An Increasing-Nogoods Global Constraint for 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. Symmetry Breaking During Search (SBDS) adds conditional symmetry breaking constraints (which are nogoods) dynamically upon backtracking to avoid exploring symmetrically equivalents of visited search space. The constraint store is proliferated with numerous such individual nogoods which are weak in constraint propagation. We introduce the notion of *increasing nogoods*, and give a global constraint of a sequence of increasing nogoods, incNGs. Reasoning globally with increasing nogoods allows extra prunings. We prove formally that nogoods accumulated for a given symmetry at a search node in SBDS and its variants are increasing. Thus we can maintain only one increasing-nogoods global constraint for each given symmetry during search. We give a polynomial time filtering algorithm for incNGs and also an efficient incremental counterpart which is stronger than GAC on each individual nogood. We demonstrate with extensive experimentation how incNGs can increase propagation and speed up search against SBDS, its variants, SBDD and carefully tailored static methods.

1 Introduction

Symmetries are common in many constraint problems. They can be broken statically [18, 1, 4, 11] or dynamically [3, 8, 19]. While there are pros and cons for each approach, the focus of the paper is on SBDS (symmetry breaking during search) [8, 6] and its variants, which add conditional symmetry breaking constraints dynamically during search. ReSBDS [12] is adapted from SBDS that tries to break extra symmetry compositions with a small overhead when only a subset of symmetries is given.

An overhead for SBDS and ReSBDS is the addition of a large number of constraints with weak pruning power. We observe that the symmetry breaking constraints added for each symmetry g at a search node are nogoods that are semantically related. We propose the notion of increasing nogoods. A global constraint (incNGs), which is logically equivalent to a set of increasing nogoods, is derived. Reasoning globally with increasing nogoods allows extra prunings. Thus we can maintain only one incNGs for each given symmetry. *Light ReSBDS* adds only a subset or implied ones of the nogoods added by ReSBDS but has a smaller overhead both in time and space. We give a polynomial time filtering algorithm for incNGs and its incremental version which is stronger

* We thank the anonymous referees and Toby Walsh for their kind and insightful comments and patience.

than GAC on each individual. Extensive experimentations are performed to demonstrate how incNGs can increase propagation and speed up search against SBDS, its variants, GAP-SBDD [7] and carefully tailored static methods.

2 Background

A *constraint satisfaction problem* (CSP) 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 $D(x)$ and C is a set of constraints, each is a subset of the cross product $\bigotimes_{i \in X} D(i)$. 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* $x = v$ 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* to P is a full assignment that satisfies every member of C.

A *nogood* is the negation of a partial assignment which cannot be contained in any solution. Nogoods can also be expressed in an equivalent implication form. A *directed nogood* ng ruling out value v_k from the initial domain of variable x_k is an implication of the form $(x_{s_1} = v_{s_1}) \wedge \cdots \wedge (x_{s_m} = v_{s_m}) \Rightarrow (x_k \neq v_k)$, meaning that the assignment $x_k = v_k$ is inconsistent with $(x_{s_1} = v_{s_1}) \wedge \cdots \wedge (x_{s_m} = v_{s_m})$. When a nogood, ng, is represented as an implication, the *left hand side* (LHS) $(lhs(ng) \equiv (x_{s_1} = v_{s_1}) \wedge \cdots \wedge (x_{s_m} = v_{s_m}))$ and the *right hand side* (RHS) $(rhs(ng) \equiv (x_k \neq v_k))$ are defined with respect to the position of $\Rightarrow$. If $lhs(ng)$ is empty, ng is *unconditional*. From now on, we call directed nogoods simply as 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. 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 associated with a search tree node is always made GAC* using an AC3-like [13] *constraint filtering algorithm* except our global constraint.

Here we consider symmetry as a property of the set of solutions. A *solution symmetry* [20] 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.

Symmetry breaking during search (SBDS) [8, 6] adds constraints to a problem during search so that after backtracking from a search node, the added constraints ensure that no symmetric equivalent of that node is ever allowed in subsequent search. An advantage is that this method can break symmetries of arbitrary kind. Partial SBDS (ParSBDS) [4, 16] is SBDS but deals with only a given subset of all symmetries. LDSB [14] is a further development of shortcut SBDS [8] which handles only active symmetries

and their compositions. Recursive SBDS [12] (ReSBDS) extends ParSBDS by breaking not only the given symmetries but also some symmetry compositions.

SBDD [3, 7] is another widely used dynamic symmetry breaking method by checking whether the current state is dominated by recorded nogoods.

3 A Global Constraint for Increasing Nogoods

A set of directed nogoods is *increasing* if the nogoods can form a sequence

$$\begin{aligned} ng_0 \equiv \quad & A_0 \Rightarrow x_{k_0} \neq v_{k_0} \\ ng_1 \equiv \quad & A_1 \Rightarrow x_{k_1} \neq v_{k_1} \\ & \vdots \\ ng_t \equiv \quad & A_t \Rightarrow x_{k_t} \neq v_{k_t} \end{aligned} \tag{1}$$

such that (i) for any $i \in [1, t]$, $A_{i-1} \subseteq A_i$ and (ii) no nogoods are implied by another. We consider the nogoods as a set or a sequence according to the context.

Every sequence of increasing nogoods has the following form:

$$\begin{aligned} ng_0 \equiv \quad & x_{s_{00}} = v_{s_{00}} \wedge \cdots \wedge x_{s_{0r_0}} = v_{s_{0r_0}} \Rightarrow x_{k_0} \neq v_{k_0} \\ ng_1 \equiv \quad & lhs(ng_0) \wedge x_{s_{10}} = v_{s_{10}} \wedge \cdots \wedge x_{s_{1r_1}} = v_{s_{1r_1}} \Rightarrow x_{k_1} \neq v_{k_1} \\ \vdots \quad & \qquad\qquad \vdots \\ ng_t \equiv \quad & lhs(ng_0) \wedge \cdots \wedge lhs(ng_{t-1}) \wedge x_{s_{t0}} = v_{s_{t0}} \wedge \cdots \wedge x_{s_{tr_t}} = v_{s_{tr_t}} \Rightarrow x_{k_t} \neq v_{k_t}. \end{aligned} \tag{2}$$

A sequence of increasing nogoods can be encoded compactly, using 3 integer lists: I (index), E (equal) and N (not equal).

$$\begin{aligned} I = \langle\, & s_{00}, \ldots, s_{0r_0}, k_0, \\ & s_{10}, \ldots, s_{1r_1}, k_1, \\ & \vdots \\ & s_{t0}, \ldots, s_{tr_t}, k_t \rangle \end{aligned} \quad \begin{aligned} E = \langle\, & v_{s_{00}}, \ldots, v_{s_{0r_0}}, \bot, \\ & v_{s_{10}}, \ldots, v_{s_{1r_1}}, \bot, \\ & \vdots \\ & v_{s_{t0}}, \ldots, v_{s_{tr_t}}, \bot \rangle \end{aligned} \quad \begin{aligned} N = \langle\, & \bot, \ldots, \bot, v_{k_0}, \\ & \bot, \ldots, \bot, v_{k_1}, \\ & \vdots \\ & \bot, \ldots, \bot, v_{k_t} \rangle \end{aligned} \tag{3}$$

We encode *in order* every equality on the LHS and every disequality on the RHS of every nogood. The lists have the same length. Consider the ith tuple (I_i, E_i, N_i) from the 3 lists. If $N_i = \bot$, then the tuple is encoding $x_{I_i} = E_i$ on the LHS of a nogood. If $E_i = \bot$, then it is encoding $x_{I_i} \neq N_i$ on the RHS of a nogood.

Suppose we have the four nogoods

$$\begin{aligned} ng_0 \equiv \quad & x_1 \neq 2, \\ ng_1 \equiv \quad & x_2 = 1 \Rightarrow x_3 \neq 1, \\ ng_2 \equiv \quad & x_2 = 1 \wedge x_4 = 1 \wedge x_5 = 1 \Rightarrow x_3 \neq 2, \\ ng_3 \equiv \quad & x_2 = 1 \wedge x_4 = 1 \wedge x_5 = 1 \wedge x_6 = 2 \Rightarrow x_1 \neq 1. \end{aligned} \tag{4}$$

These nogoods are increasing because $\emptyset \subseteq \{x_2 = 1\} \subseteq \{x_2 = 1 \wedge x_4 = 1 \wedge x_5 = 1\} \subseteq \{x_2 = 1 \wedge x_4 = 1 \wedge x_5 = 1 \wedge x_6 = 2\}$ and none is implied by another. The 3 lists are derived as follows.

$$\begin{array}{rc} & \begin{array}{cccccccc} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \end{array} \\ I = & \langle\ 1\ ,\ 2\ ,\ 3\ ,\ 4\ ,\ 5\ ,\ 3\ ,\ 6\ ,\ 1\ \rangle, \\ E = & \langle\ \bot\ ,\ 1\ ,\ \bot\ ,\ 1\ ,\ 1\ ,\ \bot\ ,\ 2\ ,\ \bot\ \rangle, \\ N = & \langle\ 2\ ,\ \bot\ ,\ 1\ ,\ \bot\ ,\ \bot\ ,\ 2\ ,\ \bot\ ,\ 1\ \rangle. \end{array} \tag{5}$$

The 3 lists derived in (3) have the following feature.

Lemma 1. *Each (I_i, E_i, N_i) in I, E, N corresponds to an equality $x_{I_i} = E_i$ or disequality $x_{I_i} \neq N_i$, which can be considered as a variable-value pair. All pairs are distinct.*

An immediate consequence is to ensure the size of these 3 lists has an upper bound.

Theorem 1. *Suppose $P = (X, D, C)$ is a CSP with $|X| = n$, and I, E and N are constructed from a sequence of increasing nogoods. The maximum size of these 3 lists is $\sum_{i=0}^{n-1} D(x_i)$.*

Next we propose a global constraint that is equivalent to these nogoods but has stronger pruning power than each individual nogood. Suppose $P = (X, D, C)$ is a CSP, I, E and N are 3 lists with the same size m in the form of (3). An *increasing-nogoods global constraint* incNGs(I,E,N)(X) specifies

$$\begin{array}{ll} \forall i \in [0, m-1], N_i = \bot\ \vee\ (& (E_0 = \bot \vee\ (x_{I_0} = E_0)) \\ & \wedge\ (E_1 = \bot \vee\ (x_{I_1} = E_1)) \\ & \wedge \quad \ldots \\ & \wedge\ (E_{i-1} = \bot \vee\ (x_{I_{i-1}} = E_{i-1})) \Rightarrow x_{I_i} \neq N_i) \end{array} \tag{6}$$

meaning that if N_i is a non-$\bot$ value and all variables with indices before i in I are assigned to the corresponding value in E when there is a non-$\bot$ value, value N_i will be pruned from $D(x_{I_i})$. Note that incNGs(I,E,N)(X) is a family of global constraints parameterized by I, E and N.

Due to space limitation, we state without proof that the global constraint constructed from a sequence of increasing nogoods is logically equivalent to the conjunction of the increasing nogoods.

Theorem 2. *Suppose $\langle ng_0, \ldots, ng_t \rangle$ are increasing nogoods. We construct I, E and N as in* (3)*. Then incNGs(I,E,N)(X) is logically equivalent to $ng_0 \wedge \cdots \wedge ng_t$.*

4 Deriving incNGs(I,E,N)(X) from SBDS and Its Variants

In this section, we first introduce an adaptation of ReSBDS with a smaller overhead. Next, we prove that constraints added by SBDS or its variants accumulated from the root node to a search node for the same symmetry forms a set of increasing nogoods.

4.1 Light ReSBDS

Domain filtering prune values by an AC3-like [13] constraint filtering algorithm. If a value v is pruned during the propagation of a constraint c, we say this pruning is *effected* by constraint c.

Recursive SBDS [12] (ReSBDS) uses a backtrackable set T to record all the assignments whose violations can indicate that a symmetry breaking constraint is already satisfied. Extra constraints would be added according to these violations. Suppose $x_j = a$ is recorded in T since the constraint $A^g \Rightarrow (x_i \neq v)^g$ is added at node P_0. Suppose further this assignment is violated at a descendant node P_1, i.e. a is pruned from $D(x_j)$. The pruning indicates that $A^g \Rightarrow (x_i \neq v)^g$ is already satisfied. This pruning is effected either by a problem constraint or a symmetry breaking constraint. Considering the latter case only, we propose a light version of the ReSBDS method without T as follows.

> **[Light ReSBDS (LReSBDS)]** Suppose G is a set of symmetries. LReSBDS always adds constraints added by ParSBDS. Once a value v is pruned from $D(x_i)$ *effected by a symmetry breaking constraint* at node P_0 with a partial assignment A, symmetry breaking constraint $A^g \Rightarrow (x_i \neq v)^g$ for all $g \in G$ is added.

Here the recursive addition of constraints is done by the propagation mechanism which stops propagation only when every variable domain does not change anymore.

Due to space limitation, we state without proof the comparison between ReSBDS and LReSBDS.

Theorem 3. *ReSBDS $\succ_n$ LReSBDS and ReSBDS $\succeq_s$ LReSBDS when given the same set of symmetries and both use the same static variable and value orderings.*

We conjecture that ReSBDS $\succ_s$ LReSBDS, but we have not found an example yet.

4.2 Deriving incNGs(I,E,N)(X)

In the following, by variants of SBDS, we mean ReSBDS and LReSBDS.

All the constraints added by SBDS or its variants down a search path for the same symmetry g forms a set of increasing nogoods.

Theorem 4. *Suppose G is a set of symmetries. Suppose further at a search node P_1, the constraints added by SBDS (or its variants) accumulated from root node to P_1 for symmetry g is R, where $g \in G$. R is a set of increasing nogoods.*

During search, all children nodes inherit I, E and N from their parent node. Every time a nogood is introduced by SBDS or its variants, the 3 lists are extended to record the nogood. Supposingly, each node should post a new increasing-nogoods constraint when a new nogood is added. We show in the following, however, that an increasing-nogoods constraint for a symmetry g posted in a parent node will always be subsumed by the corresponding one posted in its child node. Thus, each search node only has to deal with one increasing-nogoods constraint for each given symmetry.

All symmetry breaking constraints added to the parent node would also be added to the child node. It is straightforward to have the following theorem.

Theorem 5. *Suppose g is a symmetry and P_0 and P_1 are search nodes, where P_1 is a descendant of P_0. Suppose P_0 constructs I, E, N and P_1 constructs I', E', N' correspondingly by SBDS (or its variants). We have incNGs(I',E',N')(X) $\Rightarrow$ incNGs(I,E,N)(X).*

To implement LReSBDS, we need to know whether the pruning is effected by problem constraints or symmetry breaking constraints. We therefore need to get access to the filtering algorithm of the symmetry breaking constraints. This can be easily implemented in the constraint filtering algorithm of incNGs for LReSBDS.

5 A Filtering Algorithm

Suppose $\langle ng_0, \ldots, ng_t \rangle$ is a sequence of increasing nogoods. A nogood ng_i is *lower than* nogood ng_j iff $i < j$, and ng_j is *higher than* ng_i.

Suppose Λ is a sequence of increasing nogoods with the current domain D. A nogood ng is *generated* by Λ iff (i) $\exists ng'$ s.t. Λ generates ng' and $\Lambda \cup \{ng'\}$ generates ng, or (ii) we can find an $x \in X$ and a subsequence $\langle ng_{s_1}, \ldots, ng_{s_p} \rangle$ of Λ where $p = |D(x)|$, such that $\forall v \in D(x), \exists j \in [1,p], x \neq v \equiv rhs(ng_{s_j})$ and $ng \equiv \neg lhs(ng_{s_p})$ with $rhs(ng)$ being the rightmost assignment in $lhs(ng_{s_p})$.

A sequence Λ of increasing nogoods is in *reduced form* iff Λ can generate no nogoods. In the rest of the paper, we *assume that Λ is always a sequence of increasing nogoods of the form $\langle ng_0, \ldots, ng_t \rangle$.*

Theorem 6. *Λ is either in reduced form or has an equivalent sequence of increasing nogoods which is in reduced form. The size of the equivalent sequence never increases.*

The reduction procedure repeatedly checks for condition (ii) to generate a new nogood as appropriate. Assume a new nogood ng is generated from Λ according to condition (ii) and ng_k is the lowest nogood in Λ such that $\neg ng \subseteq lhs(ng_k)$. Clausal resolution ensures that $\langle ng_0, \ldots, ng_{k-1}, ng, ng_k, \ldots, ng_t \rangle$ is equivalent to $\langle ng_0, \ldots, ng_t \rangle$. As ng implies $\langle ng_k, \ldots, ng_t \rangle$, $\langle ng_0, \ldots, ng_{k-1}, ng \rangle$ is equivalent to $\langle ng_0, \ldots, ng_t \rangle$. If $\langle ng_0, \ldots, ng_{k-1}, ng \rangle$ cannot generate a new nogood, it is in reduced form. Otherwise, we continue to generate new nogood from $\langle ng_0, \ldots, ng_{k-1}, ng \rangle$.

Consider the nogoods given in (4) with $D(x_1) = D(x_2) = D(x_3) = D(x_6) = \{1,2\}$ and $D(x_4) = D(x_5) = \{1\}$. Consider ng_1 and ng_2. Between $x_3 \neq 1$ and $x_3 \neq 2$, one of them must be false since x_3 must take a value. Thus we can generate the nogood $(\neg lhs(ng_1) \vee \neg lhs(ng_2)) \Leftrightarrow \neg lhs(ng_2)$, which is $\neg(x_2 = 1 \wedge x_4 = 1 \wedge x_5 = 1)$ and can be expressed as a directed nogood $ng_4 \equiv x_2 = 1 \wedge x_4 = 1 \Rightarrow x_5 \neq 1$ by putting the rightmost assignment of $lhs(ng_2)$ to the right. Now $\langle ng_0, ng_1, ng_4 \rangle$ is a new sequence of increasing nogoods equivalent to $\langle ng_0, ng_1, ng_2, ng_3 \rangle$. Consider ng_4. Domain of x_5 is a singleton. Directed nogood $ng_5 \equiv x_2 = 1 \Rightarrow x_4 \neq 1$ is generated. Now $\langle ng_0, ng_5 \rangle$ is a new sequence of increasing nogoods equivalent to $\langle ng_0, ng_1, ng_4 \rangle$. Domain of x_4 is a singleton. Directed nogood $ng_6 \equiv x_2 \neq 1$ is generated. Now $\langle ng_0, ng_6 \rangle$ is a new sequence of increasing nogoods equivalent to $\langle ng_0, ng_5 \rangle$. No new nogood can be generated and $\langle ng_0, ng_6 \rangle$ is the reduced form of $\langle ng_0, ng_1, ng_2, ng_3 \rangle$.

It might happen that more than one subsequence of Λ satisfies condition (ii). We give the **shortest nogood rule**: whenever more than one nogood can be generated from

Λ according to condition (ii), we choose the shortest one to generate. In other words, we generate using a subsequence, the highest nogood of which is the lowest in Λ.

An assignment $x = v$ satisfies the *covering condition* in Λ iff $x = v \in lhs(ng_k) \wedge x = v \notin lhs(ng_{k-1}) \wedge (\forall v' \in D(x) - \{v\}, \exists j \in [0, k-1], rhs(ng_j) \equiv x \neq v')$. In other words, when an assignment $x = v$ appears for the first time in the sequence in the LHS of a nogood (say, ng), $x \neq v'$ would have appeared on the RHSs of nogoods lower than ng for all $v' \in D(x)$ except v.

For a sequence of increasing nogoods Λ in reduced form, we consider only a **single pruning condition**: when the LHS of a nogood in Λ is true, its RHS is enforced to effect value pruning.

The reduction procedure as described seems to call for scanning the sequence of increasing nogoods repeatedly, which is inefficient. In the following, we explain how the covering condition allows us to scan the nogood sequence only once to transform it into reduced form.

Consider the nogoods in (4) again. If, instead, $D(x_5) = \{1, 2\}$, ng_4 in $\langle ng_0, ng_1, ng_4 \rangle$ cannot generate ng_5 since condition (ii) is not met. This is because $x_5 = 2$ does not satisfy the covering condition. We state without proof the following theorem.

Theorem 7. *Assume Γ is a new and equivalent sequence of increasing nogoods transformed from Λ according to the* **shortest nogood rule** *with the current domain. Γ can generate further new nogoods only if the assignment $\neg rhs(ng)$ satisfies the covering condition in Λ, where ng is the highest nogood in Γ.*

Consider the nogoods in (4) again with the variable domains on page 6: $D(x_1) = D(x_2) = D(x_3) = D(x_6) = \{1, 2\}$ and $D(x_4) = D(x_5) = \{1\}$. We scan from ng_0 and up. Assignment $x_2 = 1$ does not satisfy the covering condition since $rhs(ng_0) \not\equiv x_2 \neq 2$. In ng_2, assignments $x_4 = 1$ and $x_5 = 1$ satisfy the covering condition since they have singleton domains. Now the new nogood $ng_4 \equiv \neg lhs(ng_2)$ is generated whose RHS is $x_5 \neq 1$. Another new nogood $ng_5 \equiv \neg lhs(ng_4)$ can be generated immediately without rescanning from the first nogood. Since $\neg rhs(ng_5) \equiv x_4 \neq 1$, another new nogood $ng_6 \equiv \neg lhs(ng_5)$ can be generated immediately. Now $\neg rhs(ng_6) \equiv x_2 \neq 1$, which does not satisfy the covering condition. We can stop and $\langle ng_0, ng_6 \rangle$ is equivalent to the original sequence and in reduced form.

As a result, we only have to scan the sequence from ng_0 and up. For every nogood, we check if condition (ii) is met for nogoods from ng_0 up to here. In addition, we also check and record whether new assignments on the LHS satisfies the covering condition or not. Once the first reduction step is launched, there is no further need to check for condition (ii). We stop once $\neg rhs(ng)$ does not satisfy the covering condition, where ng is the last generated nogood.

In our filtering algorithm, the first step is to effect prunings using nogoods whose LHSs are true under the current domain. These nogoods can then be thrown away. The remaining nogoods still form an increasing sequence, which can be turned into reduced form. The single pruning condition for reduced forms is still expensive to check. It turns out that if the leftmost assignment in the LHS of the first nogood in a sequence is not true under the current domain, we can detect pruning in the reduced form much more efficiently. This form is easy to get.

Lemma 2. *Suppose $x = v$ in $lhs(ng_0)$ is the leftmost assignment which is not true under the current domain and Δ is the set of assignments before $x = v$ in $lhs(ng_0)$. Suppose further $\Gamma \equiv \langle ng_{s_0}, \ldots, ng_{s_t} \rangle$ is a sequence of increasing nogoods such that $\forall i \in [0,t], lhs(ng_{s_i}) = lhs(ng_i) - \Delta, rhs(ng_{s_i}) = rhs(ng_i)$. Γ is equivalent to Λ and is the simplified version of Λ.*

Suppose we have an increasing sequence with the form as given in the last lemma. By repeated applications of the **single pruning condition**, we get the following result which gives a much simplified pruning condition.

Theorem 8. *Suppose Γ is the reduced form of Λ where the first assignment in $lhs(ng_0)$ is not true under the current domain D. A value $v \in D(x)$ is pruned in Γ if $\Gamma \equiv \langle x \neq v \rangle$, where $x \neq v$ is an unconditional nogood.*

In the following, we give an efficient filtering algorithm based on the following two major steps: effect prunings for nogoods whose LHSs are true and transform the simplified version of the remaining sequence of increasing nogoods into reduced form. We explain our algorithm using the I, E and N encodings of Λ. Pointer α is used to index into the 3 lists to effect prunings for nogoods whose LHSs are true.

- Pointer α is set to the *largest* index such that $(\forall i \in [0,\alpha), E_i = \bot \vee x_{I_i} = E_i)$. Note that α points at an unsatisfied equality in $lhs(ng_p) - lhs(ng_{p-1})$ where $p \in [0,t]$ and ng_p is the lowest nogood whose LHS is not true. For all the nogoods lower than ng_p, their RHSs can be enforced to prune values.

Therefore, we examine the LHSs of remaining nogoods starting from α. To transform the simplified version $\langle ng'_p, \ldots, ng'_t \rangle$ of remaining increasing nogoods $\langle ng_p, \ldots, ng_t \rangle$ into reduced form, two pointers β and γ are used to index into the 3 lists with the following conditions.

- Pointer β is used to find the shortest nogood generated by Λ according to condition (ii). If a new nogood is generated, γ is then used to check whether there are extra nogoods that can be generated according to Theorem 7 and find the last generated nogood. If extra nogoods can be generated, β is set to γ. Initially, β is set to the *largest* index such that $\forall i \in [0,\beta), E_i \in D(x_{I_i}) \vee (E_i = \bot \wedge\ D(x_{I_i}) \neq S_{I_i})$, where $S_{I_i} = \{N_j | I_j = I_i, N_j \neq \bot, N_j \in D(x_{I_i}), j \in [0,i]\}$. The key concept is S_{I_i}, which is the collection of all values N_j still in $D(x_{I_i})$ such that $x_{I_i} \neq N_j$ is a RHS disequality that appears before or at the nogood encoded at i. Note that β points to either
  (a). an equality in $lhs(ng_q) - lhs(ng_{q-1})$ where $q \in [0,t]$, in which case $E_\beta \neq \bot$ and $E_\beta \notin D(x_{I_\beta})$, i.e. ng_q is the lowest nogood whose LHS is false. All nogoods ng_i where $t \geq i \geq q$ are satisfied. We only need to enforce $\langle ng'_p, \ldots, ng_{q-1} \rangle$. No new nogoods can be generated and this increasing nogoods is in reduced form. We say β satisfies condition (a); or
  (b). the disequality in $rhs(ng_q)$ where $q \in [0,t]$, in which case $D(x_{I_\beta}) = S_{I_\beta}$, i.e. ng_q with $rhs(ng_q) \equiv x_{k_q} \neq v_{k_q}$ is the lowest nogood such that all values in the domain of $x_{k_q} (= x_{I_\beta})$ have appeared in $rhs(ng_j)$ for all $j \in [0,q]$. Now new nogood $ng \equiv \neg lhs(ng_q)$ is generated and is the shortest one. Increasing nogoods $\langle ng'_p, \ldots, ng \rangle$ are formed. We say β satisfies condition (b).

– If β satisfies condition (b), the first reduction step is launched since the new nogood ng is generated. Pointer γ is set in such a way that all equalities $x_{I_i} = E_i$ for $i \in [\gamma, s]$, where $\neg rhs(ng) \equiv x_{I_s} = E_s$, satisfy the covering condition. Pointer γ is set to the *smallest* index such that $(\forall i \in (\gamma, \beta), (E_i = \bot) \vee (D(x_{I_i}) = S_{I_i} \cup \{E_i\})) \wedge D(x_{I_\gamma}) = S_{I_\gamma} \cup \{E_\gamma\}$. Note that γ points to the disequality in $rhs(ng')$ where $\neg lhs(ng')$ is the highest nogood of increasing nogoods in reduced form (e.g. for nogoods in (4) with variable domains on page 6, ng' is ng_5 and $\neg lhs(ng')$ is ng_6). If γ takes a value, new nogoods can be generated and β is set to γ. Otherwise, we do not need to update β. Now β still satisfies condition (b).

After updating β according to the above, we have found the increasing nogoods in reduced form. If β still satisfies condition (b), i.e. new nogoods have been generated, we need to check whether values can be pruned according to Theorem 8.

Consider the nogoods given in (4) with variable domains on page 6. If incNGs(I,E,N)(X) has m as the size of the 3 lists, we do propagation in the following ways.

1. Pointer $\alpha = 0$, $\beta = m = 3$ and $\gamma = \bot$.
2. To find α, we scan with i from 0 to $m - 1$.
   – $i = 0$: prune N_0 from $D(x_1)$ $(= D(x_{I_0}))$.
   – $i = 1$: $E_1 \neq \bot$ and E_1 has not been assigned to $x_2 (= x_{I_1})$. Stop scanning and set $\alpha = 1$.
3. To find β, we scan with i from α to $m - 1$.
   – $i = 1$: $E_1 \in D(x_2)$ $(= D(x_{I_1}))$.
   – $i = 2$: $E_2 = \bot$ but value 1 in domain $D(x_3) (= D(x_{I_2}))$ is not in N_j for all $j \leq i$ and $I_j = I_2$.
   – $i = 3$: $E_3 \in D(x_4)$ $(= D(x_{I_3}))$.
   – $i = 4$: $E_4 \in D(x_5)$ $(= D(x_{I_4}))$.
   – $i = 5$: $D(x_{I_5}) = D(x_3) \subseteq \{1, 2\} = \{N_2, N_5\} = S_3$ since $I_2 = I_5 = 3$. Stop scanning and set $\beta = 5$. Now β satisfies condition (b).
4. Pointer β satisfies condition (b), the first reduction step is launched. To find γ, we scan with i from α to $\beta - 1$.
   – $i = 1$: $E_1 \in D(x_2)$ $(= D(x_{I_1}))$ but $S_2 \cup \{E_1\} \neq D(x_2)$.
   – $i = 2$: $E_2 = \bot$.
   – $i = 3$: $E_3 \in D(x_4)$ $(= D(x_{I_3}))$ and $S_4 \cup \{E_3\} = D(x_4)$, set $\gamma = 3$.
   – $i = 4$: $E_4 \in D(x_5)$ $(= D(x_{I_4}))$ and $S_5 \cup \{E_4\} = D(x_5)$, γ is still 3.
5. Pointer β satisfies condition (b) and $\gamma \neq \bot$, set $\beta = \gamma$. Now β points to the disequality in RHS of the newly generated nogood $ng' \equiv x_2 = 1 \Rightarrow x_4 \neq 1$ and $ng \equiv \neg lhs(ng')$ is the final generated nogood. Since $lhs(ng)$ is empty, value 1 is pruned from $D(x_2)$. The propagation is done.

After the propagation, $D(x_1) = \{1\}, D(x_2) = \{2\}, D(x_3) = \{1, 2\}$, $D(x_4) = D(x_5) = \{1\}$, $D(x_6) = \{1, 2\}$. GAC on individual nogoods can only prune value 2 from $D(x_1)$. Our filtering prunes also 1 from $D(x_2)$. The step to find β and γ can be done at the same time. We only need to check whether β should be set to γ or not.

Suppose $P = (X, D, C)$ is a CSP where the size of X is n. We give the filtering algorithm for an incNGs(I,E,N)(X) as follows. For the moment, please ignore highlighted codes in frame boxes, which are reserved for the incremental version of the algorithm.

Algorithm 1 *InGEnforce*()

Require:
X, D, I, E, N
m: the size of I, E and N
$\alpha = 0$
$\beta = m$ ▲$\beta = \sum_{i=0}^{n-1} D(x_i)$
$\gamma = \bot$
$p = 0$: reason of why β is updated

1: **if** $m = 0$ **then**
2: return **ENTAILED**;
3: **end if**
4: *UpdateAlpha*();
5: *UpdateBeta*();
6: **if** $\alpha = \beta$ **then**
7: **if** $p = 1$ **then**
8: return **ENTAILED**;
9: **end if**
10: **if** $p = 2$ **then**
11: return **FAILED**;
12: **end if**
13: **end if**
14: **if** $p = 2$ **then**
15: return *CheckE*();
16: **end if**

Algorithm 1 is the top level of the filtering algorithm. This algorithm is called whenever the domain of a variable in X is modified or I, E and N are extended. The pointer α is initialized to 0, β is initialized to the size of the 3 lists and γ is set to $\bot$. Integer variable p tells the reason of why β is updated, i.e. β satisfies which condition. If the 3 lists are empty (Lines 1-3), the constraint is automatically **ENTAILED**, which means the constraint can be disposed. Line 4 calls the function *UpdateAlpha*() to update the pointer α. Line 5 calls the function *UpdateBeta*() to update the pointer β according to shortest nogood rule and Theorem 7. Lines 6-13 check whether $\alpha = \beta$ or not. If it is true and β is updated as a result of condition (a), this constraint is **ENTAILED** since future pruning can take place only between α and $\beta - 1$. If the two pointers are equal and β is updated because of the condition (b), this constraint is **FAILED** since LHS of nogood at α $(= \beta)$ is satisfied but the negation of the LHS of this nogood is a nogood. Thus the current node should fail. Lines 14-16 calls the function *CheckE*() to check whether the last generated nogood satisfies the condition in Theorem 8.

In the following, *Prune*(v, x) prunes value v from $D(x)$. While x.*assigned*(v) checks whether x is assigned with value v, x.*in*(v) checks whether $v \in D(x)$ and x.*size*() returns $|D(x)|$. We assume the last three functions have constant time complexity.

Algorithm 2 *UpdateAlpha*()

1: int $i = 0$; ▲int $i = \alpha$;
2: **while** $i < m$ ♦and $i < \beta$ **do**
3: **if** $E_i = \bot$ **then**
4: *Prune*(N_i,x_{I_i});
5: **else**
6: **if** $\neg x_{I_i}$.*assigned*(E_i) **then**
7: break;
8: **end if**
9: **end if**
10: $i = i + 1$;
11: **end while**
12: $\alpha = i$;

Algorithm 2 updates α. It starts scanning from index $i = 0$, and stops only when x_{I_i} is not assigned with non-$\bot$ value E_i (lines 6-8), in which case α is set to i (line 12). During scanning, if $E_i = \bot$, since the LHS of the nogood at this point is true, N_i can be pruned from $D(x_{I_i})$ (lines 3-4).

Algorithm 3 *UpdateBeta*()

```
1: int i = α;
2: int S[n];
3: for each j ∈ [0, n − 1] do
4:     S[j] = 0;
5: end for
6: while i < m ♦and i < β do
7:     if x_{I_i}.in then(E_i)
8:         if γ ≠⊥ ∧S[I_i] ≠ x_{I_i}.size()-1 then
9:             γ =⊥;
10:        else
11:            if γ =⊥ ∧S[I_i] = x_{I_i}.size()-1 then
12:                γ = i;
13:            end if
14:        end if
15:        continue;
16:    end if
17:    if E_i ≠ ⊥ then
18:        if ¬x_{I_i}.in(E_i) then
19:            p = 1;
20:            break;
21:        end if
22:    else
23:        if x_{I_i}.in(N_i) then
24:            S[I_i] + +;
25:        end if
26:        if S[I_i] = x_{I_i}.size() then
27:            p = 2;
28:            break;
29:        end if
30:    end if
31:    i = i + 1;
32: end while
33: if i ≠ m then
34:    if p = 2 ∧ γ ≠⊥ then
35:        β = γ;
36:    else
37:        β = i;
38:    end if
39: end if
```

Algorithm 3 updates β. As $\beta \geq \alpha$, the scan starts from α. We use an array $S[]$, so that $S[i]$ records the number of encountered values during scanning for each variable x_i in the disequalities on the RHS of nogoods. $S[i]$ does not count values already pruned from the domain of x_i (lines 23-25). Lines 7-16 updates γ. Lines 8 and 9 reset γ to $\bot$ if the covering condition is not satisfied. Lines 11-13 set γ to i if γ is $\bot$ and the covering condition is satisfied. Lines 17-21 check if scanning should stop due to condition (a) and set the reason p for updating β, before updating β in lines 33-39. Lines 26-29 check if scanning should stop due to condition (b) and set the reason p for updating β, before updating β in lines 33-39. If the scanning is stopped due to condition (b) and γ is a non-$\bot$ value (line 34), β is set to γ (line 35). Or else, β is set to the interrupted i.

Algorithm 4 *CheckE*()

```
1: int i = β − 1;
2: while i > α do
3:     if E_i ≠ ⊥ then
4:         break;
5:     end if
6:     i = i − 1;
7: end while
8: if i = α then
9:     Prune(E_α,x_{I_α});
10:    return ENTAILED;
11: end if
12: return CONSISTENT;
```

If β is updated because of condition (b), Algorithm 4 is called. Lines 2-7 first check whether there exists non-$\bot$ value in E from $\alpha + 1$ to $\beta - 1$. If yes (line 12), $x_{I_\alpha} = E_\alpha$ must be in the LHS of the last generated nogood which does not satisfy the condition in Theorem 8. This constraint is **CONSISTENT** means that the domain filtering is done. Now the nogoods between α and β consist of the increasing nogoods in reduced form.

If not (lines 8-11), value E_α is pruned from its corresponding variable's domain since $x_{I_\alpha} \neq E_\alpha$ is the only nogood in the increasing nogoods in reduced form. Now this constraint is **ENTAILED** as $\beta = \alpha$.

We do not have an exact characterization on Algorithm 1's consistency level yet, but it is stronger than GAC on individual nogoods and has a polynomial time complexity.

Theorem 9. *Algorithm 1 terminates and enforces a consistency on incNGs(I,E,N)(X) that is strictly stronger than GAC on each individual nogood.*

Theorem 10. *Algorithm 1 runs in $O(db|X|)$ for constraint incNGs(I,E,N)(X), where d is the largest domain size and b is the cost of pruning a value from a variable domain.*

6 Incremental Filtering Algorithm

Though polynomial, Algorithm 1 is expensive to execute from scratch at every invocation of the global constraint during (a) constraint propagation within an AC3-like algorithm and (b) adding new increasing nogoods during search (advancing to child nodes during search and generating extra nogoods during propagation in recursive methods). We can make Algorithm 1 incremental using the following theorems.

Theorem 11. *For a global constraint incNGs(I,E,N)(X), whenever Algorithm 1 is invoked during AC3-like constraint filtering algorithm, the two pointers α and β can be carried over from the last invocation.*

Theorem 12. *Suppose I, E and N are constructed from increasing nogoods Λ. Suppose further that I', E' and N' are constructed from increasing nogoods $\langle ng_0, \ldots, ng_t, ng_{t+1} \rangle$. The two pointers α' and β' for enforcing incNGs(I',E',N')(X) can be initialized to the values of α and β respectively after domain filtering of incNGs(I,E,N)(X) with Algorithm 1.*

The incremental filtering algorithm can be obtained by *adding* the highlighted ones marked by ♦ and *substituting* codes by ones marked by ▲ to the right. Note that at the start of Algorithm 1, the initialization for α, β, γ and p is only for the root node. In subsequent nodes, these four are initialized from the ones after the latest propagation or from the ones of the previous global constraint after its domain filtering.

7 Experiments

This section gives four experiments to demonstrate empirically how globalized SBDS, ParSBDS and ReSBDS can improve the runtime substantially over their original versions. We also implemented the global version of LReSBDS but not the one using decomposed nogooods. When available, we compare our results also against state of the art static methods. All experiments are conducted using Gecode Solver 4.2.0 on Xeon E5620 2.4GHz processors.

SBC uses the static method by Puget [17] to break all variable symmetries and the value symmetries in all-different problems. **Doublelex** [4] lexicographically orders the

rows and columns in increasing order. **SBDS** uses SBDS to break *all* symmetries. **ParSBDS** and **ReSBDS** handle the given symmetries by ParSBDS and ReSBDS respectively. For matrix problems, **ReSBDS**[c] is given as symmetries that adjacent rows (columns) are interchangeable and also cartesian-product of adjacent row symmetries and adjacent column symmetries. The globalized version of **SBDS**, **ParSBDS**, **ReSBDS** and **ReSBDS**[c] are denoted by **[incNGs]**$_S$, **[incNGs]**$_P$, **[incNGs]**$_R$ and **[incNGs]**$_R$[c] respectively. By using the globalized version of LReSBDS and give the same symmetries as **ReSBDS** and **ReSBDS**[c] respectively, we have **[incNGs]**$_{LR}$ and **[incNGs]**$_{LR}$[c]. For decomposed nogood implementation, we use clause constraint whose propagation uses two-watched literals [15, 21]. For GAP-SBDD and GAP-SBDS, we do not have their implementation in Gecode, and provide an indirect but machine-independent comparison using results in the literature. LDSB is discarded in the comparison here since ReSBDS is substantially more efficient [12]. *Unless otherwise specified*, we search with input variable order and minimum value order.

In all experiments, we show only the runtime to find all solutions. Runtime is limited to 1 hour. The number of backtracks and number of solutions are in line with the theoretical predictions. We did not report them only because of lack of space. All results are shown in graphical form for easy visualization. The horizontal axis shows instances, and the vertical axis shows the runtime in seconds. N-Queens instances are sorted by size. In the other three experiments, instances are sorted by the runtime of static methods. The last two experiments use *log* graph for better visualization. *Dashed lines* give the results of methods using decomposed nogoods and *solid lines* are for methods using global constraints. *Solid lines with '+'* shows the results for static methods.

7.1 N-Queens

We model the N-Queens problem the standard way using one variable per column. All 8 geometric symmetries are given to **SBDS**. **ParSBDS** and **ReSBDS** are only given the two generators rx (reflection on the vertical axis) and $d1$ (reflection on the diagonal), which can generate all 8 geometric symmetries.

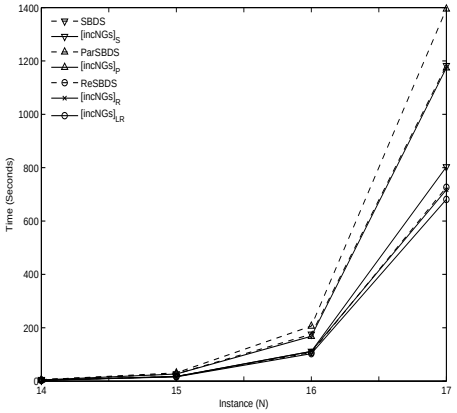

Fig. 1. N-Queens problem

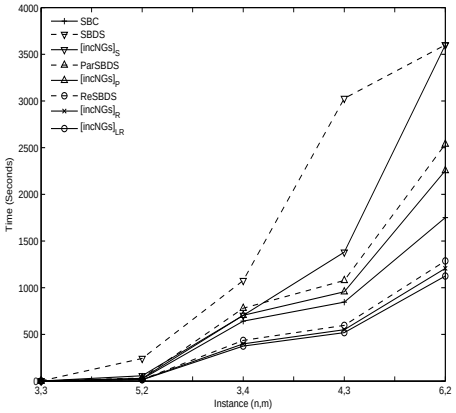

Fig. 2. The Graceful Graph problem

Fig. 1 shows the results. For complete methods, **[incNGs]**$_S$ is up to 1.82 times faster than **SBDS**, and **[incNGs]**$_S$ has up to 434825 less failures than **SBDS**. For partial symmetry breaking methods, only two symmetries are given and these two symmetries would be broken high up in the search tree. For partial SBDS, **[incNGs]**$_P$ is up to 1.23 times faster than **ParSBDS**. For ReSBDS, **[incNGs]**$_R$ improves only a little over **ReSBDS** due to the overhead to get to know when I, E and N are updated. And **[incNGs]**$_{LR}$ is the most efficient and faster than **ReSBDS**. Note that **[incNGs]**$_S$ is complete and comparable with **ReSBDS** and **[incNGs]**$_{LR}$. This shows how the global constraint can help to prune all symmetric solutions in a competitive manner.

7.2 Graceful Graph

The graceful graph problem is an all-different problem [16]. A $K_n \times P_m$ graph has intra-clique permutations, inter-clique permutations, complement symmetry, and their combinations. **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** is given $(n-1)$ symmetries to describe simultaneous permutation of adjacent nodes in each clique and also one inter-clique permutation and one complement symmetry.

Fig. 2 shows the results. For complete methods, **[incNGs]**$_S$ runs up to 4.25 times faster than **SBDS**, and **[incNGs]**$_S$ has up to 169286 less failures than **SBDS**. This shows the global constraint improves our complete method dramatically. For ParSBDS and ReSBDS, **[incNGs]**$_P$ and **[incNGs]**$_R$ are up to 1.13 and 1.09 times faster than **ParSBDS** and **ReSBDS** respectively as only a small subset of symmetries are given. Using LReSBDS, **[incNGs]**$_{LR}$ is up to 1.16 times faster than **ReSBDS** and is even up to 1.71 times faster than **SBC**. Literature results [17] show that SBC is up to 15 times faster than GAP-SBDD and GAP-SBDS. This demonstrates LReSBDS with global constraints can beat GAP-SBDD, GAP-SBDS and carefully tailored static methods.

7.3 Balanced Incomplete Block Design

A BIBD instance can be determined by its parameters (v, k, λ). We use the 0/1 model [5], which has row and column symmetries since we can permute any rows or columns freely without affecting any of the constraints. **ParSBDS** is given the symmetry that any two rows (columns) are interchangeable. **ReSBDS** is given interchangeability of adjacent rows (columns). All are solved with the maximum value heuristic.

Fig. 3 shows the results for BIBD. For partial SBDS, **[incNGs]**$_P$ runs up to 2.26 times faster than **ParSBDS**. For ReSBDS, the global constraint cannot help much due to the overhead to get to know when I, E and N are updated. Note that **[incNGs]**$_R$[c] is even 1.58 times slower than **ReSBDS**[c]. For light ReSBDS, however, **[incNGs]**$_{LR}$ and **[incNGs]**$_{LR}$[c] are up to 1.46 and 1.32 times faster than **ReSBDS** and **ReSBDS**[c] respectively. The light version improves a lot over ReSBDS. Note that **[incNGs]**$_{LR}$[c] is even 3.03 times faster than **DoubleLex**. The gains come from the advantage of LReSBDS by posting more symmetries and the efficiency of the global constraint. Literature results [7, 6] show that GAP-SBDD is about 4 times faster than GAP-SBDS and

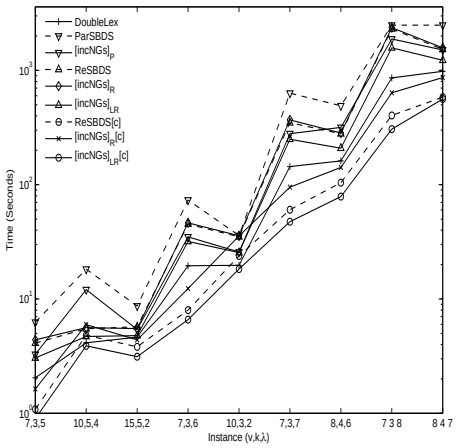

Fig. 3. The BIBD problem

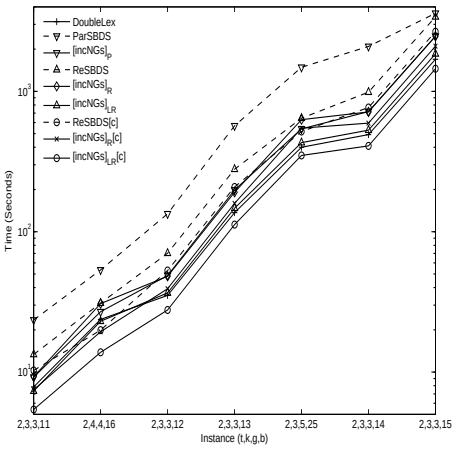

Fig. 4. The CA problem

DoubleLex is at least 10 and up to 38 times faster than GAP-SBDS, we can conclude indirectly that **[incNGs]**$_{LR}$[c] can beat GAP-SBDD and GAP-SBDS dramatically.

7.4 Cover Array Problem (CA)

The Cover Array Problem CA(t, k, g, b) is prob045 in CSPLib [9]. We use the integrated model [10], which channels an original model and a compound model. **ParSBDS** and **ReSBDS** are given the same set of symmetries as in BIBD.

Fig. 4 shows the results. For dynamic methods, **[incNGs]**$_P$, **[incNGs]**$_R$ and **[incNGs]**$_R$[c] run up to 2.91, 1.46 and 1.37 times faster than the decomposed ones, and have up to 3014, 3938 and 33383 less failures than the decomposed ones. While **[incNGs]**$_{LR}$ and **[incNGs]**$_{LR}$[c] are up to 1.92 and 1.91 times faster than **ReSBDS** and **ReSBDS**[c] respectively. Note that for the case CA$(2, 4, 4, 16)$, **ReSBDS**[c], **[incNGs]**$_R$[c] and **[incNGs]**$_{LR}$[c] leave 2250, 2076 and 2100 solutions respectively. This demonstrates that with global constraint, ReSBDS prunes more solutions than LReSBDS, and our domain filtering on global constraint can prune more solutions than GAC on each individual nogood. When compared with static methods, the best one **[incNGs]**$_{LR}$[c] runs up to 1.72 times faster than **DoubleLex**. This shows how LReSBDS with global constraint is competitive against static methods.

8 Conclusion and Future Work

Our contributions are five-fold. First, based on the special semantics and structures of increasing nogoods, we propose a global constraint with equivalent meaning but stronger pruning power. Second, we demonstrate that nogoods added by SBDS and its variants are increasng so that the methods can be adapted with the global constraints. Third, benefitting from the global constraint, we devise a light version of ReSBDS with smaller space and time overheads. Fourth, we give a polytime filtering algorithm for the increasing-nogoods constraint, which also has an efficient and simple incremental version. Fifth, extensive experimentations confirm the efficiency of our proposals.

ReSBDS has the advantage that a substantial number of symmetries can be broken with only a small given subset of them. The increasing-nogoods global constraint reduce the overhead of SBDS and its variants dramatically, making it possible to handle larger set of given symmetries which in turn can prune more search space.

Nogood learning is a general technique for improving backtracking search [2]. We envision that the increasing-nogoods constraint is applicable to other scenarios in CP, in addition to symmetry breaking.

References

1. Crawford, J., Ginsberg, M., Luks, E., Roy, A.: Symmetry breaking predicates for search problems. In: KR'96. pp. 148–159 (1996)
2. Dechter, R.: Enhancement schemes for constraint processing: Backjumping, learning, and cutset decomposition. Artificial Intelligence pp. 273–312 (1990)
3. Fahle, T., Schamberger, S., Sellmann, M.: Symmetry breaking. In: CP'01. pp. 93–107 (2001)
4. Flener, P., Frisch, A., Hnich, B., Kiziltan, Z., Miguel, I., Pearson, J., Walsh, T.: Breaking row and column symmetries in matrix models. In: CP'02. pp. 187–192 (2002)
5. Flener, P., Frisch, A., Hnich, B., Kiziltan, Z., Miguel, I., Walsh, T.: Matrix modelling. In: ModRef'01 (2001)
6. Gent, I.P., Harvey, W., Kelsey, T.: Groups and constraints: Symmetry breaking during search. In: CP'02. pp. 415–430 (2002)
7. Gent, I.P., Harvey, W., Kelsey, T., Linton, S.: Generic SBDD using computational group theory. In: CP'03. pp. 333–347 (2003)
8. Gent, I., Smith, B.: Symmetry breaking in constraint programming. In: ECAI'00. pp. 599–603 (2000)
9. Gent, I., Walsh, T.: CSPLib: a benchmark library for constraints. In: CP'99. pp. 480–481 (1999)
10. Hnich, B., Prestwich, S.D., Selensky, E., Smith, B.M.: Constraint models for the covering test problem. Constraints pp. 199–219 (2006)
11. Law, Y.C., Lee, J.: Global constraints for integer and set value precedence. In: CP'04. pp. 362–376 (2004)
12. Lee, J., Zhu, Z.: Boosting SBDS for partial symmetry breaking in constraint programming. In: (to appear) AAAI'14 (2014)
13. Mackworth, A.: Consistency in networks of relations. Artificial intelligence pp. 99–118 (1977)
14. Mears, C., de la Banda, M.G., Demoen, B., Wallace, M.: Lightweight dynamic symmetry breaking. Constraints pp. 1–48 (2013)
15. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient sat solver. In: DAC'01. pp. 530–535 (2001)
16. Petrie, K.E., Smith, B.M.: Symmetry breaking in graceful graphs. In: CP'03. pp. 930–934 (2003)
17. Puget, J.: Breaking symmetries in all different problems. In: IJCAI'05. pp. 272–277 (2005)
18. Puget, J.F.: On the satisfiability of symmetrical constrained satisfaction problems. In: ISMIS'93. pp. 350–361 (1993)
19. Roney-Dougal, C.M., Gent, I.P., Kelsey, T., Linton, S.: Tractable symmetry breaking using restricted search trees. In: ECAI'04. pp. 211–215 (2004)
20. Rossi, F., Van Beek, P., Walsh, T.: Handbook of constraint programming. Elsevier (2006)
21. Zhang, L., Malik, S.: The quest for efficient boolean satisfiability solvers. In: CAV'02. pp. 17–36 (2002)