k+1$, which indicates an order that is not accepted) are interchangeable. There are also data-dependent symmetries: if orders i and j have the same start time, end time and price, in a given instance of the problem, then the variable permutation that swaps $x[i]$ and $x[j]$ is a solution symmetry of that instance.

Suppose we want to extend our symmetries to cover the intermediate variables $neq[i, v]$ and $b[i]$ introduced by the flattening of the model as described in Example 8. Any value symmetry σ permuting the values in $\{1, \ldots, k\}$ on $x[i]$ should also permute the variables $neq[i, v]$, since they depend on the value of $x[i]$. However, $b[i]$ remains invariant, since the truth value of $x[i] \neq k + 1$ would not be changed by σ. For example, swapping values of 1 and 2 on $x[i]$ should also swap $neq[i, 1]$ with $neq[i, 2]$ for each i, and leave the $b[i]$'s unchanged. Thus, this value symmetry becomes a variable/value symmetry (that simultaneously swaps the sequence of pairs $\langle x[i] \mapsto 1, neq[i, 1] \mapsto 0$, $neq[i, 1] \mapsto 1\rangle$ with the pairs $\langle x[i] \mapsto 2$, $neq[i, 2] \mapsto 0$, $neq[i, 2] \mapsto \mathit{false}\rangle$) when extended to the intermediate variables.

Consider now a variable symmetry σ that swaps $x[i]$ and $x[j]$. Clearly, if we swap $x[i]$ with $x[j]$, then we must also swap $neq[i, v]$ with $neq[j, v]$ for each v, and $b[i]$ with $b[j]$. Thus, this variable symmetry becomes a variable sequence symmetry (that simultaneously swaps the sequence of variables $\langle x[i], b[i]\rangle$ with $\langle x[j], b[j]\rangle$). □

Whenever it is not practical to ask the user to extend the known symmetries to cover intermediate variables, we modify the solver to prevent any literals on intermediate variables from appearing in the inferred nogoods. To achieve this, we first give to the solver the set of variables that are actually covered by the known symmetries and, importantly, require the search only to make decisions on the covered variables. All other variables are considered intermediate variables. We then modify the conflict analysis algorithm described in Section 2.1, by adding an additional step: if the nogood contains any literal l from an intermediate variable, we eliminate the nogood by resolving it with the explanation for that literal. This is guaranteed to be possible *if no decisions are made on intermediate variables* since, then, literals on intermediate variables cannot be decision literals and, therefore, must have an explanation. The conflict analysis will now terminate with a nogood free of any literals on intermediate variables, for which we know how to calculate its symmetric versions.

Example 10 Assume the original conflict analysis algorithm has derived the nogood $n \equiv \neg(b[1] = 1 \wedge x[2] = 1 \wedge x[3] = 2)$, which has literals on intermediate variable $b[1]$. The modified conflict analysis algorithm will eliminate literal $b[1] = 1$ by using its explanation $neq[1, k+1] \rightarrow (b[1] = 1)$, and then eliminate $neq[1, k+1]$ by using its explanation $(x[1] \neq k + 1) \rightarrow neq[1, k + 1]$. The resulting nogood is $n \equiv \neg(x[1] \neq k+1 \wedge x[2] = 1 \wedge x[3] = 2)$, where all literals are defined on variables covered by known symmetries. □

Requiring search decisions not to be made on intermediate variables is not a strong restriction, since most search strategies already do this. If the search strategy is specified on the model variables it will automatically meet this requirement. In fact, this restriction is clearly also a requirement for SBDS, even though we are unaware of any discussion of this issue in the literature. Note however, that searching on intermediate variables is sometimes highly useful, as it is the case for domwdeg search [3] or activity-based search [26]. For these searches, extending the symmetries to cover the intermediate variables might be quite useful.

4.3 Trading Completeness for Speed

A key requirement for effective symmetry breaking is to find the right balance between the overhead of the method and the pruning it provides. The original SBDS method [14] is complete, that is, the search is guaranteed to find a single representative of each group of symmetric search nodes (representing symmetric subproblems) and, thus, solutions. While this might be required by some applications, it can impose a very significant overhead to the search. Some variants of SBDS, such as the shortcut SBDS method described in the original SBDS paper [14], the STAB method [30], and the more recent Lightweight Dynamic Symmetry Breaking (LDSB) method [21], sacrifice completeness in order to reduce this overhead. All three methods disregard any symmetry that cannot immediately be used to prune something, that is, they disregard symmetry σ if $\sigma(DS) \neq DS$ (and it is not obvious that $DS \models \sigma(DS)$), where DS is the set of decisions taken to reach the current node. Such strategies can dramatically reduce the overhead associated with keeping track of the all symmetries while retaining most (sometimes all) of the pruning power of complete methods, thus often being faster than the complete methods.

Our implementation is based on the same ideas. We are only interested in generating symmetric nogoods that can immediately propagate and prune something. We also do not want to calculate exponentially many symmetric nogoods. Thus, given a 1UIP nogood $l_1 \wedge \ldots \wedge l_{k-1} \rightarrow \neg l_k$, we calculate its symmetric nogoods only using symmetries that are: 1) pairwise swaps, 2) map $\neg l_k$ to a literal that is not already true in the current domain, and 3) map every l_i to a literal that is already true in the current domain. Further, we compose symmetries in the same way as LDSB, that is, for each symmetric nogood we find, we apply all the generator symmetries to it again to see if something new can be found. We do this recursively until we can get no new pruning. Once a symmetric nogood is found, we handle it just as we do for the 1UIP nogoods derived by the solver, i.e., an activity count is kept for each nogood and it is incremented whenever the nogood is used. Periodically, when the number of nogoods becomes too high, the least active half of the nogoods are removed.

Our implementation of SBDS-1UIP is described by procedure sbds-1uip in Figure 6. The procedure receives a nogood $B_0 \rightarrow h_0$ of a given problem P, a generating set S of the symmetry group of P to be exploited, and the current domain D. We immediately propagate the nogood—since the body B_0 must be true in D, we simply need to add the head h_0 to D (line 2). The algorithm keeps track of newly discovered nogoods New, which initially contains only $B_0 \rightarrow h_0$ (line 3). For each new nogood $B \rightarrow h$ and each symmetry $\sigma \in S$, we create the symmetric version $B' \rightarrow h'$ (lines 7-8). If the symmetric body B' is true in D and the head h' is not, we add the symmetric version to the new nogoods and propagate it by adding h' to D (lines 9-11).

Note that a nogood cannot be added twice since its head will already be true in D. Note also that, since the sbds-1uip procedure disregards some nogoods inside the while loop, it might miss some symmetric nogoods that would have otherwise been obtained by composition.

Example 11 Consider the 1UIP nogood from Example 3: $x_2 \geq 2 \wedge x_3 \geq 2 \wedge x_4 \geq 2 \rightarrow x_2 \neq 2$ which, after replacing inequalities by disequalities, appears as $x_2 \neq$

```
sbds-1uip($B_0 \to h_0, S, D$)
    $D := D \cup \{h_0\}$
    $New := \{B_0 \to h_0\}$
    while ($\exists B \to h \in New$)
        $New := New \setminus \{B \to h\}$
        for $\sigma \in S$
            $h' := \sigma(h)$
            $B' := \{\sigma(l) \mid l \in B\}$
            if ($D \models B'$ and $D \not\models h'$)
                $D := D \cup \{h'\}$
                $New := New \cup \{B' \to h'\}$
```

Fig. 6 The SBDS-1UIP algorithm.

$1 \wedge x_3 \neq 1 \wedge x_4 \neq 1 \to x_2 \neq 2$. This nogood was inferred after failure was detected due to decisions $x_1 = 1$ and $x_2 = 2$. Then, the procedure examines the generating set $S \equiv \{\sigma_1, \sigma_2, \sigma_3\}$, where variable symmetry σ_1 swaps x_1 and x_2, σ_2 swaps x_2 and x_3, and σ_3 swaps x_3 and x_4 (which generates the symmetry group where these three variables are interchangeable). Symmetry σ_1 gives $x_1 \neq 1 \wedge x_3 \neq 1 \wedge x_4 \neq 1 \to x_1 \neq 2$, which is redundant since $x_1 \neq 2$ is already known to hold. Symmetry σ_2 gives $x_3 \neq 1 \wedge x_2 \neq 1 \wedge x_4 \neq 1 \to x_3 \neq 2$, which causes $x_3 \neq 2$ to be added to D and is added to New. Symmetry σ_3 gives $x_2 \neq 1 \wedge x_4 \neq 1 \wedge x_3 \neq 1 \to x_2 \neq 2$ which is also redundant. Applying the elements of the generating set to this discovered nogood only adds the nogood $x_4 \neq 1 \wedge x_2 \neq 1 \wedge x_3 \neq 1 \to x_4 \neq 2$ to New and $x_4 \neq 2$ to D. Applying the generating set to this nogood adds nothing to New and the process finishes. □

5 SBDS-1UIP vs SBDS

It is difficult to analyse the power of SBDS-1UIP, since it depends on the strength of the propagators used and the way in which they explain their propagations. However, we can show that under some fairly reasonable assumptions regarding the *monotonicity* and *symmetricity* properties of the solver's propagation engine, a complete SBDS-1UIP system (i.e., one that applies all symmetries in the problem to its nogoods) is at least as strong as the complete SBDS method. Intuitively, a propagation engine is monotonic if when given a smaller domain as input, it always returns a smaller domain as output. It is symmetric if when given a symmetric domain as input, it returns a symmetric domain as output. More formally:

Definition 1 The propagation engine *propfix*(C, D) is *monotonic* if whenever $D' \models D$, then *propfix*$(C, D') \models$ *propfix*(C, D).

Definition 2 Let S be a symmetry group of $P \equiv (V, D, C)$. The propagation engine *propfix*(C, D) is *symmetric* w.r.t. S if: $\forall \sigma \in S$ and $\forall l, d_1, \ldots, d_k \in lit(P)$ where $d_1, \ldots, d_k$ are decision literals, if *propfix*$(C, D \wedge \bigwedge_{j=1}^{k} d_j) \models l$, then *propfix*$(C, D \wedge \bigwedge_{j=1}^{k} \sigma(d_j)) \models \sigma(l)$.

Theorem 2 below tells us that, for the monotonic and symmetric propagation engines, the symmetric versions of the 1UIP nogoods are at least as powerful as the symmetric versions of the decision nogoods in terms of their ability to find failure in subproblems. In the next section, we show that SBDS-1UIP can be strictly stronger than SBDS for some problems.

Theorem 2 *Let S be a symmetry group of problem $P \equiv (V, D, C)$, $n_{dec} \equiv \neg(d_1 \wedge \cdots \wedge d_k)$ be the decision nogood derived from the failure of subproblem $P' \equiv (V, D, C \cup \{d_1, \ldots, d_k\})$, and $n_{1uip} \equiv \neg(l_1 \wedge \cdots \wedge l_m)$ be the 1UIP nogood derived from P'. If the propagation engine $propfix(C, D)$ is monotonic and symmetric for S, then for any $\sigma \in S$, and any fixpoint domain $D' = propfix(C, D')$, if D' violates $\sigma(n_{dec})$, then it also violates $\sigma(n_{1uip})$.*

Proof Suppose domain D' violates $\sigma(n_{dec})$, i.e., all the $\sigma(d_i)$ are true in D'. By assumption, the decisions $d_1, \ldots, d_k$ in subproblem P' were enough for the propagation engine to infer each of $l_1, \ldots, l_m$ and, therefore, $propfix(C, D \wedge \bigwedge_{j=1}^{k} d_j) \models l_i, 1 \leq i \leq m$. Hence, since the propagation engine is symmetric $propfix(C, D \wedge \bigwedge_{j=1}^{k} \sigma(d_j)) \models \sigma(l_i), 1 \leq i \leq m$. Since $D' \models D$ and, by assumption $D' \models \bigwedge_{j=1}^{k} \sigma(d_j)$, we have by monotonicity that $D' = propfix(C, D') \models propfix(C, D \wedge \bigwedge_{j=1}^{k} \sigma(d_j)) \models \sigma(l_i), 1 \leq i \leq m$. Hence, D' also violates $\sigma(n_{1uip})$. □

A sufficient (and very common) condition for the propagation engine to be monotonic is for all the individual propagators to be monotonic. A sufficient condition for the propagation engine to be symmetric w.r.t. a valuation symmetry group S is for every propagator to have a symmetric counterpart of the same propagation strength. That is, $\forall c \in C, \forall \sigma \in S$, there is a propagator for $\sigma(c)$ such that $\sigma(propfix(\{c\}, D)) = propfix(\{\sigma(c)\}, \sigma(D))$.

Example 12 Consider problem $P \equiv (\{x_1, x_2, x_3\}, D, \{x_1 \neq x_2, x_2 \neq x_3, x_3 \neq x_1\})$ where variables x_1, x_2, x_3 are interchangeable. Every permutation of these variables gives a valuation symmetry. If the propagators for all three constraints are of the same consistency level (e.g., all domain consistent, or all checking only), then the propagation engine is symmetric. If on the other hand, the propagator for $x_1 \neq x_2$ and $x_2 \neq x_3$ are domain consistent but the propagator for $x_3 \neq x_1$ is only checking, then the engine is not symmetric. For example, given $x_2 = 1$, the propagation engine will remove 1 from the domain of x_1 and x_3, but given $x_3 = 1$, it will only remove the value of 1 from x_2's domain.

Consider another problem $P \equiv (\{x_1, x_2, x_3\}, D, \{x_1 = x_2, x_2 = x_3\}$ where variables x_1, x_2, x_3 are again interchangeable. Every permutation of these variables gives a valuation symmetry. However, the propagation engine may not be symmetric, as the symmetry σ that swaps x_2 and x_3 maps $x_1 = x_2$ to $x_1 = x_3$, which is not directly implemented by a propagator. If the propagators for $x_1 = x_2$ and $x_2 = x_3$ are only checking, then the engine can detect that $x_1 = 1, x_2 = 2$ fails, but will not detect that $x_1 = 1, x_3 = 2$ fails.

Most of the symmetries in the problems studied in the literature are valuation symmetries. Indeed, this property may be either guaranteed by the method used to determine the symmetries of the problem (e.g. [10,22,2]), or automatically proved (e.g. [24]). It is also typical for solvers to use the same strength of propagation for constraints of the same type. Since symmetries often map constraints to constraints of the same type, propagation engines are often symmetric.

6 Difference in power between complete methods

As proved by [31], the search tree explored by SBDS for a given problem P with symmetry group S is a *GE-tree*. That is, a tree for which (a) no two nodes are symmetric according to S, and (b) the tree contains a representative of every solution of P (i.e., for every solution θ of P, either θ or a solution in its orbit under S appears in the tree). Therefore, it may seem surprising that SBDS-1UIP can do better, as shown by Example 6. This is because, in constructing the minimal GE-tree, SBDS will visit nodes that are the root of completely failed subtrees, even though it will never visit more than one symmetric version of each such node. SBDS-1UIP is able to visit fewer nodes that are the root of completely failed subtrees.

This is in fact due to the vastly different strengths of decision nogoods and 1UIP nogoods. A decision nogood is derived via exhaustive search and tells us that the subtree which we have just fully explored is failed. It does not allow us to prune any other subtree in the search tree. Thus, the symmetric versions of decision nogoods can only prune a subtree for which we have already explored a symmetric version. A 1UIP nogood on the other hand, is derived through *constraint resolution* and gives us a sufficient set of conditions for the failure to occur again anywhere else in the search. This sufficient set of conditions may occur in many other subtrees of the search tree, allowing the 1UIP nogood to prune these subtrees. Thus, the symmetric versions of a 1UIP nogood can correctly prune a subtree even though we may never have examined a symmetric version of it before. This occurs frequently in problems with local structure, where SBDS-1UIP is able to exploit a symmetry locally in the sub-component of a problem that is the actual cause of failure, even when the symmetry may be broken outside of that sub-component.

Example 13 Consider the graph coloring problem given in Example 6. The 1UIP nogood derived after the failure of search decisions $x_1 = 1, x_2 = 2, x_3 = 3, x_4 = 4, x_5 = 5, x_6 = 1$ is: $x_7 \neq 4 \wedge x_7 \neq 5 \wedge x_8 \neq 4 \wedge x_8 \neq 5 \wedge x_9 \neq 4 \wedge x_9 \neq 5 \rightarrow x_6 \neq 1$. Thus, SBDS-1UIP identified the sub-component that caused the failure as the one formed by the variables x_6, x_7, x_8, x_9, their current domains, and the constraints linking them. The value symmetries that interchange every two values apply to this nogood regardless of the fact that $x_1, \ldots, x_5$ were labeled to values that broke all these symmetries globally. And the symmetric versions of this 1UIP nogood can prune the search node reached by decisions $x_1 = 1, x_2 = 2, x_3 = 3, x_4 = 4, x_5 = 5, x_6 = 2$, even though the two valuations do not actually lie in the same orbit! Thus, the method is able to infer the failure of this search node despite the fact that we have never examined any of its symmetric counterparts before. □

The situation is similar when using static symmetry breaking with the standard lex-leader constraints, since as soon as all values have been (correctly) used in the past decisions (as it is the case in the above graph coloring example), the symmetry breaking constraint is satisfied and causes no more pruning. The advantage for SBDS-1UIP comes from the LCG solver being able to give us very precise information about which variables are involved in the conflict. This allows SBDS-1UIP to be strictly stronger than complete methods like SBDS, SBDD or lex-leader symmetry breaking constraints.

7 Breaking Almost Symmetries with SBDS-1UIP

While symmetries occur frequently in constraint problems, it is more common for constraint problems arising from real world situations to contain *almost symmetries*, that is, symmetries that hold for all but a few constraints in the problem. As mentioned before, such symmetries are not well behaved mathematically, which makes it difficult to adapt traditional symmetry breaking techniques like lex-leader constraints, SBDS or SBDD to exploit them. Previous work on almost symmetries has concentrated on exploiting them by first transforming the original problem P into a related problem P' where the symmetries are restored, and then applying the traditional symmetry breaking techniques to P' [7,16,17,20]. In this section we show that, with a slight modification, SBDS-1UIP can also be used to exploit almost symmetries.

Definition 3 An *almost symmetry* σ of $P \equiv (V, D, C)$ is a permutation on $pairs(P)$ for which $\exists H_\sigma \subseteq C$ s.t. σ is a solution symmetry of $P' \equiv (V, D, C \setminus H_\sigma)$. We call H_σ a *breaking set* of σ w.r.t. P.

Definition 4 An *almost symmetry group* of $P \equiv (V, D, C)$ is a set S of almost symmetries of P for which $\exists H_S \subseteq C$, s.t. S is a symmetry group of $P' \equiv (V, D, C \setminus H_S)$. We call H_S a *breaking set* of S w.r.t. P.

The intuition behind almost symmetries is that the set of constraints H_σ should be a relatively small subset of C, such that if H_σ were removed from the problem, σ would be a symmetry of the resulting problem. Each symmetry σ and symmetry group S of P is a special case of almost symmetries where the sets of constraints H_σ and H_S are empty. Given an almost symmetry σ, if H_σ is small relative to C, it is likely for an LCG solver to be able to exploit σ and prune significant parts of the search space. This is because in an LCG solver, each nogood n is derived from a certain subset of the constraints $m(n) \subseteq C$, i.e., $m(n) \wedge D \models n$. In particular, n is generated either by propagating constraint c, in which case $m(n) = \{c\}$, or by resolving two nogoods n_1 and n_2 together, in which case $m(n) = m(n_1) \cup m(n_2)$. As the following theorem states, if $m(n)$ is disjoint from H_σ, the symmetry can be exploited.

Theorem 3 *Given almost symmetry σ and nogood n of $P \equiv (V, D, C)$, if $m(n)$ and H_σ are disjoint, then $\sigma(n)$ is a valid nogood of P.*

Proof Consider the problem $P' \equiv (V, D, C \setminus H_\sigma)$. By definition, σ is a symmetry of P'. Also, since $C \setminus H_\sigma \models m(n)$ and $m(n) \models n$, we have that n is a nogood of P' and by Corollary 1 we have that $\sigma(n)$ is a correct nogood for P'. Since P can be obtained by simply adding constraints to P', $\sigma(n)$ is also a correct nogood for P. □

Theorem 3 tells us that in order to decide whether we can apply almost symmetry σ to nogood n to get another valid nogood, we simply need to know H_σ and $m(n)$.

Example 14 Consider the graph coloring problem introduced in Example 6. If two vertices (that is, variables) in a given graph have the same set of neighbors, then they are symmetric (a variable symmetry). While the set of neighbors is often not

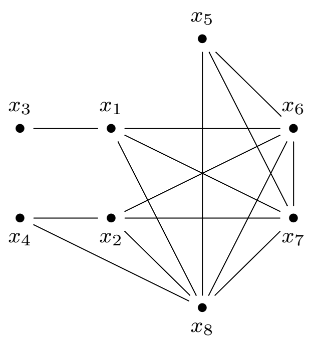

Fig. 7 A graph coloring problem where we can exploit almost symmetries.

the same, it is often similar. Consider variables x_1 and x_2 in the graph of Figure 7, where the number of colors is 4. Since the two variables have almost the same set of neighbors, we can exploit the almost symmetry σ which swaps x_1 and x_2 with $H_\sigma \equiv \{x_1 \neq x_3, x_2 \neq x_4\}$.

Suppose we try $x_5 = 1, x_1 = 2$. Propagation forces $x_6, x_7, x_8 \in \{3, 4\}$ which eventually fails and gives us the 1UIP nogood: $x_6 \neq 1 \wedge x_7 \neq 1 \wedge x_8 \neq 1 \rightarrow x_1 \neq 2$. The set of constraints used to derive this nogood is: $m(n) \equiv \{x_1 \neq x_6, x_1 \neq x_7, x_1 \neq x_8, x_5 \neq x_6, x_5 \neq x_7, x_5 \neq x_8, x_6 \neq x_7, x_6 \neq x_8, x_7 \neq x_8\}$. Clearly, $m(n) \cap H_\sigma \equiv \emptyset$ and, thus, Theorem 3 applies and the symmetric nogood $x_6 \neq 1 \wedge x_7 \neq 1 \wedge x_8 \neq 1 \rightarrow x_2 \neq 2$ is also valid, allowing us to immediately prune 2 from the domain of x_2. □

Given an almost symmetry group S, our implementation pre-computes H_σ for every $\sigma \in S$ and uses a bit-vector for each nogood n to keep track of its $m(n)$, where each bit in the bit-vector represents a constraint c that appears in some H_σ for some almost symmetry $\sigma \in S$. The bit-vectors tend to be short, since $|H_\sigma|$ is typically small as, otherwise, the symmetry is too broken for almost symmetry techniques to improve execution in any case. If n was generated by propagating constraint c, $m(n)$ is set to $\{c\}$ by setting the appropriate bit in the bit-vector to one. If n was generated by resolving two nogoods n_1 and n_2, $m(n)$ is set to $m(n_1) \cup m(n_2)$ by bitwise joining the bit-vectors of n_1 and n_2. Finally, we also need to keep track of $m(n)$ for nogoods n that are generated as symmetric versions $n = \sigma(n')$ of a previously calculated nogood n'. We assume an algorithm $\mathsf{map}(\sigma, m)$ that returns a set of constraints c' such that if $m \models n$, then $c' \models \sigma(n)$. For valuation symmetries we can define $\mathsf{map}(\sigma, m) = \{\sigma(c) \mid c \in m(n)\}$, assuming that $\sigma(c)$ is a constraint appearing in the problem. Note that we can always define $\mathsf{map}(\sigma, m) = H_S$, which is safe but will hamper the effectiveness of almost symmetries.

```
almost-sbds-1uip($B_0 \to h_0, m, S, D$)
    $D := D \cup \{h_0\}$
    $New := \{(B_0 \to h_0, m)\}$
    while ($\exists (B \to h, m) \in New$)
        $New := New \setminus \{(B \to h, m)\}$
        for $\sigma \in S$
            if $H_\sigma \cap m \neq \emptyset$ continue
            $h' := \sigma(h)$
            $B' := \{\sigma(l) \mid l \in B\}$
            if ($D \models B' \wedge \neg(D \models h')$)
                $D := D \cup \{h'\}$
                $New := New \cup \{(B' \to h', \mathsf{map}(\sigma, m)\}$
```

Fig. 8 The SBDS-1UIP algorithm for almost symmetries.

The revised SBDS-1UIP procedure almost-sbds-1uip is shown in Figure 8, where we assume the original call has argument $m = m(n)$ for initial nogood $n \equiv B_0 \to h_0$. Note that, as for full symmetries, in our implementation S is not the full symmetry group and, instead, it only contains a small set of symmetries (the pairwise swaps) that is recursively composed. The main two differences with the algorithm of Figure 6 are (a) that every element in New is a tuple containing not only a nogood but also its associated set of constraints m, and (b) that symmetric versions of a nogood n are only considered for symmetries whose breaking set H_σ is disjoint with $m(n)$ (line 7).

8 Related Work

The theoretical foundations of how symmetric nogoods can be used to exploit symmetries in propositional formulas was first given in [18]. Two approaches were proposed there: SR-I which infers symmetric nogoods from global symmetries; and SR-II which infers symmetric nogoods from local symmetries. These approaches are analogous to those given by Theorem 1 and Theorem 3, but restricted to Boolean variables and clausal constraints. No implementation or experimental evaluation of the proposed system were given in this work. Dynamic symmetry breaking techniques such as SBDS and SBDD can be considered implementations of the extension of SR-I to the more general class of constraint problems involving finite domain variables and arbitrary constraints. However, since they are only capable of using decision nogoods, they lose much of the power afforded by SR-I and they are incapable of using SR-II. In this paper, we produce a practical implementation of the SR-I and SR-II approaches for general constraint problems by combining LCG, which allows for the use of resolution to derive new nogoods, with SBDS.

As mentioned before, while there have been several other theoretical works on exploiting almost symmetries in constraint problems [16,17], we are only aware of one previous implementation of a general method for exploiting almost symmetries [20]. The general idea to exploit the almost symmetry group S is to decompose the problem $P \equiv (V, D, C)$ into two parts: P_1 which contains $C \setminus H_S$ and is fully symmetric, and P_2 which contains H_S. Traditional symmetry breaking methods can be applied to P_1 and any solutions can be extended to P_2. In [20] the authors propose a clever variant of this general idea where, rather than treating P_1 and P_2

as separate problems, an additional channeling constraint merges them into one fully symmetric problem P'. Traditional symmetry breaking methods are then applied to P', and each solution of P' converted to a solution of P.

Let $V_B = \mathit{vars}(C \setminus H_S)$, $V_H = \mathit{vars}(H_S)$, $V_{BH} = V_B \cap V_H$, and $V_{H\setminus B} = V_H \setminus V_B$. P_1 and P_2 are obtained from P as follows: $P_1 \equiv (V_B, D_{V_B}, C \setminus H_S)$ and $P_2 \equiv (V'_{BH} \cup V_{H\setminus B}, D_{V'_{BH} \cup V_{H\setminus B}}, H'_S)$, where V'_{BH} contains a copy of each variable in V_{BH}, and H'_S is formed by replacing in every $c \in H_S$ any variable in V_{BH} with its shadow copy in V'_{BH}. Note that S is now a full symmetry group on P_1.

Example 15 Consider a simple graph coloring problem $P \equiv (V, D, C)$ with domains: $x_1, x_2, x_3 \in \{1, 2, 3\}$, and constraints: $C \equiv \{x_1 \neq x_2, x_2 \neq x_3, x_3 \neq x_1, x_1 \neq 1, x_2 \neq 2\}$. Consider the symmetry group S formed by any permutation of the values in $\{1,2,3\}$. We have $H_S \equiv \{x_1 \neq 1, x_2 \neq 2\}$, $V_B = \{x_1, x_2, x_3\}$, $V_H = \{x_1, x_2\}$, $V_{BH} = \{x_1, x_2\}$ and $V_{H\setminus B} = \emptyset$. We create shadow variables $V'_{BH} = \{x'_1, x'_2\}$ yielding $P_1 \equiv (\{x_1, x_2, x_3\}, x_1, x_2, x_3 \in \{1, 2, 3\}, \{x_1 \neq x_2, x_2 \neq x_3, x_3 \neq x_1\})$ where S holds, and $P_2 \equiv (\{x'_1, x'_2\}, x'_1, x'_2 \in \{1, 2, 3\}, \{x'_1 \neq 1, x'_2 \neq 2\})$. □

P_1 and P_2 are then merged into one problem P' to which we add a symmetry variable sv whose domain is S, and a channeling constraint Q such that $\theta \in \mathit{solns}(Q)$ iff: $\forall \sigma \in S$, $\forall x \in V_2$, $\forall x' \in V'_2$ where x' is the shadow version of x, we have $(\theta(sv) = \sigma) \rightarrow \theta(x') = \sigma(\theta)(x)$. That is, the channeling constraint forces V'_{BH} to take on the values that V_{BH} would take if we had applied σ to the valuation of V_B. The new problem P' is fully symmetric. Every solution θ of P corresponds to an orbit of solutions in P' and, conversely, every solution θ' of P' corresponds to a solution θ of P.

Example 16 In the problem from Example 15, we add new variable sv and domain constraint $sv \in S$ and we post the constraint Q given by $(sv = \sigma) \rightarrow (x'_1 = \sigma(1) \wedge x'_2 = \sigma(2))$. Thus, for the solution $x_1 = 1, x_2 = 2, x_3 = 3$ and sv equal to the symmetry that swaps 1 and 3 leaving other values unchanged, denoted $(1 \leftrightarrow 3)$, Q forces $x'_1 = 3, x'_2 = 2$.

The solution of P given by $x_1 = 2, x_2 = 1, x_3 = 3$ corresponds to the following solutions of P': $\{x_1 = 2, x_2 = 1, x_3 = 3, sv = id, x'_1 = 2, x'_2 = 1\}, \{x_1 = 2, x_2 = 3, x_3 = 1, sv = (2 \leftrightarrow 3), x'_1 = 2, x'_2 = 1\}, \ldots, \{x_1 = 3, x_2 = 1, x_3 = 2, sv = (1 \leftrightarrow 3), x'_1 = 2, x_2 = 1\}$, where id is the identity and $v_1 \leftrightarrow v_2$ represents a symmetry that swaps values v_1 and v_2, leaving the rest unchanged.

Conversely, suppose θ' is a solution of P' for which $\theta'(sv) = \sigma$, then θ defined as $\forall x \in V_B$, $\theta(x) = \sigma(\theta')(x)$ and $\forall x \in V_3$, $\theta(x) = \theta'(x)$, is a solution of P. In our example, $x_1 = 1, x_2 = 2, x_3 = 3, sv = \sigma, x'_1 = 3, x'_2 = 1$, where σ maps 1 to 3, 2 to 1, and 3 to 2, is a solution of P'. The corresponding solution of P is $x_1 = 3, x_2 = 1, x_3 = 2$. □

In our experiments we will break symmetries in P' using SBDS-1UIP with this remodeling method and compare these results to those obtained by breaking almost symmetries directly with our method.

9 Experiments

9.1 Exploiting Symmetries

We now provide experimental evidence that SBDS-1UIP can be much stronger than SBDS on some problems, and that symmetry breaking can work well with lazy clause generation. To do this we extended CHUFFED, which is a state of the art LCG solver, with three different versions of SBDS. The first version (denoted by dec in the tables) only uses symmetric versions of decision nogoods (in addition to the 1UIP itself, of course, as all versions implemented within CHUFFED). The second version (1UIP) only uses symmetric versions of 1UIP nogoods. And the third version (crippled) only uses symmetric versions of 1UIP nogoods whose associated symmetric decision nogood would have caused pruning, that is, posts $\sigma(n_{1uip})$ only if $\sigma(n_{dec})$ can prune something. This is to show that the symmetries that only SBDS-1UIP can exploit can give additional speedup. We compare against CHUFFED with no symmetry breaking (none) and with standard lex-leader symmetry breaking constraints (static). Finally, we compare against the Gecode implementation of LDSB (LDSB), which gives the fastest execution with dynamic symmetry breaking on the first two problems we examine, beating GAP-SBDS and GAP-SBDD by significant margins. Note that all methods run are incomplete, in the sense that they might only obtain a subset of all possible symmetric nogoods. All versions of CHUFFED are run on Xeon Pro 2.4GHz processors. The results for LDSB were run on a Core i7 920 2.67 GHz processor.

The first two problems we examine are Concert Hall Scheduling and Graph Coloring, introduced in Example 8 and Example 6, respectively. These problems have a distributed constraint structure, that is, their failures are often only the result of the interaction of a few local constraints. Hence, they tend to generate small local 1UIP nogoods. The instances are randomly generated following [19] to be highly symmetrical. The problems are directly defined in C++ for CHUFFED and Gecode for LDSB, and they do not require temporary variables (they are available at `http://www.cmears.id.au/symmetry/symcache.tar.gz`). For the Concert Hall problem all instances have 8 identical halls and N orders are generated in partitions of uniformly distributed size, with all orders in a partition being identical (i.e., having the same start time, end time and price). Thus, the 8 halls (values) are interchangeable and all orders within a partition (variables) are also interchangeable. There are 20 randomly generated instances per N. For the Graph Coloring problem, N nodes are partitioned so that each partition (of maximum size 8) is either an independent set or a complete graph, and each subgraph with nodes in two partitions is either an independent or a complete bipartite graph. Instances are divided into classes according to how the partition sizes are distributed ("uniform" or "biased"). Again, there are 20 instances of each class. Colors (values) are interchangeable, as are nodes (variables) within each partition. Our tables group the 20 instances by size N, so that the times and number of fails displayed in the tables are the mean run times and fails for the instances of each size. A timeout of 600 seconds per instance was used.

The results are shown in Tables 1 and 2. LDSB and dec fail to solve a few instances. 1UIP, crippled and static all solve every instance in the benchmarks. In fact, the set of instances for Graph Coloring, which is of an appropriate size for normal CP solvers, is a bit too easy for lazy clause solvers such as CHUFFED, which

Table 1 Comparison of three SBDS implementations in Chuffed, static symmetry breaking in Chuffed, and LDSB in Gecode, on the Concert Hall Scheduling problem

	Chuffed										Gecode	
N	none		1UIP		crippled		dec		static		LDSB	
	Time	Fails	Time	Fails	Time	Fails	Time	Fails	Time	Fails	Time	Fails
20	259.8	686018	**0.04**	**84**	0.05	130	0.07	350	0.05	134	0.01	496
22	381.5	749462	**0.07**	**181**	0.08	299	0.17	1207	**0.07**	183	**0.07**	3285
24	576.9	1438509	0.10	**275**	0.11	316	0.78	3426	0.15	486	**0.04**	913
26	483.4	1189930	**0.10**	**282**	0.19	677	2.26	5605	0.25	685	0.11	2895
28	530.7	1282797	0.68	1611	1.12	2613	3.64	10530	**0.42**	**1041**	0.83	21738
30	581.3	1251980	**0.27**	**761**	0.53	2042	19.52	48474	0.52	2300	20.19	530100
32	542.4	936019	**0.40**	**1522**	1.01	4845	21.48	65157	1.31	5712	1.27	49382
34	600.0	1039051	**1.10**	**2636**	3.22	8761	19.86	48837	1.60	4406	43.79	2128875
36	600.0	1223864	**1.40**	**3156**	5.02	13606	59.70	131142	2.37	5707	10.25	275917
38	600.0	1027778	**1.91**	**5053**	12.56	26556	82.77	178170	3.51	10518	37.37	564961
40	600.0	1447604	**2.96**	**6648**	10.27	27028	102.1	219454	6.40	18169	13.07	164685

is clear from the run times. Note that dec cannot compete with LDSB on Concert Hall Scheduling because LDSB is more aggressive in generating symmetric nogoods arising from compositions of generators.

Comparison between dec and 1UIP shows that posting symmetric 1UIP nogoods is better than symmetric decision nogoods. Comparison between crippled and 1UIP shows that the additional symmetries that we can only exploit with SBDS-1UIP indeed gives us reduced search and additional speedup. Comparison with static shows that dynamic symmetry breaking can be superior to static symmetry breaking on appropriate problems. The comparison with LDSB shows that lazy clause solvers can be much faster than normal CP solvers, and that they retain this advantage when integrated with symmetry breaking methods. It also shows (by proxy) that SBDS-1UIP is superior to GAP-SBDS or GAP-SBDD on these problems.

The total speed difference between 1UIP and dec is up to 2 orders of magnitude for the Concert Hall problems and up to 4 orders of magnitude for the Graph Coloring problems. Most of this speedup can be explained by the dramatic reduction in search space, which is apparent from the node counts in the results table. The redundancies exploited by lazy clause solvers are different from those redundancies caused by symmetries, and it is very clear here that by exploiting both at the same time with SBDS-1UIP, we get much higher speedups than possible with either of them separately. It should also be noted that Chuffed with static symmetry breaking (static) is reasonably fast (only about 2 times slower for Concert Hall) even though it cannot exploit the extra redundancies that SBDS-1UIP can, it has very low overhead (as shown by Graph Coloring) and integrates well with lazy clause generation.

Graph Coloring and Concert Hall Scheduling are problems with a structure that exposes the advantages of SBDS-1UIP. We also consider a more standard set of problems with symmetries. The results, presented in Table 3, show that once we have learning, static symmetry breaking is highly competitive. The advantage of 1UIP over dec arises from taking advantage of locality in the constraint graph. For problems where every variable is tightly constrained with every other variable it has no advantage over dec and, hence, it is also usually surpassed by static

Table 2 Comparison of three SBDS implementations in Chuffed, static symmetry breaking in Chuffed, and LDSB in Gecode, on the Graph Coloring problems

Uniform

	Chuffed										Gecode	
Size	none		1UIP		crippled		dec		static		LDSB	
	Time	Fails	Time	Fails	Time	Fails	Time	Fails	Time	Fails	Time	Fails
30	140.7	282974	**0.00**	**14**	0.06	474	0.26	3049	0.02	277	0.32	51331
32	211.4	390392	**0.00**	**17**	**0.00**	146	0.24	3677	**0.00**	84	0.16	14324
34	213.9	272772	**0.00**	**25**	0.29	1182	3.53	11975	0.03	433	4.26	629905
36	195.9	296358	**0.00**	**36**	0.04	467	6.91	23842	0.01	200	3.99	446275
38	224.0	297138	**0.00**	**55**	0.04	516	23.55	69480	0.03	526	7.42	912076
40	250.9	423326	**0.00**	**83**	0.31	1879	21.07	78918	0.06	878	21.24	2605242

Biased

	Chuffed										Gecode	
Size	none		1UIP		crippled		dec		static		LDSB	
	Time	Fails	Time	Fails	Time	Fails	Time	Fails	Time	Fails	Time	Fails
20	13.25	39551	**0.00**	**27**	**0.00**	32	0.01	639	**0.00**	29	**0.00**	65
22	11.53	63984	**0.00**	**25**	**0.00**	34	0.02	727	**0.00**	25	**0.00**	301
24	66.60	154409	**0.00**	**35**	**0.00**	47	0.07	1992	**0.00**	32	0.02	2149
26	74.77	277290	**0.00**	**55**	**0.00**	93	0.12	3385	**0.00**	104	0.02	4117
28	130.5	280649	**0.00**	**62**	**0.00**	84	0.58	6402	**0.00**	103	0.62	137965
30	267.6	480195	**0.00**	**101**	0.01	239	10.48	43835	0.01	359	18.47	2562808
32	331.7	600772	**0.01**	**232**	0.24	1597	9.98	44216	0.16	1864	29.98	3746913
34	219.9	387213	**0.20**	**806**	0.40	1946	10.26	47470	0.45	1730	54.31	8365913
36	442.6	709888	**0.01**	**317**	0.04	857	27.39	113252	0.80	3226	43.93	5488843
38	382.5	631403	**0.10**	**798**	1.01	5569	31.63	138787	4.12	9413	76.00	7861140
40	465.6	531285	**0.02**	**410**	0.36	2660	24.68	91847	0.12	2133	52.05	7413304

symmetry breaking. Note that on these examples dec and 1UIP always find exactly the same set of solutions, but sometimes static symmetry breaking finds less (since dec and 1UIP are incomplete).

9.2 Exploiting Almost Symmetries

We have extended Chuffed to support almost symmetries as described in Section 7. We compare four methods for handling almost symmetries, all of them running under Chuffed: 1) ignore all almost symmetries, but exploit full symmetries if any exist using standard lex-leader symmetry breaking constraints (denoted by Lex-FullOnly in the tables); 2) use SBDS-1UIP to exploit both types of symmetries (1UIP-All); and use the remodeling method as described in Section 8 to exploit both types of symmetries using either 3) lex-leader symmetry breaking (Remodel-Lex-ALL) or 4) SBDS-1UIP (Remodel-1UIP-All). We use the following four problems (all models and instances are available at `http://www.csse.unimelb.edu.au/~pjs/almostsym/`).

Black Hole Problem The Black Hole Problem [12] seeks a solution to the Black Hole patience game. In this game the 52 playing cards (of a standard 52-card deck) are laid out in 17 piles of 3, with the ace of spades starting in a "blackhole". Each turn, a card at the top of a pile can be played into the blackhole if it is ±1 from the card that was played previously, with king wrapping back around to ace. The aim is to play all 52 cards. This problem has variable almost symmetries since

Table 3 Comparison of SBDS, SBDS-1UIP and static symmetry breaking implementations in CHUFFED on a variety of problems

	1UIP		dec		static	
Benchmark	Time	Fails	Time	Fails	Time	Fails
bibd [7, 7, 3, 3, 1]	**0**	**13**	**0**	**13**	0	17
bibd [12, 12, 6, 6, 4]	**0.04**	**389**	**0.04**	398	0.1	1777
bibd [13, 13, 6, 6, 4]	**0.04**	**391**	**0.04**	400	0.17	2528
bibd [8, 14, 4, 7, 3]	0.13	908	0.08	915	**0.06**	**533**
bibd [9, 12, 3, 4, 1]	0.01	**39**	**0**	**39**	0.01	72
bibd [11, 5, 2]	0.03	196	0.02	196	**0.01**	**65**
bibd [16, 6, 2]	22.11	111980	18.69	111993	**3.71**	**29767**
bibd [11, 6, 3]	0.08	1094	0.07	1094	**0.01**	**92**
bibd [9, 6, 5]	0.05	787	0.05	800	**0**	**69**
graceful [3, 3]	1.17	32583	1.12	32690	**0.94**	**31562**
graceful [4, 2]	0.39	10126	0.39	10558	**0.27**	**8180**
latin [4]	**0**	11	**0**	11	**0**	**4**
latin [5]	**0**	119	**0**	119	**0**	**57**
latin [6]	0.41	17098	0.37	17101	**0.17**	**9426**
magic square [3]	**0**	8	**0**	8	**0**	**6**
magic square [4]	2.12	38677	2.09	38391	**1.95**	**37252**
nn_queens [7]	**0.01**	177	**0.01**	185	**0.01**	**172**
nn_queens [8]	0.11	2071	0.11	2061	**0.09**	**2054**
queens [12]	1.97	30266	1.92	**30236**	**1.82**	30254
queens [13]	27.74	131564	26.97	131221	**26.17**	**130819**
steiner [7]	**0**	**13**	**0**	**13**	0	19
steiner [9]	0.03	597	0.02	669	**0.01**	**160**

each card of the same value is symmetric with others of the same value except for the ordering constraints from the piles. We use 100 random instances.

Graph Coloring As discussed in Example 14, almost variable symmetries arise in this problem when pairs of vertices have almost the same set of neighbors. We construct instances as follows. We first randomly partition 50 vertices into 8 sets. For each pair of sets S_i, S_j, we decide with 0.5 probability whether they are densely connected or sparely connected. If dense, then for each pair of vertices $v_1 \in S_i$, $v_2 \in S_j$, we add an edge between them with probability 0.99. If sparse, then for each pair of vertices $v_1 \in S_i$, $v_2 \in S_j$, we add an edge between them with probability 0.01. As a result, pairs of vertices in the same set $v_1, v_2 \in S_i$ typically have very similar sets of neighbors, with the number of difference varying between about 0 to 3. Roughly 40% of the variable pairs are symmetric (0 difference) while the rest are almost symmetric (> 0 differences), resulting in a mixture of variable symmetries and variable almost symmetries. There is also the standard symmetry group in the problem created by value interchangeability.

Quasi-Group Completion The Quasi-Group Completion Problem [15] is an instance of the Latin Squares Problem with side constraints of the form $x_i = v$. While QCP typically has no symmetries, the Latin Square Problem has many row/column symmetries and value symmetries. We can model these underlying symmetries in QCP as almost symmetries. We try 100 instances of $n = 8$ with 25% of the squares pre-filled and find all solutions of each instance.

Table 4 Comparison between ignoring almost symmetries and breaking full with lex(Lex-FullOnly), breaking all with SBDS-1UIP (1UIP), or breaking all by remodeling using lex (Remodel-Lex) or using SBDS-1UIP (Remodel-1UIP)

Problem	Lex-FullOnly		1UIP		Remodel-Lex		Remodel-1UIP	
	Time	Fails	Time	Fails	Time	Fails	Time	Fails
BlackHole	45.8	50427	21.8	24380	**0.1**	**303**	25.7	33481
GraphColor	35.2	73133	**0.5**	**1488**	51.4	179837	129.2	598644
QCP	252.5	4392785	**17.7**	**279118**	300	2497296	300	1137630
ConcertHall	257.1	600190	224.7	463213	11.9	24310	**1.1**	**1866**

Concert Hall Problem As shown before, the Concert Hall Problem of Example 8 has value interchangeability for the k identical concert halls. However, in more realistic versions of this problem the halls may vary in size, equipment, location, etc. As a result, it is possible that for each concert only a certain subset of the halls are acceptable, corresponding to side constraints of form $x_i \neq v$. This creates almost value symmetries and almost variable symmetries. We use randomly generated instances following [19], with an additional 50 random side constraints of the form $x_i \neq v$.

The experiments were run on Xeon Pro 2.4GHz processors with a timeout of 300 seconds. Table 4 shows the average fails and times on each set of instances, with timeouts counting as 300 seconds. It can be seen from Table 4 that all three methods for almost symmetries can produce significant speedups on at least some of the problems compared to the baseline of Lex-FullOnly. SBDS-1UIP is the fastest on Graph Coloring and QCP, while Remodel-Lex is the fastest on Black Hole and Remodel-1UIP is the fastest on Concert Hall. The different techniques are effective on different problems. Their effectiveness appears to be very sensitive to the structure of the problem and the types of constraints in them, so that one method that performs well on one problem may perform very poorly on a different problem.

SBDS-1UIP tends to be effective if the constraints in H_S are not often directly involved in the conflict. This happens, for example, whenever those constraints are weak constraints (Graph Coloring), become satisfied by the partial assignment, or have already propagated at a higher decision level (QCP). This is the only method that could get any speedup on QCP. The remodeling method tends to be effective if the constraints in H_S are weak since, then, not too much information is lost when the set of constraints is split into two. This is true for Black Hole and Concert Hall, where we get very good speedups. On the other hand, if H_S contains strong constraints that prune many solutions of $C \setminus H_S$, then the remodeling severely weakens the model, potentially significantly increasing the run time (Graph Coloring, QCP).

10 Conclusion

We have shown how LCG solvers can be extended to post symmetric versions of their 1UIP nogoods. We proved that SBDS-1UIP is at least as strong as SBDS, and showed that it can be strictly stronger on some problems. In particular, it is capable of exploiting symmetries in the sub-component of a subproblem which actually caused the failure. This kind of local symmetry is very difficult to exploit and we are not aware of any other general symmetry breaking method that can exploit it. To implement SBDS-1UIP, we had to solve a number of new issues, such as how to map disequality literals, inequality literals, and literals from intermediate variables under the symmetries. Our experimental evaluation showed that SBDS-1UIP can be orders of magnitude faster than the original SBDS on appropriate problems. We also defined almost symmetries and showed how they can be exploited using a modified version of SBDS-1UIP. Our experimental evaluation showed that SBDS-1UIP can give significant speedup on problems with almost symmetries, and can be much faster than the previous almost symmetry breaking method.

Acknowledgments

NICTA is funded by the Australian Government through the Department of Communications and the Australian Research Council through the ICT Centre of Excellence Program. This work was partially supported by Asian Office of Aerospace Research and Development 12-4056.

References

1. R. Backofen and S. Will. Excluding symmetries in constraint-based search. In J. Jaffar, editor, *Proceedings of the 5th International Conference on Principles and Practice of Constraint Programming*, volume 1713 of *Lecture Notes in Computer Science*, pages 73–87. Springer, 1999.
2. B. Benhamou. Study of symmetry in constraint satisfaction problems. In *PPCP'94: Second International Workshop on Principles and Practice of Constraint Programming*, pages 246–254, 1994.
3. F. Boussemart, F. Hemery, C. Lecoutre, and L. Sais. Boosting systematic search by weighting constraints. In *Proceedings of the European Conference on Artificial Intelligence*, pages 146–150, 2004.
4. G. Chu, M. Garcia de la Banda, C. Mears, and P. Stuckey. Symmetries and lazy clause generation. In *Proceedings of the 22nd International Joint Conference on Artificial Intelligence*, pages 516–521, 2011.
5. D. A. Cohen, P. Jeavons, C. Jefferson, K. E. Petrie, and B. M. Smith. Symmetry definitions for constraint satisfaction problems. *Constraints*, 11(2-3):115–137, 2006.
6. J. M. Crawford, M. L. Ginsberg, E. M. Luks, and A. Roy. Symmetry-Breaking Predicates for Search Problems. In *Proceedings of the 5th International Conference on Principles of Knowledge Representation and Reasoning*, pages 148–159. Morgan Kaufmann, 1996.
7. A. Donaldson. Partial Symmetry in Model Checking. In *SymNet Workshop on Almost-Symmetry in Search*, pages 17–21, 2005. `http://www.allydonaldson.co.uk/edited_volumes/SymNet2005.pdf`.
8. T. Fahle, S. Schamberger, and M. Sellmann. Symmetry Breaking. In T. Walsh, editor, *Proceedings of the 7th International Conference on Principles and Practice of Constraint Programming*, volume 2239 of *Lecture Notes in Computer Science*, pages 93–107. Springer, 2001.
9. T. Feydy and P. J. Stuckey. Lazy Clause Generation Reengineered. In I. P. Gent, editor, *Proceedings of the 15th International Conference on Principles and Practice of Constraint Programming*, volume 5732 of *Lecture Notes in Computer Science*, pages 352–366. Springer, 2009.
10. A. Frisch, I. Miguel, and T. Walsh. CGRASS: A system for transforming constraint satisfaction problems. In *Recent Advances in Constraints, Joint ERCIM/CologNet International Workshop on Constraint Solving and Constraint Logic Programming*, pages 15–30, 2003.

11. A. Gargani and P. Refalo. An Efficient Model and Strategy for the Steel Mill Slab Design Problem. In C. Bessiere, editor, *Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming*, volume 4741 of *Lecture Notes in Computer Science*, pages 77–89. Springer, 2007.
12. I. P. Gent, C. Jefferson, T. Kelsey, I. Lynce, I. Miguel, P. Nightingale, B. M. Smith, and A. Tarim. Search in the patience game 'Black Hole'. *AI Commun.*, 20(3):211–226, 2007.
13. I. P. Gent, T. Kelsey, S. Linton, I. McDonald, I. Miguel, and B. M. Smith. Conditional Symmetry Breaking. In P. van Beek, editor, *Proceedings of the 11th International Conference on Principles and Practice of Constraint Programming*, volume 3709 of *Lecture Notes in Computer Science*, pages 256–270. Springer, 2005.
14. I. P. Gent and B. M. Smith. Symmetry Breaking in Constraint Programming. In W. Horn, editor, *Proceedings of the 14th European Conference on Artificial Intelligence*, pages 599–603. IOS Press, 2000.
15. C. Gomes and D. B. Shmoys. Completing Quasigroups or Latin Squares: A structured Graph Colouring Problem. In D. S. Johnson, A. Mehrotra, and M. Trick, editors, *Proceedings of the Computational Symposium on Graph Colouring and Extensions*, pages 22–39, 2002.
16. P. Gregory. Almost-Symmetry in Planning. In *SymNet Workshop on Almost-Symmetry in Search*, pages 14–16, 2005. `http://www.allydonaldson.co.uk/edited_volumes/SymNet2005.pdf`.
17. W. Harvey. Symmetric Relaxation Techniques for Constraint Programming. In *SymNet Workshop on Almost-Symmetry in Search*, pages 50–59, 2005. `http://www.allydonaldson.co.uk/edited_volumes/SymNet2005.pdf`.
18. B. Krishnamurthy. Short proofs for tricky formulas. *Acta Inf.*, 22(3):253–275, 1985.
19. Y. C. Law, J. H. M. Lee, T. Walsh, and J. Y. K. Yip. Breaking symmetry of interchangeable variables and values. In C. Bessiere, editor, *Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming*, volume 4741 of *Lecture Notes in Computer Science*, pages 423–437. Springer, 2007.
20. R. Martin. The Challenge of Exploiting Weak Symmetries. In B. Hnich, M. Carlsson, F. Fages, and F. Rossi, editors, *Proceedings of the International Workshop on Constraint Solving and Constraint Logic Programming*, volume 3978 of *Lecture Notes in Computer Science*, pages 149–163. Springer, 2005.
21. C. Mears. *Automatic Symmetry Detection and Dynamic Symmetry Breaking for Constraint Programming.* PhD thesis, Clayton School of Information Technology, Monash University, 2009.
22. C. Mears, M. Garcia de la Banda, and M. Wallace. On implementing symmetry detection. *Constraints*, 14(4):443–477, 2009.
23. C. Mears, M. Garcia de la Banda, M. Wallace, and B. Demoen. A novel approach for detecting symmetries in CSP models. In L. Perron and M. Trick, editors, *Proceedings of the 8th International Conference on Integration of Artificial Intelligence (AI) and Operations Research (OR) techniques in Constraint Programming*, volume 5015 of *Lecture Notes in Computer Science*, pages 158–172. Springer, 2008.
24. C. Mears, T. Niven, M. Jackson, and M. Wallace. Proving symmetries by model transformation. In J. Lee, editor, *Proceedings of the 17th International Conference on Principles and Practice of Constraint Programming*, volume 6876 of *Lecture Notes in Computer Science*, pages 591–605. Springer, 2011.
25. P. Meseguer and C. Torras. Exploiting symmetries within constraint satisfaction search. *Artif. Intell.*, 129(1-2):133–163, 2001.
26. M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of the 38th Design Automation Conference*, pages 530–535. ACM, 2001.
27. N. Nethercote, P. Stuckey, R. Becket, S. Brand, G. Duck, and G. Tack. Minizinc: Towards a standard CP modelling language. In C. Bessiere, editor, *Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming*, volume 4741 of *Lecture Notes in Computer Science*, pages 529–543. Springer-Verlag, 2007.
28. O. Ohrimenko, P. J. Stuckey, and M. Codish. Propagation = Lazy Clause Generation. In C. Bessiere, editor, *Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming*, volume 4741 of *Lecture Notes in Computer Science*, pages 544–558. Springer, 2007.
29. J.-F. Puget. Symmetry breaking revisited. In P. Van Hentenryck, editor, *Proceedings of the 8th International Conference on Principles and Practice of Constraint Programming*, volume 2470 of *Lecture Notes in Computer Science*, pages 446–461. Springer, 2002.

30. J.-F. Puget. Symmetry breaking using stabilizers. In F. Rossi, editor, *Proceedings of the 9th International Conference on Principles and Practice of Constraint Programming*, volume 2470 of *Lecture Notes in Computer Science*, pages 585–589. Springer, 2003.
31. C. Roney-Dougal, I. Gent, T. Kelsey, and S. Linton. Tractable symmetry breaking using restricted search trees. In *Proceedings of the 16th European Conference on Artificial Intelligence*, pages 211–215. IOS Press, 2004.
32. A. Schutt, T. Feydy, P. J. Stuckey, and M. Wallace. Why cumulative decomposition is not as bad as it sounds. In I. P. Gent, editor, *Proceedings of the 15th International Conference on Principles and Practice of Constraint Programming*, volume 5732 of *Lecture Notes in Computer Science*, pages 746–761. Springer, 2009.
33. A. Schutt, P. J. Stuckey, and A. R. Verden. Optimal carpet cutting. In J. Lee, editor, *Proceedings of the 17th International Conference on Principles and Practice of Constraint Programming*, volume 6876 of *Lecture Notes in Computer Science*, pages 69–84. Springer, 2011.
34. L. Zhang, C. F. Madigan, M. W. Moskewicz, and S. Malik. Efficient Conflict Driven Learning in Boolean Satisfiability Solver. In *Proceedings of the International Conference on Computer-Aided Design*, pages 279–285. ACM, 2001.