Boosting SBDS for Partial Symmetry Breaking in Constraint Programming

Jimmy H.M. Lee and **Zichen Zhu***
Department of Computer Science and Engineering
The Chinese University of Hong Kong
Shatin, N.T., Hong Kong
{jlee,zzhu}@cse.cuhk.edu.hk

Abstract

The paper proposes a dynamic method, *Recursive SBDS (ReSBDS)*, for efficient partial symmetry breaking. We first demonstrate how (partial) Symmetry Breaking During Search (SBDS) misses important pruning opportunities when given only a subset of symmetries to break. The investigation pinpoints the culprit and in turn suggests rectification. The main idea is to add extra conditional constraints during search recursively to prune also symmetric nodes of some pruned subtrees. Thus, ReSBDS can break extra symmetry compositions, but is carefully designed to break only the ones that are easy to identify and inexpensive to break. We present theorems to guarantee the soundness and termination of our approach, and compare our method with popular static and dynamic methods. When the variable (value) heuristic is static, ReSBDS is also complete in eliminating all interchangeable variables (values) given only the generator symmetries. Extensive experimentations confirm the efficiency of ReSBDS, when compared against state of the art methods.

Introduction

Symmetries are common in many constraint problems. They can be broken statically (Puget 1993; Crawford et al. 1996; Flener et al. 2002; Law and Lee 2004) or dynamically (Fahle, Schamberger, and Sellmann 2001; Gent and Smith 2000; Roney-Dougal et al. 2004). Static methods alter the original problem by adding new constraints to eliminate symmetric solutions. In contrast, dynamic methods modify the search procedure to exclude exploration of symmetric regions. The focus of this paper is dynamic symmetry breaking, in particular the SBDS (symmetry breaking during search) method (Gent and Smith 2000) that adds conditional symmetry breaking constraints during search. The completeness of SBDS, however, relies on the fact that all symmetries are given. For problems with exponential number of symmetries, direct use of SBDS is impractical (Gent and Smith 2000). Apart from the large number of symmetry functions to implement, many conditional constraints may be added to the constraint store, slowing down search significantly. If only a subset of all symmetries is to be broken, (partial) SBDS misses important pruning opportunities.

Gent and Smith (2000) propose shortcut SBDS, which reduces overheads by breaking only active symmetries (the symmetric counterpart of the partial assignment at the current node being true) during search and thus adding only unconditional constraints. Similar to shortcut SBDS, LDSB (lightweight dynamic symmetry breaking) (Mears et al. 2013), handles only active symmetries and also their compositions. LDSB is restricted to breaking *only certain* kinds of symmetries, which enjoy a compact representation and efficient processing, but gives little flexibility for users to specify the symmetries to break.

Our goal is to have a dynamic symmetry breaking method that (a) can handle arbitrary kinds of symmetries, (b) breaks both active and inactive symmetries, (c) is flexible in choosing symmetries to break, (d) has relatively small overheads and (e) is efficient in identifying and breaking symmetry compositions. We propose *Recursive SBDS (ReSBDS)*, the main idea of which is to add extra conditional constraints during search recursively to prune also symmetric nodes of some pruned subtrees. Thus, ReSBDS can break extra symmetry compositions. Our proposal features a careful tradeoff between the number of constraints added and the benefits of extra pruning. We give theoretical characterization on the soundness and termination of our method, and comparisons on pruning strengths against other well-known symmetry breaking methods, such as LDSB (Mears et al. 2013) and the LexLeader method (Crawford et al. 1996). When given generators of interchangeable variables (values) according to a static search heuristic, ReSBDS is complete in eliminating the entire symmetry group. We perform extensive experimentation on benchmarks of different natures and compare against state of the art static and dynamic methods. Results confirm the feasibility and competitiveness of our proposal.

Background

A *constraint satisfaction problem* (CSP) $P = (X, D, C)$ is a tuple where X is a finite set of variables, D is a function mapping each $x \in X$ to a finite set D_x which is the domain of x, and C is a set of constraints, each of which is a subset of the cartesian product $\Pi_{x \in X} D_x$. A constraint is *generalized arc consistent (GAC)* iff when a variable in the scope of a

*We are grateful to the anonymous referees of CP-13 and AAAI-14 for their comments.

constraint is assigned any value in its domain, there exist compatible values in the domains of all other variables in the scope of the constraint. A CSP is GAC iff every constraint is GAC. An *assignment* $x = v$ assigns value v to variable x. A *full assignment* is a set of assignments, one for each variable in X. A *partial assignment* is a subset of a full assignment. A *solution* for P is a full assignment that makes every member of C true.

In order to make sensible comparisons against other methods, we consider only search trees with static variable and value orderings. A *search tree* for a CSP P with variables X is finite and has CSPs as nodes. The root is P. A node P_0 is a *leaf node* iff either P_0 has a variable with empty domain or the domains of all variables of P_0 are singletons. Without loss of generality, we consider search trees with binary branching, in which every non-leaf node has exactly two descendants. Suppose a non-leaf node P_1 has x and $v \in D_x$ as the branching variable and value. The left and right children of P_1 are $P_1 \cup \{x = v\}$ and $P_1 \cup \{x \neq v\}$ respectively. Each node P_1 is *associated with a partial assignment* A_1 which is the set of assignments collected from the root P to P_1.

We assume that *the CSP associated with a node is always made domain consistent.* Given a node P_0 with branching variable x and value v. The left child $P_0 \cup \{x = v\}$ of P_0 and the entire subtree are *pruned* by its ancestor P_k during search if v is pruned from D_x in P_k. In addition, the right child of P_0 is then merged with P_0.

Here we consider symmetry as a property of the set of solutions. A *solution symmetry* (Rossi, van Beek, and Walsh 2006) is a solution-preserving permutation on assignments. A *variable symmetry* σ is a bijection on variables that preserves solutions: if $\{x_i = v_i | 1 \leq i \leq n\}$ is a solution, then $\{x_{\sigma(i)} = v_i | 1 \leq i \leq n\}$ is also. A *value symmetry* θ is a bijection on values that preserves solutions: if $\{x_i = v_i | 1 \leq i \leq n\}$ is a solution, then $\{x_i = \theta(v_i) | 1 \leq i \leq n\}$ is also. A set of variables X (values V) is *interchangeable* iff any bijection mapping from $X \rightarrow X$ ($V \rightarrow V$) is a variable (value) symmetry. A *symmetry class* (Flener et al. 2002) is an equivalence class of full assignments, where two assignments are equivalent if there is some symmetry mapping one into the other. Given two nodes P_0 and P_1 with partial assignments A_0 and A_1 respectively. P_1 is a *symmetric node* of P_0 w.r.t. symmetry g if $A_0^g = A_1$.

A set of symmetries G *generates* a group Σ if every element of Σ can be written as a product of elements in G with the composition operator $\circ$ and every product of any sequence of elements of G is in Σ. G is called a set of *generators* for Σ, which is in turn the *symmetry group* of G.

A symmetry breaking method is *sound* (complete) iff it leaves at least (most) one solution in each symmetry class. A symmetry breaking method *breaks* a symmetry g iff there *exists* a remaining solution S after applying this method, S^g is pruned. A symmetry breaking method *eliminates* a symmetry g iff *for each* remaining solution S after applying this method, S^g must be pruned. A symmetry breaking method *eliminates* a symmetry group Σ iff all symmetries in Σ except the identity symmetry are eliminated.

Symmetry breaking method m_1 is *stronger in node (solution) pruning than* method m_2, denoted $m_1 \geq_n m_2$ ($m_1 \geq_s m_2$), if all nodes (solutions) pruned by m_2 would also be pruned by m_1. Note that $\geq_n$ implies $\geq_s$. If $m_1 \geq_n m_2$ and $m_2 \geq_n m_1$, then $m_1 =_n m_2$, and similarly for $m_1 =_s m_2$.

The LexLeader method (Crawford et al. 1996) adds one lexicographical ordering constraint, $\leq_{lex}$ (Frisch et al. 2002), per variable symmetry according to a fixed variable order.

Given the set of all symmetries to a CSP, symmetry breaking during search (SBDS) (Gent and Smith 2000) adds conditional constraints for each symmetry upon backtracking. Consider a node P_0 in the search tree with partial assignment A, branching variable x and value v. After backtracking from the node $P_0 \cup \{x = v\}$, for each solution symmetry g, SBDS adds the following conditional constraint to the node $P_0 \cup \{x \neq v\}$:

$$A \& A^g \& (x \neq v) \Rightarrow (x \neq v)^g \quad (1)$$

meaning that once $A\&(x = v)$ has been searched, its symmetric partial assignments $(A\&(x = v))^g$ for any g in the symmetry set under this subtree should not be searched at all. Note that Equation (1) can be simplified to $A^g \Rightarrow (x \neq v)^g$ as A and $x \neq v$ must hold in the subtree to be searched. If A^g for symmetry g is true at a node with partial assignment A, we call g an *active* symmetry at this node; otherwise, it is *inactive*. If A^g for a symmetry g is false at a search node with partial assignment A, we say g is *broken* at this node; otherwise, it is *unbroken*. Note that SBDS does not add symmetry breaking constraints for symmetries that are broken, as the left hand side of (1) is false already.

Partial SBDS and Its Pitfall

Partial SBDS (ParSBDS) is SBDS but handles only a given subset of all symmetries, which are usually generator symmetries (Flener et al. 2002; Petrie and Smith 2003). In the following, we analyze how LexLeader (Crawford et al. 1996) is stronger in node (solution) pruning than ParSBDS when variable and value orders are fixed and both are given the same subset of symmetries. We use as example the 2×2 unconstrained matrix problem with domain size 2. The symmetries are interchangeable rows and columns, which are denoted by R and C respectively.

Figure 1 gives the symmetry classes under matrix symmetries of the unconstrained matrix problem with generators $\{R, C\}$. Symmetric solutions are connected by lines with double arrow marked by the corresponding symmetries. There are 7 symmetry classes. Modeling the problem with 4 variables $\{x_{11}, x_{12}, x_{21}, x_{22}\}$, one for each square with the domain of each variable being $\{1, 2\}$, we can break $\{R, C\}$ using the LexLeader method (Crawford et al. 1996) by statically adding two constraints

$$\langle x_{11}, x_{12} \rangle \leq_{lex} \langle x_{21}, x_{22} \rangle, \langle x_{11}, x_{21} \rangle \leq_{lex} \langle x_{12}, x_{22} \rangle$$

to choose the lexicographically least solutions according to the input order $(x_{11}, x_{12}, x_{21}, x_{22})$.

Only 7 solutions are left: ①, ②, ④, ⑥, ⑦, ⑧ and ⑯. Solutions ③ and ⑤ are pruned by ②. Solution ⑨ is pruned by ③ and ⑤. Solution ⑩ is pruned by ⑦. Solution ⑪ is pruned by ⑥. Solution ⑬ is pruned by ④. Solutions ⑫ and ⑭ are pruned by ⑧. Solution ⑮ is pruned by ⑫ and ⑭.

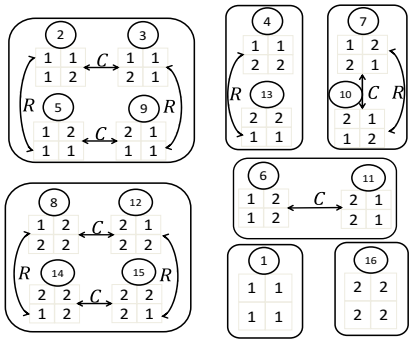


Figure 1: Symmetry classes of the unconstrained matrix problem with generators $\{R, C\}$.

The ParSBDS method leaves 8 solutions: ①, ②, ④, ⑥, ⑦, ⑧, ⑮ and ⑯ by breaking $\{R, C\}$ with input variable order $(x_{11}, x_{12}, x_{21}, x_{22})$ and minimum value heuristic. We show the depth-first search tree in Figure 2 where each solution leaf node is marked by its solution number. Upon each backtracking, ParSBDS adds a conditional constraint $A^g \Rightarrow (x \neq v)^g$ for each unbroken given symmetry g, where A is the partial assignment of the parent node, and x and v are the branching variable and value of the parent node respectively. The constraints are labeled by $\triangle$.

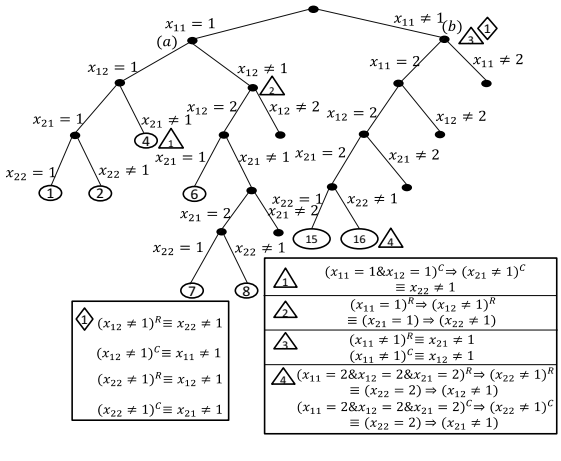


Figure 2: Search tree of the unconstrained matrix problem by breaking generators $\{R, C\}$.

LexLeader can prune solution ⑮ but ParSBDS *cannot*. LexLeader prunes ⑮ because it is lexicographically larger than ⑫ and ⑭. For ParSBDS, after backtracking from node (a) at node (b) in Figure 2, symmetry breaking constraints prune value 1 from $D_{x_{12}}$ and $D_{x_{21}}$. Solutions ⑫ and ⑭ are thus pruned. There would not be any backtracking from these pruned nodes in the subtree of (b). However, ParSBDS is triggered to add symmetry breaking constraints only by backtracking. Thus, no constraints are added to prune ⑮, which is symmetric to ⑧ by simply composing R and C.

We compare ParSBDS and LexLeader theoretically by considering only symmetries of the form $\sigma \circ \theta$, which are compositions of a variable symmetry σ and a value symmetry θ. Such symmetries $\sigma \circ \theta$ are common in practice (variable and value symmetries are special cases) and are bijections on assignments that preserve solutions: if $\{x_i = v_i | 1 \leq i \leq n\}$ is a solution, then $\{x_{\sigma(i)} = \theta(v_i) | 1 \leq i \leq n\}$ is also. Note $\sigma \circ \theta = \theta \circ \sigma$. LexLeader breaks value symmetries using the `element` constraint (Puget 2006b).

With a proof technique similar to that by Puget (2006a) in comparing SBDS and dynamic lex constraints, we give the following theorem.

Theorem 1. *Given a set of symmetries G. LexLeader $\geq_n$ ParSBDS and LexLeader $\geq_s$ ParSBDS when both search with the same static variable order used by LexLeader and min (max) value ordering.*

Recursive SBDS

In order to circumvent the pitfall of ParSBDS for a common class of symmetries, we propose *Recursive SBDS*, which adds extra constraints that are easy to derive, inexpensive to compute and can break many of the composition symmetries. Experimental results show our method can find a good balance between the number of symmetry breaking constraints to add and the extra pruning induced.

Consider the problem in Figure 2 again. Being given the generators $\{R, C\}$, ParSBDS will add at most two conditional constraints in each backtracking. Our proposed additional constraints are indicated by $\diamondsuit$.

After backtracking from (a) at (b), as A is empty, both symmetries are intact. ParSBDS adds $x_{12} \neq 1$ and $x_{21} \neq 1$. After 1 is pruned from $D_{x_{12}}$, symmetric nodes P_0 containing partial assignment $x_{12} = 1$ are thus pruned in the subtree. Why should we not also prune P_0's symmetric nodes? Adding $(x_{12} \neq 1)^R$, we get $x_{22} \neq 1$. Solution ⑮ is pruned. Since $x_{12} \neq 1$ is derived from $(x_{11} \neq 1)^C$ and $C \circ C$ is identity, adding $(x_{12} \neq 1)^C \equiv x_{11} \neq 1$ cannot give us any new prunings. Continuing with this recursive addition of symmetry breaking constraints, we add $(x_{22} \neq 1)^R$ and $(x_{22} \neq 1)^C$. This gives no further pruning, and the recursive addition of constraints is stopped. The extra constraints indicated by $\diamondsuit$ in Figure 2 allow us to prune also solution ⑮. Thus symmetry $C \circ R$ is eliminated.

Given a set of symmetries, breaking the potentially exponential number of all symmetry compositions is expensive in general. An important lesson we learn from the last example is that some such compositions can be easy to identify and break. Generalizing from the example, we give *Recursive SBDS (ReSBDS)* as follows.

1. Given the input symmetries G. Upon each backtracking, ReSBDS adds conditional constraints added by ParSBDS.
2. ReSBDS maintains a *backtrackable* set T of assignments, which is initially empty at the root node. Whenever ReSBDS adds a conditional constraint, $A^g \Rightarrow (x \neq v)^g$ (where $g \in G$), we add the following into T:
   - each $(x_i = v_i) \in A^g$, and

- $(x = v)^g$.

3. After constraint propagation at every node P_0 with partial assignment E during search, ReSBDS performs:
   - For each violated $(x_i = v_i) \in T$
     - add $E^h \Rightarrow (x_i \neq v_i)^h$ for $\forall h \in G$
     - delete $(x_i = v_i)$ from T in P_0's subtree
     - update T accordingly as in Step 2
   - Initiate constraint propagation and repeat Step 3 if there is pruning; otherwise, go on branching.

Since T is backtrackable, when the search backtracks, modifications to T must be undone. We elaborate the meaning of T briefly in the following.

There are two different reasons for ReSBDS to add a conditional constraint $A^g \Rightarrow (x \neq v)^g$ for some $g \in G$ at a node P_0. First, ReSBDS follows ParSBDS to add a symmetry breaking constraint upon backtracking. Second, P_0 has partial assignment A with v being pruned from D_x by constraint propagation. ReSBDS needs to detect *when exactly* the intended symmetric nodes to prune for an added conditional constraint are *actually* pruned. To achieve this, we record in T all the assignments whose violations can indicate that $A^g \Rightarrow (x \neq v)^g$ is already satisfied, in which case all nodes containing the partial assignments $A^g \& (x = v)^g$ under the subtree of P_0 are pruned.

At every node after constraint propagation, Step 3 of ReSBDS checks T to see if any stored assignment is violated. Suppose node P_1 has partial assignment E with v_i being pruned from D_{x_i} and $(x_i = v_i) \in T$. It means that every node P_s in the subtree of P_1 containing partial assignments $E \& (x_i = v_i)$ are pruned. Suppose $x_i = v_i$ is added to T because of the previously posted conditional constraint $A^g \Rightarrow (x \neq v)^g$. P_s is associated with the partial assignment A_s. Consider all assignments $x_j = v_j$ (which can be none) in $A^g \& (x = v)^g$ but not in E. Therefore, $x_j = v_j$ can be true or false in A_s. If all such assignments are true in A_s, P_s is a symmetric node of a previously visited or pruned node w.r.t. g, which is intended to be pruned by $A^g \Rightarrow (x \neq v)^g$. Posting $E^h \Rightarrow (x_i = v_i)^h$ can *potentially* break $g \circ h$ (when this constraint has pruned a solution) for all $h \in G$. Otherwise, P_s is just an ordinary pruned node. Posting $E^h \Rightarrow (x_i = v_i)^h$ to prune the symmetric nodes of such ordinary pruned nodes is sound but not useful for breaking compositions.

Similar to SBDS, ReSBDS does not add symmetry breaking constraints for broken symmetries. Now we give theorems on the termination, space complexity, soundness, and pruning strength of ReSBDS over ParSBDS.

Theorem 2. *Step 3 of ReSBDS always terminates.*

Proof. The number of all possible assignments is limited. Once an assignment in T is violated, the assignment will be removed from T under the subtree. Thus Step 3 of ReSBDS will always terminate. □

Theorem 3. *Given a CSP $P = (X, D, C)$ with $|X| = n$. The maximum size of T is $\sum_{i=0}^{n-1} D(x_i)$.*

Proof. The maximum size of T is $\sum_{i=0}^{n-1} D(x_i)$, which is the number of all possible assignments for the CSP P. □

Theorem 4. *ReSBDS is sound.*

Proof. Gent and Smith (2000) prove that partial SBDS does not exclude the solution occurring at the leftmost position in the search tree of every symmetry class. The extra breaking constraints added by ReSBDS are used only to prune the symmetric equivalent subtrees of pruned subtrees which either contain no solutions or solutions symmetric to the leftmost solution discovered earlier. □

Theorem 5. *Given the same set of symmetries. ReSBDS $\geq_n$ ParSBDS and ReSBDS $\geq_s$ ParSBDS when both use the same static variable and value orderings.*

Proof. All constraints added by ParSBDS are also added by ReSBDS. □

ReSBDS is stronger in solution pruning than LexLeader when we *consider only* symmetries that are compositions $\sigma \circ \theta$ of a variable symmetry σ and a value symmetry θ.

Theorem 6. *Given the same set of symmetries. ReSBDS $\geq_s$ LexLeader when both search with the static variable ordering used by LexLeader and min (max) value ordering.*

Sketch of Proof. We prove that all symmetric solutions pruned by LexLeader will also be pruned by ReSBDS. Suppose a solution s_1 is pruned by LexLeader at a node P_1. There must exist a solution s_2 symmetric to s_1 and $s_2 <_{lex} s_1$. Suppose s_1 cannot be pruned by ReSBDS. With the same search heuristic, s_2 or its ancestor should be searched earlier than s_1. There are two cases to consider. If s_2 is not pruned, symmetry breaking constraints added because of backtracking would prune s_1. If s_2 is pruned by some conditional symmetry breaking constraints, ReSBDS would record the partial assignments in these constraints. Symmetry breaking constraints added because of violation of these recorded assignments would prune s_1. Both cases contradict the assumption. The theorem is proved. □

The DOUBLELEX (Flener et al. 2002) method, a special case of LexLeader, breaks adjacent rows and columns interchangeability in matrix problems when searching with row-wise or column-wise variable ordering and min (max) value ordering.

Corollary 1. *ReSBDS $\geq_s$ DOUBLELEX.*

ReSBDS is, however, not stronger in node pruning than LexLeader. Consider the CSP with variables $\{x_1, \ldots, x_6\}$ and $\{1, 2, 3\}$ as domains. The constraint is: $x_1 + x_2 + x_3 = x_4 + x_5 + x_6$. There are several variable symmetries. Here we consider only the symmetry mapping x_i onto x_{7-i}. LexLeader adds constraint $\langle x_1, \ldots x_6\rangle \leq_{lex} \langle x_6, \ldots, x_1\rangle$ at the root node to break this symmetry. ReSBDS adds conditional symmetry breaking constraints during search.

Both ReSBDS and LexLeader return 84 solutions, but LexLeader has 175 nodes in the search tree while ReSBDS has 187. We shall demonstrate empirically in our experiments that ReSBDS prunes many more symmetric solutions than LexLeader in practice, and thus also more search space.

Even when ReSBDS is given only generators, it can be complete in specific situations.

Theorem 7. *Given a fixed variable (value) ordering γ. Suppose G is the set of symmetries such that adjacent variables (values) are interchangeable. Applying ReSBDS on G and searching with the γ variable (value) ordering eliminates all symmetries in Σ, where Σ is the symmetry group of G.*

Sketch of Proof. For interchangeable variables, LexLeader can eliminate Σ by being only given G according to any γ ordering. For interchangeable values, Walsh (2007) proves VALSYMBREAK(G,X) eliminates Σ, where VALSYMBREAK is a global constraint generated from LexLeader. Thus, results follow directly from Theorem 6. □

LDSB (Mears et al. 2013) is a further development of shortcut SBDS (Gent and Smith 2000), which is a light version of SBDS by considering only active symmetries. LDSB handles only active symmetries and also their compositions. In addition, when v is pruned from variable x, for every active symmetry f, LDSB not only asserts $(x \neq v)^f$ but also $(x \neq v)^{f \circ g}$ for each active symmetry g. This recursive step is repeated in a breadth-first manner until no more new prunings are obtained (Mears et al. 2013). LDSB is close in spirit to ReSBDS, has smaller overhead but also misses important pruning opportunities. We give theorems to compare the node and solution pruning power of ReSBDS and LDSB. Since LDSB (Mears et al. 2013) can deal only with variable symmetries and value symmetries, we *restrict our attention* to these symmetries.

Theorem 8. *Given the same set of symmetries to ReSBDS and LDSB. Suppose both use the same static variable and value orderings. If all unbroken symmetries at each node during search are active symmetries, ReSBDS $=_n$ LDSB and ReSBDS $=_s$ LDSB, otherwise, ReSBDS $\geq_n$ LDSB and ReSBDS $\geq_s$ LDSB.*

Sketch of Proof. If all unbroken symmetries are active, both methods add only unconditional symmetry breaking constraints and we can easily check that ReSBDS and LDSB add exactly the same constraints to prune the same nodes and solutions.

Otherwise, LDSB breaks only active unbroken symmetries, but ReSBDS breaks also inactive unbroken symmetries and their compositions. Hence, ReSBDS is stronger. □

Experimental Results

This section gives three experiments. In case of partial symmetry breaking, the symmetries to break are chosen according to the standard generators suggested in the literature. SBDS and ParSBDS share the same code base as the ReSBDS implementation. We compare ReSBDS against ParSBDS and LDSB as well as state of the art static symmetry breaking methods. We also compare against symmetry breaking with lazy clause generation (Chu et al. 2011) indirectly using results from the literature since we do not have access to the implementation. GAP-SBDS (Gent, Harvey, and Kelsey 2002) and GAP-SBDD (Gent et al. 2003) are not compared since we do not have their implementations in Gecode. These two methods are complete but the overhead is big in general. We also skip comparing against SNAKELEX (Grayland, Miguel, and Roney-Dougal 2009), which is another partial symmetry breaking method using LexLeader. Literature results (Katsirelos, Narodytska, and Walsh 2010) show it has similar efficiency as DOUBLELEX. All experiments are conducted using Gecode Solver 4.2.0 on Xeon E5620 2.4GHz processors.

Although Mears et al. (2013) has designed a pattern syntax to specify the input symmetries for LDSB, the syntax is not available in the Gecode implementation, which gives only a restricted syntax to specify the types of symmetries and parameters. It is unclear exactly what symmetries are processed by LDSB. In subsequent sections, we state only the *types* of symmetries given to LDSB in our experiments.

In our tables, $\#s$ denotes the number of solutions, $\#f$ denotes the number of failures (number of failed leaf nodes) and t denotes the running time. An entry with the symbol " – " indicates that the search timed out after the 1 hour limit. The best results are highlighted in bold. **Unique** uses SBDS to break *all* symmetries. **Doublelex** lexicographically orders the rows and columns in increasing order. **ParSBDS**, **LDSB** and **ReSBDS** handle the given symmetries by ParSBDS, LDSB and ReSBDS respectively. *Unless otherwise specified*, the search order is defaulted to input variable order and minimum value order.

N-Queens

We model the N-Queens problem the standard way using one variable per column. All 8 geometric symmetries are given to **Unique**. We give **ParSBDS** and **ReSBDS** only the two generators rx (reflection on the vertical axis) and $d1$ (reflection on the diagonal), which can generate all 8 geometric symmetries. **LDSB** can only handle the $rx, ry, r180$ symmetries, which are reflections on the vertical and horizontal axes and rotation of 180 degrees. They form a symmetry group. Following Mears et al. (2013), we give **LDSB** only the two generators rx and ry.

Table 1 shows the results. Being given two symmetries, **ReSBDS** enjoys over 36% reduction in solution size and over 62% in failures when compared to **ParSBDS**. In terms of runtime, **ReSBDS** is almost 2 times faster. This shows the additional constraints we add can break more compositions and prune more symmetric subtrees in an efficient way. **Unique** eliminates the symmetries in N-Queens completely and has the smallest solution size and search tree. **ReSBDS**, which breaks the symmetries only partially, is 1.6 times faster than **Unique**. This shows partial symmetry breaking method can be efficient even for polynomially symmetric CSPs (McDonald and Smith 2006). **LDSB** performs worse than **ParSBDS** in search tree size and runtime. This example demonstrates the main advantages of **ReSBDS**: flexibility in choosing symmetries to break, and good balance in overhead and extra pruning. For **ReSBDS**, the maximum size of T is 2. In at least 99.99% of the cases, the size of T is 0.

Table 1: N-Queens (all solutions)

N	Unique			ParSBDS			LDSB			ReSBDS		
	$\#s$	$\#f$	t	$\#s$	$\#f$	t	$\#s$	$\#f$	t	$\#s$	$\#f$	t
14	**45,752**	**823,621**	4.60	140,438	1,361,836	4.75	99,883	1,454,958	5.85	51,876	875,600	**2.69**
15	**285,053**	**4,825,631**	25.53	859,654	7,951,827	30.49	613,978	8,452,568	33.69	324,173	5,131,707	**15.99**
16	**1,846,955**	**30,221,334**	161.64	5,646,963	50,928,933	182.14	3,985,771	53,277,940	216.16	2,071,568	31,934,685	**101.58**
17	**11,977,939**	**200,005,686**	1,127.83	36,078,885	337,758,336	1,242.52	25,549,837	351,039,264	1,469.31	13,388,788	211,409,624	**694.73**

Error Correcting Code - Lee Distance (ECCLD)

The ECCLD problem is prob036 in CSPLib (Gent and Walsh 1999). We follow Lee and Li (2012) in the modeling and turning the optimization problem into a satisfaction problem to illustrate the effect on solution set size. We only try to break a subset of the available matrix symmetries in the problem. **ParSBDS** is given the symmetry that any two rows (columns) are interchangeable. **ReSBDS** is given interchangeability of adjacent rows (columns). **ReSBDS**^{c} is given as symmetries that adjacent rows (columns) are interchangeable and also cartesian-product of adjacent row symmetries and adjacent column symmetries. **LDSB** is given the interchangeable rows and columns. Note **ParSBDS** is given more symmetries than **ReSBDS**. Given the same symmetries as **ReSBDS** or **ReSBDS**^{c}, **ParSBDS** fails to solve most of the problems within the time limit.

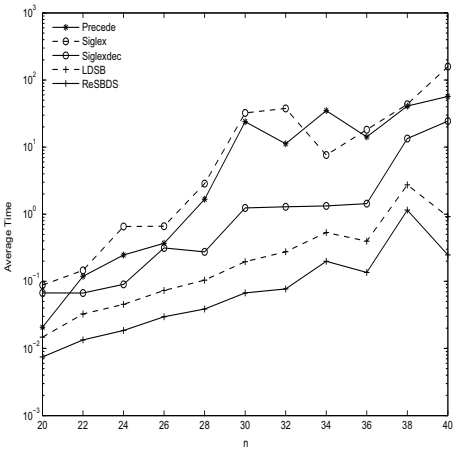
Table 2 shows the results. **ReSBDS** is again much more time efficient and breaks more symmetries than **ParSBDS**. **ReSBDS** has a smaller number of solutions and search tree size than **Doublelex**. The runtime, however, do not gain from the smaller search tree size because of the larger overhead of **ReSBDS**. After given more symmetries, **ReSBDS**^{c} has half the number of solutions and failures as those of **Doublelex**, and is 1.5 times faster than **Doublelex** on average. We can also see that **LDSB** leaves many more solutions and is drastically less efficient than **ReSBDS**. This shows **ReSBDS** can gain a lot by breaking also inactive symmetries and their compositions. Another advantage over **LDSB** is again that **ReSBDS** has the flexibility of choosing given symmetries to break. For **ReSBDS**, the maximum size of T is 99, and the average size is 5. In 15.7% of the cases, the size of T is 0.

Concert Hall Scheduling

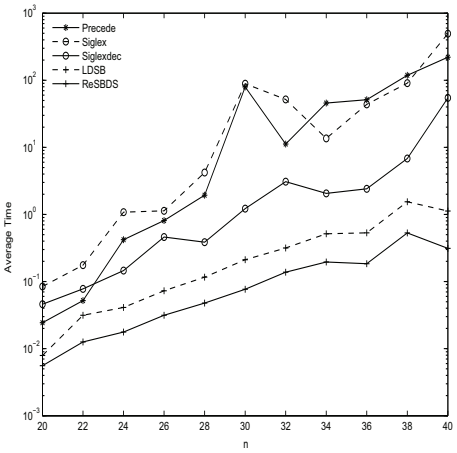
Concert hall scheduling (Law and Lee 2006) is to choose among n applications specifying a period and an offered price to use k identical concert halls, to maximize profit. We take the benchmarks used by Law and Lee (2007), and test with $k \in \{8, 10\}$. The concert halls are interchangeable, and so are applications within each partition.

We compare against the value precedence constraint (Law and Lee 2006) for breaking value interchangeability and $\leq_{lex}$ for breaking variable interchangeability (**Precede**), SIGLEX constraint (**Siglex**) for breaking variable and value interchangeability (Law et al. 2007) and its descending-partition-size variation (**Siglexdec**). **ReSBDS** is given two symmetries: adjacent variables within each partition being interchangeable and adjacent values being interchangeable so that **ReSBDS** only adds unconditional constraints. **LDSB** is given the interchangeable variables and values.

For the three static methods, smallest domain first heuristic is used. For **ReSBDS** and **LDSB**, the variable ordering heuristic chooses the variable with the most constraints and breaks ties by the size of the partition containing the variables. This can make more symmetries being active at the upper level. Due to space limitation, we do not show the results of running the three static methods with this search heuristic as the heuristic does not improve the performance.



(a) k =8



(b) k =10

Figure 3: Concert hall scheduling (optimal solution)

We do not compare the solution set size since the problem

Table 2: The ECCLD problem (all solutions)

n, c, b	**Doublelex**			**ParSBDS**			**LDSB**			**ReSBDS**			**ReSBDS**^{c}		
	$\#s$	$\#f$	t	$\#s$	$\#f$	t	$\#s$	$\#f$	t	$\#s$	$\#f$	t	$\#s$	$\#f$	t
4,4,8	32,469	839,251	44.09	38,235	1,073,244	86.89	430,079	4,538,862	214.22	25,543	631,717	40.49	**15,657**	**394,468**	**25.21**
5,2,10	87	41,571	5.25	136	94,472	22.81	572,041	8,439,178	735.09	76	33,672	5.31	**72**	**24,653**	**4.09**
5,6,4	710,731	725,837	17.09	849,724	900,549	29.51	770,956	822,157	21.85	530,819	537,118	16.74	**344,762**	**354,993**	**11.78**
5,6,5	1,441,224	5,508,192	123.03	1,661,689	6,523,348	203.20	1,944,489	7,669,846	183.17	1,044,674	4,005,819	115.79	**649,864**	**2,560,522**	**76.41**
5,6,6	297,476	11,709,068	312.95	334,519	13,579,038	541.34	538,688	19,051,585	523.68	221,236	8,553,011	300.59	**137,279**	**5,484,120**	**196.30**
6,4,4	4,698,842	4,139,211	117.11	7,607,152	6,726,710	270.11	13,710,850	10,718,458	348.79	3,746,439	3,278,563	122.12	**2,331,405**	**2,181,802**	**88.04**
6,4,5	29,345,816	73,522,873	1,992.05	-	-	-	-	-	-	23,059,956	57,225,832	1,957.51	**12,832,628**	**33,062,865**	**1,187.37**
6,8,4	59,158	2,469,211	37.71	62,380	2,992,204	67.89	64,262	2,588,227	43.72	45,024	1,778,978	36.84	**28,382**	**1,176,239**	**28.15**
8,4,4	35,626,714	48,525,827	1,332.26	-	-	-	-	-	-	28,982,779	39,163,464	1,433.60	**16,965,773**	**25,052,546**	**1,039.86**

Table 3: Concert hall scheduling, $k = 8$

n	**SBDS-1UIP** Xeon Pro 2.4GHz		**Static-1UIP** Xeon Pro 2.4GHz		**ReSBDS** Xeon E5620 2.4GHz	
	$\#f$	t	$\#f$	t	$\#f$	t
20	84	0.04	134	0.05	100	0.01
22	181	0.07	183	0.07	155	0.01
24	275	0.10	486	0.15	172	0.02
26	282	0.10	685	0.25	445	0.03
28	1,611	0.68	1,041	0.42	501	0.04
30	761	0.27	2,300	0.52	987	0.07
32	1,522	0.40	5,712	1.31	1,080	0.08
34	2,636	1.10	4,406	1.60	4,951	0.20
36	3,156	1.40	5,707	2.37	1,941	0.14
38	5,053	1.91	10,518	3.51	50,015	1.15
40	6,648	2.96	18,169	6.40	2,564	0.25

is optimization in nature. Figure 3 shows the timing results in graphical form for easy visualization. The horizontal axis shows instance size, and the vertical axis shows the *logarithmic* mean time in seconds taken to solve the set of instances to optimality. Any instance that is not solved within the time limit is considered to have taken 1 hour for the purpose of calculating the mean. Results show **ReSBDS** achieves the best performance, and **Precede** and **Siglex** perform the worst. **ReSBDS** is also significantly better than **Siglexdec** since **ReSBDS** has a much lower overhead and collaborates relatively well with the most-constraining heuristic. **LDSB** has similar performance trends as **ReSBDS**, but is at least 2 times and up to 3.7 times slower than **ReSBDS**. For **ReSBDS**, the maximum size of T is 8, and the average size is 1.65. In 19% of the cases, the size of T is 0.

Symmetry breaking with lazy clause generation (Chu et al. 2011) can also handle this problem well. **SBDS-1UIP** and **Static-1UIP** combine SBDS and static methods with lazy clause generation respectively. The experiments were run on different generations of the same processor family, but with different architectures and threading mechanisms. Results should not be compared directly, but Table 3 gives the competitive performance of **ReSBDS** against the replicated results (Chu et al. 2011) of **SBDS-1UIP** and **Static-1UIP** implemented with CHUFFED which is one of the best CSP solvers. This again shows our method can break a large number of symmetries with a small overhead.

Concluding Remarks

Our contributions are five fold. First, we have identified the inadequacy of ParSBDS in pruning symmetric solutions with respect to LexLeader. Second, we propose ReSBDS which can utilize symmetry breaking constraints' information to break extra symmetry compositions with low overhead. Third, we give formally the soundness and termination of ReSBDS and theoretical comparisons against ParSBDS, LexLeader, DOUBLELEX and LDSB. Fourth, ReSBDS is shown to be complete to break all interchangeable variables (values) given only generators when the variable (value) ordering is fixed. Fifth, we demonstrate empirically the efficiency of ReSBDS against state of the art static and dynamic methods via extensive experimentation.

McDonald and Smith (2006) have shown how dynamic methods are affected by variable and value heuristics in partial symmetry breaking. The ReSBDS method is no exception. Our experiments on concert hall scheduling have shown how a good heuristic can make ReSBDS much more efficient than the Siglex method. It is worthwhile to investigate the interaction between ReSBDS and search heuristics.

References

Chu, G.; Stuckey, P.; de la Banda, M.; and Mears, C. 2011. Symmetries and lazy clause generation. In *IJCAI'11*, 516–521.

Crawford, J.; Ginsberg, M.; Luks, E.; and Roy, A. 1996. Symmetry breaking predicates for search problems. In *KR'96*, 148–159.

Fahle, T.; Schamberger, S.; and Sellmann, M. 2001. Symmetry breaking. In *CP'01*, 93–107.

Flener, P.; Frisch, A.; Hnich, B.; Kiziltan, Z.; Miguel, I.; Pearson, J.; and Walsh, T. 2002. Breaking row and column symmetries in matrix models. In *CP'02*, 187–192.

Frisch, A.; Hnich, B.; Kiziltan, Z.; Miguel, I.; and Walsh, T. 2002. Global constraints for lexicographic orderings. In *CP'02*, 93–108.

Gent, I., and Smith, B. 2000. Symmetry breaking in constraint programming. In *ECAI'00*, 599–603.

Gent, I., and Walsh, T. 1999. CSPLib: a benchmark library for constraints. In *CP'99*, 480–481.

Gent, I. P.; Harvey, W.; Kelsey, T.; and Linton, S. 2003.

Generic SBDD using computational group theory. In *CP'03*, 333–347.

Gent, I. P.; Harvey, W.; and Kelsey, T. 2002. Groups and constraints: Symmetry breaking during search. In *CP'02*, 415–430.

Grayland, A.; Miguel, I.; and Roney-Dougal, C. M. 2009. Snake lex: An alternative to double lex. In *CP'09*, 391–399.

Katsirelos, G.; Narodytska, N.; and Walsh, T. 2010. On the complexity and completeness of static constraints for breaking row and column symmetry. In *CP'10*, 305–320.

Law, Y. C., and Lee, J. 2004. Global constraints for integer and set value precedence. In *CP'04*, 362–376.

Law, Y., and Lee, J. 2006. Symmetry breaking constraints for value symmetries in constraint satisfaction. *Constraints* 221–267.

Law, Y. C.; Lee, J.; Walsh, T.; and Yip, J. 2007. Breaking symmetry of interchangeable variables and values. In *CP'07*, 423–437.

Lee, J., and Li, J. 2012. Increasing symmetry breaking by preserving target symmetries. In *CP'12*, 422–438.

McDonald, I., and Smith, B. 2006. Partial symmetry breaking. In *CP'06*, 207–213.

Mears, C.; de la Banda, M. G.; Demoen, B.; and Wallace, M. 2013. Lightweight dynamic symmetry breaking. *Constraints* 1–48.

Petrie, K. E., and Smith, B. M. 2003. Symmetry breaking in graceful graphs. In *CP'03*, 930–934.

Puget, J.-F. 1993. On the satisfiability of symmetrical constrained satisfaction problems. In *ISMIS'93*, 350–361.

Puget, J.-F. 2006a. A comparison of SBDS and Dynamic Lex Constraints. In *SymCon'06*, 56–60.

Puget, J.-F. 2006b. An efficient way of breaking value symmetries. In *AAAI'06*, 117–122.

Roney-Dougal, C. M.; Gent, I. P.; Kelsey, T.; and Linton, S. 2004. Tractable symmetry breaking using restricted search trees. In *ECAI'04*, 211–215.

Rossi, F.; van Beek, P.; and Walsh, T. 2006. *Handbook of constraint programming*.

Walsh, T. 2007. Breaking value symmetry. In *CP'07*, 880–887.