Towards Breaking More Composition Symmetries in Partial Symmetry Breaking

Jimmy H.M. Lee and Zichen Zhu*

Department of Computer Science and Engineering
The Chinese University of Hong Kong
Shatin, N.T., Hong Kong

Abstract

The paper proposes a dynamic method, *Recursive Symmetry Breaking During Search (ReSBDS)*, for efficient partial symmetry breaking. We first demonstrate how Partial Symmetry Breaking During Search (ParSBDS) 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 symmetry breaking 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. We propose further a light version of ReSBDS method (LReSBDS), which has a slightly weaker pruning power of ReSBDS but with a reduced overhead. We give theoretical characterization on the soundness and termination of LReSBDS, and comparisons on pruning strengths against other symmetry breaking methods including ReSBDS. Extensive experimentations confirm the efficiency of ReSBDS and LReSBDS, when compared against state of the art methods.

1. Introduction

Constraint Programming (CP) is a practical framework for modeling and solving combinatorial search and optimization problems, which are NP-complete in general. Mainstream constraint-solving mechanisms encompass systematic exploration of the search tree augmented with various forms and degrees of constraint propagation [1] to prune the search space. Symmetries are transformations that map a problem solution into an equivalent solution (and also from non-solutions to non-solutions). Visiting symmetrical equivalents of traversed subtrees is fruitless. In many cases, all symmetrical variants of every deadend encountered during the search must be explored before a solution can be found. The goal of symmetry breaking is to avoid searching the symmetrical equivalents of visited search nodes, so as to increase solving efficiency.

*Corresponding author

Table 1: Comparing ParSBDS, LDSB and ReSBDS/LReSBDS

Features	ParSBDS	LDSB	ReSBDS LReSBDS
(a) can handle arbitrary kinds of symmetries	Yes	No	Yes
(b) has relatively small overheads	No	Yes	Yes
(c) is efficient in identifying and breaking symmetry compositions, i.e., has strong side-effects	No	Yes	Yes

Symmetries can be broken statically [2, 3, 4, 5] or dynamically [6, 7, 8]. Static methods alter the original problem by adding new constraints to eliminate symmetric parts and solutions. In contrast, dynamic methods modify the search procedure to exclude exploration of symmetric regions. There are pros and cons for each approach. Static methods are more commonly adopted since they are easier to implement and incur relatively less overhead. Static symmetry breaking constraints interact well (a) with one another to prune also extra composition symmetries in addition to the target symmetries, and (b) with problem constraints to increase constraint propagation. However, static techniques are usually tailored for specialized symmetry types and can be in conflict with search heuristics. An important advantage of dynamic techniques is the flexibility to handle symmetries of all kinds and the compatibility with search heuristics.

In this paper, we focus on dynamic symmetry breaking, in particular the Symmetry Breaking During Search (SBDS) method [7] that adds conditional symmetry breaking constraints during search. The completeness of SBDS, however, relies on the fact that all symmetries are given to SBDS to break. For problems with exponential number of symmetries, direct use of SBDS is impractical [7]. Apart from the large number of symmetry functions to implement, many symmetry breaking constraints may be added to the constraint store, slowing down search significantly. The first method to make SBDS practical is partial SBDS (ParSBDS). While SBDS tries to break all symmetries in the problem, ParSBDS tackles only a selected subset of symmetries [4, 9]. This is a direct application of partial symmetry breaking [10], which trades completeness for efficiency. Apart from ParSBDS, other adaptations of SBDS include shortcut SBDS [7] and Lightweight Dynamic Symmetry Breaking (LDSB) [11].

A starting point of our work is based on a simple observation: in many partial variants of static methods, posting only a few symmetry breaking constraints can sometimes eliminate most if not all symmetries in a problem. This is due to the fact that symmetry breaking constraints intended for a particular symmetry often break more than just the intended symmetry as a side-effect [12]. Thus, by carefully choosing the subset of symmetries to break, static methods can break also a large number of composition symmetries. Unfortunately, ParSBDS fails to generate the same strong side-effect. In a subsequent section, we give a detailed example showing how ParSBDS misses pruning opportunities when compared to a static method and given the same subset of symmetries to break.

Another starting point is LDSB, which is a further extension of shortcut SBDS designed for small overhead and ability to break composition symmetries. However, LDSB [11] targets only at symmetries that are common in constraint problems, has compact representation, and can be easily and efficiently processed. This limits the applicability of LDSB.

We propose a generalization of ParSBDS that enjoys the benefits of both ParSBDS and LDSB as summarized in Table 1. The main idea of *Recursive SBDS (ReSBDS)* is to add extra symmetry breaking 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 [11] and the LexLeader method [3]. When given generators of interchangeable variables (values) according to a static search heuristic, ReSBDS is complete in eliminating the entire symmetry group. We propose further a light version of ReSBDS method (LReSBDS), which has a slightly weaker pruning power of ReSBDS but with a reduced overhead. We also give theoretical characterization on the soundness and termination of LReSBDS, and comparisons on pruning strengths against other symmetry breaking methods including ReSBDS. 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.

As evidenced by Walsh's Spotlight Talk [13] at AAAI 2012, most recent successful symmetry breaking work [4, 14, 15, 16, 17, 5, 18, 19, 20, 21, 12, 22, 23] has been static in nature. Our work can help to revive interests of researchers in dynamic symmetry breaking and is the starting point of a series of further work on dynamic symmetry breaking [24, 25, 26]. The paper enhances the work of Lee and Zhu [27] and the Light ReSBDS algorithm by Lee and Zhu [24].

2. Background

In this section, we give some background on CSPs, symmetries as well as the two symmetry breaking methods: LexLeader and SBDS.

2.1. Basic Definitions

A *Constraint Satisfaction Problem* (CSP) [1] P is a tuple (X, D, C) where X is a finite set of variables, D is a finite set of domains such that each $x \in X$ has a domain $D(x)$ and C is a set of constraints, each a subset of the Cartesian product $D(x_{i_1}) \times \cdots \times D(x_{i_k})$ of the domains of the involved variables (*scope*). 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. A constraint is *generalized arc consistent (GAC)* iff when a variable in the scope of a 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. In the following, given a CSP $P = (X, D, C)$, we use the short form $P \cup \{c\}$ for $(X, D, C \cup \{c\})$ where c is a constraint.

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 *cons*$(P_1 \cup \{x = v\})$ and *cons*$(P_1 \cup \{x \neq v\})$ respectively where *cons*() enforces some form of consistency to a CSP. We call $x = v$ the *branching assignment* from P_1 to *cons*$(P_1 \cup \{x = v\})$. Each node P_1 is *associated with a partial assignment* A_1 which is the set of branching assignments collected from the root P to P_1. If a node P_0 is in a subtree under node P_1, P_0 is the *descendant node* of P_1 and P_1 is the *ancestor node* of P_0.

Throughout the paper, we assume that *cons() enforces generalized arc consistency to the CSP associated with each node.* Given a node P_0 with branching variable x and value v. The left child *cons*$(P_0 \cup \{x = v\})$ 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.

2.2. Symmetry, Group and Symmetry Breaking Methods

Here we consider symmetry as a property of the set of solutions. A *solution symmetry* [28] 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 solution. 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 solution. 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* [4] 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 $A_0^g \subseteq W_1$ where W_1 is the set of assignments $x = v$ where $D(x) = \{v\}$ in P_1.

A *group* is a non-empty set Σ with a composition operator $\circ$ such that:

1. Σ is closed under $\circ$. That is, for all $g, h \in \Sigma$, $g \circ h \in \Sigma$; and
2. there is an identity $id \in \Sigma$. That is, for all $g \in \Sigma$, $g \circ id = id \circ g = g$; and
3. every element g of Σ has an inverse g^{-1} such that $g \circ g^{-1} = g^{-1} \circ g = id$; and
4. $\circ$ is associative. That is, for all $f, g, h \in \Sigma$, $(f \circ g) \circ h = f \circ (g \circ h)$.

A set of symmetries G *generates* a group Σ iff 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 if $S \neq S^g$. A symmetry breaking method *eliminates* a symmetry group Σ iff all symmetries in Σ except the identity one are eliminated.

Symmetry breaking method m_1 is *stronger in nodes (resp. solutions) pruning* than method m_2, denoted by $m_1 \succeq_n (resp. \succeq_s) m_2$, when all the nodes (resp. solutions) pruned by m_2 would also be pruned by m_1. Symmetry breaking method m_1 is *strictly stronger in nodes (resp. solutions) pruning* than method m_2, denoted by $m_1 \succ_n (resp. \succ_s) m_2$, when $m_1 \succeq_n (resp. \succeq_s) m_2$ and $m_2 \not\succeq_n (resp. \not\succeq_s) m_1$. Note that $\succeq_n$ and $\succ_n$ imply $\succeq_s$ and $\succ_s$ respectively. Symmetry breaking method m_1 is *incomparable in nodes (resp. solutions) pruning* than method m_2, denoted by $m_1 \nparallel_n (resp. \nparallel_s) m_2$, when $m_1 \not\succ_n (resp. \not\succ_s) m_2$ and $m_2 \not\succ_n (resp. \not\succ_s) m_1$. Note that $\nparallel_s$ implies $\nparallel_n$.

2.3. LexLeader and Its Partial Versions

The LexLeader method [3] adds constraints for each symmetry to the problem to ensure that only the lexicographically least full assignments among all the symmetric full assignments are allowed. Thus only one full assignment is chosen by LexLeader in each symmetry class. In this way, all symmetries are broken. Given two vectors, $\overline{x} = \langle x_1, \ldots, x_n \rangle$ and $\overline{y} = \langle y_1, \ldots, y_n \rangle$ of

n variables, the *lexicographic ordering (lex) constraint*, $\overline{x} \leq_{lex} \overline{y}$, ensures that $\overline{x}$ is lexicographically less than or equal to $\overline{y}$. Such lexicographic ordering constraint is added for each variable symmetry according to a fixed variable order. Generalised arc consistency on lexicographic ordering between a pair of vectors, denoted as $\leq_{lex}$ or $\geq_{lex}$, has been enforced [29]. For row and column matrix symmetries, which are of exponential size, posting a lexicographic ordering constraint for each symmetry is impractical. DoubleLex [4] is a special case of LexLeader which posts constraints under row-wise or column-wise canonical order, to break only adjacent row and column interchangeability generator symmetries. Another partial symmetry breaking method based on LexLeader is SnakeLex [21] which posts lexicographical ordering constraints under snake ordering to break the same subset of symmetries as DoubleLex as well as an extra linear number of row or column symmetries. Allperm [16] is another method to partially break the matrix symmetries by constraining that the first row is less than or equal to all permutations of all other rows.

2.4. Symmetry Breaking During Search and Its Variants

Given the set of all symmetries to a CSP, SBDS [7] adds symmetry breaking 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 *cons*$(P_0 \cup \{x = v\})$, for each solution symmetry g, SBDS adds the following symmetry breaking constraint to the node *cons*$(P_0 \cup \{x \neq v\})$:

$$A \wedge A^g \wedge (x \neq v) \Rightarrow (x \neq v)^g \tag{1}$$

meaning that once $A \wedge (x = v)$ has been searched, its symmetric partial assignments $(A \wedge (x = v))^g$ 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 satisfied 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 *non-broken*. 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 (ParSBDS) is SBDS but handles only a given subset of all symmetries, which are usually generator symmetries [4, 9].

Gent and Smith [7] also 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, Lightweight Dynamic Symmetry Breaking (LDSB) [11], 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. It provides a pattern syntax [11] for users to specify symmetries to break, but works best on only symmetries for which these patterns are very compact.

3. Partial SBDS and Missing Pruning Opportunities

In partial symmetry breaking using static methods, Lee and Li [12] show that symmetry breaking constraint intended for a particular symmetry always breaks more than just the intended symmetry with a side-effect. This explains the reason why a few static symmetry breaking constraints can sometimes eliminate most if not all symmetries in a problem. We will show

partial symmetry breaking using SBDS method (ParSBDS) cannot generate the same side-effect as the static method of Crawford *et al.* [3].

In the following, we analyze how LexLeader [3] is stronger in both nodes and solutions pruning than ParSBDS when variable and value orders are fixed and both are given the same subset of symmetries. We use as example the matrix model of the $n \times n$ unconstrained matrix problem with domain size d which contains only variables but no constraints. Suppose $n = d = 2$. The symmetries to break 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 horizontal straightline and vertical curved double arrows 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 [3] 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.

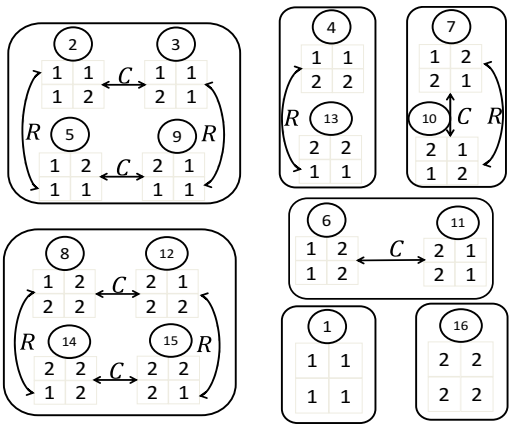

Figure 1: Symmetry classes of the unconstrained matrix problem with generators $\{R, C\}$.

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 min 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 backtrack, ParSBDS adds a symmetry breaking constraint $A^g \Rightarrow (x \neq v)^g$ for each non-broken 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 $\triangle$.

- After backtracking from $x_{22} = 1$ at (a), R and C are broken as $x_{22} = 1$ is false.
- After backtracking from $x_{21} = 1$ at (b), symmetry R is broken as $x_{21} = 1$ is false. One constraint $(x_{11} = 1 \wedge x_{12} = 1)^C \Rightarrow (x_{21} \neq 1)^C$ is added. Solution ③ is pruned.
- After backtracking from $x_{12} = 1$ at (c), symmetry C is broken as $x_{12} = 1$ is false. One constraint $(x_{11} = 1)^R \Rightarrow (x_{12} \neq 1)^R$ is added. Solution ⑤ is pruned.
- After backtracking from $x_{21} = 1$ at (d), symmetry R is broken as $x_{21} = 1$ is false. Symmetry C has been broken at (c). No symmetries are left in the subtree.
- After backtracking from $x_{11} = 1$ at (e), A is empty. Constraints $(x_{11} \neq 1)^R$ and $(x_{11} \neq 1)^C$ are added. Solutions ⑨, ⑩, ⑬ and ⑭ are pruned by the first constraint. Solutions ⑨, ⑩, ⑪ and ⑫ are pruned by the second constraint.
- After backtracking from $x_{22} = 1$ at (f), two constraints $(x_{11} = 2 \wedge x_{12} = 2 \wedge x_{21} = 2)^R \Rightarrow (x_{22} \neq 1)^R$ and $(x_{11} = 2 \wedge x_{12} = 2 \wedge x_{21} = 2)^C \Rightarrow (x_{22} \neq 1)^C$ are added.

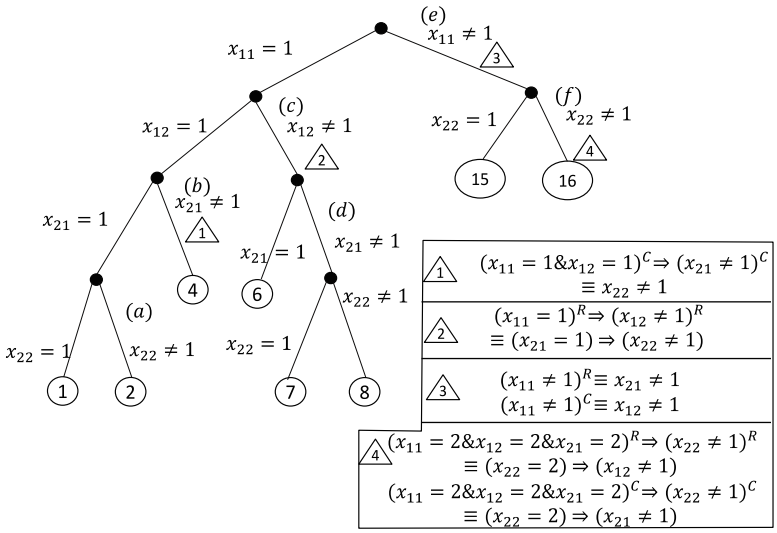

Figure 2: Search tree of utilizing ParSBDS to break generators $\{R, C\}$.

LexLeader can prune solution ⑮ but ParSBDS *cannot*. LexLeader prunes ⑮ because ⑮ is lexicographically larger than ⑫ and ⑭. For ParSBDS, after backtracking from $x_{11} = 1$ at (e) 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 more backtrack 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 [30].

When we compare the pruning power of LexLeader and ParSBDS in the following, we give the set of inverse symmetries to LexLeader of the ones given by ParSBDS. The reason is because LexLeader and ParSBDS are different in how they break symmetries. Consider two solutions s_1 and s_2 where $s_2 = s_1^g$ for a symmetry g. Suppose $s_1 <_{lex} s_2$, and ParSBDS and LexLeader are used to break g under the static variable ordering used by LexLeader and min value ordering. In this order, s_1 is searched before s_2. Once s_1 is searched, s_2 would be pruned by ParSBDS since $s_2 = s_1^g$. However, LexLeader cannot prune s_2 since $s_1 <_{lex} s_2$ satisfies the LexLeader constraint $X \leq_{lex} X^g$. If, otherwise, symmetry g^{-1} is given to LexLeader, the constraint $s_2 \leq_{lex} s_2^{g^{-1}}$ is violated since $s_1 = s_2^{g^{-1}}$ and $s_2 >_{lex} s_1$. Thus s_2 can be pruned now.

With a proof technique similar to that by Puget [31] in comparing SBDS and dynamic lex constraints, we give the following theorem.

Theorem 1. *Suppose G and H are sets of variable and value symmetries so that $\gamma \in G \Leftrightarrow \gamma^{-1} \in H$. LexLeader $\succ_n$ ParSBDS and LexLeader $\succ_s$ ParSBDS by posting G to ParSBDS and breaking H by LexLeader when both search with the same static variable ordering used by LexLeader and min (max) value ordering.*

Proof. We prove the symmetry breaking constraints added by ParSBDS during search will always be implied by the static symmetry breaking constraints added by LexLeader at the root node. Suppose we are at the node *cons*$(P_0 \cup \{x_t \neq v_t\})$ after assigning $t-1$ variables and backtracking from a node *cons*$(P_0 \cup \{x_t = v_t\})$. ParSBDS adds to node *cons*$(P_0 \cup \{x_t \neq v_t\})$ the following symmetry breaking constraints

$$(x_{\sigma(1)} = \theta(v_1)) \wedge \cdots \wedge (x_{\sigma(t-1)} = \theta(v_{t-1})) \rightarrow (x_{\sigma(t)} \neq \theta(v_t)) \tag{2}$$

for all $\gamma \equiv \sigma \circ \theta$ in G where σ is a variable symmetry and θ is a value symmetry. LexLeader (choosing the lexicographically least solution) adds the following constraints

$$\langle x_1, \ldots x_n \rangle \leq_{lex} \langle x_1, \ldots, x_n \rangle^{\sigma^{-1} \circ \theta^{-1}} \tag{3}$$

at the root node for all $\gamma^{-1} \equiv \sigma^{-1} \circ \theta^{-1}$ in H where n is the number of variables. Each of these constraints implies the following n constraints

$$(x_1 = \theta^{-1}(x_{\sigma(1)})) \wedge \cdots \wedge (x_{k-1} = \theta^{-1}(x_{\sigma(k-1)})) \rightarrow (x_k \leq \theta^{-1}(x_{\sigma(k)})) \tag{4}$$

where $k \in \{1, \ldots, n\}$. The tth constraint in the above n constraints can be rewritten as

$$(v_1 = \theta^{-1}(x_{\sigma(1)})) \wedge \cdots \wedge (v_{t-1} = \theta^{-1}(x_{\sigma(t-1)})) \rightarrow (x_t \leq \theta^{-1}(x_{\sigma(t)})) \tag{5}$$

at node *cons*$(P_0 \cup \{x_t \neq v_t\})$. With min value ordering, $x_t > v_t$ at node *cons*$(P_0 \cup \{x_t \neq v_t\})$. Therefore, combining (5) and $x_t > v_t$, we have:

$$(v_1 = \theta^{-1}(x_{\sigma(1)})) \wedge \cdots \wedge (v_{t-1} = \theta^{-1}(x_{\sigma(t-1)})) \rightarrow (v_t < \theta^{-1}(x_{\sigma(t)})) \tag{6}$$

which can be rewritten as

$$(x_{\sigma(1)} = \theta(v_1)) \wedge \cdots \wedge (x_{\sigma(t-1)} = \theta(v_{t-1})) \rightarrow (v_t < \theta^{-1}(x_{\sigma(t)})) \tag{7}$$

It is straightforward to derive that (7) implies (2). Thus LexLeader implies (7) and also (2). In other words, the symmetry breaking constraints added by ParSBDS during search will always be implied by the static constraints added by LexLeader at the root node.

Now we show LexLeader can prune more nodes and solutions than ParSBDS. Consider the unconstrained matrix problem in the above. Given only two generators $\{R, C\}$ where $R^{-1} = R$ and $C^{-1} = C$, LexLeader has 13 nodes and leaves 7 solutions while ParSBDS has 15 nodes and leaves 8 solutions. Therefore LexLeader $\succ_n$ ParSBDS and LexLeader $\succ_s$ ParSBDS. □

4. Enhancing Partial SBDS

In order to circumvent the pitfalls of ParSBDS, we propose *Recursive SBDS*, which adds extra constraints that are easy to derive, inexpensive to compute and can break many of the composition symmetries. After that, we propose a lightweight version of ReSBDS to further reduce the overhead. Formal characterizations and properties of these two methods, and comparisons with other state of the art methods are given in the form of theorems. Experimental results show our methods can strike good balances between the number of symmetry breaking constraints to add and the extra prunings induced.

4.1. Recursive SBDS

Consider the unconstrained matrix problem in Figure 2 again. Given the generators $\{R, C\}$, ParSBDS will add at most two symmetry breaking constraints in each backtrack. After backtracking from $x_{11} = 1$ at (e), 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. Thus symmetry $C \circ R$ is eliminated by adding this extra constraint.

Given a set of symmetries, breaking the potentially exponential number of all symmetry compositions is expensive in general. An important observation 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 backtrack, ReSBDS adds symmetry breaking 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 symmetry breaking 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 $(x_i = v_i) \in T$
     - delete $(x_i = v_i)$ from T in P_0's subtree
     - add $E^h \Rightarrow (x_i \neq v_i)^h$ for $\forall h \in G$ and 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.

Similar to ParSBDS, ReSBDS also does not add symmetry breaking constraints for symmetries that are broken. Moreover, if $v \notin D(x)$, the assignment $x = v$ would never be violated in subsequent search and thus does not need to be recorded into T. Note also that T is a set. This means if an assignment $x = v$ has already been recorded in T, we do not need to record it again in subsequent search.

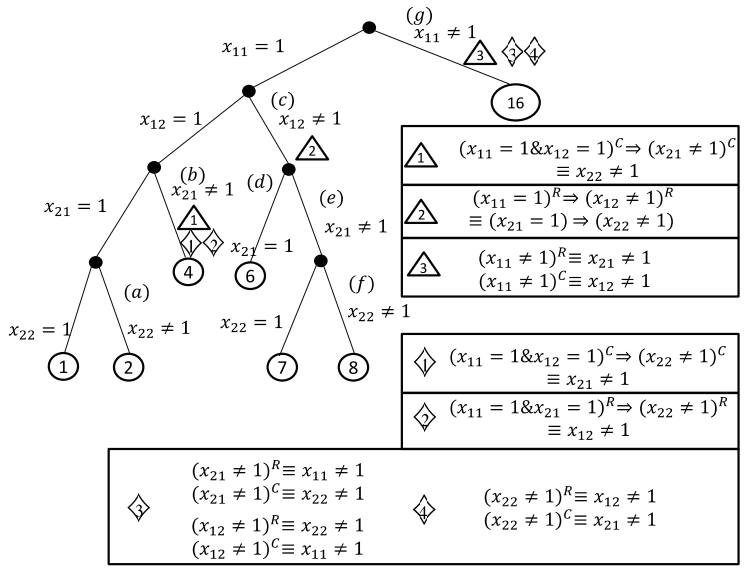

Figure 3: Search tree of utilizing ReSBDS to break generators $\{R, C\}$.

We show the depth-first search tree of the 2×2 unconstrained matrix problem with domain size 2 in Figure 3 which utilizes the above ReSBDS method to break the two generators R and C. Upon each backtrack, ReSBDS adds a symmetry breaking constraint $A^g \Rightarrow (x \neq v)^g$ for each non-broken 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. This kind of constraints are also added by ParSBDS and are labeled by $\triangle$ in Figure 3. Once an assignment $x_i = v_i$ in T is violated at a node P, ReSBDS adds a symmetry breaking constraint $E^h \Rightarrow (x_i \neq v_i)^h$ for each non-broken given symmetry h, where E is the partial assignment of P. This kind of constraints are additional constraints added at Step 3 and are indicated by $\diamondsuit$ in Figure 3.

- After backtracking from $x_{22} = 1$ at (a), R and C are broken as $x_{22} = 1$ is false. $T = \emptyset$.
- After backtracking from $x_{21} = 1$ at (b), symmetry R is broken as $x_{21} = 1$ is false. One constraint $(x_{11} = 1 \wedge x_{12} = 1)^C \Rightarrow (x_{21} \neq 1)^C$ is added. This constraint is simplified to $x_{22} \neq 1$ after doing symmetry computations. ReSBDS records $x_{22} = 1$ into T. $T = \{x_{22} = 1\}$. After constraint propagation, 1 is immediately pruned from $D(x_{22})$ and solution ③ is pruned. Assignment $x_{22} = 1$ is violated now and is deleted from T. Extra constraints can be added according to this violation. Since symmetry R is broken, constraint $(x_{11} = 1 \wedge x_{12} = 1)^C \Rightarrow (x_{22} \neq 1)^C$ is added. This constraint is simplified to $x_{21} \neq 1$ after doing symmetry computations. Assignment $x_{21} = 1$ has been violated

since (b) backtracks from $x_{21} = 1$. No assignments are put into T. $T = \emptyset$. And no further pruning occurs.

- After backtracking from $x_{12} = 1$ at (c), symmetry C is broken as $x_{12} = 1$ is false. One constraint $(x_{11} = 1)^R \Rightarrow (x_{12} \neq 1)^R$ is added. This constraint is simplified to $(x_{21} = 1) \Rightarrow (x_{22} \neq 1)$ after doing symmetry computations. ReSBDS records $x_{21} = 1$ and $x_{22} = 1$ into T. $T = \{x_{21} = 1, x_{22} = 1\}$.

- After branching with $x_{21} = 1$ at (d), propagation of the symmetry breaking constraint $(x_{21} = 1) \Rightarrow (x_{22} \neq 1)$ prunes 1 from $D(x_{22})$ and prunes solution ⑤ in subsequent search. Assignment $x_{22} = 1$ is violated. Now $T = \{x_{21} = 1\}$ after deleting $x_{22} = 1$ from T. Extra constraints can be added. Since symmetry C has been broken, constraint $(x_{11} = 1 \wedge x_{21} = 1)^R \Rightarrow (x_{22} \neq 1)^R$ is added. This constraint is simplified to $x_{12} \neq 1$ after doing symmetry computations. Assignment $x_{12} = 1$ has been violated since its parent node backtracks from $x_{12} = 1$ at (c). No assignments are put into T. And no more prunings take place.

- After backtracking from $x_{21} = 1$ at (e), symmetry R is broken as $x_{21} = 1$ is false. Symmetry C has been broken at (c). No symmetries are left in the subtree. Moreover, symmetry breaking constraint $(x_{21} = 1) \Rightarrow (x_{22} \neq 1)$ is satisfied as $x_{21} \neq 1$. Assignment $x_{21} = 1$ in T is violated. However, both generator symmetries are broken, thus no extra symmetry breaking constraints are added.

- After backtracking from $x_{22} = 1$ at (f), symmetries R and C have been broken at (e). Assignment $x_{22} = 1$ in T is violated. However, both generator symmetries are broken, no symmetry breaking constraints are added. $T = \emptyset$.

- After backtracking from $x_{11} = 1$ at (g), A is empty. Constraints $(x_{11} \neq 1)^R$ and $(x_{11} \neq 1)^C$ are added. These constraints are $x_{21} \neq 1$ and $x_{12} \neq 1$ after doing symmetry computations. ReSBDS records $x_{21} = 1$ and $x_{12} = 1$ into T. $T = \{x_{21} = 1, x_{12} = 1\}$. After constraint propagation, 1 is immediately pruned from $D(x_{21})$ and $D(x_{12})$. Solutions ⑨, ⑩, ⑪, ⑫, ⑬ and ⑭ are pruned. Assignments $x_{21} = 1$ and $x_{12} = 1$ are violated and are deleted from T. Constraints $(x_{21} \neq 1)^R$, $(x_{21} \neq 1)^C$, $(x_{12} \neq 1)^R$ and $(x_{12} \neq 1)^C$ are added. They are $x_{11} \neq 1$, $x_{22} \neq 1$, $x_{22} \neq 1$ and $x_{11} \neq 1$ after doing symmetry computations respectively. Only assignment $x_{22} = 1$ is not violated yet and is put into T. Constraint propagation immediately prunes 1 from $D(x_{22})$ and prunes solution ⑮. Assignment $x_{22} = 1$ is violated and is deleted from T. Two constraints $(x_{22} \neq 1)^R$ and $(x_{22} \neq 1)^C$ are added. No values can be further pruned. $T = \emptyset$.

We elaborate the meaning of T briefly in the following. There are two different reasons for ReSBDS to add a symmetry breaking constraint $A^g \Rightarrow (x \neq v)^g$ for some $g \in G$ at a node P. First, ReSBDS follows ParSBDS to add a symmetry breaking constraint upon backtracking ($\triangle$ in Figure 3). Second, P has partial assignment A with v being pruned from $D(x)$ by constraint propagation ($\diamondsuit$ in Figure 3). ReSBDS needs to detect *when exactly* the intended symmetric nodes to prune for an added symmetry breaking 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 $A^g \wedge (x = v)^g$ under the subtree of P are pruned.

At every node after constraint propagation, Step 3 of ReSBDS checks T to see if any stored assignment is violated. Suppose assignment $x_i = v_i$ is added to T at P_0 because of the posted symmetry breaking constraint $A^g \Rightarrow (x \neq v)^g$. Suppose further a descendent node P_1 of P_0 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 $E \wedge (x_i = v_i)$ are pruned. ReSBDS adds additional constraints $E^h \Rightarrow (x_i \neq v_i)^h$ for all non-broken given symmetries h. Suppose P_s is associated with the partial assignment A_s. Consider all assignments $x_j = v_j$ (which can be none) in $A^g \wedge (x = v)^g$ but not in $E \wedge (x_i = v_i)$. 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. Consider constraints added at ③ in Figure 3. All P_s containing $x_{21} = 1$ or $x_{12} = 1$ are pruned under (g) after doing constraint propagation. Each P_s is symmetric to a previously visited node according to R or C. ReSBDS adds constraints at ④ in Figure 3. These constraints break the generator symmetric nodes of all P_s. While each P_s is also generator symmetric to a visited node, composition symmetries are broken. Constraints at ④ break $R \circ C$ (and $C \circ R$) as they prune solution ⑮.

ReSBDS, however, might have useless recordings which cannot help to break composition symmetries. We will elaborate and propose a light version of ReSBDS to avoid such useless recordings in the next section.

Given a CSP $P = (X, D, C)$. We present the ReSBDS algorithm with given symmetries G in the following[1].

Algorithm 1 *Branch*(G, X, D, C)

Require:
G: non-broken symmetries;
X: variables;
D: domain of variables
C: constraints in the constraint store
E: the current partial assignment
x: the branching variable
v: the branching value
b: search direction;
T: recorded assignments
VT: violated assignments in T

Choice(x,v,b);
if $b = 0$ **then**
    x.*assign*(v);
    $E = E \wedge (x = v)$;
else
    x.*prune*(v);
    AddCon(G, E, T, x, v);
end if
EnforceConsistency(C);
CheckT(T, VT);
while $VT \neq \emptyset$ **do**
    for each $x_j = v_j \in VT$ **do**
        AddCon(G, E, T, x_j, v_j);
    end for
    EnforceConsistency(C);
    CheckT(T, VT);
end while

We show how to implement ReSBDS in a depth first search solver in Algorithm 1. To add the symmetry breaking constraint, we need to know the current partial assignment, which is

[1]In GitHub https://github.com/zichenzu/Recursive-SBDS, the implementation of ReSBDS is given with the help of N-Queens as an illustrative example.

recorded in E. T is the backtrackable set that is maintained by ReSBDS while VT records the set of violated assignments in T after constraint propagation at a search node.

Algorithm 1 shows the branching algorithm, i.e. how to branch at a search node. *Choice*() creates a choice according to the current node which is used to decide the branching variable and value, and whether to branch to its left or right child. If $b = 0$ (line 2), the node branches to the left child with $x = v$ (line 3) and E is updated (line 4); otherwise, it signifies a backtrack: search is branched to the right child with $x \neq v$ (line 6) and symmetry breaking constraints are added by calling *AddCon*() (line 7). All the assignments in the added constraints are also recorded in T by *AddCon*() (line 7). After enforcing GAC to each constraint in the constraint store using *EnforceConsistency*(), we need to check whether some recorded assignments in T are violated or not by calling *CheckT*() (line 10), where the violated assignments are recorded into VT (line 10). While there are violations (line 11), we add symmetry breaking constraints according to the current E and G for each of the violated assignments (lines 12-14). We need to do consistency enforcement (line 15), check violations (line 16), and add symmetry breaking constraints (lines 12-14) until there are no more violations (line 11).

Algorithm 2 *AddCon*(G, E, T, x, v)

```
1: for each h ∈ G do
2:     add E^h ⇒ (x ≠ v)^h into C;
3:     for each (x_i = v_i) ∈ (E^h ∧ (x = v)^h) do
4:         if (x_i = v_i) ∉ T and v_i ∈ D(x_i) then
5:             T = T ∧ (x_i = v_i);
6:         end if
7:     end for
8: end for
```

Algorithm 2 shows how to add symmetry breaking constraints according to the symmetries in G, current partial assignment E and the branching assignment (from which the backtrack happens) or violated assignment $x = v$. For each of the non-broken symmetries in G, one symmetry breaking constraint is added (line 2). We record in T all assignments in these symmetry breaking constraints that are not violated and have not been recorded into T (lines 3-7) yet.

Algorithm 3 *CheckT*(T, VT)

```
1: VT = ∅;
2: for each x_i = v_i ∈ T do
3:     if v_i ∉ D(x_i) then
4:         VT = VT ∧ (x_i = v_i);
5:         delete x_i = v_i from T;
6:     end if
7: end for
```

Algorithm 3 checks whether some assignments recorded in T are violated. If an assignment is violated (line 3), we record this assignment into VT and delete this assignment from T (lines 4-5). Such an assignment will never be recorded into T again in subsequent search.

Similar to the implementation of SBDS [7], ReSBDS does not add symmetry breaking constraints for broken symmetries and a boolean variable b_h^P is also constructed for each symmetry

$h \in G$ at each node P representing whether E^h is satisfied or not where E is the partial assignment of P. Along branching, b_h^P is incrementally computed in the following way. Suppose P is branched to P' with branching assignment $x = v$. The partial assignment E' of P' is extended to $E' \equiv (E \wedge (x = v))$. Now the value of $b_h^{P'}$ for symmetry $h \in G$ at P' is the conjunction of the satisfaction of E^h and $(x = v)^h$, i.e. $b_h^P \wedge (x = v)^h$. Hence, we can compute the symmetric partial assignment of the current partial assignment incrementally for each symmetry during search. A further advantage of these boolean variables is that when a b_h^P is proven to be false at P, its corresponding symmetry is broken at P as well as under the subtree of P.

Now we give theorems on the termination, space and time complexity of Algorithm 1.

Theorem 2. *Algorithm 1 always terminates.*

Proof. The number of all possible assignments is limited. Once an assignment is recorded into VT, it would be removed from T (line 5 of Algorithm 3) and never recorded back to T due to the condition in line 4 of Algorithm 2. Thus the while loop (lines 11-17) of Algorithm 1 always terminates. □

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. *Given a subset of symmetries G and a CSP $P = (X, D, C)$ with $|X| = n$. The time complexity of Algorithm 1 is $O(\sum_{i=0}^{n-1} |D(x_i)|(n|G| + \sum_{i=0}^{n-1} |D(x_i)|/2))$.*

Proof. The time complexity of Algorithm 2 and Algorithm 3 are $O(|G||X|)$ and $O(|T|)$ respectively. The worst time complexity of Algorithm 1 happens when $|VT| = 1$ after calling Algorithm 3 every time until T is empty. Thus the total time complexity is $O(|T||G||X| + |T|^2/2)$. While T has the maximum size $\sum_{i=0}^{n-1} |D(x_i)|$ and $|X| = n$, the time complexity of Algorithm 1 is $O(\sum_{i=0}^{n-1} |D(x_i)|(n|G| + (\sum_{i=0}^{n-1} |D(x_i)|)/2))$. □

4.2. Light ReSBDS

Domain filtering prunes values by an AC3-like [1] constraint propagation mechanism. If a value v is pruned from the domain of a variable during the propagation of a constraint c, we say this pruning is *effected* by constraint c.

The ReSBDS method utilizes a backtrackable set T to record useful assignments which might be violated in the future. Suppose $x_i = v_i$ is recorded in T since the constraint $A^g \Rightarrow (x \neq v)^g$ is added at node P_0. Suppose further this assignment is violated at a descendant node P_1, i.e. v_i is pruned from $D(x_i)$. The pruning indicates that $A^g \Rightarrow (x \neq v)^g$ is already satisfied. This pruning is effected either by a problem constraint or a symmetry breaking constraint. The latter case has the following feature.

Lemma 1. *Given a set of symmetries G. If, using the ReSBDS method, value v_i is pruned from $D(x_i)$ at node P_1 effected by a symmetry breaking constraint, $x_i = v_i$ must have been recorded into T at node P_1.*

Proof. Given any symmetry breaking constraint $A^g \Rightarrow (x \neq v)^g$ added by ReSBDS, ReSBDS records all assignments in $A^g \wedge (x = v)^g$. If v_i is pruned from $D(x_i)$ at node P_1 and is effected by a symmetry breaking constraint $c \equiv A^g \Rightarrow (x \neq v)^g$, $x_i = v_i$ must be in $A^g \wedge (x = v)^g$. Thus $x_i = v_i$ must have been recorded into T at P_1. □

ReSBDS has the opportunity to record assignments whose violations generate extra constraints but cannot help to prune composition symmetries. Suppose a symmetry breaking constraint $A^g \Rightarrow (x \neq v)^g$ is added at node P_0 and $x_i = v_i$ and $x_j = v_j$ are two of its assignments. ReSBDS records both $x_i = v_i$ and $x_j = v_j$ into T and generates symmetry breaking constraints once they are violated in subsequent search. Suppose $x_i = v_i$ is violated at a descendant node P_1 of P_0 and $x_j = v_j$ is violated at a descendant node P_2 of P_1. The symmetry breaking constraints added at P_1 due to the violation of $x_i = v_i$ prune all symmetric nodes of nodes (if any) containing $A^g \wedge (x = v)^g$ and in the subtree of P_1. The violation of $x_j = v_j$ in P_2 which is a descendant node of P_1 thus cannot prune any symmetric nodes of nodes containing $A^g \wedge (x = v)^g$ and in the subtree of P_2. Constraints added due to the violation of $x_j = v_j$ in P_2 is therefore useless to prune composition symmetries.

We would like to propose an adaptation of ReSBDS where extra symmetry breaking constraints are generated due to the violation of at most one assignment in each symmetry breaking constraint. Lemma 1 gives us some hints on a light ReSBDS method (LReSBDS) by considering only the assignments violated by the propagation of symmetry breaking constraints. In this way, we only need to add extra symmetry breaking constraints when a pruning is effected by a symmetry breaking constraint. T is not needed anymore and time is saved since there is no need to record assignments and check violations. Moreover, this ensures at most one assignment in each symmetry breaking constraint is used to generate extra symmetry breaking constraints.

The LReSBDS method is given as follows.

1. Given the input symmetries G. Upon each backtrack, LReSBDS adds symmetry breaking constraints added by ParSBDS.
2. After constraint propagation at every node P_0 with partial assignment A during search, LReSBDS performs:
   - For each assignment $(x_i = v_i)$ whose pruning is *effected by a symmetry breaking constraint*
     - add $A^g \Rightarrow (x_i \neq v_i)^g$ for $\forall g \in G$
   - Initiate constraint propagation and repeat Step 2 if new constraints are added; otherwise, go on branching.

The depth-first search tree of the 2×2 unconstrained matrix problem with domain size 2 is exactly the same as the one in Figure 3 after utilizing the LReSBDS method to break the two generators R and C. Upon each backtrack, LReSBDS adds a symmetry breaking constraint $A^g \Rightarrow (x \neq v)^g$ for each non-broken 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. This kind of constraints are also added by ParSBDS and are labeled by $\triangle$ in Figure 3. Once a symmetry breaking constraint prunes a value v_i from $D(x_i)$ at a node P, LReSBDS adds a symmetry breaking constraint $E^h \Rightarrow (x_i \neq v_i)^h$ for each non-broken given symmetry h, where E is the partial assignment of P. This kind of constraints are additional constraints added at Step 2 and are indicated by $\diamondsuit$ in Figure 3.

The reason that ReSBDS and LReSBDS have the same search tree for the above example is that there are no problem constraints. Thus the prunings of all violated assignments in T that can add extra constraints are effected only by the symmetry breaking constraints when using ReSBDS. Combining Lemma 1 and the fact that LReSBDS only adds extra constraints according to the prunings effected by the symmetry breaking constraints, we introduce the following lemma.

Lemma 2. *Given the same set of symmetries to ReSBDS and LReSBDS. Suppose both use the same variable and value orderings. If the prunings of all violated assignments in T are effected by the symmetry breaking constraints when using ReSBDS, ReSBDS $=_n$ LReSBDS and ReSBDS $=_s$ LReSBDS.*

What about the cases when the conditions of Lemma 2 are not satisfed? Consider the ECCLD problems in Section 5.1.4. Given the same set of interchangeability of adjacent rows (columns) and using the same input variable ordering and minimum value ordering heuristic, LReSBDS and ReSBDS result in search trees of different sizes, 3648007 nodes and 3648003 nodes respectively, for the instance (6,8,4). In Section 4.3, we give formal characterizations and comparisons of the node and solution pruning powers of LReSBDS and ReSBDS in the general case.

Similar with ReSBDS, there are two different reasons for LReSBDS to add a symmetry breaking constraint. First, LReSBDS follows ParSBDS to add a symmetry breaking constraint upon backtracking. Second, an assignment is violated due to the pruning effected by a symmetry breaking constraint (rather than any constraint in ReSBDS). Now we do not need extra effort to detect *when exactly* the intended assignment are *actually* pruned. We can "hack" into the propagation algorithm so that once a value is pruned by the symmetry breaking constraint, extra constraints are added.

Given a CSP $P = (X, D, C)$. We present the LReSBDS algorithm with given symmetries G in the following[2].

Algorithm 4 *BranchL*(G, X, D, C)

Require:
G: non-broken symmetries;
X: variables;
D: domain of variables
C: constraints in the constraint store
E: the current partial assignment
x: the branching variable
v: the branching value
b: search direction;

1: *Choice*(x,v,b);
2: **if** $b = 0$ **then**
3: x.*assign*(v);
4: $E = E \wedge (x = v)$;
5: **else**
6: x.*prune*(v);
7: *AddConL*(G, E, x, v);
8: **end if**

We show how to implement LReSBDS with the depth first search engine. Algorithm 4 shows the branching algorithm, i.e. how to branch at a search node. *Choice*() has the same functionality as in Algorithm 1. Here we do not need backtrackable sets T and VT. When branching to the left, E needs to be updated (line 4). Otherwise, symmetry breaking constraints are added by calling *AddConL*() upon backtracking (line 7).

Algorithm 5 shows how to add symmetry breaking constraints according to the given symmetries G, current partial assignment E and the branching assignment $x = v$ where backtrack happens or violated assignment $x = v$. For each of the non-broken symmetry in G, one symmetry breaking constraint is added (line 2).

Once a constraint is added to the constraint store or some form of consistency is enforced at a search node, Algorithm 6, an AC3-like constraint propagation algorithm, would be called

[2]In GitHub https://github.com/zichenzu/Light-Recursive-SBDS, the implementation of LReSBDS is further given with the help of N-Queens as an illustrative example.

Algorithm 5 *AddConL*(G, E, x, v)

1: **for** each $h \in G$ **do**
2: add $E^h \Rightarrow (x \neq v)^h$ into C;
3: **end for**

Algorithm 6 *AC3*(C)

1: $Q = C$;
2: **while** $Q \neq \emptyset$ **do**
3: Choose c from Q;
4: Remove c from Q;
5: **if** *Propagate*(c) **then**
6: $Q = Q \cup \{c' | c' \in C - \{c\}$ and c' and c have same variable in their scope$\}$
7: **end if**
8: **end while**

to trigger propagators of all constraints in the constraint store. In this algorithm, a constraint c would be chosen from the constraint store (line 3) and deleted from Q (line 4). After that, we call its propagator (line 5). If its propagator can prune some values (line 5), all other constraints that have been deleted from Q but have the same variable in their scope with that in c would be added back to Q (line 6). The while-loop would trigger all constraints in Q until Q is empty (line 2).

Algorithm 7 *SymConPro*(G, E, A, x, v)

// constraint propagation algorithm of $A \Rightarrow (x \neq v)$
1: **if** all assignments except $x_i = v_i$ in $A \wedge (x = v)$ are true **then**
2: x_i.*prune*(v_i);
3: *AddConL*(G, E, x_i, v_i);
4: **end if**

Algorithm 7 shows the constraint propagation algorithm of all symmetry breaking constraints added by LReSBDS which is in the form $A \Rightarrow (x \neq v)$. Values are pruned when all assignments except $x_i = v_i$ in the assignment set $A \wedge (x = v)$ are true (line 1). Now $x_i = v_i$ is enforced to be false (line 2). Additional symmetry breaking constraints are added according to this pruning (lines 3). And these symmetry breaking constraints' propagation is also done by Algorithm 7. Hence the recursive addition of constraints is done by the propagation mechanism which stops propagation only when every variable's domain does not change and no extra constraints are added then.

Similar to the implementation of SBDS and ReSBDS, LReSBDS does not add symmetry breaking constraints for broken symmetries and also a boolean variable is constructed for each symmetry $h \in G$ at each node P representing whether E^h is satisfied or not where E is the partial assignment of P. Along branching, b_h^P is incrementally computed in the similar way of that in ReSBDS.

We give theorems on the termination of Step 2.

Theorem 5. *Step 2 of LReSBDS always terminates.*

Proof. The number of all possible assignments is limited. Once a value is pruned from its variable's domain, this value never be pruned from its variable's domain again in subsequent constraint propagation. Thus the recursive addition of constraints will always terminate. □

4.3. Theoretical Results

In this section, we give theorems on the properties of ReSBDS and LReSBDS in addition to comparing the pruning power of ReSBDS and LReSBDS as well as against other state of the art methods.

Theorem 6. *(Soundness) Symmetry breaking constraints added by ReSBDS/LReSBDS leave at least one solution in each symmetry class.*

Proof. Gent and Smith [7] prove that ParSBDS does not exclude the solution occurring at the leftmost position in the search tree in every symmetry class. The extra symmetry breaking constraints added by ReSBDS are generated according to the violation of some assignments. These assignments comes from some already added symmetry breaking constraints, which are either symmetry breaking constraints also added by ParSBDS or the extra constraints added by ReSBDS. While ParSBDS leaves the leftmost solution in each symmetry class, these extra constraints 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.

Similar to ReSBDS, the extra symmetry breaking constraints added by LReSBDS 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. The result for LReSBDS follows directly. □

In ParSBDS, the symmetry breaking constraints added at a descendant node are implied by the constraints added upon backtracking from its ancestor node. Consider the depth-first search tree in Figure 2 again. The symmetry breaking constraint added by ParSBDS at node (c) does not need to be added upon backtracking to node (e) since the left hand side of the constraint $(x_{11} = 1)^R \Rightarrow (x_{12} \neq 1)^R$ is implied by the symmetry breaking constraint $(x_{11} \neq 1)^R$ posted at (e). Thus all the symmetry breaking constraints added by ParSBDS at a search node P_0 only need to be posted locally to the subtree of P_0 which means that they are removed from the constraint store upon backtracking from an ancestor node of P_0. This property also holds in ReSBDS and LReSBDS.

Theorem 7. *Given a set of symmetries G. Suppose P_0 is an ancestor node of P_1. The symmetry breaking constraints generated by ReSBDS/LReSBDS at node P_1 are implied by the constraints generated by ReSBDS upon backtracking from P_0.*

Proof. There are two kinds of constraints added by ReSBDS at node P_1.

1. P_1 is backtracking from a node and symmetry breaking constraints are thus added. For the same reason that ParSBDS posts constraints locally, these symmetry breaking constraints are implied by the constraints added upon backtracking from P_0.
2. A recorded assignment $x_i = v_i$ is violated at node P_1 with the current partial assignment E_1. Constraints $E_1^h \Rightarrow (x_i \neq v_i)^h$ for all $h \in G$ are added at P_1. Suppose P_0 has current partial assignment E_0 and P_2 backtracks from P_0. The constraints $\neg E_0^h$ for all $h \in G$ are thus added upon backtracking from P_0 at P_2. While $E_0 \subseteq E_1$, $E_0^h \subseteq E_1^h$ for all $h \in G$. Thus $\neg E_0^h$ implies $\neg E_1^h$ and also implies $E_1^h \Rightarrow (x_i \neq v_i)^h$.

Therefore, the symmetry breaking constraints generated by ReSBDS at node P_1 are implied by the constraints generated by ReSBDS upon backtracking from P_0.

Similarly, LReSBDS adds two kinds of constraints at node P_1. The first one is the same with that of ReSBDS. The second kind of constraint is added when a value v_i is pruned from $D(x_i)$ which is effected by a symmetry breaking constraint at node P_1 with the current partial assignment E_1. Constraints $E_1^h \Rightarrow (x_i \neq v_i)^h$ for all $h \in G$ are added at P_1. Suppose P_0 has current partial assignment E_0 and P_2 backtracks from P_0. The constraints $\neg E_0^h$ for all $h \in G$ are thus added upon backtracking from P_0 at P_2. While $E_0 \subseteq E_1$, $E_0^h \subseteq E_1^h$ for all $h \in G$. Thus $\neg E_0^h$ implies $\neg E_1^h$ and also implies $E_1^h \Rightarrow (x_i \neq v_i)^h$. Therefore, the symmetry breaking constraints added by LReSBDS at node P_1 are implied by the constraints added by LReSBDS upon backtracking from P_0. □

The above theorem shows the symmetry breaking constraints added by ReSBDS and LReSBDS only need to be posted locally to the subtree of the node where they are generated.

We give theorems to compare the pruning power of LReSBDS with ReSBDS.

Theorem 8. *Given a CSP with only variables, domains but no problem constraints and a subset of symmetries G. ReSBDS $=_n$ LReSBDS when both use the same static variable and value orderings and given the same set of symmetries.*

Proof. Since there are no problem constraints, the prunings of all violated assignments in T are effected by the symmetry breaking constraints when using ReSBDS. The result follows directly from Lemma 2. □

When there are problem constraints, things can get complicated. We have symmetry breaking constraints added by ReSBDS but not by LReSBDS when the violation of a recorded assignment in T is effected by a problem constraint. However, LReSBDS without these symmetry breaking constraints can have more backtracks and results in more symmetry breaking constraints being added. Thus their node pruning powers are incomparable in general.

Theorem 9. *Given the same set of symmetries. ReSBDS $\nparallel_n$ LReSBDS when both use the same static variable and value orderings.*

Proof. Suppose C_R and C_L are the sets of all symmetry breaking constraints added by ReSBDS and LReSBDS respectively.

When a violated assignment in T whose pruning is effected by a problem constraint, ReSBDS will be triggered to add a symmetry breaking constraint c for each symmetry to C_R. However, LReSBDS will be triggered to add symmetry breaking constraints only when the pruning is effected by symmetry breaking constraints. Thus the symmetry breaking constraints are added only to C_R but not C_L.

The added symmetry breaking constraints c in ReSBDS may interact with problem constraints and can cause further non-solution parts to be pruned by problem constraints. If the pruned assignments are not already in T, these prunings cannot trigger ReSBDS into adding new symmetry breaking constraints. In LReSBDS, however, there will be eventually backtracks from these non-solution parts and such backtracks will trigger addition of a symmetry breaking constraint for each symmetry. Thus there are symmetry breaking constraints added to C_L but not C_R.

Thus, the node pruning powers of ReSBDS and LReSBDS are incomparable in general. □

According to our empirical results (to be reported below), ReSBDS always prunes more than LReSBDS. However, their solution pruning powers remain the same.

Theorem 10. *ReSBDS $=_s$ LReSBDS when given the same set of symmetries and both use the same static variable and value orderings.*

Proof. Suppose C_R and C_L are the sets of all symmetry breaking constraints added by ReSBDS and LReSBDS respectively. We prove that each constraint that can prune solutions in C_L are also in C_R or implied by the constraints in C_R. Consider the resulting search trees Υ_R of ReSBDS and Υ_L of LReSBDS. There are three cases to consider.

1. LReSBDS adds a constraint $c_0 \in C_L$ at node P_0 in Υ_L due to a backtrack from node P_1 and symmetry g, and P_1 is pruned in Υ_R. Suppose the parent node of P_0 and P_1 is P in Υ_L. Thus P_0 is merged with P in Υ_R. Constraint c_0 in Υ_L can prune solutions only if P_1 has solutions in its subtree. Thus P_1 must be pruned by the the symmetry breaking constraints added by ReSBDS. An assignment $x_i = v_i$ in P_1 must have been recorded into T and violates at node P or its ancestor node. Assume this violation results constraint c' being added by ReSBDS according to symmetry g. If the violation occurs at an ancestor node of P in Υ_R, c' implies c_0. Otherwise, $c' = c_0$.
2. LReSBDS adds a constraint $c_0 \in C_L$ at node P_0 in Υ_L due to a backtrack from node P_1 and symmetry g, and P_1 is not pruned in in Υ_R. If P_0 is pruned in Υ_R, since c_0 is posted locally to the subtree of P_0, c_0 cannot prune more nodes than ReSBDS. Otherwise, c_0 is added by ReSBDS at node P_0.
3. LReSBDS adds a constraint $c_1 \in C_L$ at node P_1 in Υ_L because the propagation of a symmetry breaking constraint on symmetry g prunes value v_i from $D(x_i)$ at P_1. If P_1 is pruned in Υ_R, c_1 cannot prune more nodes than ReSBDS. Otherwise, assignment $x_i = v_i$ must have been recorded into T and violates at node P_1 or its ancestor node. Assume this violation results constraint c' being added by ReSBDS according to symmetry g. If the violation occurs at an ancestor node of P_1 in Υ_R, c' implies c_1. Otherwise, $c' = c_1$.

Thus we have ReSBDS $\succeq_s$ LReSBDS. Next, we prove symmetry breaking constraints in C_R but not in C_L cannot prune any solutions. Since these constraints are generated due to the violations of recorded assignments in T which are effected by problem constraints, they can only prune symmetric nodes of non-solution nodes. Thus ReSBDS $=_s$ LReSBDS. □

We give theorems to compare the pruning power of ReSBDS and LReSBDS with with the variants of LexLeader and SBDS methods.

Theorem 11. *Given a CSP with only variables, domains but no problem constraints and a subset of symmetries G. ReSBDS $\succ_n$ ParSBDS and LReSBDS $\succ_n$ ParSBDS when all use the same static variable and value orderings and given the same set of symmetries.*

Proof. Suppose C_P and C_R are the sets of all symmetry breaking constraints added by ParSBDS and ReSBDS respectively. We prove that each constraint in C_P are also in C_R or implied by the constraints in C_R. Consider the resulting search trees Υ_P of ParSBDS and Υ_R of ReSBDS. Suppose a constraint $c \in C_P$ is added at a node P_0 by ParSBDS in Υ_P due to the backtracking from node P_1.

1. If P_0 and P_1 are both in Υ_R, c is also added by ReSBDS and in C_R.

2. If P_0 is in Υ_R but P_1 is not, P_1 must have been pruned at an ancestor node of P_0 in Υ_R. It must be pruned due to the extra symmetry breaking constraints added by ReSBDS. Thus, the pruning that occurs at the ancestor node would trigger ReSBDS to add extra symmetry breaking constraints c' which implies c.
3. If both P_0 and P_1 are not in Υ_R, one of their ancestor nodes must have been pruned from Υ_R. Since c are only added at the subtree of P_0 in Υ_P, it is implied by the constraints added by ReSBDS.

Thus we have ReSBDS $\succeq_n$ ParSBDS. ReSBDS adds extra symmetry breaking constraints if an assignment recorded in T is violated. Consider the unconstrained matrix problem again. Given only two generators $\{R, C\}$, ReSBDS has 13 nodes while ParSBDS has 15 nodes. Thus ReSBDS $\succ_n$ ParSBDS.

Since we further have ReSBDS $=_n$ LReSBDS, LReSBDS $\succ_n$ ParSBDS by Theorem 8. □

With similar reasoning as in the proof of Theorem 9, the node pruning powers of ReSBDS/LReSBDS and ParSBDS are incomparable if there are problem constraints. Empirically, however, ReSBDS/LReSBDS usually results in a much smaller search tree than ParSBDS does.

Theorem 12. *Given the same set of symmetries. ReSBDS/LReSBDS $\nparallel_n$ ParSBDS when both use the same static variable and value orderings.*

Proof. Similar to the proof of Theorem 9, ReSBDS/LReSBDS adds more extra composition symmetry breaking constraints than ParSBDS, but ParSBDS without these symmetry breaking constraints can have more backtracks and result in more symmetry breaking constraints being added. Thus their node pruning powers are incomparable in general. □

However, solution pruning power is not affected by the above case.

Lemma 3. *Given the same set of symmetries. Symmetry breaking constraints added by ParSBDS but not by ReSBDS cannot prune solutions when both use the same static variable and value orderings.*

Proof. Consider the resulting search trees Υ_P of ParSBDS and Υ_R of ReSBDS. Suppose a constraint c is added at a node P_0 by ParSBDS in Υ_P due to the backtracking from node P_1. Constraint c cannot be added or implied by the constraints added by ReSBDS in Υ_R only when (a) P_0 is in Υ_R but P_1 is not, (b) P_1 is pruned by a problem constraint and (c) this pruned assignment is not recorded into T. Since P_1 is pruned by problem constraint in Υ_R, no solutions exist in the subtree of P_1 in Υ_P. Thus c cannot prune any symmetric solutions. □

Theorem 13. *Given the same set of symmetries. ReSBDS $\succ_s$ ParSBDS and LReSBDS $\succ_s$ ParSBDS when all use the same static variable and value orderings.*

Proof. Lemma 3 shows all symmetry breaking constraints added by ParSBDS but not by ReSBDS cannot prune solutions. Thus ReSBDS $\succeq_s$ ParSBDS. Consider the unconstrained matrix problem again. Given only two generators $\{R, C\}$, ReSBDS leaves 7 solutions while ParSBDS leaves 8 solutions. Therefore, ReSBDS $\succ_s$ ParSBDS.

Since we further have ReSBDS $=_s$ LReSBDS, LReSBDS $\succ_s$ ParSBDS by Theorem 10. □

When comparing ReSBDS/LReSBDS with LexLeader and its partial methods, similar to what we have done when comparing ParSBDS and LexLeader, we give ReSBDS and LReSBDS

the inversions of the symmetries broken by the LexLeader family of methods. ReSBDS and LReSBDS are stronger in solutions 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 14. *Suppose G and H are sets of symmetries so that $\gamma \in G \Leftrightarrow \gamma^{-1} \in H$. ReSBDS $\succ_s$ LexLeader and LReSBDS $\succ_s$ LexLeader by posting G to ReSBDS and LReSBDS and breaking H by LexLeader when all search with the same static variable ordering used by LexLeader and min (max) value ordering.*

Proof. We prove that all the symmetric solutions pruned by LexLeader will always be pruned by ReSBDS. LexLeader (retaining the lexicographically least solution) adds the following constraints

$$\langle x_1, \dots x_n \rangle \leq_{lex} \langle x_1, \dots x_n \rangle^{\sigma^{-1} \circ \theta^{-1}} \tag{8}$$

at the root node for all $\gamma^{-1} \equiv \sigma^{-1} \circ \theta^{-1}$ in H where n is the number of variables, σ^{-1} is a variable symmetry and θ^{-1} is a value symmetry. Each of these constraints implies the following n constraints

$$(x_1 = \theta^{-1}(x_{\sigma(1)})) \wedge \dots \wedge (x_{k-1} = \theta^{-1}(x_{\sigma(k-1)})) \rightarrow (x_k \leq \theta^{-1}(x_{\sigma(k)})) \tag{9}$$

where $k \in \{1, \dots, n\}$. ReSBDS adds the following constraint for all $\gamma \equiv \sigma \circ \theta$ in G

$$(x_{\sigma(1)} = \theta(v_1)) \wedge \dots \wedge (x_{\sigma(k-1)} = \theta(v_{k-1})) \rightarrow (x_{\sigma(r)} \neq \theta(v_r)) \tag{10}$$

at a node P_0 after assigning $k-1$ variables with the backtracking or violated recorded assignment $x_r = v_r$.

Suppose solution s_1 is pruned by the LexLeader constraints (8). There must exists a solution s_2 (pruned or not) and a symmetry $(\sigma')^{-1} \circ (\theta')^{-1}$ in H such that $s_2 = s_1^{(\sigma')^{-1} \circ (\theta')^{-1}}$ and $s_1 >_{lex} s_2$. Suppose $s_1 \equiv (x_1 = a_1 \wedge \dots \wedge x_n = a_n)$ and s_1 has the first $t-1$ assignments (according to variable order) same with s_2. We must have

$$(a_{\sigma'(1)} = \theta'(a_1)) \wedge \dots \wedge (a_{\sigma'(t-1)} = \theta'(a_{t-1})) \rightarrow (a_t > (\theta')^{-1}(a_{\sigma'(t)})) \tag{11}$$

since $s_1 >_{lex} s_2$. *Suppose s_1 cannot be pruned by ReSBDS.* With the static variable ordering used by LexLeader and min value ordering, s_2 is searched earlier than s_1 since $s_1 >_{lex} s_2$. *Assume* node P_i is the deepest common ancestor of s_1 and s_2 with the partial assignment $x_1 = a_1 \wedge \dots \wedge x_{t-1} = a_{t-1}$ and the branching variable x_t and value $(\theta')^{-1}(a_{\sigma'(t)})$ respectively. There are two cases to consider.

1. The node $cons(P_i \cup \{x_t = (\theta')^{-1}(a_{\sigma'(t)})\})$ is not pruned. Upon backtracking from $cons(P_i \cup \{x_t = (\theta')^{-1}(a_{\sigma'(t)})\})$, ReSBDS adds symmetry breaking constraint

   $$(x_{\sigma'(1)} = \theta'(a_1)) \wedge \dots \wedge (x_{\sigma'(t-1)} = \theta'(a_{t-1})) \rightarrow (x_{\sigma'(t)} \neq \theta'((\theta')^{-1}(a_{\sigma'(t)}))) \tag{12}$$

   according to the symmetry $\sigma' \circ \theta' \in G$, which can be rewritten as

   $$(x_{\sigma'(1)} = \theta'(a_1)) \wedge \dots \wedge (x_{\sigma'(t-1)} = \theta'(a_{t-1})) \rightarrow (x_{\sigma'(t)} \neq a_{\sigma'(t)}). \tag{13}$$

   Thus solution $s_1 \equiv (x_1 = a_1 \wedge \dots \wedge x_n = a_n)$ is pruned. *CONTRADICTION.*

2. The node $P_i \wedge \{x_t = (\theta')^{-1}(a_{\sigma'(t)})\}$ is pruned. It is pruned because $x_t = (\theta')^{-1}(a_{\sigma'(t)})$ is violated by the propagation of a symmetry breaking constraint at P_i or an ancestor node of P_i since there exists a solution s_2 in the subtree. Thus assignment $x_t = (\theta')^{-1}(a_{\sigma'(t)})$ must have been recorded in T by ReSBDS at the time it is violated. Suppose $x_t = (\theta')^{-1}(a_{\sigma'(t)})$ is violated at node P_j with the partial assignment $x_1 = a_1 \wedge \cdots \wedge x_{s-1} = a_{s-1}$ where $s \leq t$. ReSBDS adds the following constraint

$$(x_{\sigma'(1)} = \theta'(a_1)) \wedge \cdots \wedge (x_{\sigma'(s-1)} = \theta'(a_{s-1})) \rightarrow (x_{\sigma'(t)} \neq \theta'((\theta')^{-1}(a_{\sigma'(t)}))) \quad (14)$$

when this violation happens according to the symmetry $\sigma' \circ \theta' \in G$, which can be rewritten as

$$(x_{\sigma'(1)} = \theta'(a_1)) \wedge \cdots \wedge (x_{\sigma'(s-1)} = \theta'(a_{s-1})) \rightarrow (x_{\sigma'(t)} \neq a_{\sigma'(t)}). \quad (15)$$

Thus solution $s_1 \equiv (x_1 = a_1 \wedge \cdots \wedge x_n = a_n)$ is pruned. *CONTRADICTION*.

To show strictly stronger, we consider the $n \times n$ unconstrained matrix problem with $n = 2$ and $d = 3$. Given only two generators $\{R, C\}$ where $R^{-1} = R$ and $C^{-1} = C$, LexLeader leaves 29 solutions while ReSBDS leaves 28 solutions. Therefore ReSBDS $\succ_s$ ParSBDS.

Theorem 10 further shows ReSBDS $=_s$ LReSBDS. Thus LReSBDS $\succ_s$ LexLeader follows directly. □

The DoubleLex [4] method, a special case of LexLeader, breaks adjacent rows and columns interchangeability in matrix problems. Following directly from Theorem 14, we show that when ReSBDS and LReSBDS are given the same adjacent rows and columns interchangeability symmetries, and searching with row-wise or column-wise variable ordering and min (max) value ordering, ReSBDS and LReSBDS are strictly stronger in solutions pruning than DoubleLex.

Theorem 15. *ReSBDS $\succ_s$ DoubleLex and LReSBDS $\succ_s$ DoubleLex.*

Proof. ReSBDS $\succeq_s$ DoubleLex and LReSBDS $\succeq_s$ DoubleLex following directly from Theorem 14. To show strictness, consider the ECCLD results in Table 6. **ReSBDS** and **LReSBDS** are given the same subset of symmetries as that to **DoubleLex**, but ReSBDS and LReSBDS leave less solutions than DoubleLex does for all cases. □

Another partial symmetry breaking method of LexLeader, SnakeLex [21], breaks the same subset of symmetries as DoubleLex as well as an extra linear number of symmetries that rows (columns) with distance two are interchangeable. Following directly from Theorem 14, we show that when ReSBDS and LReSBDS are given the same generator symmetries as SnakeLex, and searching with snake-wise ordering and min (max) value ordering, ReSBDS and LReSBDS are strictly stronger in solutions pruning than SnakeLex.

Theorem 16. *ReSBDS $\succ_s$ SnakeLex and LReSBDS $\succ_s$ SnakeLex.*

Proof. ReSBDS $\succeq_s$ SnakeLex and LReSBDS $\succeq_s$ SnakeLex following directly from Theorem 14. To show strictness, consider the ECCLD results in Table 7. **ReSBDS** and **LReSBDS** are given the same subset of symmetries as that to **SnakeLex**, but ReSBDS and LReSBDS leave less solutions than SnakeLex does for all cases. □

Even though LexLeader leaves 57 nodes while ReSBDS leaves 55 nodes for the $n \times n$ unconstrained matrix problem with $n = 2$ and $d = 3$, ReSBDS is not stronger in nodes pruning than LexLeader in general.

Theorem 17. *Given the same set of symmetries. ReSBDS/LReSBDS $\nparallel_n$ LexLeader when both search with the same static variable ordering used by LexLeader and min (max) value ordering.*

Proof. ReSBDS $\succ_s$ LexLeader and LReSBDS $\succ_s$ LexLeader by Theorem 14. Thus LexLeader $\not\succ_n$ ReSBDS/LReSBDS. Next, we show ReSBDS/LReSBDS $\not\succ_n$ LexLeader using two examples that exhibit this property for two different reasons.

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} which is identical to its inverse symmetry. 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. Suppose we are at node P_0 with partial assignment $\{x_1 = 1, x_2 = 3, x_3 = 1\}$. After backtracking from $cons(P_0 \cup \{x_4 = 1\})$, the problem constraint would prune value 3 from $D(x_5)$ and $D(x_6)$. $D(x_5)$ is $\{1, 2\}$, $x_2 \leq x_5$ is false. LexLeader thus guarantees $x_1 < x_6$. This prunes value 1 from $D(x_6)$. The problem constraint again prunes value 3 from $D(x_4)$ and value 2 from $D(x_5)$. The solution $\{1, 3, 1, 2, 1, 2\}$ is found. ReSBDS adds the following two constraints

$$x_6 = 1 \rightarrow x_5 \neq 1,\ x_6 = 1 \rightarrow x_5 \neq 2$$

along branching to P_0. Even though $D(x_5)$ is $\{1, 2\}$, these two constraints cannot prune value 1 from $D(x_6)$. Both ReSBDS and LexLeader return 84 solutions, but LexLeader has 175 nodes in the search tree while ReSBDS has 187. The reason is because LexLeader adds one symmetry breaking constraint for each symmetry, while ReSBDS posts the symmetry breaking constraints separately and thus loses pruning opportunities. This shows that there are nodes pruned by LexLeader that cannot be pruned by ReSBDS, i.e. ReSBDS $\not\succ_n$ LexLeader.

Consider another CSP with variables $\{x_1, x_2\}$ and $\{0, 1\}$ as domains. The constraint is: $x_1 \neq x_2$. The two values are interchangeable. LexLeader adds constraint $\langle x_1, x_2\rangle \leq_{lex} A_{(01)}[\langle x_1, x_2\rangle]$ at the root node to break this symmetry where $A_{(01)} = [1, 0]$ and $A_{(01)}[X]$ is defined as the application of an element constraint to the variables in X. ReSBDS adds conditional symmetry breaking constraints during search. Once the LexLeader constraint is added at the root node, values 1 and 0 would be pruned from $D(x_1)$ and $D(x_2)$ respectively. ReSBDS, however, would not add constraint $x_1 \neq 1$ until backtrack happens before which x_1 is assigned the value 0 and the solution $\{x_1 = 0, x_2 = 1\}$ is found. Both ReSBDS and LexLeader return 1 solution, but LexLeader only has the root node while ReSBDS has an extra search node. The reason is LexLeader posts all the symmetry breaking constraints at the root node, while ReSBDS has to wait until there is a backtrack or a value v_i being pruned from the domain of a variable x_i and $(x_i = v_i)$ is already recorded in T. Only at that point are symmetry breaking constraints added to do the value pruning. This shows again that there are nodes pruned by LexLeader that cannot be pruned by ReSBDS, i.e. ReSBDS $\not\succ_n$ LexLeader.

Similar analysis can also show that there are nodes pruned by LexLeader that cannot be pruned by LReSBDS in the last two examples. Thus LReSBDS $\not\succ_n$ LexLeader.

□

Even though their nodes pruning powers are incomparable, we shall demonstrate empirically in our experiments, however, that ReSBDS/LReSBDS prunes many more symmetric solutions and also more nodes than LexLeader in practice.

LDSB [11] is an improvement of shortcut SBDS. LDSB handles only active symmetries and also their compositions. In addition, when v is pruned from the domain of variable x, LDSB not

only asserts $(x \neq v)^g$ but also $(x \neq v)^{g \circ h}$ for each active symmetries g and h. This recursive step is repeated in a breadth-first manner until no more new prunings are obtained [11]. LDSB is close in spirit to ReSBDS and has smaller overhead, but also misses important pruning opportunities since it only handles active symmetries. We give theorems to compare the nodes and solution pruning power of ReSBDS/LReSBDS and LDSB. Since LDSB [11] can deal only with variable symmetries and value symmetries, we *restrict our attention* to these symmetries.

Theorem 18. *Given the same set of variable and value symmetries to ReSBDS and LDSB. Suppose both use the same static variable and value orderings. If all non-broken symmetries at each node during search are active symmetries, ReSBDS* $=_n$ *LDSB and ReSBDS* $=_s$ *LDSB; otherwise, ReSBDS* $\succ_n$ *LDSB and ReSBDS* $\succ_s$ *LDSB.*

Proof. If all non-broken symmetries g are active, ReSBDS only adds unconditional symmetry breaking constraints and records these constraints' variable-value pairs. After posting these unconditional symmetry breaking constraints $(x \neq v)^g$, constraint propagation will immediately prune the value v^g from the domain of x^g. ReSBDS continues to add constraints $(x \neq v)^{g \circ h}$ also in a breadth-first manner. Thus the set of constraints added by ReSBDS is same as that added by LDSB when all non-broken symmetries are active symmetries. Now their pruning powers are the same.

Otherwise, LDSB breaks only active non-broken symmetries and their compositions, but ReSBDS breaks all non-broken symmetries (active and inactive) and their compositions. Hence, ReSBDS is strictly stronger. □

Theorem 19. *Given the same set of variable and value symmetries to LReSBDS and LDSB. Suppose both use the same static variable and value orderings. If all non-broken symmetries at each node during search are active symmetries and the prunings of all assignments in each unconditional symmetry breaking constraint are effected by symmetry breaking constraint, LReSBDS* $=_n$ *LDSB and LReSBDS* $=_s$ *LDSB; otherwise, LReSBDS* $\succ_n$ *LDSB and LReSBDS* $\succ_s$ *LDSB.*

Proof. If all non-broken symmetries g are active, LReSBDS only adds unconditional symmetry breaking constraints. After posting these unconditional symmetry breaking constraints $(x \neq v)^g$, constraint propagation will immediately prune the value v^g from the domain of x^g. Since these pruning is effected by symmetry breaking constraint, LReSBDS continues to add constraints $(x \neq v)^{g \circ h}$ for all non-broken symmetries h and also doing this in a breadth-first manner. Thus the set of constraints added by LReSBDS is the same as that added by LDSB when all non-broken symmetries are active symmetries. Now their pruning powers are the same.

Otherwise, LDSB breaks only active non-broken symmetries and their compositions, but LReSBDS breaks all non-broken symmetries (active and inactive) and their compositions. Hence, LReSBDS is strictly stronger. □

Even when ReSBDS and LReSBDS are given only generators, they can be complete in specific situations.

Theorem 20. *Given a fixed variable (value) ordering* Π*. Suppose* G *is the set of symmetries such that adjacent variables (values) are interchangeable. Applying ReSBDS or LReSBDS on* G *and searching with the* Π *variable (value) ordering eliminates all symmetries in* Σ*, where* Σ *is the symmetry group of* G*.*

Proof. For interchangeable variables, LexLeader can eliminate Σ by being only given G according to any Π ordering. For interchangeable values, Walsh [32] proves VALSYMBREAK(G,X) eliminates Σ, where VALSYMBREAK is a global lexicographic ordering constraint to break value symmetries. Thus, results follow directly from Theorem 14. □

Therefore, when breaking interchangeable variables and values, ReSBDS and LReSBDS only need to be given adjacent variable or value interchangeabilities to eliminate the complete symmetry group.

Theorem 21. *Suppose G is the set of symmetries such that adjacent rows (columns) are interchangeable. Applying ReSBDS/LReSBDS on G and searching with the input variable variable ordering and static value ordering eliminates all symmetries in Σ, where Σ is the symmetry group of G.*

Proof. For interchangeable rows (columns), LexLeader can eliminate Σ by being only given G according to the input variable variable ordering and static value ordering. Thus, results follow directly from Theorem 14. □

Therefore, when breaking interchangeable rows and columns, ReSBDS and LReSBDS only need to be given adjacent row or column interchangeabilities to eliminate the complete symmetry group.

5. Experimental Results

This section reports experiments on five satisfaction problems and three optimization problems. In case of partial symmetry breaking, the symmetries to break are some generators of the entire symmetry group. Our ReSBDS and LReSBDS implementations are modified directly from the code base of SBDS and ParSBDS. When we propagate symmetry breaking constraints which are actually nogoods, dynamic subscriptions (dynamic event sets [33], also known as dynamic triggers [34]) are used.

We compare ReSBDS and LReSBDS against SBDS, ParSBDS and LDSB as well as state of the art static symmetry breaking methods: (a) two partial symmetry breaking variants of LexLeader, DoubleLex [4] and SnakeLex [21] for breaking matrix symmetries, (b) value precedence [18] for breaking value interchangeability, (c) the SIGLEX constraint [19] for breaking variable and value interchangeability and its descending-partition-size variation and (d) the static method by Puget [17, 35] to break all variable symmetries in all-different problems and all value symmetries in surjection problems. We do not compare with allperm [16] since its implementation is not available. All experiments are conducted using Gecode Solver 4.2.0 on Xeon E5620 2.4GHz processors.

We also compare against other state of the art dynamic symmetry breaking methods: symmetry breaking with lazy clause generation [36], GE-trees [8], GAP-SBDD [37] and GAP-SBDS [38]. Here, we can do the comparison only indirectly for these four methods by using results from the literature since they are not available on the Gecode platform. In Table 2, we give the Passmark CPU marks [39] on the reported CPUs in the literature and the CPU of our experimental machines. The higher the mark, the better the efficiency of the CPU.

We compare against SBDS, ParSBDS and LDSB since they are state of the art dynamic methods and also starting point of our work. The other methods chosen for comparison are best static and dynamic methods on handling the respective problems in the literature.

Table 2: Passmark CPU Marks on CPUs

CPU Name	Passmark CPU Mark
2.6GHz Pentium IV processor	289
600MHz Intel PIII processor	238
Xeon Pro 2.4GHz processor	4875
Xeon E5620 2.4GHz processor	4875

Although Mears et al. [11] 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) , $\#bt$ denotes the number of backtrack 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. **SBDS** uses SBDS to break *all* symmetries. **DoubleLex** and **SnakeLex** lexicographically orders variable sequences in increasing order according to the DoubleLex and SnakeLex methods. **ParSBDS**, **LDSB ReSBDS** and **LReSBDS** handle the given symmetries by ParSBDS, LDSB, ReSBDS and LReSBDS respectively. *Unless otherwise specified*, the search order is defaulted to input variable order and min value order.

5.1. Satisfaction Problems

In this part, we give five experiments to show the benefit of our methods on finding all solutions in satisfaction problems.

5.1.1. N-Queens

The N-Queens problem is prob054 in CSPLib [40], which is to place N queens on an $N \times N$ chessboard so that none of the queens can attack each other. We model the N-Queens problem the standard way using one variable per column. This model has 8 geometric symmetries which are classified into variable symmetries, value symmetries and variable-value symmetries. All 8 geometric symmetries are given to **SBDS**. We give **ParSBDS**, **ReSBDS** and **LReSBDS** only the two generators rx (reflection on the vertical axis) and $d1$ (reflection on the diagonal), which can generate all 8 geometric symmetries. Note that all the symmetry breaking constraints added by **ReSBDS** and **LReSBDS** according to these two generators are unconditional constraints. This means that all non-broken symmetries at each node during search are active symmetries. Thus the pruning power of **ReSBDS** and **LReSBDS** are identical according to Lemma 2. We thus show their solutions and failures together and use T_R and T_L to denote the runtime of **ReSBDS** and **LReSBDS** respectively. **LDSB** can only handle the rx, ry and $r180$ symmetries, which are reflections on the vertical and horizontal axes and rotation of 180 degrees. They form a symmetry group which is a subset of the geometric symmetry group. Following Mears et al. [11], we give **LDSB** only the two generators rx and ry. **SBC** uses the static method by Puget [17, 35] to break all variable symmetries in all-different problems and all value symmetries. Thus only rx and ry can be broken by SBC.

Table 3 shows the results. **SBC** eliminates the smallest set of symmetric solutions and search parts. Being given two symmetries, **ReSBDS** and **LReSBDS** achieve over 63% reduction in

Table 3: N-Queens (all solutions)

N	**SBC**			**SBDS**			**ParSBDS**		
	$\#s$	$\#f$	t	$\#s$	$\#f$	t	$\#s$	$\#f$	t
14	141,988	1,819,871	7.25	**45,752**	**823,621**	5.92	140,438	1,361,836	6.64
15	940,824	11,242,568	45.38	**285,053**	**4,825,631**	34.33	859,654	7,951,827	39.50
16	5,726,294	67,306,747	268.22	**1,846,955**	**30,221,334**	219.05	5,646,963	50,928,933	265.16
17	39,043,343	469,901,695	1,891.50	**11,977,939**	**200,005,686**	1,502.88	36,078,885	337,758,336	1,796.78

N	**LDSB**			**ReSBDS/LReSBDS**			
	$\#s$	$\#f$	t	$\#s$	$\#f$	t_R	t_L
14	99,883	1,454,958	7.75	51,876	875,600	4.50	**4.29**
15	613,978	8,452,568	45.43	324,173	5,131,707	25.70	**24.13**
16	3,985,771	53,277,940	288.97	2,071,568	31,934,685	164.33	**157.33**
17	25,549,837	351,039,264	1,915.12	13,388,788	211,409,624	1,109.95	**1,062.43**

solution size and over 36% in failures when compared to **ParSBDS**. In terms of runtime, **ReSBDS** and **LReSBDS** are 1.56 and 1.64 times faster than **ParSBDS** on average respectively. This shows the additional constraints we add can break more compositions and prune more symmetric subtrees in an efficient way. **LReSBDS** runs 1.05 times faster than **ReSBDS** on average since there is no need to record and check the violations of assignments in T. The saving is not that prominent since there are only two symmetries added and both of them are active symmetries if they are not broken yet during search. **SBDS** eliminates the symmetries in N-Queens completely and has the smallest solution size and search tree. **ReSBDS** and **LReSBDS**, which break the symmetries only partially, are 1.34 and 1.40 times faster than **SBDS** on average respectively. This shows partial symmetry breaking method can be efficient even for polynomially symmetric CSPs [10]. **LDSB** performs worse than **ParSBDS** in search tree size and runtime. This example demonstrates the main advantages of **ReSBDS** and **LReSBDS**: flexibility in choosing symmetries to break, and good balance in overhead and extra pruning. For **ReSBDS**, the maximum size of T is 2. In more than 99.99% of the cases, the size of T is 0.

5.1.2. Graceful Graph

The graceful graph problem is prob053 in CSPLib [40], which is to find a labelling f of the nodes of a graph with q edges so that each node is assigned a unique label from $0, \ldots, q$ and when each edge xy is labelled with $f(x) - f(y)$, the edge labels are all different. The graceful graph problem is an all-different problem [9]. We model this problem by one variable for a node. For $K_n \times P_m$ graph, it has intra-clique permutations, inter-clique permutations, complement symmetry, and their combinations. All variable and value symmetries and their combinations are given to **SBDS**. **ParSBDS** is given $n * (n-1)/2$ symmetries to describe any two nodes in each clique being permutable simultaneously and two more symmetries to describe inter-clique permutation and complement symmetry.

ReSBDS, **LReSBDS** and **LDSB** are given $(n-1)$ symmetries to describe simultaneous permutation of adjacent nodes in each clique and also one inter-clique permutation and one complement symmetry. The intra-clique permutations and inter-clique permutations are actually row and column symmetries in this model. All row and column interchangeabilities are eliminated by ReSBDS and LReSBDS when given only interchangeabilities of adjacent rows (columns)

Table 4: Graceful Graph (all solutions)

n, m	**SBC**			**GAP-SBDD**			**GAP-SBDS**			**LDSB**		
	$\#s$	$\#f$	t	$\#s$	$\#bt$	t	$\#s$	$\#bt$	t	$\#s$	$\#f$	t
3,2	8	652	0.01	**4**	13	0.73	**4**	9	0.53	8	470	0.01
4,2	30	40,749	0.62	**15**	173	9.9	**15**	165	8.5	30	27,103	0.57
5,2	2	2,735,546	34.31	**1**	4,402	426.89	**1**	4,390	388.71	2	1,794,653	31.84
6,2	**0**	164,725,514	2,674.42							**0**	**103,682,981**	2,299.51
3,3	568	200,500	1.83							380	128,151	1.51
4,3	1,408	116,706,469	1,225.29							958	72,564,505	1,032.01
2,4	354	12,666	0.13							233	7,678	0.11
3,4	25,508	109,204,827	949.21							16,625	63,845,298	720.35

n, m	**SBDS**			**ParSBDS**			**ReSBDS/LReSBDS**			
	$\#s$	$\#f$	t	$\#s$	$\#f$	t	$\#s$	$\#f$	t_R	t_L
3,2	**4**	**343**	0.01	16	797	0.01	8	470	0.01	**0.00**
4,2	**15**	**21,140**	1.35	60	54,762	0.81	30	27,103	0.53	**0.30**
5,2	**1**	**1,421,771**	363.03	4	3,668,283	48.26	2	1,794,653	26.12	**25.80**
6,2	–	–	–	–	–	–	**0**	**103,682,981**	1,979.78	**1,941.06**
3,3	**284**	**109,808**	2.68	761	224,060	2.09	380	128,151	1.38	**1.32**
4,3	**661**	**47,742,168**	3,600.00	2,046	135,853,331	1,565.89	958	72,564,505	911.36	**904.44**
2,4	**177**	**6,980**	0.12	484	14,018	0.11	233	7,678	0.06	**0.06**
3,4	**12,754**	**56,727,197**	1,681.39	41,559	117,618,370	1,138.84	16,625	63,845,298	662.61	**636.14**

according to Theorem 21. Posting any extra symmetries to ReSBDS and LReSBDS as in the case of ParSBDS is fruitless. We have also tried giving the same symmetries for **ReSBDS** to **ParSBDS**. In this case, **ParSBDS** fails to solve most of the problem instances within the time limit. Similarly, LDSB needs to be given only adjacent symmetries due to its power in pruning composition symmetries.

Note that all the symmetry breaking constraints added by **ReSBDS** and **LReSBDS** according to these two generators are unconditional constraints. This means that all non-broken symmetries at each node during search are active symmetries. Thus the pruning power of **ReSBDS**, **LReSBDS** and **LDSB** are identical according to Lemma 2. We thus show the solutions and failures of **ReSBDS** and **LReSBDS** together and use t_R and t_L to denote the runtime of **ReSBDS** and **LReSBDS** respectively. **SBC** uses the static method by Puget [17, 35] to break all variable symmetries in all-different problems and all value symmetries. For **GAP-SBDD** [37] and **GAP-SBDS** [38], we replicate their results from the literature [41].

Table 4 shows the results. Since we run more instances than those reported in the literature, instances not tested by **GAP-SBDD** and **GAP-SBDS** are given empty entries in the table. **SBDS** eliminates all symmetries and has the smallest solution size and search tree. **ReSBDS** and **LReSBDS**, which break the symmetries only partially, are 4.05 and 4.29 times faster than **SBDS** on average respectively. **LReSBDS** performs the best and runs slightly faster than **ReSBDS**. When comparing with the other two complete methods, **GAP-SBDD** is solved under a 2.6GHz Pentium IV processor and **GAP-SBDS** is solved under a 600MHz Intel PIII processor. According to Table 2, our CPU (Xeon E5620 2.4GHz processor) is 16.87 and 20.48 times faster than the two reported CPUs respectively. However, **LReSBDS** runs 100 and 80 times faster than **GAP-SBDD** and **GAP-SBDS**. This shows the gains of our efficient partial symmetry breaking

methods. Being given a smaller subset of symmetries, **ReSBDS** and **LReSBDS** achieve over 46% reduction in solution size and over 47% in failures when compared to **ParSBDS**. In terms of runtime, **ReSBDS** and **LReSBDS** are 1.68 and 1.89 times faster than **ParSBDS** on average. This demonstrates ReSBDS and LReSBDS break more composition symmetries and prune more symmetric subtrees in an efficient way. **LDSB** has the same performance with **ReSBDS** and **LReSBDS** in number of solutions left and search tree size. This demonstrates Theorem 18. The runtime of **LDSB**, however, are 1.24 and 1.37 times slower than **ReSBDS** and **LReSBDS** on average respectively due to the high overheads to handle symmetries. **ReSBDS** and **LReSBDS** better the static partial method **SBC** by 19% reduction in solution size on average, 37% reduction in failures on average and 1.40 and 1.47 times faster in runtime on average respectively. For **ReSBDS**, the maximum size of T is 4. In 99.11% of the cases on average, the size of T is 0.

5.1.3. The $n \times n$ Queen Problem

The $n \times n$ queen problem is to color an $n \times n$ chessboard with n colors, such that no line (row, column or diagonal) contains the same color twice [42]. This can be seen as searching for n non-intersecting solutions to the n queens problem. Each solution is given by the squares containing one of the n colors. This problem can be modeled with n^2 variables, one per square of the chess board, and one all different constraint per line. Like the N-Queens problem, it has 8 geometric symmetries. It also has value interchangeability and their compositions with variable symmetries. All these symmetries are given to **SBDS**. **ParSBDS** is given the 8 geometric symmetries and any two of values are interchangeable. **ReSBDS** and **LReSBDS** are given the 8 geometric symmetries and adjacent value interchangeability. All value interchangeabilities have been eliminated by ReSBDS and LReSBDS when given only adjacent value interchangeabilities according to Theorem 20. Thus posting the extra number of value symmetries to ReSBDS and LReSBDS as in the case of ParSBDS is fruitless. Since value symmetries are inactive symmetries, LDSB needs to be given only the adjacent value interchangeabilities by similar reasoning. **LDSB** is given the 8 geometric symmetries and value interchangeability. For the static method, variable symmetries are broken by the following **VAR** constraints [43]: $x_0 < x_{n-1}, x_0 < x_{n(n-1)}, x_0 < x_{n^2-1}, x_1 < x_n$. Value symmetries are broken by **OCC** which is proposed by Puget [35] to break all value symmetries in surjection problems. The results of **GAP-SBDD** and **GE-tree** are replicated from the literature [42] where **GE-tree** uses GE-tree construction for the value symmetries and GAP-SBDD on the symmetry group for the variables. Unfortunately, the CPU/machine used in their experimentation is not reported.

Table 5 shows the results. Again, we leave the entries empty if an instance is not tested by **GE-tree** and **GAP-SBDD**. **ReSBDS** and **LReSBDS** are again much more time efficient and break more symmetries than **ParSBDS**. **ReSBDS**, **LReSBDS** and **LDSB** leave the same number of solutions and failures. This shows all non-broken symmetries are active symmetries under this searching order. But **LReSBDS** runs the fastest. The static partial method **VAR+OCC** leaves the smallest search tree size. **LReSBDS** leaves only slightly larger number of failures than **VAR+OCC**, but its runtime is 3.8 times faster on average due to its small overhead. When compared to the two complete methods, **LReSBDS** is 922 and 561 times faster than **GE-tree** and **GAP-SBDD** on average respectively. For **ReSBDS**, the maximum size of T is 4. In 87.70% of the cases on average, the size of T is 0.

5.1.4. Error Correcting Code - Lee Distance (ECCLD)

The ECCLD problem is prob036 in CSPLib [40]. The task is to find the maximum number b of codes of length n drawn from 4 symbols $\{1, 2, 3, 4\}$ such that the Lee distance between

Table 5: $n \times n$ Queens (all solutions)

N	**GE-tree**		**GAP-SBDD**		**VAR+OCC**			**ParSBDS**		
	$\#s$	t	$\#s$	t	$\#s$	$\#f$	t	$\#s$	$\#f$	t
5	**1**	0.80	**1**	0.68	2	**2**	**0.00**	2	7	**0.00**
6	**0**	1.25	**0**	0.96	**0**	**33**	**0.00**	**0**	53	**0.00**
7	**1**	10.61	**1**	8.36	4	**1,070**	0.04	4	1,787	0.02
8	**0**	2,077.23	**0**	927.36	**0**	**207,048**	9.10	**0**	387,731	3.78
9					–	–	–	–	–	–

N	**LDSB**			**ReSBDS**			**LReSBDS**		
	$\#s$	$\#f$	t	$\#s$	$\#f$	t	$\#s$	$\#f$	t
5	2	6	**0.00**	2	6	**0.00**	2	6	**0.00**
6	**0**	38	**0.00**	**0**	38	**0.00**	**0**	38	**0.00**
7	4	1,076	**0.01**	4	1,076	**0.01**	4	1,076	**0.01**
8	**0**	207,055	2.51	**0**	207,055	2.40	**0**	207,055	**2.14**
9	**0**	**232,274,005**	3,285.32	**0**	**232,274,005**	3,311.21	**0**	**232,274,005**	**2,863.27**

any pair of codes is exactly c, where the Lee distance between two symbols a and b is min $\{|a-b|, 4-|a-b|\}$. We model it into a $b \times n$ matrix with domain $\{1 \dots 4\}$. Similar to Lee and Li [12], in order to illustrate the effect on solution set size, we transform the optimization problem to a satisfaction one by setting b in advance. This model has matrix symmetries which is exponential to the problem size. We only try to break a subset of the available matrix symmetries in the problem.

In Table 6, the symmetries are given as the following. **ParSBDS** is given the symmetry that any two rows (columns) are interchangeable. **ReSBDS**, **LReSBDS** and **DoubleLex** are given interchangeability of adjacent rows (columns). **LDSB** is given the interchangeable rows and columns. Note **ParSBDS** is given more symmetries than **ReSBDS**. Given the same symmetries as **ReSBDS**, **ParSBDS** fails to solve most of the problems within the time limit. To show the improvement of LReSBDS over ReSBDS and also their flexibility in choosing symmetries to break, we also use these two methods to break interchangeability of adjacent rows (columns) as well as cartesian-product of any two rows are interchangeable and any two columns are interchangeable. Now we get **ReSBDS**^{c} and **LReSBDS**^{c} respectively.

Table 6 shows the results by using input order heuristic. **ReSBDS** and **LReSBDS** are 1.92 and 2.29 times faster than **ParSBDS** on average respectively with more symmetries broken more efficiently. **ReSBDS** has smaller number of solutions and search tree size than **DoubleLex**. The runtime of **ReSBDS**, however, does not gain too much from the smaller search tree size because of the larger overhead of **ReSBDS**. After introducing our light version, **LReSBDS** leaves the same number of solutions and almost the same number of failures as those of **ReSBDS**, and is 1.31 times faster than **DoubleLex** on average. Given more symmetries, **LReSBDS**^{c} is 1.89 times faster than **ReSBDS**^{c} and has only slightly increase on number of failures. This shows LReSBDS is much more efficient than ReSBDS and does not loss too much pruning power. Now **LReSBDS**^{c} is 2.48 times faster than **DoubleLex** on average. We can also see that **LDSB** leaves many more solutions and is drastically less efficient than **ReSBDS** and **LReSBDS**. This shows ReSBDS and LReSBDS can gain a lot by breaking also inactive symmetries and their compositions. Another advantage over LDSB is again that ReSBDS and LReSBDS have the flexibility of choosing the given symmetries to break. For **ReSBDS**, the maximum size of T is

Table 6: The ECCLD problem (all solutions)

n, c, b	**DoubleLex**			**ParSBDS**			**LDSB**		
	$\#s$	$\#f$	t	$\#s$	$\#f$	t	$\#s$	$\#f$	t
5,2,10	87	41,571	8.88	136	94,472	26.18	572,041	8,439,178	1067.64
5,6,4	710,731	725,837	27.29	849,724	900,549	41.68	770,956	822,157	32.81
6,8,4	59,158	2,469,211	58.14	62,380	2,992,204	92.80	64,262	2,588,227	66.86
4,4,8	32,469	839,251	66.10	38,235	1,073,244	109.94	430,079	4,538,862	319.43
6,4,4	4,698,842	4,139,211	182.65	7,607,152	6,726,710	376.90	13,710,850	10,718,458	531.02
5,6,5	1,441,224	5,508,192	188.63	1,661,689	6,523,348	287.50	1,944,489	7,669,846	277.34
5,6,6	297,476	11,709,068	475.63	334,519	13,579,038	715.24	538,688	19,051,585	791.28
8,4,4	35,626,714	48,525,827	2,040.84	–	–	–	–	–	–
6,4,5	29,345,816	73,522,873	3,076.87	–	–	–	–	–	–

n, c, b	**ReSBDS**			**LReSBDS**		
	$\#s$	$\#f$	t	$\#s$	$\#f$	t
5,2,10	76	33,672	6.88	76	33,672	**6.51**
5,6,4	530,819	537,118	24.79	530,819	537,118	20.30
6,8,4	45,024	1,778,978	54.87	45,024	1,778,980	43.75
4,4,8	25,543	631,717	57.19	25,543	631,717	50.13
6,4,4	3,746,439	3,278,563	191.46	3,746,439	3,278,563	146.96
5,6,5	1,044,674	4,005,819	170.31	1,044,674	4,005,819	141.16
5,6,6	221,236	8,553,011	449.17	221,236	8,553,011	353.14
8,4,4	28,982,779	39,163,464	2,122.88	28,982,779	39,163,464	1,700.11
6,4,5	23,059,956	57,225,832	2,952.21	23,059,956	57,225,832	2,406.25

n, c, b	**ReSBDS**^{c}			**LReSBDS**^{c}		
	$\#s$	$\#f$	t	$\#s$	$\#f$	t
5,2,10	**56**	**20,813**	9.39	**56**	20,820	7.50
5,6,4	**235,866**	**253,358**	20.46	**235,866**	253,729	**10.71**
6,8,4	**18,933**	**792,644**	54.55	**18,933**	793,258	**23.53**
4,4,8	**7,698**	**231,956**	29.03	**7,698**	235,074	**21.16**
6,4,4	**1,608,536**	**1,568,196**	181.52	**1,608,536**	1,568,776	**85.79**
5,6,5	**392,221**	**1,611,688**	112.74	**392,221**	1,614,088	**63.82**
5,6,6	**72,150**	**3,204,456**	262.61	**72,150**	3,210,014	**151.61**
8,4,4	**11,582,467**	**17,462,366**	2,865.98	**11,582,467**	17,464,022	**1,069.40**
6,4,5	**7,631,833**	**20,481,227**	2,033.60	**7,631,833**	20,494,554	**1,073.90**

99, and the average size is 1.16. In 55.91% of the cases, the size of T is 0. For **ReSBDS**^{c}, the maximum size of T is 99, and the average size is 1.97. In 52.67% of the cases, the size of T is 0.

To compare with LexLeader in another variable order, we do the following experiments. In Table 7, the variable heuristic is row-wise snake order. We choose row-wise snake order rather than column-wise snake order because the static method **SnakeLex** performs better in row-wise snake order for this problem. **ParSBDS** is given the same subset of symmetries as in Table 6. **ReSBDS**, **LReSBDS** and **SnakeLex** are given interchangeability of adjacent rows (columns) and interchangeability of rows with distance 1 which is exactly the symmetries that are broken by SnakeLex. **LDSB** is given the interchangeable rows and columns. Given the same symmetries as **ReSBDS**, **ParSBDS** again fails to solve most of the problems within the time limit. **ReSBDS**^{c} and **LReSBDS**^{c} break the extra cartesian-product of any two rows are interchangeable and any two columns are interchangeable than **ReSBDS** and **LReSBDS** respectively.

Table 7 shows the results with row-wise snake ordering. **ReSBDS** and **LReSBDS** are 1.83 and 2.24 times faster than **ParSBDS** on average respectively. **ReSBDS** has smaller number of solutions and search tree size than **SnakeLex**. **LReSBDS** leaves the same number of solutions and almost the same number of failures as those of **ReSBDS** but runs faster, and is 1.33 times faster than **SnakeLex** on average. Given more symmetries, **LReSBDS**^{c} is 1.56 times faster than **ReSBDS**^{c} and has only slightly increase on number of failures. This shows LReSBDS is much more efficient than ReSBDS and does not loss too much pruning power. Note that for the last instance, **LReSBDS**^{c} leaves more solutions than **ReSBDS**^{c}. This is because the heuristic is not a static one. If we use static input variable order, they all leave 7,631,833 number of solutions. Now **LReSBDS**^{c} is 2.08 times faster than **SnakeLex** on average. Again, **LDSB** leaves many more solutions and is drastically less efficient than **ReSBDS** and **LReSBDS**. For **ReSBDS**, the maximum size of T is 91, and the average size is 1.32. In 54.71% of the cases, the size of T is 0. For **ReSBDS**^{c}, the maximum size of T is 105, and the average size is 1.62. In 53.42% of the cases, the size of T is 0.

5.1.5. Cover Array Problem (CA)

The Cover Array Problem CA(t, k, g, b), prob045 in CSPLib [40], is to construct a $k \times b$ array A over $Z_g = 0, 1, 2, \ldots, g-1$ with the property that for any t distinct rows $1 \leq r_1 \leq r_2 \leq \cdots \leq r_t \leq k$, and any member $(x_1, x_2, \ldots, x_t)$ of Z_g there exists at least one column c such that x_i equals the $(r_i, c) - th$ element of A for all $1 \leq i \leq t$. We use the integrated model [44], which channels an original model and a compound model. This model also has matrix symmetries. We again only try to break a subset of the available matrix symmetries in the problem for each method which is same as that for ECCLD.

Table 8 shows the results by using input order heuristic. **ReSBDS** and **LReSBDS** are 2.23 and 3.08 times faster than **ParSBDS** on average respectively with more symmetries broken. **ReSBDS** and **LReSBDS** have smaller number of solutions and search tree size than **DoubleLex**. They gain little from the smaller search tree size because of the larger overhead of dynamically generating and handling nogoods even though **LReSBDS** is 1.39 times faster than **ReSBDS** on average. Given more symmetries, **LReSBDS**^{c} is 1.60 times faster than **ReSBDS**^{c} with almost the same number of failures. This shows again LReSBDS is much more efficient than ReSBDS and does not lose too much pruning power. Now **LReSBDS**^{c} is 1.75 times faster than **DoubleLex** on average. **LDSB** cannot solve any of the instances. This shows ReSBDS and LReSBDS can gain a lot by breaking also inactive symmetries and their compositions. For **ReSBDS**, the maximum size of T is 74, and the average size is 1.28. In 58.14% of the cases, the size of T is 0. For

Table 7: The ECCLD problem with row-wise snake ordering (all solutions)

n, c, b	**SnakeLex**			**ParSBDS**			**LDSB**		
	$\#s$	$\#f$	t	$\#s$	$\#f$	t	$\#s$	$\#f$	t
5,2,10	107	93,517	14.07	108	176,753	37.88	875,690	11,018,305	1198.45
5,6,4	748,248	1,093,588	35.92	822,094	1,247,706	54.58	872,705	1,140,871	40.91
6,8,4	55,618	3,788,403	91.42	62,215	4,235,825	142.69	60,845	3,598,001	92.04
4,4,8	29,384	1,116,810	84.67	32,566	1,300,117	140.41	402,315	4,687,018	335.52
6,4,4	5,061,729	6,065,447	215.97	7,181,939	8,802,453	427.19	9,940,119	7,321,232	346.34
5,6,5	1,468,811	8,309,086	280.82	1,599,097	9,479,158	440.63	2,343,133	11,729,808	420.77
5,6,6	299,821	17,799,185	750.26	314,523	19,914,985	1,151.36	701,349	29,994,954	1,255.18
8,4,4	38,629,753	70,340,164	2,396.48	–	–	–	–	–	–
6,4,5	29,668,229	84,182,877	3,257.11	–	–	–	–	–	–

n, c, b	**ReSBDS**			**LReSBDS**		
	$\#s$	$\#f$	t	$\#s$	$\#f$	t
5,2,10	76	64,308	12.46	76	64,358	10.01
5,6,4	545,702	799,420	36.03	545,702	807,629	29.31
6,8,4	45,890	2,766,514	98.51	45,890	2,780,946	80.53
4,4,8	20,714	734,565	77.62	20,714	764,219	63.07
6,4,4	3,187,304	3,637,032	181.79	3,187,304	3,687,033	150.47
5,6,5	1,085,771	6,097,164	281.18	1,085,771	6,186,192	234.19
5,6,6	223,894	12,955,195	779.37	223,894	13,178,222	618.44
8,4,4	23,786,633	39,901,361	1,922.32	23,786,633	40,151,576	1,599.00
6,4,5	17,861,886	50,359,248	2,568.60	17,861,886	50,954,100	2,142.48

n, c, b	**ReSBDS**^{c}			**LReSBDS**^{c}		
	$\#s$	$\#f$	t	$\#s$	$\#f$	t
5,2,10	**56**	**38,037**	12.12	**56**	38,116	**9.18**
5,6,4	**231,826**	**423,171**	28.70	**231,826**	430,280	**18.06**
6,8,4	**18,422**	**1,410,697**	99.70	**18,422**	1,422,932	**55.47**
4,4,8	**7,773**	**309,456**	42.52	**7,773**	326,890	**31.73**
6,4,4	**1,610,913**	**2,027,286**	192.32	**1,610,913**	2,070,904	**117.29**
5,6,5	**379,167**	**2,470,662**	174.91	**379,167**	2,527,264	**117.96**
5,6,6	**68,001**	**4,842,499**	427.90	**68,001**	4,966,572	**277.87**
8,4,4	**11,650,172**	**21,915,319**	2,813.94	**11,650,172**	22,160,606	**1,632.08**
6,4,5	**7,665,345**	**23,327,437**	2,090.00	7,665,359	23,707,465	**1,289.86**

Table 8: The CA problem (all solutions)

t,k,g,b	**DoubleLex**			**ParSBDS**			**LDSB**		
	$\#s$	$\#f$	t	$\#s$	$\#f$	t	$\#s$	$\#f$	t
2,3,3,11	6,824	778,271	9.81	7,111	839,993	37.25	–	–	–
2,4,4,16	3,456	661,726	32.31	3,456	661,919	80.37	–	–	–
2,3,3,12	86,960	4,195,254	50.87	91,905	4,538,035	211.67	–	–	–
2,3,3,13	820,844	16,611,790	202.16	875,686	17,974,093	880.72	–	–	–
2,3,5,25	161,280	16,542,398	580.65	161,280	16,542,485	2,282.55	–	–	–
2,3,3,14	6,096,380	57,811,324	722.98	6,543,422	62,503,556	3,262.27	–	–	–
2,3,3,15	37,309,730	188,015,587	2,493.98	–	–	–	–	–	–

t,k,g,b	**ReSBDS**			**LReSBDS**		
	$\#s$	$\#f$	t	$\#s$	$\#f$	t
2,3,3,11	6,676	750,606	18.98	6,676	750,606	12.68
2,4,4,16	3,456	659,587	40.59	3,456	659,587	32.77
2,3,3,12	84,464	4,046,266	97.81	84,464	4,046,266	67.90
2,3,3,13	794,015	16,037,217	389.42	794,015	16,037,217	271.50
2,3,5,25	161,280	16,541,873	849.22	161,280	16,541,873	638.84
2,3,3,14	5,884,689	55,870,750	1,411.71	5,884,689	55,870,750	1,026.92
2,3,3,15	–	–	–	35,982,708	181,894,353	3,369.66

t,k,g,b	**ReSBDS**^{c}			**LReSBDS**^{c}		
	$\#s$	$\#f$	t	$\#s$	$\#f$	t
2,3,3,11	**2,808**	**346,073**	11.82	**2,808**	**346,073**	**7.07**
2,4,4,16	**424**	**120,799**	13.62	**424**	120,815	**9.98**
2,3,3,12	**34,147**	**1,861,886**	62.80	**34,147**	**1,861,886**	**37.76**
2,3,3,13	**314,250**	**7,272,149**	250.12	**314,250**	**7,272,149**	**150.43**
2,3,5,25	**51,328**	**5,401,345**	397.89	**51,328**	**5,401,345**	**265.19**
2,3,3,14	**2,306,002**	**24,838,306**	878.16	**2,306,002**	**24,838,306**	**526.12**
2,3,3,15	**14,064,057**	**79,305,753**	2,944.38	**14,064,057**	**79,305,753**	**1,778.12**

Table 9: The CA problem with column-wise snake ordering (all solutions)

t,k,g,b	**SnakeLex**			**ParSBDS**			**LDSB**		
	$\#s$	$\#f$	t	$\#s$	$\#f$	t	$\#s$	$\#f$	t
2,3,3,11	6,905	5,641	**0.12**	7,111	5,588	0.23	7,182	5,650	0.14
2,4,4,16	3,456	11,705	**0.73**	3,456	12,622	2.03	4,158	12,093	0.79
2,3,3,12	88,515	55,203	**1.16**	91,903	54,568	2.39	93,042	55,451	1.47
2,3,3,13	839,869	445,851	**10.30**	875,645	440,645	21.81	887,409	447,618	12.73
2,3,5,25	161,280	21,893	**5.78**	161,280	21,893	9.01	161,280	21,893	6.02
2,3,3,14	6,265,234	2,939,321	**73.30**	6,542,989	2,906,360	165.32	6,632,928	2,945,668	92.75
2,3,3,15	38,484,136	16,175,606	**427.60**	40,199,167	16,004,730	1,053.86	40,748,373	16,180,801	541.86

t,k,g,b	**ReSBDS**			**LReSBDS**		
	$\#s$	$\#f$	t	$\#s$	$\#f$	t
2,3,3,11	6,676	5,254	0.21	6,676	5,254	0.14
2,4,4,16	3,456	12,442	2.11	3,456	12,622	0.83
2,3,3,12	84,464	51,213	2.03	84,464	51,213	1.43
2,3,3,13	794,015	414,400	17.52	794,015	414,400	12.45
2,3,5,25	161,280	21,893	9.46	161,280	21,893	6.88
2,3,3,14	5,884,689	2,744,326	125.79	5,884,689	2,744,326	88.24
2,3,3,15	35,982,708	15,185,725	743.58	35,982,708	15,185,725	528.70

t,k,g,b	**ReSBDS**^{c}			**LReSBDS**^{c}		
	$\#s$	$\#f$	t	$\#s$	$\#f$	t
2,3,3,11	**5,643**	**4,889**	1.19	**5,643**	**4,889**	0.46
2,4,4,16	**3,456**	**12,428**	14.90	**3,456**	12,608	9.65
2,3,3,12	**68,133**	**48,113**	11.43	**68,133**	**48,113**	3.84
2,3,3,13	**620,802**	**382,688**	100.22	**620,802**	**382,688**	31.37
2,3,5,25	**161,280**	**21,656**	695.91	**161,280**	**21,656**	436.66
2,3,3,14	**4,505,830**	**2,484,332**	743.61	**4,505,830**	**2,484,332**	233.97
2,3,3,15	–	–	–	**27,177,359**	**13,514,690**	1,369.44

ReSBDS^c, the maximum size of T is 74, and the average size is 1.48. In 55.63% of the cases, the size of T is 0.

Table 9 shows the results with column-wise snake ordering. We choose column-wise snake order rather than row-wise snake order because the static method **SnakeLex** performs better in column-wise snake order for this problem. **ReSBDS** and **LReSBDS** are 1.13 and 1.77 times faster than **ParSBDS** on average respectively. **ReSBDS** and **LReSBDS** have smaller number of solutions and search tree size than **SnakeLex** but no gains on the runtime due to their larger overheads. **LReSBDS** is 1.60 times faster than **ReSBDS** on average. Given more symmetries, **LReSBDS**^{c} is 2.53 times faster than **ReSBDS**^{c} and has almost the same number of failures. This shows again LReSBDS is much more efficient than ReSBDS and does not loss too much pruning power. Given more symmetries, ReSBDS and LReSBDS cannot further prune many more symmetries so that the reduction of the search tree size cannot overweight the overhead of handling more symmetry breaking constraints. However, **LDSB** leaves only slightly number of solutions than **ReSBDS** and **LReSBDS** which shows that most non-broken symmetries are active ones under this search order. For **ReSBDS**, the maximum size of T is 74, and the average size is 1.75. In 55.77% of the cases, the size of T is 0. For **ReSBDS**^{c}, the maximum size of T is

74, and the average size is 1.92. In 55.10% of the cases, the size of T is 0.

5.2. Optimization Problems

A *Constraint Optimization Problem* (COP) P is a tuple (X, D, C) with an *objective function* f that maps full assignments to real numbers. An *optimal solution* s is a solution to (X, D, C) such that $f(s) \leq f(s')$ for all other solutions s' of (X, D, C). We note that maximization can be easily reformulated into a minimization problem. In this part, we give three experiments to show the benefit of our methods on finding optimal solutions in optimization problems. Optimization problems are quite sensitive to search heuristics. We employ dynamic heuristics for solving all benchmarks in this section.

5.2.1. Concert Hall Scheduling

Concert hall scheduling [18] 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 [19], 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 [18] for breaking value interchangeability and LexLeader for breaking variable interchangeability (**Precede**), SIGLEX constraint (**Siglex**) for breaking variable and value interchangeability [19] and its descending-partition-size variation (**Siglexdec**). **ParSBDS** are given two sets of symmetries: any two variables within each partition being interchangeable and any two values being interchangeable. All variable interchangeabilities are eliminated by ReSBDS and LReSBDS when given only adjacent variable interchangeabilities according to Theorem 20. Thus posting the composition variable symmetries to ReSBDS and LReSBDS is fruitless. **ReSBDS** and **LReSBDS** are given two sets of symmetries: adjacent variables within each partition being interchangeable and adjacent values being interchangeable. Thus **ReSBDS** and **LReSBDS** only adds unconditional constraints and have the same pruning power according to Lemma 2. According to the symmetry pattern syntax, we give **LDSB** variable interchangeabilities and value interchangeabilities.

This problem is quite sensitive to search heuristics. For each symmetry breaking method, we experiment with several commonly used heuristics in the literature and report the best heuristic in our results. For the three static methods, smallest domain first heuristic is used. For **ParSBDS**, **ReSBDS**, **LReSBDS** 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 broken at the upper level. Running the three static methods with this search heuristic would reduce their performance.

We do not compare the solution set size since the problem is optimization in nature. Figures 4 and 5 show the timing results in logarithmic scale and 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** and **LReSBDS** perform similar and achieve the best performance, while **Precede**, **Siglex** and **ParSBDS** perform the worst. **ReSBDS** and **LReSBDS** are significantly better than **Siglexdec** since they have a much lower overhead and collaborate relatively well with the most-constraining heuristic. **ReSBDS** and **LReSBDS** are also significantly better than **ParSBDS** due to their lower overhead and stronger pruning power. **LDSB** has similar performance trends as **ReSBDS** and **LReSBDS**, but is 3.18 and 3.28 times slower than **ReSBDS** and **LReSBDS** on average respectively.

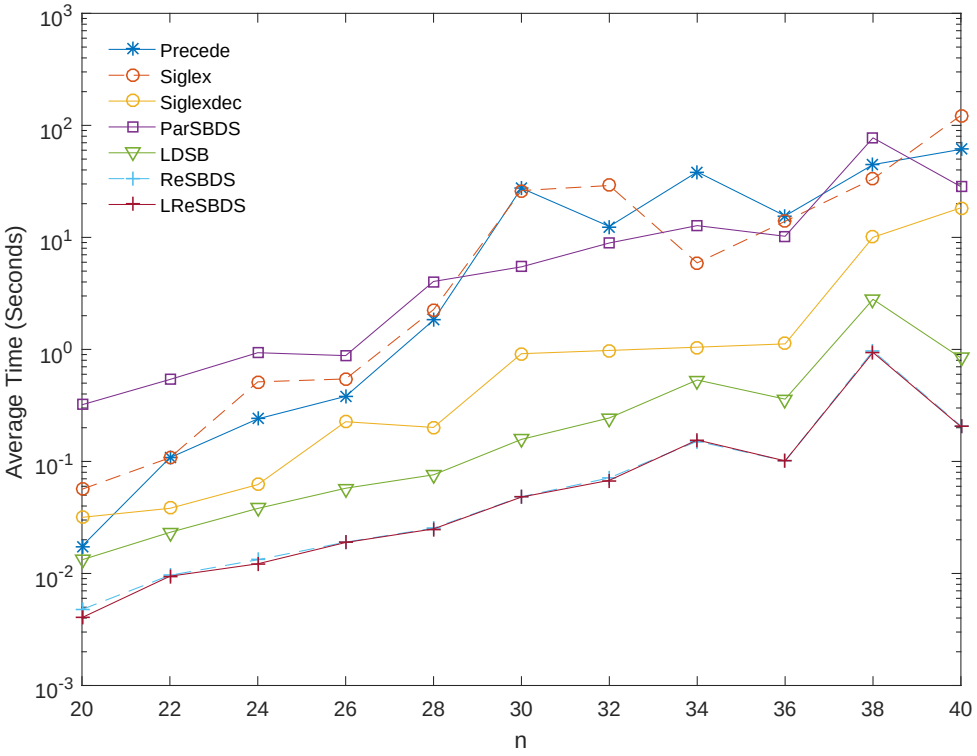

Figure 4: Concert hall scheduling with $k = 8$ (optimal solution)

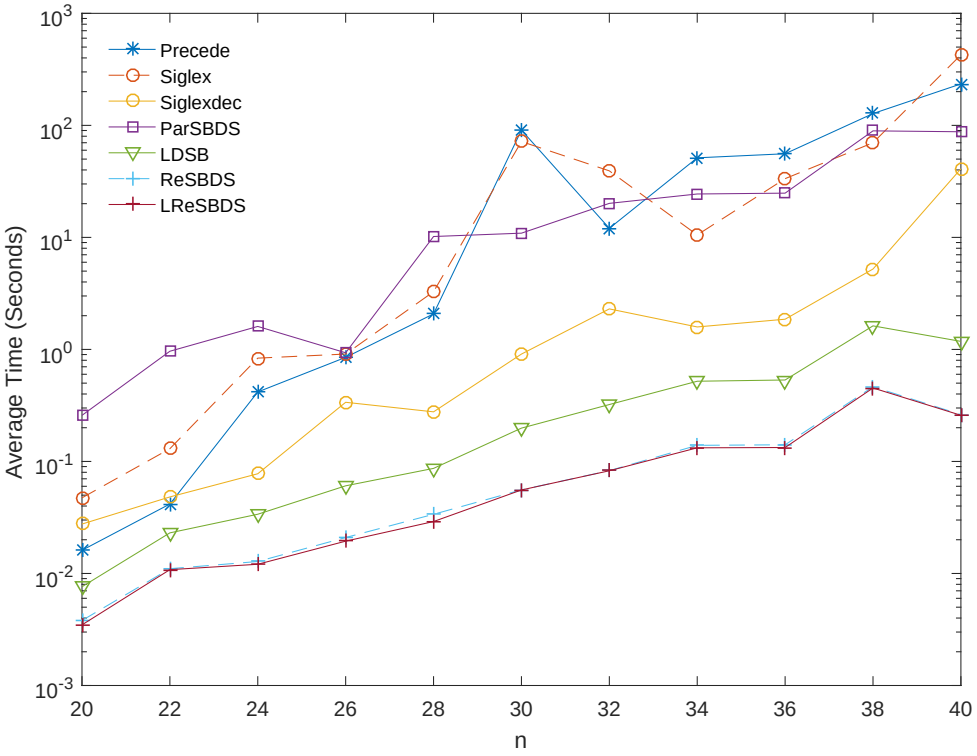

Figure 5: Concert hall scheduling with k =10 (optimal solution)

Table 10: Concert hall scheduling, $k = 8$

n	**SBDS-1UIP** Xeon Pro 2.4GHz		**Static-1UIP** Xeon Pro 2.4GHz		**LReSBDS** 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	0.01
26	282	0.10	685	0.25	445	0.02
28	1,611	0.68	1,041	0.42	501	0.02
30	761	0.27	2,300	0.52	987	0.05
32	1,522	0.40	5,712	1.31	1,080	0.07
34	2,636	1.10	4,406	1.60	4,951	0.16
36	3,156	1.40	5,707	2.37	1,941	0.10
38	5,053	1.91	10,518	3.51	50,015	0.94
40	6,648	2.96	18,169	6.40	2,564	0.20

Symmetry breaking with lazy clause generation [36] can also handle this problem well. **SBDS-1UIP** and **Static-1UIP** combine SBDS and static methods with lazy clause generation respectively. Their experiments were run on the Xeon Pro 2.4GHz processor, which is a dual CPU version of our Xeon E5620 2.4GHz processor running on a Mac Pro. Thus, according to Table 2 both CPUs have the same Passmark CPU marks, but their methods are implemented on top of the CHUFFED solver [45] which is one of the fastest existing CP solvers. Table 10 shows **LReSBDS** is 9.72 and 14.64 times faster than the replicated results [36] of **SBDS-1UIP** and **Static-1UIP**. This again shows our method is very efficient to break a large number of symmetries with a small overhead.

5.2.2. Optimization Versions of CA

To achieve the near-orthogonal case of covering array which happens when every symbol appears with the same frequency in each column, Kim *et al.* [46] introduce several metrics to evaluate the quality of covering array. The second metric is to evaluate a covering array by the average of the absolute equal occurrence discrepancy of each column. Our objective is to minimize this metric. We use the smallest domain first heuristic to search the problem.

Table 11 shows the results. **LDSB** cannot solve most of the instances. **ReSBDS** and **LReSBDS** are 2.38 and 3.02 times faster than **ParSBDS** on average respectively with more symmetries broken. **LReSBDS**^{c} performs the best, and is 1.75 and 1.93 times faster than **DoubleLex** and **SnakeLex** on average. This demonstrates that our methods are competitive against state of the art symmetry breaking methods also in solving optimization problems.

We attempt also another metric: the sum variance in each column. Another objective is to to minimize this metric. We again use the smallest domain first heuristic to search the problem. Table 12 shows the results. Again, **LDSB** cannot solve most of the instances. **LReSBDS**^{c} performs the best for all cases and is 2.02 and 2.26 times faster than **DoubleLex** and **SnakeLex** respectively on average. This gives an indication of the flexibility of our methods in dealing with objective functions of different natures.

Table 11: The CA with equal occurrence discrepancy metric using smallest domain first heuristic (optimal solutions)

t,k,g,b	**DoubleLex**		**SnakeLex**		**LDSB**		**ParSBDS**	
	$\#f$	t	$\#f$	t	$\#f$	t	$\#f$	t
2,3,3,14	2,962,642	51.25	3,516,052	59.22	–	–	3,229,760	128.71
3,4,2,19	2,160,192	**54.89**	2,158,553	56.89	–	–	2,560,706	238.44
2,4,3,12	3,517,349	78.27	1,332,316	**37.34**	34,322,416	951.08	4,064,546	177.89
3,4,2,21	6,315,708	**163.37**	6,280,422	168.49	–	–	7,478,426	760.85
2,4,2,21	25,533,154	605.45	27,060,075	650.25	–	–	35,616,203	3,197.30
2,3,4,17	65,581,343	1,395.84	89,065,478	1,760.79	–	–	67,174,354	2,714.50
2,4,2,23	58,532,147	1,416.13	61,428,368	1,534.31	–	–	–	–
2,4,3,11	129,861,034	2,804.75	151,142,159	3,428.13	–	–	–	–
2,4,2,25	126,811,077	3,210.55	131,998,700	3,491.29	–	–	–	–

t,k,g,b	**ReSBDS**		**ReSBDS**^{c}		**LReSBDS**		**LReSBDS**^{c}	
	$\#f$	t	$\#f$	t	$\#f$	t	$\#f$	t
2,3,3,14	2,839,581	74.67	**1,060,902**	39.38	2,839,581	56.54	**1,060,902**	**28.06**
3,4,2,19	2,160,192	83.44	**723,655**	64.97	2,160,192	65.35	**723,655**	62.51
2,4,3,12	3,338,965	99.91	**981,468**	73.36	3,338,965	82.43	**981,468**	61.24
3,4,2,21	6,315,708	253.85	**2,256,500**	201.20	6,315,708	196.54	**2,256,500**	191.19
2,4,2,21	25,533,154	907.72	**7,066,329**	531.98	25,533,154	729.45	**7,066,329**	**414.11**
2,3,4,17	63,366,807	1,959.11	**19,862,487**	839.64	63,366,807	1,537.27	19,862,639	**627.70**
2,4,2,23	58,532,147	2,182.98	**17,520,892**	1,398.21	58,532,147	1,754.50	**17,520,892**	**1,120.61**
2,4,3,11	117,049,008	3,479.94	**16,651,231**	820.64	117,049,008	2,812.39	16,651,274	**584.22**
2,4,2,25	–	–	**40,796,022**	3,525.42	–	–	**40,796,022**	**2,861.07**

5.2.3. Optimization Versions of ECCLD

Two optimization versions of the ECCLD problem are formed by adopting the same two metrics used in the CA problem. Again, we use the smallest domain first heuristic to search the problems.

Tables 13 and 14 show the results using the two metrics respectively. **LReSBDS**^{c} performs the best in most of the cases under both metrics. It is 2.21 and 3.23 times faster than **DoubleLex** and **SnakeLex** respectively on average for the first metric and is 2.27 and 4.09 times faster than **DoubleLex** and **SnakeLex** respectively on average for the second metric. Our results confirm the empirical efficiency of ReSBDS and LReSBDS in solving optimization problems.

5.3. Discussion

We have performed extensive experimentation on five satisfaction benchmarks and three optimization benchmarks to demonstrate the benefits of ReSBDS and LReSBDS over state of the art symmetry breaking methods, including LexLeader and its partial variants, and SBDS and its partial variant.

ReSBDS and LReSBDS often achieve great reductions in number of failures when they are given the same or smaller subset of symmetries as ParSBDS. The substantial reduction in search tree size implies shorter time to find all solutions or optimal solutions. The extra prunings are attributed to the additional constraints added by ReSBDS and LReSBDS, which can break more composition symmetries and prune more symmetric subtrees with small overheads. Moreover, given the same subset of symmetries, LReSBDS runs slightly faster than ReSBDS since it does not need to record and check the violations of assignments in the table T. Such savings become

Table 12: The CA with sum variance metric using smallest domain first heuristic (optimal solution)

t,k,g,b	**DoubleLex**		**SnakeLex**		**LDSB**		**ParSBDS**	
	$\#f$	t	$\#f$	t	$\#f$	t	$\#f$	t
2,3,3,11	182,679	6.94	194,705	7.21	16,552,961	814.08	195,163	10.60
2,4,2,11	133,158	9.41	170,842	11.98	–	–	196,840	21.41
3,4,2,13	220,868	24.31	228,780	26.40	–	–	266,877	40.22
2,4,2,13	316,551	29.39	402,125	37.32	–	–	466,709	65.25
3,4,2,15	1,150,037	135.29	1,172,823	139.70	–	–	1,372,782	217.06
2,3,3,13	4,265,947	183.73	4,399,931	191.55	–	–	4,636,439	294.41
3,4,2,17	4,001,900	490.75	4,054,157	503.36	–	–	4,758,423	832.77
2,3,3,14	12,614,944	598.24	11,485,742	568.68	–	–	13,861,109	995.31
2,3,5,25	8,607,119	1,245.33	15,321,558	2,156.11	–	–	8,607,206	1,672.81
3,4,2,19	9,669,491	1,331.30	9,781,795	1,372.92	–	–	11,473,843	2,339.41
2,3,3,15	29,984,808	1,659.95	22,485,875	1,260.35	–	–	33,320,943	2,607.11
2,4,3,11	–	–	–	–	–	–	–	–

t,k,g,b	**ReSBDS**		**ReSBDS**^{c}		**LReSBDS**		**LReSBDS**^{c}	
	$\#f$	t	$\#f$	t	$\#f$	t	$\#f$	t
2,3,3,11	176,772	8.25	**70,808**	4.48	176,772	6.84	**70,808**	**3.43**
2,4,2,11	133,158	11.37	**31,967**	5.01	133,158	9.42	**31,967**	**4.26**
3,4,2,13	220,868	26.83	**82,799**	17.94	220,868	21.70	**82,799**	**13.96**
2,4,2,13	316,551	32.13	**80,962**	15.37	316,551	26.37	**80,962**	**14.60**
3,4,2,15	1,150,037	150.79	**431,902**	82.08	1,150,037	122.46	**431,902**	**74.78**
2,3,3,13	4,119,016	227.06	**1,638,050**	112.17	4,119,016	188.27	**1,638,050**	**88.68**
3,4,2,17	4,001,900	558.13	**1,585,370**	318.00	4,001,900	455.89	**1,585,370**	**298.66**
2,3,3,14	12,165,319	742.95	**4,916,415**	365.77	12,165,319	608.04	**4,916,415**	**292.09**
2,3,5,25	8,606,789	1,488.10	**2,863,366**	572.01	8,606,789	1,263.93	**2,863,366**	**476.16**
3,4,2,19	9,669,491	1,488.84	**4,022,919**	909.50	9,669,491	1,206.59	**4,022,919**	**890.41**
2,3,3,15	28,874,496	1,925.34	**11,865,199**	960.42	28,874,496	1,601.75	**11,865,199**	**779.63**
2,4,3,11	–	–	**26,942,200**	1,872.37	–	–	**26,942,200**	**1,477.06**

Table 13: The ECCLD with equal occurrence discrepancy metric using smallest domain first heuristic (optimal solutions)

n, c, b	**DoubleLex**		**SnakeLex**		**LDSB**		**ParSBDS**	
	$\#f$	t	$\#f$	t	$\#f$	t	$\#f$	t
6,8,4	495	0.02	**137**	**0.01**	521	0.02	703	0.04
5,6,4	162,081	4.91	178,342	5.69	142,396	4.77	183,166	6.43
5,2,10	47,151	10.87	130,435	18.02	8,428,880	1,058.29	117,798	26.96
4,4,8	930,664	78.95	1,894,579	165.79	1,628,566	138.42	1,154,009	115.94
5,6,5	3,659,377	149.23	5,894,245	249.33	3,560,112	147.70	4,316,250	196.55
6,4,4	6,657,790	213.84	6,792,322	214.44	15,597,402	514.47	10,616,495	413.78
5,6,6	14,806,304	734.99	29,272,068	1,537.98	15,840,594	820.11	17,215,257	993.83
8,4,4	70,888,317	2,658.47	71,133,987	2,570.20	–	–	–	–
6,4,5	–	–	–	–	–	–	–	–

n, c, b	**ReSBDS**		**ReSBDS**^{c}		**LReSBDS**		**LReSBDS**^{c}	
	$\#f$	t	$\#f$	t	$\#f$	t	$\#f$	t
6,8,4	485	0.02	431	0.06	485	0.02	431	0.03
5,6,4	122,144	4.16	**61,444**	3.70	122,144	3.89	61,455	**2.47**
5,2,10	32,148	6.62	**18,825**	8.91	32,148	**6.26**	18,840	7.04
4,4,8	701,833	64.17	**246,831**	32.84	701,833	61.01	248,068	**25.39**
5,6,5	2,769,574	120.29	**1,169,259**	79.81	2,769,574	114.45	1,169,874	**54.10**
6,4,4	5,220,364	202.72	**2,344,101**	170.36	5,220,364	176.21	2,344,368	**98.42**
5,6,6	11,537,662	623.48	**4,445,326**	372.96	11,537,662	586.70	4,449,381	**252.85**
8,4,4	56,921,268	2,546.01	**24,060,919**	2,868.86	56,921,268	2,266.00	24,062,397	**1,277.98**
6,4,5	–	–	**26,482,932**	2,254.43	76,716,709	3,319.74	26,496,956	**1,385.54**

Table 14: The ECCLD with sum variance metric using smallest domain first heuristic (optimal solution)

n, c, b	**DoubleLex**		**SnakeLex**		**LDSB**		**ParSBDS**	
	$\#f$	t	$\#f$	t	$\#f$	t	$\#f$	t
5,2,10	39,716	10.67	118,191	21.86	5,311,415	1,117.84	92,232	27.63
5,6,4	512,937	21.76	786,251	35.08	482,430	21.29	667,859	33.15
6,4,4	438,043	23.14	575,503	31.02	1,263,337	63.88	921,254	56.23
6,8,4	2,548,910	115.29	4,177,088	198.57	2,388,895	110.13	3,211,257	172.67
4,4,8	1,235,878	133.76	2,808,024	313.12	2,129,594	229.22	1,587,733	199.66
5,6,5	3,183,651	187.61	6,055,240	376.73	3,265,069	189.54	3,938,041	259.63
8,4,4	4,233,238	248.23	5,128,823	299.15	17,622,276	1,017.33	10,510,559	733.42
6,4,5	6,122,535	453.20	8,014,546	613.88	34,590,383	2,406.83	13,787,698	1,149.83
5,6,6	17,132,058	1,213.02	32,961,053	2,466.64	19,271,921	1,300.51	20,076,249	1,571.86

n, c, b	**ReSBDS**		**ReSBDS**^{c}		**LReSBDS**		**LReSBDS**^{c}	
	$\#f$	t	$\#f$	t	$\#f$	t	$\#f$	t
5,2,10	27,178	7.83	**15,995**	8.77	27,178	**7.16**	16,010	7.18
5,6,4	381,195	17.91	**186,315**	13.64	381,195	16.62	186,572	**9.30**
6,4,4	355,032	21.47	**200,693**	23.32	355,158	18.93	200,959	**14.06**
6,8,4	2,016,686	99.89	**938,192**	80.26	2,016,686	92.12	938,994	**47.18**
4,4,8	953,623	110.33	**365,531**	57.21	953,623	103.84	367,806	**44.99**
5,6,5	2,437,614	151.48	**1,054,715**	96.20	2,437,614	142.04	1,055,841	**67.48**
8,4,4	3,593,791	243.57	**1,953,238**	322.82	3,595,151	213.55	1,955,421	**175.71**
6,4,5	4,611,994	382.12	**2,181,030**	308.97	4,612,836	346.04	2,183,365	**209.26**
5,6,6	13,510,979	981.19	**5,249,356**	545.90	13,510,979	920.31	5,255,319	**384.98**

more prominent with the increase in size of the subset of symmetries to break. When compared against LDSB which is designed for breaking only active symmetries but limits the form of symmetries to break, ReSBDS and LReSBDS perform significant better for their capabilities to break both active and inactive symmetries. Unlike LDSB, ReSBDS and LReSBDS are not restricted by pattern syntax and thus can break arbitrary symmetries.

In matrix problems, DoubleLex and SnakeLex are state of the art methods. ReSBDS and LReSBDS are comparable in running time to DoubleLex and SnakeLex when given the same subset of symmetries. When given slightly more symmetries, ReSBDS and LReSBDS can break substantially more composition symmetries, which results in tangible reduction in search space and runtime when compared to DoubleLex and SnakeLex.

When solving satisfaction problems in the experiments, we find all solutions. In case of searching only for the first solution, symmetry breaking methods in general and the ReSBDS family of methods in particular function similarly, and can help reduce search space by pruning symmetric counterparts of visited search regions. However, we did some simple experiments and confirmed that substantially less search effort is required to find the first solution in general. There are two consequences. First, we are searching only an initial part of the search tree. With less visited regions, there are less symmetric counterparts to avoid. Also some of these symmetric counterparts may lie even beyond the first solution in the search tree. Second, there are less branching and backtracking, and thus also less symmetry breaking nogoods being added during search by the ReSBDS family of methods. Very often, these characteristics translate to less symmetries being broken. As a result, symmetry breaking helps relatively little for first solution search, as compared to all solution search.

6. Related Work

Symmetries can be broken statically or dynamically. We give the main and recent work on symmetry breaking techniques.

6.1. Static Symmetry Breaking

In the static symmetry breaking approach, *symmetry breaking constraints* [2] are added to a CSP to allow only some of the symmetrical regions to be traversed during search. Crawford *et al.* [3] suggest a general scheme, called LexLeader, to add symmetry breaking predicates to satisfiability problems. This constraint selects the lexicographically least solution to break symmetries of indistinguishable objects. Aloul *et al.* [47] improve this scheme by constructing more efficient CNF representations of symmetry-breaking predicates. Efficient consistency enforcing algorithms [29, 48, 49] are given for propagating the global lexicographical ordering constraints $\leq_{lex}$. Walsh [14] gives an overview of the application and propagators of $\leq_{lex}$ to deal with various symmetries for both integer and set variables. DoubleLex [4] is an incomplete but efficient method for handling matrix symmetries by breaking only the row and column symmetries. Multiset ordering constraints [15] and allperm constraints [16] are also available for breaking row and column symmetries in matrix models [4], which are commonly found in many CSPs. SnakeLex [21] is similar to DoubleLex but is based on the snake ordering of variables. Yip and Van Hentenryck [22] break matrix symmetries by utilizing LexLeader feasibility checkers to verify, during search, whether the current partial assignment can be extended into a canonical solution. Narodytska and Walsh [23] study extensively the effect of variable ordering in the LexLeader method. When the CSP has an AllDiff on all variables, the exponential number of

lex ordering constraints can be simplified to a linear number of inequality constraints [17]. Katsirelos *et al.* [20] suggest to put together lexicographical ordering symmetry breaking constraints and common global constraints to increase the propagation.

Several static symmetry breaking methods are also proposed for value symmetries. Petrie and Smith [9] adapt LexLeader [3] for breaking value symmetries by imposing an appropriate lexicographical ordering constraint on each value symmetry. Puget [30] and Walsh [14] give propagation methods for value symmetry breaking constraints around the same time but independently. Having to handle an exponential number of such constraints, Puget [30] propose a polytime global filtering algorithm which performs forward checking. To eliminate symmetric solutions due to interchangeable values [50, 51], Law and Lee [5, 18] formally define value precedence and propose a specialized propagator for a pair of interchangeable values. Walsh [52] extends the work of Law and Lee [5, 18] to a propagator for any number of interchangeable values. Assuming two redundant models connected by channeling constraints, Law and Lee [5, 18] also show how value symmetries in one model can be broken using variable symmetry breaking constraints in the dual. The Siglex constraint [19] is proposed for breaking variable and value interchangeability together.

Lee and Li [12] introduce the novel notion of symmetry preservation, and demonstrate its benefits in terms of more symmetries broken, and a smaller solution set and search space.

6.2. Dynamic Symmetry Breaking

Another approach is to break symmetries dynamically [53, 7, 6, 54]. Dynamic methods modify the search procedure to exclude exploration of symmetric regions. A representative dynamic approach is *Symmetry Breaking During Search* (SBDS) [7, 38]. Upon backtracking from a search decision, SBDS [7] adds a conditional symmetry breaking constraint for each symmetry to remove all future nodes symmetric to the current node. Backofen and Will [53] introduce *Symmetry Excluding Search* (SES), which is similar to but more general than SBDS. SES allows a search tree to branch over arbitrary constraints instead of simple unary assignment constraints in SBDS. Symmetries form groups. Gent *et al.* [38] incorporate GAP, a computational group theory system, to SBDS such that large symmetry groups can be handled efficiently. Recently, Mears *et al.* [11] propose a formalization of the shortcut SBDS method called Lightweight Dynamic Symmetry Breaking (LDSB) that handles only active symmetries and also their compositions.

Based on the notion of dominance detection, *Symmetry Breaking via Dominance Detection* (SBDD) [6, 55, 56] tests at each search node if the next node is symmetric to a previously explored node. Barnier and Brisset [56] propose SBDD+, an improvement of SBDD. The key idea of the improvement is a deep pruning technique which allows to prune higher in the search tree whenever possible. Gent *et al.* [37] again use computation group theory to extend SBDD, by proposing a generic dominance checker, which avoids the necessity of implementing a specific dominance checker in SBDD for each problem by a constraint programmer.

There is also research on the intersection of symmetry reasoning and nogood learning. Benhamou *et al.* [57] add to the clause base all its symmetrical assertive clauses when an assertive clause is detected during the search. Chu *et al.* [36] combine Lazy Clause Generation [58, 59] and create the SBDS-1UIP method. Lazy Clause Generation generates 1UIP nogood [60] to help reducing search in constraint programming. Once a 1UIP nogood is generated upon backtrack, its symmetric parts are also posted as nogoods. SBDS-1UIP can exploit symmetries that cannot be exploited by other static or dynamic symmetry breaking methods.

Another dynamic symmetry breaking method is by constructing GE-trees [8] which is a search tree containing a unique representative of each class of full assignments. Such a search

tree also has the property that no node is isomorphic to any other node. Puget [61] proposes dynamic lexicographic constraints (DLC) to use a dynamic variable order (the one used during search) in the lexicographic constraints to avoid incompatibility with search heuristics.

Sellmann and Van Hentenryck [62] devise a polytime dominance tester for eliminating all symmetries of interchangeable variables and values. Flener *et al.* [63] also propose dominance detection search for other tractable symmetries such as piecewise variable and value symmetry. They prove intractability results for some classes of CSPs to show the limits of dominance-detection based symmetry breaking methods. To detect and break symmetries partially, Prestwich *et al.* [64] propose Symmetry Breaking by Nonstationary Optimisation (SBNO) which combines local search with standard backtrack search.

Most recent successful symmetry breaking work has been static in nature, as evidenced by Walsh's Spotlight Talk [13] at AAAI 2012. Our starting point is partial SBDS which is general and can cater for all symmetry types. The overhead of SBDS is big in general. Partial symmetry breaking trades completeness for efficiency by breaking on a subset of symmetries. By carefully controlling the overheads and cleverly breaking symmetry compositions in the context of partial symmetry breaking, we come up with dynamic methods that are competitive both theoretically and practically against the state of the art static methods, and yet enjoy the benefits of the dynamic approaches.

7. Concluding Remarks

Our contributions are eight fold. First, we have identified the inadequacy of ParSBDS in pruning symmetric solutions with respect to LexLeader. Based on this observation, second, we propose ReSBDS which can utilize symmetry breaking constraints' information to break extra symmetry compositions with low overhead. Third, we give formally the time complexity, soundness and termination of ReSBDS and theoretical comparisons against ParSBDS, LexLeader, DoubleLex, SnakeLex 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 also propose a light version of ReSBDS, LReSBDS, which can break extra symmetry compositions without recording assignments, and thus has lower overhead than ReSBDS. Sixth, we give formally the soundness and termination of LReSBDS and theoretical comparisons against ReSBDS, ParSBDS, LexLeader, DoubleLex, SnakeLex and LDSB. Seventh, LReSBDS is also shown to be complete to break all interchangeable variables (values) given only generators when the variable (value) ordering is fixed. Eighth, we demonstrate empirically the efficiency of ReSBDS and LReSBDS against state of the art static and dynamic methods via extensive experimentation.

Acknowledgement

We are grateful to the insightful comments and valuable suggestions by the anonymous reviewers of AAAI'14, CP'14 and the Artificial Intelligence Journal. Zhizhen Ye's assistance with lots of last minute experimentation work is much appreciated. This research has been supported by the grant CUHK413713 from the Research Grants Council of Hong Kong SAR and a Direct Grant from The Chinese University of Hong Kong.

References

[1] A. K. Mackworth, Consistency in networks of relations, Artificial intelligence 8 (1) (1977) 99–118.

[2] J.-F. Puget, On the satisfiability of symmetrical constrained satisfaction problems, in: Proceeding of the 7th International Symposium on Methodologies for Intelligent Systems, 1993, pp. 350–361.

[3] J. Crawford, M. Ginsberg, E. Luks, A. Roy, Symmetry breaking predicates for search problems, in: Proceedings of the Fifth International Conference on Principles of Knowledge Representation and Reasoning, 1996, pp. 148–159.

[4] P. Flener, A. Frisch, B. Hnich, Z. Kiziltan, I. Miguel, J. Pearson, T. Walsh, Breaking row and column symmetries in matrix models, in: Proceeding of the 8th International Conference on Principles and Practice of Constraint Programming, 2002, pp. 187–192.

[5] Y. C. Law, J. Lee, Global constraints for integer and set value precedence, in: Proceeding of the 10th International Conference on Principles and Practice of Constraint Programming, 2004, pp. 362–376.

[6] T. Fahle, S. Schamberger, M. Sellmann, Symmetry breaking, in: Proceeding of the 7th International Conference on Principles and Practice of Constraint Programming, 2001, pp. 93–107.

[7] I. Gent, B. Smith, Symmetry breaking in constraint programming, in: Proceeding of the 14th European Conference on Artificial Intelligence, 2000, pp. 599–603.

[8] C. M. Roney-Dougal, I. P. Gent, T. Kelsey, S. Linton, Tractable symmetry breaking using restricted search trees, in: Proceeding of the 16th European Conference on Artificial Intelligence, 2004, pp. 211–215.

[9] K. E. Petrie, B. M. Smith, Symmetry breaking in graceful graphs, in: Proceeding of the 9th International Conference on Principles and Practice of Constraint Programming, 2003, pp. 930–934.

[10] I. McDonald, B. Smith, Partial symmetry breaking, in: Proceeding of the 8th International Conference on Principles and Practice of Constraint Programming, 2002, pp. 431–445.

[11] C. Mears, M. G. de la Banda, B. Demoen, M. Wallace, Lightweight dynamic symmetry breaking, Constraints 19 (3) (2014) 195–242.

[12] J. H. Lee, J. Li, Increasing symmetry breaking by preserving target symmetries, in: Proceeding of the 18th International Conference on Principles and Practice of Constraint Programming, 2012, pp. 422–438.

[13] T. Walsh, Symmetry breaking constraints: Recent results, in: Proceeding of the 26th AAAI Conference on Artificial Intelligence, 2012, pp. 2192–2198.

[14] T. Walsh, General symmetry breaking constraints, in: Proceeding of the 12th International Conference on Principles and Practice of Constraint Programming, 2006, pp. 650–664.

[15] A. Frisch, I. Miguel, Z. Kiziltan, B. Hnich, T. Walsh, Multiset ordering constraints, in: Proceeding of the 18th International Joint Conference on Artificial Intelligence, 2003, pp. 221–226.

[16] A. Frisch, C. Jefferson, I. Miguel, Constraints for breaking more row and column symmetries, in: Proceeding of the 9th International Conference on Principles and Practice of Constraint Programming, 2003, pp. 318–332.

[17] J. Puget, Breaking symmetries in all different problems, in: Proceeding of the 19th International Joint Conference on Artificial Intelligence, 2005, pp. 272–277.

[18] Y. Law, J. Lee, Symmetry breaking constraints for value symmetries in constraint satisfaction, Constraints (2006) 221–267.

[19] Y. C. Law, J. Lee, T. Walsh, J. Yip, Breaking symmetry of interchangeable variables and values, in: Proceeding of the 13th International Conference on Principles and Practice of Constraint Programming, 2007, pp. 423–437.

[20] G. Katsirelos, N. Narodytska, T. Walsh, Combining symmetry breaking and global constraints, in: Recent Advances in Constraints, Springer, 2009, pp. 84–98.

[21] A. Grayland, I. Miguel, C. M. Roney-Dougal, Snake Lex: An alternative to Double Lex, in: Proceeding of the 15th International Conference on Principles and Practice of Constraint Programming, 2009, pp. 391–399.

[22] J. Yip, P. Van Hentenryck, Symmetry breaking via LexLeader feasibility checkers, in: Proceeding of the 22th International Joint Conference on Artificial Intelligence, 2011, pp. 687–692.

[23] N. Narodytska, T. Walsh, Breaking symmetry with different orderings, in: Proceeding of the 19th International Conference on Principles and Practice of Constraint Programming, 2013, pp. 545–561.

[24] J. Lee, Z. Zhu, An increasing-nogoods global constraint for symmetry breaking during search, in: Proceeding of the 20th International Conference on Principles and Practice of Constraint Programming, 2014, pp. 465–480.

[25] J. Lee, Z. Zhu, Filtering nogoods lazily in dynamic symmetry breaking during search, in: Proceeding of the 24th International Joint Conference on Artificial Intelligence, 2015, pp. 339–345.

[26] J. Lee, Z. Zhu, Breaking more composition symmetries using search heuristics, in: Proceeding of the 30th AAAI Conference on Artificial Intelligence, 2016, pp. 3418–3425.

[27] J. Lee, Z. Zhu, Boosting SBDS for partial symmetry breaking in constraint programming, in: Proceeding of the 28th AAAI Conference on Artificial Intelligence, 2014, pp. 2695–2702.

[28] F. Rossi, P. Van Beek, T. Walsh, Handbook of constraint programming, Elsevier, 2006.

[29] A. Frisch, B. Hnich, Z. Kiziltan, I. Miguel, T. Walsh, Global constraints for lexicographic orderings, in: Proceeding of the 8th International Conference on Principles and Practice of Constraint Programming, 2002, pp. 93–108.

[30] J.-F. Puget, An efficient way of breaking value symmetries, in: Proceeding of the 21st AAAI Conference on Artificial Intelligence, 2006, pp. 117–122.
[31] J.-F. Puget, A comparison of SBDS and Dynamic Lex Constraints, in: The Sixth International Workshop on Symmetry and Constraint Satisfaction Problems, 2006, pp. 56–60.
[32] T. Walsh, Breaking value symmetry, in: Proceeding of the 13th International Conference on Principles and Practice of Constraint Programming, 2007, pp. 880–887.
[33] C. Schulte, P. J. Stuckey, Efficient constraint propagation engines, ACM Transactions on Programming Languages and Systems 31 (1) (2008) 2:1–2:43.
[34] I. Gent, C. Jefferson, I. Miguel, Watched literals for constraint propagation in Minion, in: Proceeding of the 12th International Conference on Principles and Practice of Constraint Programming, 2006, pp. 182–197.
[35] J.-F. Puget, Breaking all value symmetries in surjection problems, in: Proceeding of the 11th International Conference on Principles and Practice of Constraint Programming, 2005, pp. 490–504.
[36] G. Chu, P. Stuckey, M. de la Banda, C. Mears, Symmetries and lazy clause generation, in: Proceeding of the 22th International Joint Conference on Artificial Intelligence, 2011, pp. 516–521.
[37] I. P. Gent, W. Harvey, T. Kelsey, S. Linton, Generic SBDD using computational group theory, in: Proceeding of the 9th International Conference on Principles and Practice of Constraint Programming, 2003, pp. 333–347.
[38] I. P. Gent, W. Harvey, T. Kelsey, Groups and constraints: Symmetry breaking during search, in: Proceeding of the 8th International Conference on Principles and Practice of Constraint Programming, 2002, pp. 415–430.
[39] PassMark Software, Passmark CPU mark, `https://www.cpubenchmark.net/cpu_list.php`, captured on May 7, 2017 (2017).
[40] I. Gent, T. Walsh, CSPLib: a benchmark library for constraints, in: Proceeding of the 5th International Conference on Principles and Practice of Constraint Programming, 1999, pp. 480–481.
[41] K. Petrie, Combining SBDS and SBDD, Tech. rep., APES-86-2004. Available from http://www.dcs.st-and.ac.uk/apes/apesreports.html (2004).
[42] T. Kelsey, S. Linton, C. Roney-Dougal, New developments in symmetry breaking in search using computational group theory, in: Artificial Intelligence and Symbolic Computation, 2004, pp. 199–210.
[43] J.-F. Puget, Elimination des symétries dans les problemes injectifs, in: Premières Journées Francophones de Programmation par Contraintes, 2005, pp. 259–266.
[44] B. Hnich, S. D. Prestwich, E. Selensky, B. M. Smith, Constraint models for the covering test problem, Constraints (2006) 199–219.
[45] G. Chu, Improving combinatorial optimization, Ph.D. thesis, The University of Melbourne (2011).
[46] Y. Kim, D.-H. Jang, C. M. Anderson-Cook, Selecting the best wild card entries in a covering array, Quality and Reliability Engineering International.
[47] F. Aloul, K. Sakallah, I. Markov, Efficient symmetry breaking for Boolean Satisfiability, in: Proceeding of the 18th International Joint Conference on Artificial Intelligence, 2003, pp. 271–276.
[48] M. Carlsson, N. Beldiceanu, Revisiting the lexicographic ordering constraint, Tech. Rep. T2002-17, Swedish Institute of Computer Science (2002).
[49] M. Carlsson, N. Beldiceanu, Arc-consistency for a chain of lexicographic ordering constraints, Tech. Rep. T2002-18, Swedish Institute of Computer Science (2002).
[50] B. Benhamou, Study of symmetry in Constraint Satisfaction Problems, in: Proceedings of the 2nd Workshop on Principles and Practice of Constraint Programming, 1994, pp. 246–254.
[51] I. Gent, A symmetry breaking constraint for indistinguishable values, in: Proceedings of the 1st International Workshop on Symmetry in Constraint Satisfaction Problems, 2001, pp. 469–473.
[52] T. Walsh, Symmetry breaking using value precedence, in: Proceeding of the 17th European Conference on Artificial Intelligence, 2006, pp. 168–172.
[53] R. Backofen, S. Will, Excluding symmetries in constraint-based search, in: Proceeding of the 5th International Conference on Principles and Practice of Constraint Programming, 1999, pp. 73–87.
[54] F. Focacci, M. Milano, Global cut framework for removing symmetries, in: Proceeding of the 7th International Conference on Principles and Practice of Constraint Programming, 2001, pp. 77–92.
[55] J.-F. Puget, Symmetry breaking revisited, in: Proceeding of the 8th International Conference on Principles and Practice of Constraint Programming, 2002, pp. 446–461.
[56] N. Barnier, P. Brisset, Solving the Kirkman's schoolgirl problem in a few seconds, in: Proceeding of the 8th International Conference on Principles and Practice of Constraint Programming, 2002, pp. 477–491.
[57] B. Benhamou, T. Nabhani, R. Ostrowski, M. R. Saidi, Enhancing clause learning by symmetry in SAT solvers, in: ICTAI'10, 2010, pp. 329–335.
[58] T. Feydy, P. J. Stuckey, Lazy clause generation reengineered, in: Proceeding of the 15th International Conference on Principles and Practice of Constraint Programming, 2009, pp. 352–366.
[59] O. Ohrimenko, P. Stuckey, M. Codish, Propagation via lazy clause generation, Constraints (2009) 357–391.
[60] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, S. Malik, Chaff: Engineering an efficient SAT solver, in:

Proceedings of the 38th annual Design Automation Conference, 2001, pp. 530–535.
[61] J.-F. Puget, Dynamic Lex constraints, in: Proceeding of the 12th International Conference on Principles and Practice of Constraint Programming, Springer, 2006, pp. 453–467.
[62] M. Sellmann, P. Van Hentenryck, Structural symmetry breaking, in: Proceeding of the 19th International Joint Conference on Artificial Intelligence, 2005, pp. 298–303.
[63] P. Flener, J. Pearson, M. Sellmann, P. Van Hentenryck, M. Ågren, Dynamic structural symmetry breaking for constraint satisfaction problems, Constraints 14 (4) (2009) 506–538.
[64] S. D. Prestwich, B. Hnich, H. Simonis, R. Rossi, S. A. Tarim, Partial symmetry breaking by local search in the group, Constraints 17 (2) (2012) 148–171.