Filtering Nogoods Lazily in Dynamic Symmetry Breaking During Search

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

Abstract

The generation and GAC enforcement of a large number of weak nogoods in Symmetry Breaking During Search (SBDS) is costly and often not worthwhile in terms of prunings. In this paper, we propose weak-nogood consistency (WNC) for nogoods and a lazy propagator for SBDS (and its variants) using watched literal technology. We give formal results on the strength and relatively low space and time complexities of the lazy propagator. Nogoods collected for each symmetry are increasing. We further define generalized weak-incNGs consistency (GWIC) for a conjunction of increasing nogoods, and give a lazy propagator for the incNGs global constraint. We prove GWIC on a conjunction is equivalent to WNC on individual nogoods, and give the space and time complexities. Various lazy versions of SBDS and its variants are implemented. We give experimentation to demonstrate the efficiency of the lazy versions as compared to state of the art symmetry breaking methods.

Introduction

Advantages of dynamic symmetry breaking methods, such as SBDS [Gent and Smith, 2000; Gent, Harvey, and Kelsey, 2002] and SBDD [Fahle, Schamberger, and Sellmann, 2001; Gent et al., 2003], include completeness, ability to break symmetries of arbitrary kinds and compatibility with variable and value heuristics. When considering runtime, dynamic methods often lose out to widely used static methods, such as the LexLeader method [Crawford et al., 1996] and value precedence [Law and Lee, 2006], which require no modifications to the solver and often have less overheads. Based on SBDS, ReSBDS [Lee and Zhu, 2014a] and LReSBDS [Lee and Zhu, 2014b] break additional symmetry compositions thereby, further pruning the solution and search space. Instead of maintaining the large number of symmetry breaking nogoods in the constraint store, the increasing-nogood (incNGs) global constraint [Lee and Zhu, 2014b] collects all nogoods for a symmetry and filters them with an incremental and stronger filtering algorithm. Such recent advances greatly increase the competitiveness of dynamic approaches against the state of the art static methods.

SBDS adds conditional symmetry breaking nogoods dynamically upon backtracking to avoid exploring symmetrically equivalents of visited search space. However, nogoods are weak in pruning and maintaining GAC is not cost effective even when the watched literal technique [Moskewicz et al., 2001] is utilized. In addition, the number of such nogoods is often large, incurring big overheads to the constraint store.

In this work, we propose weak-nogood consistency (WNC), a weaker consistency notion for nogoods to trade pruning power for efficiency. To enforce GAC on a nogood, usual implementation watches two assignments (as literals) of the nogood. We present an efficient lazy propagator to enforce WNC for SBDS (and its variants) using *one* watched literal. First, our propagator generates the watched assignment on demand from partial assignment of the current search path. Second, the propagator is triggered lazily when the watched assignment becomes true, and effects prunings only when all but the last assignment of the nogood is true.

A similar weaker consistency, generalized weak-incNGs consistency (GWIC), together with a lazy propagator is also proposed for the incNGs constraint [Lee and Zhu, 2014b]. By exploiting the increasing property of the nogoods in incNGs, our lazy propagator watches also one assignment for each global constraint, and operates and benefits from a similar lazy principle. We give formal characterization of the pruning strengths of the proposed propagators, and also the space and time complexities. Various lazy versions of SBDS and its variants are implemented. We give experimentation to demonstrate the efficiency of the lazy versions as compared to state of the art symmetry breaking methods.

Background

A *constraint satisfaction problem* (CSP) P is a tuple (X, D, C) where X is a finite set of variables $\{x_0, \ldots, x_{n-1}\}$, D is a finite set of domains such that each $x \in X$ has a $D(x)$ and C is a set of constraints, each is a subset of the Cartesian product $D(x_{i_1}) \times \ldots \times D(x_{i_k})$ of the domains of the involved variables. 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 (called *supports*) in the domains of all the other variables in the scope of the constraint. A CSP is *GAC* iff every constraint is GAC. An *assignment* $x = v$ is an equality constraint assigning 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* to P is a full assignment that satisfies every member of C. An assignment $x = v$ is *satisfied* iff $D(x) = \{v\}$. An assignment $x = v$ is *falsified* iff $v \notin D(x)$. If an assignment is neither satisfied nor falsified, it is *unresolved*.

A *nogood* is the negation of a partial assignment which is not in any solution. Nogoods can also be expressed in an equivalent implication form. A *directed nogood* ng is an implication of the form $(x_{s_0} = v_{s_0}) \wedge \ldots \wedge (x_{s_m} = v_{s_m}) \Rightarrow (x_k \neq v_k)$, where the *left hand side* (LHS) $(lhs(ng) \equiv (x_{s_0} = v_{s_0}) \wedge \ldots \wedge (x_{s_m} = v_{s_m}))$ and the *right hand side* (RHS) $(rhs(ng) \equiv (x_k \neq v_k))$ are defined with respect to the position of $\Rightarrow$. We call all assignments in the LHS as *the LHS assignments* and the negation of the RHS as *the RHS assignment*. The meaning of ng is that the RHS assignment $x_k = v_k$ is incompatible with the LHS assignments, and v_k should be ruled out from $D(x_k)$ when the LHS is true. If $lhs(ng)$ is empty, ng is *unconditional*. Hereafter, directed nogoods are simply called nogoods when the context is clear.

In this paper, we consider search trees with binary branching, in which every non-leaf node has exactly two children. Suppose a non-leaf node P_1 has x and $v \in D(x)$ as the branching variable and value. The left and right children of P_1 are $P_1 \cup \{x = v\}$ and $P_1 \cup \{x \neq v\}$ respectively. We call $x = v$ the *branching assignment* from P_1 to $P_1 \cup \{x = v\}$ and *backtracking assignment* from $P_1 \cup \{x = v\}$ to $P_1 \cup \{x \neq v\}$. We say that backtracking *takes place* at P_1. 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.

We assume that *the CSP, except the generated symmetry breaking nogoods, at a search tree node is always made GAC* using an AC3-like [Mackworth, 1977] algorithm to coordinate the triggering of the propagators associated with the individual constraints. Consistency Φ_1 is *stronger than or equal to* ($\geq$) consistency Φ_2 iff any Φ_1-consistent CSP is also Φ_2-consistent. Consistency Φ_1 is *strictly stronger than* ($>$) Φ_2 if $\Phi_1 \geq \Phi_2$ but not *vice versa*. Consistency Φ_1 is *equivalent to* ($=$) Φ_2 if $\Phi_1 \geq \Phi_2$ and $\Phi_2 \geq \Phi_1$.

Here we consider symmetry as a property of the set of solutions. A *solution symmetry* [Rossi, Van Beek, and Walsh, 2006] is a solution-preserving permutation on assignments.

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.

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

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

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

meaning that once $A \wedge (x = v)$ has been searched, its symmetric partial assignment $(A \wedge (x = v))^g$ for any g in the symmetry set under this subtree should not be searched at all. Note that the added constraint is a nogood.

Weak-Nogood Consistency

The watched literal technique [Moskewicz et al., 2001] is extended to implementing propagators for CSPs [Gent, Jefferson, and Miguel, 2006]. When an assignment (as a literal) is *watched*, the satisfaction of the assignment *triggers* the propagator in the AC-3 like algorithm. To enforce GAC for a nogood, only two assignments that are not satisfied need to be watched [Moskewicz et al., 2001]. Upon each backtracking, SBDS would add new nogoods into the constraint store. Maintaining such a large set of nogoods is costly since the triggers to their propagators are often fruitless. We propose to watch only one assignment to trade pruning power for efficiency by enforcing nogood with a weaker consistency.

A nogood $(x_{s_0} = v_{s_0}) \wedge \ldots \wedge (x_{s_m} = v_{s_m}) \Rightarrow (x_k \neq v_k)$ is *weak-nogood consistent (WNC)* iff $(\exists i \in [0, m], (v_{s_i} \notin D(x_{s_i}) \vee (v_{s_i} \in D(x_{s_i}) \wedge |D(x_{s_i})| > 1))) \vee ((\forall i \in [0, m], D(x_{s_i}) = \{v_{s_i}\}) \wedge v_k \notin D(x_k))$.

A nogood is WNC if either (1) one of the LHS assignments is not satisfied or (2) all LHS assignments are satisfied and its RHS assignment is falsified.

We now compare the consistency level of GAC and WNC.

Theorem 1. *GAC $>$ WNC on a nogood.*

Proof. We first prove that GAC implies WNC. A nogood is GAC if either (a) it is true or (b) it has at least two unresolved assignments. For (a), either one of the LHS assignments is falsified, or all LHS assignments are satisfied and its RHS assignment is falsified. Both cases satisfy the two conditions of WNC respectively and the nogood is also WNC. For (b), one of the unresolved assignments must be in the LHS. Thus one of the LHS assignments is not satisfied and condition (1) of WNC is satisfied. The nogood is WNC.

To prove GAC is strictly stronger than WNC, consider the nogood $ng \equiv (x_1 = 1) \wedge (x_2 = 1) \Rightarrow (x_3 \neq 1)$ with domains $D(x_1) = D(x_3) = \{1\}, D(x_2) = \{1, 2\}$. The nogood is WNC since $(x_2 = 1)$ is not satisfied yet but not GAC since 1 from $D(x_2)$ has no support. □

WNC loses pruning opportunities in which an unresolved LHS assignment can be enforced to be true when all other assignments in this nogood are all satisfied. Experimental results confirm however that such cases happen not very often.

To enforce WNC for a symmetry breaking nogood, we propose a lazy propagator, *LazyNgProp*, in Algorithm 1. The propagator can end with two different outputs: **ENTAILED** or **WNCed**. A stores the current partial assignment. Without loss of generality, we assume that variables in A are in input order $\langle x_0, \ldots, x_{n-1} \rangle$ for ease of explanation. We also assume

that a propagator is immediately triggered once when it is first added to the constraint store.

Algorithm 1 *LazyNgProp*()

Require:
A: current partial assignment
g: symmetry
x: variable of the watched LHS assignment
v: value of the watched LHS assignment
$\tilde{x}$: variable of the RHS
$\tilde{v}$: value of the RHS
$\alpha = -1$: position of the assignment in A whose symmetric assignment is being watched
β: the length of the LHS

1: **for** each $(x_i = v_i) \in A$ **and** $i \in [\alpha + 1, \beta)$ **do**
2: $(x_i{}' = v_i{}') = (x_i = v_i)^g$;
3: **if** $v_i{}' \notin D(x_i{}')$ **then** return **ENTAILED**;
4: **else**
5: **if** $|D(x_i{}')| > 1$ **then**
6: $x = x_i{}'$; $v = v_i{}'$;
7: $\alpha = i$;
8: watch $x = v$;
9: return **WNCed**;
10: prune $\tilde{v}$ from $D(\tilde{x})$;
11: return **ENTAILED**;

Upon each backtracking, we create a propagator for each generated symmetry breaking nogood and make available the symmetry g, the length of the LHS and the RHS $\tilde{x} \neq \tilde{v}$ of the associated nogood to the propagator. The pointer α is initialized to -1 meaning that the propagator does not watch any assignment when it is initially triggered. The **for**-loop in lines 1-9 starts from the current watched assignment to look for the next unresolved assignment to watch. Once found, the nogood is WNC and the propagator can exit (line 9) with **WNCed** returned. If, however, any LHS assignment is found falsified, the nogood is entailed and can be removed from the constraint store (line 3, exit with **ENTAILED**). If no unresolved assignments are found, all LHS assignments are satisfied and pruning is effected using the RHS (line 10) and again the nogood is entailed (line 11).

As can be seen in Algorithm 1, *LazyNgProp* watches only one assignment at a time. Symmetric assignments in the LHS of nogoods are also only computed on demand but not eagerly. In addition, the current partial assignment A can be shared by all propagators and backtrackable.

Theorem 2. *LazyNgProp enforces WNC for a nogood.*

Proof. If the propagator for a nogood ng returns **WNCed** at line 9, it has an unresolved assignment $x = v$ being watched. From line 1, we know $x = v$ is in the LHS of ng. Thus condition (1) of WNC is satisfied and ng is WNC. The propagator can return **ENTAILED** at line 3 or 11. In line 3, condition (1) of WNC is satisfied. In line 11, condition (2) is satisfied since all LHS assignments are satisfied and the RHS assignment is falsified. Again, ng is WNC. □

Theorem 3. *Given a CSP $P = (X, D, C)$ with $|X| = n$. The space complexity of LazyNgProp for a nogood is $O(1)$, and the time complexity is $O(n)$.*

Proof. *LazyNgProp* records only the symmetry (just a name or pointer), the LHS watched assignment, the RHS assignment, the LHS watched assignment's corresponding assignment position and the length of the LHS. The total space complexity is $O(1)$. The maximum number of assignments in each nogood is n and we scan the assignments only once in *LazyNgProp*. The time complexity is $O(n)$. □

Theorem 4. *Given a CSP $P = (X, D, C)$ with $|X| = n$ and a set of symmetries G. The total space and time complexity of SBDS utilizing LazyNgProp to enforce WNC on all nogoods at each search tree node is $O(|G| \sum_{i=0}^{n-1}(|D(x_i)| - 1) + n)$ and $O(n|G| \sum_{i=0}^{n-1}(|D(x_i)| - 1))$ respectively.*

Proof. The total number of nogoods in SBDS is at most $|G| \sum_{i=0}^{n-1}(|D(x_i)| - 1)$ and the length of a partial assignment can be at most n. While each filtering algorithm has $O(1)$ space complexity, the total space complexity is $O(|G| \sum_{i=0}^{n-1}(|D(x_i)| - 1) + n)$.

LazyNgProp scans the LHS assignments incrementally during the AC3-like algorithm by maintaining α. The maximum number of such assignments is n. Thus the total time complexity for *LazyNgProp* to handle all the nogoods in SBDS is $O(n|G| \sum_{i=0}^{n-1}(|D(x_i)| - 1))$. □

GAC vs WNC in SBDS and Its Variants

In addition to ReSBDS and LReSBDS, Partial SBDS (ParSBDS) [Flener et al., 2002] is SBDS but deals with only a given subset of all symmetries. LDSB [Mears et al., 2013] is a further development of shortcut SBDS [Gent and Smith, 2000] which handles only active symmetries and their compositions. WNC can also be used in ParSBDS, LDSB, ReSBDS and LReSBDS since they generate symmetry breaking nogoods in a similar fashion.

In this section, we compare the node and solution pruning power between GAC and WNC in SBDS and its variants respectively. We denote the methods enforcing WNC with subscript WNC, and those enforcing GAC without a subscript.

In the following, we assume all methods use the same static variable and value ordering heuristics.

Theorem 5. *SBDS $=_s$ SBDS$_{WNC}$ and SBDS $\succ_n$ SBDS$_{WNC}$.*

Proof. Suppose SBDS$_{WNC}$ leaves two symmetric solutions s_1 and s_2. Suppose further P is their deepest common ancestor node and s_1 is searched earlier than s_2. Upon backtracking from the left subtree to the right at P, the nogood which prunes the symmetric solution of s_1 must be posted. Since *LazyNgProp* can only postpone some pruning opportunities to deeper nodes, s_2 is pruned by the nogood eventually. Thus SBDS$_{WNC}$ has the *same* solution pruning power as SBDS.

As the entire symmetry group is posted, SBDS would never leave any symmetric subtrees. While SBDS$_{WNC}$ can postpone the pruning opportunity into deeper nodes, it leaves some symmetric parts and prunes less nodes. SBDS$_{WNC}$ can

backtrack more and generate more nogoods, but these nogoods break no new symmetries (since all symmetries are broken in SBDS). Thus the search tree of SBDS$_{WNC}$ *subsumes* that of SBDS. □

In partial symmetry breaking, such as ParSBDS and ReSBDS, only a subset of all symmetries is posted. Since WNC is weaker, there are symmetric subtrees that are pruned by GAC but not by WNC. In turn, WNC searches and also backtracks more, thus sometimes posting more nogoods than GAC. Such nogoods can potentially break composition symmetries that are not posted. As a result, it is difficult to compare theoretically the strength of GAC and WNC on ParSBDS, ReSBDS and LReSBDS respectively, but is interesting future work.

Theorem 6. *WNC = GAC for unconditional nogoods.*

Proof. Unconditional nogoods are simply disequality constraints. WNC and GAC handle them equally. □

Theorem 7. *LDSB* $=_n$ *(resp.* $=_s$*) LDSB*$_{WNC}$.

Proof. All symmetry breaking nogoods in LDSB are unconditional since LDSB handles only active symmetries and their compositions. Results follow directly from Theorem 6. □

Generalized Weak-incNGs Consistency

Lee and Zhu (2014b) show that the set of symmetry breaking nogoods added by SBDS and its variants for one symmetry at a search node are increasing nogoods. They also give the incNGs global constraint and a filtering algorithm which is stronger than GAC on each nogood. Now we propose a weaker consistency for the incNGs constraint.

A set of directed nogoods Λ is *increasing* [Lee and Zhu, 2014b] if the nogoods can form a sequence $\langle ng_0, \ldots, ng_t \rangle$ where $ng_i \equiv (A_i \Rightarrow x_{k_i} \neq v_{k_i})$ such that (i) for any $i \in [1, t]$, $A_{i-1} \subseteq A_i$ and (ii) no nogoods are implied by another. A nogood ng_j in Λ is *lower than* nogood ng_i iff $j < i$, and ng_i is *higher than* ng_j.

A set of increasing nogoods $\Lambda = \langle ng_0, \ldots, ng_t \rangle$ where $ng_i \equiv (A_i \Rightarrow x_{k_i} \neq v_{k_i})$ is *generalized weak-incNGs consistent (GWIC)* iff $(\exists i \in [0,t], \exists (x_{s_i} = v_{s_i}) \in A_i, ((v_{s_i} \notin D(x_{s_i})) \vee (v_{s_i} \in D(x_{s_i}) \wedge |D(x_{s_i})| > 1))) \wedge (\forall j < i, (\forall (x_{s_j} = v_{s_j}) \in A_j, D(x_{s_j}) = \{v_{s_j}\}) \wedge v_{k_j} \notin D(x_{k_j}))) \vee ((\forall i \in [0,t], (\forall (x_{s_i} = v_{s_i}) \in A_i, D(x_{s_i}) = \{v_{s_i}\}) \wedge v_{k_i} \notin D(x_{k_i}))$. Thus, Λ is GWIC if either (1) one of the LHS assignments of nogood ng_i is not satisfied and for each ng_j lower than ng_i, ng_j's LHS assignments are satisfied and its RHS assignment is falsified or (2) the LHS assignments of all nogoods are satisfied and all RHS assignments are falsified. GWIC has the following theorems.

Theorem 8. *Given increasing nogoods* $\Lambda = \langle ng_0, \ldots, ng_t \rangle$*, GWIC on* Λ *is equivalent to WNC on each* ng_k*,* $k \in [0, t]$.

Proof. We first prove that if Λ is GWIC, all individual nogoods are WNC. One case is that there exists ng_i such that Λ satisfies condition (1) of GWIC. Now all nogoods lower than ng_i satisfy condition (2) of WNC, and the remaining nogoods whose LHSs subsume $lhs(ng_i)$ satisfy condition (1) of WNC. Thus all nogoods are WNC. Another case is that Λ satisfies condition (2) of GWIC, which implies that all nogoods satisfy condition (2) of WNC.

Secondly, we prove if all individual nogoods are WNC, Λ is GWIC. Suppose we can find a nogood ng_j which is the lowest nogood in Λ and satisfies condition (1) of WNC. All nogoods lower than ng_j must satisfy condition (2) of WNC since all their LHS assignments must be satisfied. Thus condition (1) of GWIC is satisfied. If we cannot find such an ng_j, all nogoods satisfy condition (2) of WNC. Now condition (2) of GWIC is satisfied. Therefore, Λ is GWIC iff all individual nogoods are WNC. □

Theorem 9. *GAC > GWIC on an incNGs constraint.*

Proof. Given increasing nogoods $\Lambda = \langle ng_0, \ldots, ng_t \rangle$. Lee and Zhu (2014b) show that GAC on Λ is strictly stronger than GAC on individual nogoods (ng_i), which is strictly stronger than WNC on the individual nogoods (ng_i) by Theorem 1. In addition, WNC on the individual nogoods (ng_i) is equivalent to GWIC on Λ by Theorem 8. Result follows. □

A sequence of increasing nogoods $\Lambda = \langle ng_0, \ldots, ng_t \rangle$ for a symmetry g can be encoded compactly. The increasing property guarantees that $lhs(ng_j) \subseteq lhs(ng_i)$ for all $j < i \leq t$. Thus the LHS assignments of all nogoods are available in $lhs(ng_t)$. Suppose Λ is added at an ancestor node P' of P and A is the partial assignment of P. We must have $lhs(ng_t) \subseteq A^g$. Therefore, we can always construct (by applying symmetry transformation) $lhs(ng_t)$ from the partial assignment at any descendant node of P'. We also collect all the backtracking assignments (refer to Section 2) from root to a search node, and call it B. Each backtracking assignment is also associated with its *depth*, which is the length of the partial assignment associated with the node where backtracking takes place. We can construct (by applying symmetry transformation) $rhs(ng_k)$ for $k \in [0, t]$ from the backtracking assignments in B. Using A and B together, we can reconstruct all nogoods in Λ using symmetry g.

Section 3 shows that watching one unresolved assignment in the LHS is enough to enforce WNC on a nogood. An important consequence of $lhs(ng_j) \subseteq lhs(ng_i)$ for all $j < i \leq t$ in Λ is that watching an assignment in $lhs(ng_0)$ implies watching the same assignment in $lhs(ng_1), \ldots, lhs(ng_t)$. In general, when we watch an assignment that first appears only in $lhs(ng_i)$, we are also watching the same assignment in $lhs(ng_{i+1}), \ldots, lhs(ng_t)$. Thus, we can watch only one unresolved assignment for our lazy propagator for Λ. We scan the symmetric assignments of the current partial assignment at each search node to find the first unresolved assignment to watch. During scanning, when we encounter an assignment Γ, there are three possibilities. (a) Γ is satisfied. We should make use of B to look for all nogoods with a true LHS, and enforce the RHSs of all discovered nogoods to be true to effect prunings. Then we continue our previous scanning to the next assignment. (b) Γ is falsified. We can stop scanning since Λ is entailed. (c) Γ is unresolved. We stop scanning and watch Γ, since Λ is now GWIC.

The incNGs constraint grows since nogoods are added dynamically upon backtracking. It can happen that, at one

search node, all symmetric equivalents of those in the current assignments A are true. There are no assignments to watch. In such a case, the propagator should still be triggered upon backtracking (when B is updated), which is when a new nogood is added to the propagator.

We now present our lazy propagator, *LazyincNGsProp*, for the incNGs constraint as shown in Algorithm 2. The propagator can return two different results: **ENTAILED** or **GWICed**. A stores the current partial assignment. B stores all backtracking assignments and their depth as a pair from root up to the current search node. If we view A and B as arrays, their indices start from 0. Without loss of generality, we assume that variables in A are in input order $\langle x_0, \ldots, x_{n-1} \rangle$.

Algorithm 2 *LazyincNGsProp*()

Require:
A: current partial assignment
B: all backtracking assignments and their depth from root up to the current node
g: symmetry
$x = \perp$: variable of the watched LHS assignment
$v = \perp$: value of the watched LHS assignment
$\alpha = -1$: position of the assignment in A whose symmetric assignment is being watched
$\beta = 0$: position of the first assignment in B whose symmetric assignment has not been enforced by this constraint

```
1: if α = |A| − 1 then
2:     pruneB(α + 1);
3: else
4:     for each (x_i = v_i) ∈ A and i ∈ [α + 1, |A|) do
5:         (x_i' = v_i') = (x_i = v_i)^g;
6:         if v_i' ∉ D(x_i') then return ENTAILED;
7:         else
8:             if |D(x_i')| > 1 then
9:                 x = x_i'; v = v_i';
10:                α = i;
11:                watch x = v;
12:                return GWICed;
13:            pruneB(i + 1);
14:    x = ⊥; v = ⊥; α = |A| − 1;
15: return GWICed;
```

Algorithm 3 *pruneB*(k)

```
1: for each (x_j = v_j, d_j) ∈ B and j ≥ β and d_j ≤ k do
2:     (x_j' = v_j') = (x_j = v_j)^g;
3:     prune v_j' from D(x_j');
4:     β = j + 1;
```

For each given symmetry g, an incNGs global constraint would be posted at the root node. We create a propagator *LazyincNGsProp* and make available the symmetry g to the propagator. This propagator is triggered by the watched assignment $x = v$. If there are no assignments to watch, it is triggered when B is updated. The pointer α always points to the watched assignment, and is initialized to -1 to watch nothing. If the propagator is triggered by B's update, and no assignments are watched and A is not updated from the last propagation (line 1), this means the LHSs of all increasing nogoods are satisfied and *pruneB* (line 2) is called to prune all symmetric equivalents of the backtracking assignments in B starting from pointer β whose corresponding nogoods are not enforced yet. After doing that, the constraint is GWIC and the propagator can exit (line 15). If the propagator is triggered by the watched assignment $x = v$ or A's update, the **for**-loop in lines 4-13 looks for the next unresolved assignment to watch as in Algorithm 1. The only thing to take care is if an assignment in the LHS is satisfied, *pruneB* (line 13) is called to prune all symmetric equivalents of the backtracking assignments in B starting from pointer β whose LHSs are satisfied and corresponding nogoods are not enforced yet.

As can be seen in Algorithm 2, *LazyincNGsProp* watches only one assignment at a time, and also gets triggered when B is updated. All symmetric assignments are computed on demand but not eagerly. Note that A and B should be backtrackable and can be shared by all propagators.

Theorem 10. *LazyincNGsProp enforces GWIC for an incNGs global constraint.*

Proof. The propagator returns **GWICed** at line 12 or 15. Line 12 is reached when there is an unresolved assignment $x = v$ being watched. Now all nogoods whose LHSs do not contain $x = v$ are true since their LHS assignments are satisfied and their RHSs are enforced at line 2 or 13 by *pruneB*. For all nogoods containing $x = v$, $x = v$ must be in their LHSs according to line 4. Thus condition (1) of GWIC is satisfied and the increasing nogoods is GWIC. Line 15 means all LHS assignments of current nogoods are satisfied and all RHS assignments are falsified. Now all nogoods are true. Condition (2) of GWIC is satisfied and this increasing nogoods is GWIC. If the propagator returns **ENTAILED** at line 6, there exists an ng_i such that its LHS contains $x'_i = v'_i$ and all lower nogoods have their LHS assignments satisfied and RHS assignment falsified. Thus, condition (1) of GWIC is satisfied and this increasing nogoods is GWIC. □

Theorem 11. *Given a CSP $P = (X, D, C)$ with $|X| = n$. The space complexity of LazyincNGsProp for an incNGs global constraint is $O(1)$, and the time complexity is $O(\sum_{i=0}^{n-1} |D(x_i)| - n/2)$.*

Proof. LazyincNGsProp needs to record the symmetry, the LHS watched assignment, two pointers α and β. The total space complexity is $O(1)$. There are at most $n + \sum_{i=0}^{n-1}(|D(x_i)| - 1)$ number of symmetric assignments to generate and $\sum_{i=0}^{n-1}(|D(x_i)| - 1)$ to prune. The time complexity is $O(\sum_{i=0}^{n-1} |D(x_i)| - n/2)$. □

Experimental Results

This section gives three experiments, all with matrix symmetries (variable symmetries). Our method works for arbitrary symmetries, but better as the number of symmetry grows. Matrix symmetries are common in many CSP models and their numbers are exponential with problem size. We first solve the benchmarks using the efficient and widely used static method

Table 1: Error Correcting Code - Lee Distance (all solutions)

n,c,b	DoubleLex			LexLeader		
	$\#s$	$\#f$	t	$\#s$	$\#f$	t
4,4,8	32,469	839,251	42.19	7,863	267,815	15.14
5,2,10	87	41,571	4.73	**56**	34,659	7.48
5,6,4	710,731	725,837	16.39	269,841	354,184	8.1
5,6,5	1,441,224	5,508,192	116.88	451,303	1,918,579	45.17
5,6,6	297,476	11,709,068	303.4	82,742	3,837,292	112.16
6,4,4	4,698,842	4,139,211	112.17	1,690,229	3,404,499	72.47
6,4,5	29,345,816	73,522,873	1909.09	8,052,126	23,457,604	678.49
6,8,4	59,158	2,469,211	35.9	22,756	1,082,827	18.05
8,4,4	35,626,714	48,525,827	1303.08	12,246,480	47,273,927	1171.17

ParSBDS$_{GAC}$			ParSBDS$_{incNGs}$			ParSBDS$_{WNC/GWIC}$			
$\#s$	$\#f$	t	$\#s$	$\#f$	t	$\#s$	$\#f$	t_W	t_G
8,918	281,720	21.62	8,918	281,553	16.68	8,654	275,236	18.01	14.43
66	25,824	6.22	66	25,810	4.96	66	26,732	5	4.09
297,819	307,148	12.91	297,819	306,376	8.32	289,822	334,263	9.73	7.80
508,585	2,042,200	76.83	508,585	2,036,425	51.68	490,687	2,095,836	59.5	46.83
95,380	4,143,524	185.03	95,380	4,129,503	126.99	90,767	4,163,933	147.23	116.14
1,943,608	1,890,047	102.97	1,943,604	1,888,679	62.08	1,859,800	1,961,610	71.84	53.99
9,472,056	25,509,615	1,260.44	9,472,021	25,493,887	806.56	9,034,353	25,186,758	920.08	696.83
24,355	1,033,529	32.45	24,355	1,031,161	18.92	24,192	1,090,154	22.69	17.39
14,541,826	21,963,988	1,450.38	14,541,822	21,958,213	750.56	13,877,574	22,137,598	889.59	637.30

LReSBDS$_{GAC}$			LReSBDS$_{incNGs}$			LReSBDS$_{WNC/GWIC}$			
$\#s$	$\#f$	t	$\#s$	$\#f$	t	$\#s$	$\#f$	t_W	t_G
7,698	235,074	15.01	**7,698**	**234,806**	14.47	**7,698**	237,306	19.63	**13.44**
56	20,820	5.6	**56**	**20,806**	4.42	**56**	21,801	6.36	**2.76**
235,866	253,729	7.34	**235,866**	**252,866**	6.94	**235,866**	262,483	8.78	**6.34**
392,221	1,614,088	44.47	**392,221**	**1,608,523**	41.42	**392,221**	1,645,389	53.79	**39.55**
72,150	3,210,014	106.3	**72,150**	**3,197,525**	101.79	**72,150**	3,260,502	139.17	**94.70**
1,608,536	1,568,776	59.95	**1,608,536**	**1,566,251**	54.06	**1,608,536**	1,650,210	71.88	**49.41**
7,631,833	20,494,554	747.13	**7,631,833**	**20,474,513**	670.31	**7,631,833**	21,024,022	904.68	**618.61**
18,933	793,258	16.28	**18,933**	**790,810**	15.03	**18,933**	806,897	20.56	**13.63**
11,582,467	17,464,022	768.35	**11,582,467**	**17,453,197**	629.5	**11,582,467**	18,047,630	998.21	**532.95**

Table 2: Cover Array Problem (all solutions)

t,k,g,b	DoubleLex			LexLeader		
	$\#s$	$\#f$	t	$\#s$	$\#f$	t
2,4,4,16	3,456	661,726	23.67	**424**	123,439	8.19
3,4,2,13	29,738	202,723	2.86	**11,047**	**76,235**	3.09
3,4,2,14	107,224	496,246	7.29	**38,007**	**185,786**	6.96
3,4,2,15	348,857	1,149,974	17.91	**120,832**	**431,794**	15.09
3,4,2,16	1,039,641	2,548,941	42.21	**357,662**	**965,531**	31.93
3,4,2,17	2,870,734	5,433,943	94.55	**991,700**	**2,085,752**	65.24
3,4,2,18	7,413,394	11,181,194	210.61	**2,590,000**	**4,362,860**	**130.88**
3,4,2,19	18,043,630	22,265,801	443.24	**6,404,281**	**8,851,675**	**257.07**
3,4,3,27	24	64,777	5.66	**8**	26,193	17.69

ParSBDS$_{GAC}$			ParSBDS$_{incNGs}$			ParSBDS$_{WNC/GWIC}$			
$\#s$	$\#f$	t	$\#s$	$\#f$	t	$\#s$	$\#f$	t_W	t_G
432	130,811	14.18	432	130,707	8.41	432	128,629	11.21	6.37
16,085	95,343	7.01	16,085	95,343	3.86	15,361	93,896	3.84	2.20
54,702	229,588	17.53	54,702	229,588	9.34	52,655	226,715	9.81	5.45
170,263	526,766	41.98	170,263	526,766	22.04	165,093	521,431	24.21	13.08
491,135	1,162,225	97.74	491,135	1,162,225	50.45	479,200	1,152,823	58.01	31.25
1,325,254	2,477,180	222.36	1,325,254	2,477,180	113.12	1,299,657	2,461,304	135.76	71.07
3,370,156	5,114,350	495.8	3,370,156	5,114,350	246.88	3,318,533	5,088,491	314.99	158.25
8,127,249	10,247,830	1,077.68	8,127,249	10,247,830	525.8	8,028,436	10,206,998	-	347.16
8	26,690	8.19	**8**	26,690	6.45	**8**	26,654	5.99	3.56

LReSBDS$_{GAC}$			LReSBDS$_{incNGs}$			LReSBDS$_{WNC/GWIC}$			
$\#s$	$\#f$	t	$\#s$	$\#f$	t	$\#s$	$\#f$	t_W	t_G
424	120,815	8.54	**424**	**120,758**	8.81	**424**	121,353	10.38	**6.12**
11,047	**76,235**	4.8	**11,047**	**76,235**	3.39	**11,047**	**76,235**	6.44	**1.85**
38,007	**185,786**	11.78	**38,007**	**185,786**	8.31	**38,007**	**185,786**	17.67	**4.68**
120,832	**431,794**	27.78	**120,832**	**431,794**	19.48	**120,832**	**431,794**	46.57	**11.12**
357,662	**965,531**	63.73	**357,662**	**965,531**	44.8	**357,662**	**965,531**	116	**26.61**
991,700	**2,085,752**	142.94	**991,700**	**2,085,752**	99.75	**991,700**	**2,085,752**	291.92	**60.45**
2,590,000	**4,362,860**	309.37	**2,590,000**	**4,362,860**	221.28	**2,590,000**	**4,362,860**	702.6	136.78
6,404,281	**8,851,675**	653.30	**6,404,281**	**8,851,675**	469.64	**6,404,281**	**8,851,675**	-	300.97
8	**25,962**	6.99	**8**	**25,962**	8.66	**8**	25,985	7.19	**3.49**

Doublelex [Flener et al., 2002], and also LexLeader to break a much larger subset of symmetries. We then report the results of two dynamic methods ParSBDS and LReSBDS. Each dynamic method would be implemented with the four propagators: GAC on each nogood (GAC), the filtering algorithm of incNGs given by Lee and Zhu [2014b] ($incNGs$), WNC on each nogood (WNC) and GWIC on each incNGs constraint ($GWIC$).

We also did extra experiments to find the best subset of matrix symmetries for each method. ParSBDS is given any two rows or columns being permutable and the Cartesian products of these two subsets. LReSBDS and LexLeader can break the entire row symmetries and column symmetries by only posting adjacent rows or columns being permutable with the input order variable heuristic [Lee and Zhu, 2014a]. We thus only post adjacent rows or columns being permutable and the Cartesian products of any two rows or columns being permutable to LReSBDS and LexLeader. LDSB and ReSBDS are discarded in the comparison since LReSBDS is substantially more efficient [Lee and Zhu, 2014a; 2014b] than these two methods. All experiments are conducted using Gecode Solver 4.2.0 on Xeon E5620 2.4GHz processors with 7GB.

Due to the many columns, each table is split into three rows. The first column always gives the instance parameters. In addition, $\#s$ denotes the number of solutions, $\#f$ denotes the number of failures and t denotes the runtimes. Since WNC and GWIC have the same pruning power, we show their solutions and failures together and use t_W and t_G to denote the runtime of WNC and GWIC respectively. The search time out limit is 1 hour. An entry with the symbol " $-$ " indicates that memory is exhausted. The best results are highlighted in **bold**. *Unless otherwise specified*, search is defaulted to input variable order and minimum value order.

Error Correcting Code-Lee Distance (ECCLD)

Each ECCLD instance is parameterized by (n, c, b). We use the same model by Lee and Zhu (2014a).

Table 1 shows the results for ECCLD. ParSBDS$_{WNC}$ is 1.35 times faster than ParSBDS$_{GAC}$ on average. LReSBDS$_{WNC}$ does not perform well since the number of extra nogoods added by LReSBDS to prune extra composition symmetries is big and the symmetries are broken late. ParSBDS$_{GWIC}$ and LReSBDS$_{GWIC}$ run 1.13 and 1.15 times faster than ParSBDS$_{incNGs}$ and LReSBDS$_{incNGs}$ on average respectively. The improvement is not that much due to the small number of given symmetries. When we compare the number of failures for ParSBDS and LReSBDS, WNC and GWIC increase only slightly the search tree size. This shows our weaker consistencies lose few pruning opportunities. LReSBDS$_{GWIC}$ performs the best and runs 1.50 and 2.67 times faster than LexLeader and DoubleLex on average respectively. ParSBDS$_{GWIC}$ performs slightly slower than LReSBDS$_{GWIC}$ due to its bigger search tree size and more symmetries to handle. Thus, laziness can save us time comparing with other symmetry breaking methods.

Cover Array Problem (CA)

CA instances are parameterized by (t, k, g, b) We use the same model by Lee and Zhu (2014b).

Table 2 shows the results for CA. ParSBDS$_{WNC}$ is 1.61 times faster than ParSBDS$_{GAC}$ on average. LReSBDS$_{WNC}$ still does not perform well. ParSBDS$_{GWIC}$ and LReSBDS$_{GWIC}$ run 1.62 and 1.75 times faster than ParSBDS$_{incNGs}$ and LReSBDS$_{incNGs}$ on average respectively. For LReSBDS, WNC and GWIC only slightly increase the search tree size. LReSBDS$_{GWIC}$ performs the best and runs 1.67 and 1.82 times faster than LexLeader and Dou-

Table 3: BIBD with Maximum Value Ordering (all solutions)

v, k, λ	DoubleLex			LexLeader		
	$\#s$	$\#f$	t	$\#s$	$\#f$	t
7,3,5	33,304	191,223	**2.12**	**5,979**	**41,978**	65.64
7,3,6	250,878	1,814,425	21.06	**33,824**	**292,634**	172.44
7,3,7	1,460,332	13,149,270	154.79	**203,296**	**2,069,840**	611.51
7,3,8	6,941,124	76,463,115	886.95	-	-	-
8,4,6	2,058,523	14,156,697	157.75	**596,399**	3,873,360	**118.12**

ParSBDS$_{GAC}$			ParSBDS$_{incNGs}$			ParSBDS$_{WNC/GWIC}$			
$\#s$	$\#f$	t	$\#s$	$\#f$	t	$\#s$	$\#f$	t_W	t_G
12,936	83,578	34.95	12,936	83,578	7.25	7,916	54,608	2.39	4.84
93,713	717,959	377.91	93,713	717,959	44.37	41,388	353,232	18.07	20.50
476,752	4,486,587	3,349.82	476,752	4,486,587	270.15	226,176	2,292,110	137.22	114.92
305,312	3,583,192	3,600.00	-	-	-	1,134,253	13,599,864	-	694.02
932,022	6,450,151	2,366.52	932,022	6,450,183	281.22	925,504	6,483,468	351.32	177.09

LReSBDS$_{GAC}$			LReSBDS$_{incNGs}$			LReSBDS$_{WNC/GWIC}$			
$\#s$	$\#f$	t	$\#s$	$\#f$	t	$\#s$	$\#f$	t_W	t_G
5,979	**41,978**	13.24	**5,979**	**41,978**	5.46	**5,979**	**41,978**	2.89	4.20
33,824	**292,634**	152.99	**33,824**	**292,634**	24.58	**33,824**	**292,634**	25.14	**18.37**
-	-	-	**203,296**	**2,069,840**	145.44	**203,296**	**2,069,840**	224.6	**111.09**
-	-	-	**1,075,694**	**12,921,639**	927.34	**1,075,694**	**12,921,639**	-	**654.21**
596,399	**3,873,339**	445.98	**596,399**	**3,873,339**	169.36	**596,399**	3,956,200	287.32	129.06

bleLex on average respectively. This again shows the advantage of our lazy incNGs propagator.

Balanced Incomplete Block Design (BIBD)

A BIBD instance can be determined by its parameters (v, k, λ). We use the same model by Lee and Zhu (2014b). The value ordering is maximum value ordering and DoubleLex orders rows and columns decreasingly.

Table 3 shows the results for BIBD. ParSBDS$_{WNC}$ and LReSBDS$_{WNC}$ run 16.67 and 4.07 times faster than ParSBDS$_{GAC}$ and LReSBDS$_{GAC}$ on average respectively. One reason for the improvement is the reduction of overhead. The other reason for the good efficiency of ParSBDS$_{WNC}$ is that it prunes symmetries late and can break much more composition symmetries. ParSBDS$_{GWIC}$ and LReSBDS$_{GWIC}$ run 1.90 and 1.34 times faster than ParSBDS$_{incNGs}$ and LReSBDS$_{incNGs}$ on average respectively. The search tree size by enforcing WNC and GWIC still does not increase too much more than GAC. LReSBDS$_{GWIC}$ performs the best and runs 7.90 and 1.12 times faster than LexLeader and DoubleLex on average respectively. From the above, we can conclude that laziness can save much time and memory comparing with other symmetry breaking methods.

Conclusion and Future Work

Our contributions are four fold. First, we propose WNC for nogoods and give a lazy propagator to symmetry breaking nogoods added by SBDS-based methods. We give also the space and time complexities of the WNC lazy propagator. Second, we propose GWIC for incNGs global constraint and also give a lazy propagator and its space and time complexities. Third, we prove that GWIC on a conjunction is equivalent to WNC on the individual nogoods. Fourth, we use experiments to show the lazy methods' pruning loss is small while the gain in efficiency is worthwhile.

The improvement from the original incNGs to the lazy version is not as good as that from GAC on nogoods to WNC on nogoods. That is because the global constraint version must be incremental in nature to cater for the addition of new nogoods, and is triggered every time when new nogoods are generated, which is often. So it is not as lazy as we hope it to be. This points to new research opportunities to investigate how best to incorporate laziness into incNGs constraints.

Other possibilities include investigating symmetric nogoods collected from restarts [Lecoutre and Tabary, 2011] and detecting symmetry dynamically.

References

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

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

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

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

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

[Gent et al., 2003] Gent, I.; Harvey, W.; Kelsey, T.; and Linton, S. 2003. Generic SBDD using computational group theory. In *CP'03*, 333–347.

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

[Gent, Jefferson, and Miguel, 2006] Gent, I.; Jefferson, C.; and Miguel, I. 2006. Watched literals for constraint propagation in Minion. In *CP'06*, 182–197.

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

[Lecoutre and Tabary, 2011] Lecoutre, C., and Tabary, S. 2011. Symmetry-reinforced nogood recording from restarts. In *SymCon'11*, 13–27.

[Lee and Zhu, 2014a] Lee, J., and Zhu, Z. 2014a. Boosting SBDS for partial symmetry breaking in constraint programming. In *AAAI'14*, 2695–2702.

[Lee and Zhu, 2014b] Lee, J., and Zhu, Z. 2014b. An increasing-nogoods global constraint for symmetry breaking during search. In *CP'14*, 465–480.

[Mackworth, 1977] Mackworth, A. 1977. Consistency in networks of relations. *Artificial intelligence* 99–118.

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

[Moskewicz et al., 2001] Moskewicz, M.; Madigan, C.; Zhao, Y.; Zhang, L.; and Malik, S. 2001. Chaff: Engineering an efficient sat solver. In *DAC'01*, 530–535.

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