

Lightweight Dynamic Symmetry Breaking

C. Mears, M. Garcia de la Banda, B. Demoen, M. Wallace

Abstract

LDSB is a dynamic symmetry breaking method and, therefore, it does not interfere with the given search heuristic. Like the shortcut SBDS method, it might trade completeness for efficiency, and does not require complex group theory computations. Like the GAP-SBDS method, it only requires the user to provide the symmetries, and pruning of symmetric parts of the search tree occurs as early as it is useful. Importantly, the overhead introduced by LDSB to exploit symmetries is very small, since it targets those that can be represented and checked efficiently. An implementation of LDSB is described. Results are so promising that we believe LDSB can be used as a default search method even when looking only for the first solution.

1 Introduction

Symmetry breaking in constraint satisfaction problems can lead to speedups, since exploring parts of the search tree that are symmetric to those already explored is unnecessary. There are two main approaches to symmetry breaking *static* and *dynamic*.

Static symmetry breaking alters the original problem by adding new constraints that are satisfied by a single representative of each group of symmetric solutions (thus eliminating symmetric solutions). Dynamic symmetry breaking alters the search, rather than the problem. Unfortunately, both approaches can result in a slowdown, rather than a speedup. Static approaches introduce a small run-time overhead but might interfere with the search strategy used. Dynamic approaches, like SBDD [Focacci and Milano, 2001; Fahle *et al.*, 2001] and SBDS [Backofen and Will, 1999; Gent and Smith, 2000], do not interfere with the search but might introduce a significant amount of overhead.

As shown in [Flener *et al.*, 2006], it is possible to add static constraints that do not interfere with the search for problems in which the static variable and value ordering can be chosen in advance. If so, the static symmetry breaking method explores the same search tree that

would be obtained using SBDS. However, determining in advance the best order is often not possible.

Our aim is to develop a practical and general symmetry breaking method, i.e., one that can be used for any CSP with any search strategy, while introducing as little overhead as possible. In particular, the overhead needs to be small enough for our method to be used as a default method. For these two reasons, we decided to focus on improving the practicality of dynamic symmetry breaking and, in particular, on the SBDS method. This is because SBDS results in equal or stronger pruning than SBDD [Petrie and Smith, 2003]. Note however, that having stronger pruning does not always mean faster execution: SBDS can perform worse than SBDD when the number of symmetries is too high. To achieve our aim we must therefore significantly reduce this overhead.

We decided to do this by targeting symmetries that are common in CSPs and can be handled easily, with little time and space complexity, and without the need to rely on group theory packages such as GAP. Moreover, we are willing to give up breaking all the symmetries, if that yields better performance. The shortcut SBDS method informally introduced in [Gent and Smith, 2000] by means of several examples, shares some of our goals: restrict the symmetries to those that can be efficiently dealt with, and trade completeness in favour of performance.

Our work can thus be seen as a further development of the shortcut SBDS: we formalise and generalise it, and offer a uniform framework for optimisations that otherwise appear scattered in the literature. In particular, our contributions are as follows. First, we identify very common symmetry groups that can be efficiently broken. Second, we define an instance of the shortcut method for these symmetries and prove the method is correct and sometimes complete. Third, we extend the shortcut method to be able to combine symmetries. Fourth, we provide interesting insights regarding the particular time at which the posting of symmetry breaking constraints occurs in dynamic symmetry breaking methods. Fifth, we provide a thorough experimental evaluation whose results show that LDSB significantly reduces the overhead of the operations required to exploit the symmetries, compared to other methods: LDSB is uniformly

faster than GAP-SBDS and faster than GAP-SBDD in all but three benchmark instances. This confirms the validity of the shortcut approach, i.e., that trading completeness for performance is often a good choice. The results are so promising that we believe LDSB can be used as a default search method even when looking only for the first solution. Finally, LDSB is implemented in ECLⁱPS^e with an interface similar to the `ic_gap_sbds` and `ic_gap_sbdd` libraries, and can therefore be easily used as an alternative not only in ECLⁱPS^e but also in any Prolog system, even if it does not have an interface to GAP.

Section 2 introduces the necessary background on symmetries in CSPs, on the SBDS method and its shortcut and GAP variants. Section 3 presents the symmetry groups specifically targeted by LDSB. Section 4 describes how the targeted symmetry groups are represented in LDSB, how this representation evolves during search, and how it is used to achieve pruning. Section 5 discusses in more detail the LDSB implementation. Section 6 shows experimentally the excellent performance of LDSB. Section 7 indicates future work and concludes.

2 Background

2.1 Basic symmetry definitions

A CSP is a tuple (X, D, C) where X represents a set of variables, D a set of possible values for these variables, and C a set of constraint(s). Given a CSP, a *literal* is of the form $x = d$ where $x \in X$ and $d \in D$. The set of all literals of a CSP P is denoted by $lit(P)$. An *assignment* A over a set of variables $V \subseteq X$ is a set of literals with exactly one literal $x = d$ for each $x \in V$. We say that $scope(A) = V$ and $range(A) = \{d \mid (x = d) \in A\}$. Assignment A is *complete* if $scope(A) = X$. The projection of A over a set of variables W is defined as $proj(A, W) = \{(x = d) \in A \mid x \in W\}$. A constraint c is a set of assignments over the set of variables denoted by $vars(c)$. Assignment A *satisfies* c if $vars(c) \subseteq scope(A)$ and $proj(A, vars(c)) \in c$. A *solution* is a complete assignment that satisfies every constraint in C .

Any permutation f of $lit(P)$ induces a function σ_f over $\wp(lit(P))$, where \wp denotes the powerset. Thus σ_f is defined on solutions, complete assignments, partial assignments and all other sets of literals. A *solution symmetry* (or just *symmetry* for short) f of a CSP P is a permutation of $lit(P)$ whose induced σ_f preserves the set of solutions [Cohen *et al.*, 2005], i.e., a bijection from literals to literals whose induced σ_f maps solutions to solutions. Two important kinds of solution symmetries are induced by permuting either variables or values.

A variable permutation r of X induces a permutation f_r of $lit(P)$ defined as $f_r(x = d) = (r(x) = d)$. A *variable symmetry* is a variable permutation r whose induced permutation f_r is a solution symmetry [Puget, 2002]. We use $\langle x_1, \dots, x_n \rangle \leftrightarrow \langle x_{1'}, \dots, x_{n'} \rangle$, where $x_1, \dots, x_n, x_{1'}, \dots, x_{n'} \in X$ to denote the variable symmetry that maps each x_i to $x_{i'}$ leaving the remaining variables in X unchanged.

A value permutation l of D induces a permutation f_l of $lit(P)$ defined as $f_l(x = d) = (x = l(d))$. A *value symmetry* is a value permutation l whose induced permutation f_l is a solution symmetry. We write $\langle d_1, \dots, d_n \rangle \leftrightarrow \langle d_{1'}, \dots, d_{n'} \rangle$, where $d_1, \dots, d_n, d_{1'}, \dots, d_{n'} \in D$. A *variable-value symmetry* of P is any solution symmetry that is not a variable or a value symmetry of P . Note that it is not necessarily a composition of variable and value symmetries of P .

Example 2.1 The *Latin square problem* of size 3 involves a 3×3 square, where each of the 9 cells in the square takes a value from $[1..3]$, in such a way that each value occurs exactly once in each row and once in each column. The associated CSP can be defined as follows:

$$\begin{aligned} X &= \{x_{11}, x_{12}, x_{13}, x_{21}, x_{22}, x_{23}, x_{31}, x_{32}, x_{33}\} \\ D &= \{1, 2, 3\} \\ C &= \{x_{11} \neq x_{12}, x_{11} \neq x_{13}, x_{12} \neq x_{13}, x_{21} \neq x_{22}, x_{21} \neq x_{23}, \\ &\quad x_{22} \neq x_{23}, x_{31} \neq x_{32}, x_{31} \neq x_{33}, x_{32} \neq x_{33}, x_{11} \neq x_{21}, \\ &\quad x_{11} \neq x_{31}, x_{21} \neq x_{31}, x_{12} \neq x_{22}, x_{12} \neq x_{32}, x_{22} \neq x_{32}, \\ &\quad x_{13} \neq x_{23}, x_{13} \neq x_{33}, x_{23} \neq x_{33}\} \end{aligned}$$

where constraints in C are given in their usual intensional form (rather than as sets of assignments). This CSP has (a) variable symmetries that swap any columns: $\langle x_{11}, x_{21}, x_{31} \rangle \leftrightarrow \langle x_{12}, x_{22}, x_{32} \rangle$, $\langle x_{11}, x_{21}, x_{31} \rangle \leftrightarrow \langle x_{13}, x_{23}, x_{33} \rangle$, and $\langle x_{12}, x_{22}, x_{32} \rangle \leftrightarrow \langle x_{13}, x_{23}, x_{33} \rangle$, (b) similar variable symmetries that swap any rows, (c) a variable symmetry that transposes the rows and columns corresponding to flipping the 3×3 square using a diagonal, and (d) the variable-value symmetry f defined as $f(x_{ij} = k) = (x_{jk} = i)$. \square

2.2 SBDS

Since LDSB is related to (shortcut) SBDS, we give a quick overview of this method, which was introduced in [Backofen and Will, 1999] and formally defined in [Gent and Smith, 2000]. The method is based on a search tree explored depth-first, and whose nodes are *decision points*, i.e., each node in the tree has either zero or two descendants, the first (labelled $x = v$) binds a variable x to a value v , while the second (labelled $x \neq v$) eliminates v from the domain of x . This kind of search tree does not restrict the order in which variables or values are explored, nor requires all values of a variable to be tried consecutively.

Consider a CSP $P = (X, D, C)$ with set of symmetries Sym , and let α be the assignment at a node n , obtained by collecting all variable/value bindings from n to the root of the tree. Let $x = v$ and $x \neq v$ be the labels of its two children nodes. The aim of SBDS is to eliminate bindings for node $x \neq v$ that could lead to assignments symmetric to $\alpha \cup \{x = v\}$. To achieve this, when the SBDS execution arrives at node $x \neq v$, it adds the conditional constraint $\sigma_f(\alpha) \Rightarrow \neg f(x = v)$ for any $f \in Sym$, thus ensuring that binding $f(x = v)$ will not occur in any node whose assignment is consistent with assignment $\sigma_f(\alpha)$.

The shortcut method was introduced in [Gent and Smith, 2000] by Gent and Smith as a special purpose

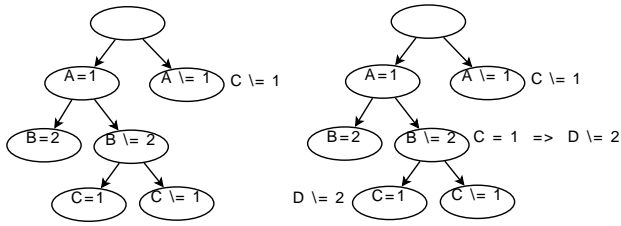


Figure 1: Shortcut SBDS (left) vs SBDS (right). At node $C=1$, the implication constraint posted at $B \neq 2$ becomes active and $D \neq 2$ is posted. Shortcut SBDS does not post such conditional constraints.

SBDS aimed at improving efficiency by only adding unconditional constraints. This is ensured to be correct for symmetries that leave the current assignment α unchanged, i.e., any $f \in \text{Sym}$ such that $\sigma_f(\alpha) = \alpha$ (see Figure 1). We will say that such symmetries are *active* for α . Gent and Smith restricted the symmetries handled by the shortcut method to those for which this condition is easy to check, although the actual choice of symmetries was left open, with two examples provided: a value symmetry for a general graph colouring, and the transposition of two nodes of a Ramsey graph colouring (complete graph with n nodes and c colours) whenever they are either uncoloured or have the same colour. Their results show considerable speedups when searching for all solutions when compared to a search with no symmetry breaking.

GAP-SBDS: In [Gent *et al.*, 2002] Gent *et al.* introduced an optimised implementation of SBDS that uses the computational group theory system GAP. GAP-SBDS allows the user to represent the symmetries by a (usually small) set of symmetry *group generators* (since the symmetries of any CSP form a group) that can be combined to describe all existing symmetries. This is a considerable improvement over previous systems which explicitly required coding of each symmetry.

There are four main sources of potential inefficiency when implementing SBDS. The first comes from *vacuous* posting, i.e., from posting a conditional constraint $\sigma_f(\alpha) \Rightarrow \neg f(x = v)$ such that while $\sigma_f(\alpha)$ is consistent with α , $f(x = v) = (x = v)$, since these constraints achieve no pruning.¹ The second source of potential inefficiency comes from *multiple posting*, i.e., from considering different symmetries which result in posting identical conditional constraints (see [McDonald, 2001] for an approach to eliminate multiple postings). The third source comes from not eliminating *broken* symmetries, i.e., those that result in constraints whose condition $\sigma_f(\alpha)$ will never be true, since it is inconsistent with α . The last source comes from incompleteness, i.e., from not posting a constraint that could have achieved pruning. GAP-SBDS avoids these issues by using GAP.

¹Since $\neg f(x = v)$ is equal to the already posted $\neg x = v$

3 Optimising for Common Symmetry Groups

As mentioned before, our aim is to develop a symmetry breaking method that can be used by any CSP with any search strategy while introducing as little overhead as possible. To achieve this we have targeted symmetry groups that are common in practice, can be represented compactly and manipulated efficiently. In particular, given $P = (X, D, C)$, the targeted symmetry groups are:

- **Interchangeable variables:** The set of variables $W \subseteq X$ are interchangeable for P if the extension f_r of every variable permutation r of W that leaves variables in $X \setminus W$ unchanged, is a variable symmetry for P . This can be used in a straightforward manner to define piecewise variable symmetries [Van Hentenryck *et al.*, 2003].
- **Interchangeable values:** The set of values $W \subseteq D$ are interchangeable for P if the extension f_l of every value permutation l of W that leaves values in $D \setminus W$ unchanged, is a value symmetry for P . Similarly, this can be used to define piecewise value symmetries [Van Hentenryck *et al.*, 2003].
- **Simultaneously interchangeable variables:** Two sequences of variables $\langle x_1, \dots, x_n \rangle$ and $\langle y_1, \dots, y_n \rangle$, such that $x_1, \dots, x_n, y_1, \dots, y_n \in X$ are simultaneously interchangeable for P if $\langle x_1, \dots, x_n \rangle \leftrightarrow \langle y_1, \dots, y_n \rangle$, i.e., if the bijection that maps each x_i to y_i and vice versa, leaving the remaining variables in X unchanged, is a variable symmetry for P . S is a set of simultaneously interchangeable variable sequences for P if $\forall s_1, s_2 \in S$, s_1 and s_2 are simultaneously interchangeable.
- **Simultaneously interchangeable values:** Two sequences of values $\langle x_1, \dots, x_n \rangle$ and $\langle y_1, \dots, y_n \rangle$, such that $x_1, \dots, x_n, y_1, \dots, y_n \in D$ are simultaneously interchangeable for P if $\langle x_1, \dots, x_n \rangle \leftrightarrow \langle y_1, \dots, y_n \rangle$. S is a set of simultaneously interchangeable value sequences for P if $\forall s_1, s_2 \in S$, s_1 and s_2 are simultaneously interchangeable.

Variable (and value) interchangeability is common in problems where the variables (or values) form the elements of a set (e.g., set of golfers that will play a tournament), and thus their order is irrelevant. Simultaneous variable and value interchangeability is common in problems that can be represented geometrically (e.g., swapping of any two rows in a matrix). Note that all these symmetries can be automatically detected by the method of [Mears *et al.*, 2008].

Example 3.1 Consider the Latin square CSP of Example 2.1. In this CSP, the set of values $\{1, 2, 3\}$ are interchangeable. Also, the variable sequences in the set $\{\langle x_{11}, x_{21}, x_{31} \rangle, \langle x_{12}, x_{22}, x_{32} \rangle, \langle x_{13}, x_{22}, x_{33} \rangle\}$ are simultaneously interchangeable. This corresponds to swapping the columns of the square. Simultaneous interchangeability of variables also exists for the rows. \square

Note that we are currently not targeting variable-value interchangeability. Our focus is on variable and on value

interchangeability since it often results in large symmetry groups that can be compactly represented. For example, a set of interchangeable variables W of size n represents $n!$ variable symmetries.

4 Representing and Using Symmetries

This section explains how symmetries are represented in LDSB. It also shows that the associated operations are correct and, for the variable and value interchangeability case, complete. Finally, it shows how LDSB avoids some of the inefficiencies described in Section 2.2. Our implementation integrates and extends some of the specialisations presented in [Focacci and Milano, 2001].

4.1 Interchangeable variables

Consider a CSP $P = (X, D, C)$, where $W \subseteq X$ is a set of interchangeable variables for P . Let f_r denote the induced variable symmetry of a variable permutation r of W . Let f_r also denote (by an abuse of notation) its associated solution preserving function σ_{f_r} . Consider the search tree up to node n at which the current assignment is α , and let $G(W, n)$ be the group of variable symmetries of W that are active at node n . Let us assume that none of the variables in W appears in α (i.e., $scope(\alpha) \cap W = \emptyset$). It is clear that $G(W, root) = G(W, n)$, i.e., all variable symmetries of W are active at any node from the root to n . Our implementation represents group $G(W, root)$ by the list of program variables **ListW** corresponding to those in W .

Suppose now that the first child of n is labelled $x = d$ with associated assignment $\beta = \alpha \cup \{x = d\}$ where $x \in W$, i.e., x is the first variable in W to occur in β . Clearly, $f_r(\beta) \neq \beta$ for every r such that $r(x) \neq x$ and, therefore, every such f_r is not active at node $x = d$. As a result, **ListW** does not need to represent these broken symmetries any longer. The removal simply consists in deleting from our initial **ListW** any variable that appears in the assignment α associated to n , i.e., $\mathbf{ListW} = W \setminus scope(\alpha)$. This can easily be done incrementally by, at each forward step in the search, eliminating the variable involved in the binding.

Note that f_r might become active again at some descendant n' of node $x = d$, with potential to prune the search tree again. This can occur if there is a set $V \subset W$, $x \in V$ with two or more variables and the assignment at n' contains $y = d : y \in V$. Eliminating f_r after node n is not a problem, since there is another variable symmetry $f_{r'}$ with $r'(y) = y : y \in V$ and $r'(z) = r(z) : z \notin V$, which has remained active throughout the path to n' , is represented by the remaining list $\mathbf{ListW} \setminus V$, and in the current path is broken by the same constraints as r .

Upon backtracking, the second child of node n will be labelled $x \neq d$. At this point, for any $f_r \in G(W, n)$ such that $f_r(\beta) \neq \beta$, we add $\neg r(x) = d$ and, thus, prune every part of the subtree symmetric to β under f_r . This is exactly the same as adding $Y \neq d$ for every Y in **ListW**. This is correct since, for every such Y , the variable symmetry $f_r(x = d) = (y = d)$ is represented by **ListW**.

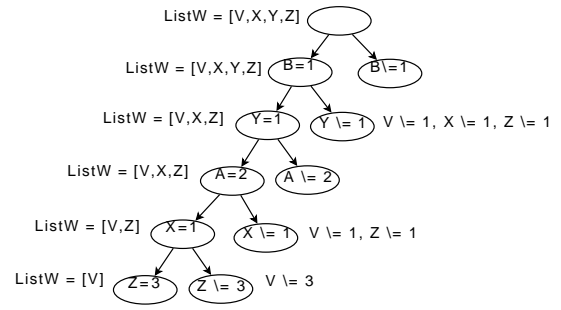


Figure 2: Evolution of **ListW** during search for interchangeable variables

Example 4.1 Consider the CSP $(\{x, y, z, v, a, b\}, \{1, 2, 3\}, C)$ where $W = \{x, y, z, v\}$ are interchangeable variables, and Figure 2 which shows the effect on **ListW** of each particular binding. Importantly, **ListW** only changes whenever the binding involves a variable in **ListW**. Also, note that at node $y = 1$, the variable symmetry $r \equiv \langle x, z \rangle \leftrightarrow \langle y, v \rangle$ is not active, since it maps x to y , and it is thus not represented by the **ListW** list of node $y = 1$. However, r becomes again active at node $x = 1$. The fact that r is not represented by the **ListW** list at node $x = 1$ is however not a problem, since it does represent the symmetry $\langle z \rangle \leftrightarrow \langle v \rangle$ which will achieve as much pruning as r once both x and y have the same value.

LDSB is correct for variable interchangeability. Furthermore, it is also optimally efficient, i.e., it does not post vacuous constraints, does not perform multiple postings, eliminates all broken symmetries, and is complete. It builds the same search tree as SBDS if a single variable interchangeable set is considered.

4.2 Interchangeable Values

Similarly to the case of variable interchangeability, our implementation represents the initial group of value symmetries for values in M by the list of values **ListM** in M . The invariant maintained during the execution is that **ListM** contains the values that are still involved in active symmetries. This means that whenever the node n is labeled $x = d$, we remove d from **ListM**. Upon backtracking, for the node labeled $x \neq d$, we add the symmetry-breaking constraint $X \neq m$ for every m in **ListM**. This leads to an algorithm similar to that given in [Van Hentenryck *et al.*, 2003].

Reasoning about the correctness, the efficiency properties and the completeness of dealing with interchangeable values, is similar to the variable case. It follows that for value interchangeability LDSB is also correct and optimally efficient. Thus, the LDSB and the SBDS methods build the same search tree if a single value interchangeability set is considered.

4.3 Simultaneous Interchangeable Variables

Consider a CSP $P = (X, D, C)$, where S is a set of simultaneously interchangeable sequences for P . Let G be

the group of variable symmetries of S , and for each two sequences $s_1, s_2 \in S$, let f_r denote their induced variable symmetry, and (by an abuse of notation) f_r also denote its associated solution preserving function, previously denoted by σ_{f_r} . Consider the search tree up to node n where, as before, α is the current assignment and $G(S, n)$ is the subgroup of G whose symmetries are active at node n . Let us assume that none of the variables in $\text{vars}(S)$ appears in α (i.e., $\text{scope}(\alpha) \cap \text{vars}(S) = \emptyset$). As before, $G(S, \text{root}) = G(S, n)$ and therefore all variable symmetries in G are active at any node from root to n . Our implementation represents $G(S, \text{root})$ by the list **Seqs** of lists of program variables, where each list $[X_1, \dots, X_n]$ corresponds to a sequence $\langle x_1, \dots, x_n \rangle \in S$.

Suppose now that the first child of n is labelled $x = d$ with associated assignment $\beta = \alpha \cup \{x = d\}$ where $x \in \text{vars}(S)$, i.e., x is the first variable in $\text{vars}(S)$ to occur in β . Clearly, $f_r(\beta) \neq \beta$ for every r such that $r(x) \neq x$ and, therefore, every such f_r is not active at node $x = d$. At this point, and analogously to what we did for broken variable symmetries, one could remove from **Seqs** every list that contains X . While doing this is correct, it would eliminate too many symmetries, since this time the remaining sequences might not represent symmetries $f_{r'}$ that achieve as much pruning as the reactivated f_r . Instead, we will simply keep all sequences, even if variables in them are ground. This allows our implementation to take advantage of reactivated symmetries.

Example 4.2 Consider a search where **Seqs** is initially $[[X, Y, Z], [A, B, C], [U, V, W]]$, representing (among others) symmetry $f_r = \langle x, y, z \rangle \leftrightarrow \langle a, b, c \rangle$. At node $X=1$ list **Seqs** becomes $[[1, Y, Z], [A, B, C], [U, V, W]]$, at which point f_r is broken. However, if the next node binds A to 1 , f_r is reactivated. As we will see later, this can be detected from the new instantiation of **Seqs** $= [1, Y, Z], [1, B, C], [U, V, W]$.

Upon backtracking, the second child of node n will be labelled $x \neq d$. At this point, for any $f_r \in G(S, n)$ such that $f_r(\beta) \neq \beta$, we add $\neg r(x) = d$ and, thus, prune every part of the subtree symmetric to β under f_r . This is achieved by finding every sequence $S \in \text{Seqs}$ that contains X , and comparing S to every other sequence $R \in \text{Seqs}$ to determine whether $S \leftrightarrow R$ is an active symmetry for α . This is true if every two corresponding elements in S and R have the same value, or are not fixed yet. For every R for which this is true, we add $Y \neq d$, where Y occurs in R in the same position as X in S .

LDSB is correct for simultaneous variable interchangeability. However, it might post vacuous constraints whenever the two sequences contain the same variable in the same position. As this is rare, avoiding this case will often result in loss of performance. It might also perform multiple postings, whenever there is more than one symmetry that maps variable X to Y . It does not eliminate broken symmetries from its representation, since it might need them to detect reactivated symmetries. And finally, as shown in Figure 3, it is incomplete due

to its adherence to the shortcut method, in contrast to the cases of variable and value interchangeability. LDSB and SBDS might not build the same search tree (depending on the choice of variable order followed by the search strategy) whenever simultaneous variable interchangeability is considered.

4.4 Simultaneous Interchangeable Values

Similarly to the case of simultaneous variable interchangeability, our implementation represents the initial group of simultaneous value symmetries of set of sequences S by the list **Seqs** of lists of values in S . The invariant maintained during the execution is that no list in **Seqs** contains a value in the range of the current assignment α . Thus, whenever the node n is labeled $x = d$, we remove any list containing d from **Seqs**. Upon backtracking, for the node labeled $x \neq d$, we find every sequence S in **Seqs** that contains d , and for every other sequence R in **Seqs**, add $X \neq m$, where m occurs in R in the same position as d in S .

LDSB is correct for simultaneous value interchangeability. However, it might post vacuous constraints whenever the two sequences contain the same value in the same position. Again, avoiding this will often result in loss of performance. It might also perform multiple postings, whenever there is more than one symmetry that maps value d to m . It does eliminate broken symmetries from its representation, but it might eliminate too many leading to further incompleteness. This can only happen if the same value appears at the same position in two sequences of **Seqs**. LDSB might thus build a larger tree than the SBDS method.

4.5 Combining Symmetries

Since the symmetries of a CSP form a group, any number of symmetries represented in LDSB can be composed to form a potentially new symmetry that might be used for pruning. The method described thus far does not compose symmetries, and thus misses pruning opportunities. We can easily adapt the implementation to compose symmetries by noticing that the active symmetries form a group themselves. Therefore, for active symmetries f and g , on asserting $x \neq a$ we not only assert $\neg f(x = a)$ but also $\neg g(f(x = a))$, and keep doing this in a breadth-first manner until no more new prunings are obtained.

As discussed in Section 6, our experiments indicate that, while this introduces a small overhead, it is sometimes crucial for the performance of LDSB. With the composition of symmetries, LDSB is also complete for problems with more than one set of interchangeable variables and/or interchangeable values.

4.6 Extending the Shortcut Method

Let n be the parent of the two nodes labeled $x = d$ and $x \neq d$, with associated assignment α . Let n' be a descendant of node $x \neq d$ (and thus of n), with associated assignment α' . Let f be an active symmetry at node n' . Then, it is correct to post $\neg f(x = d)$ at

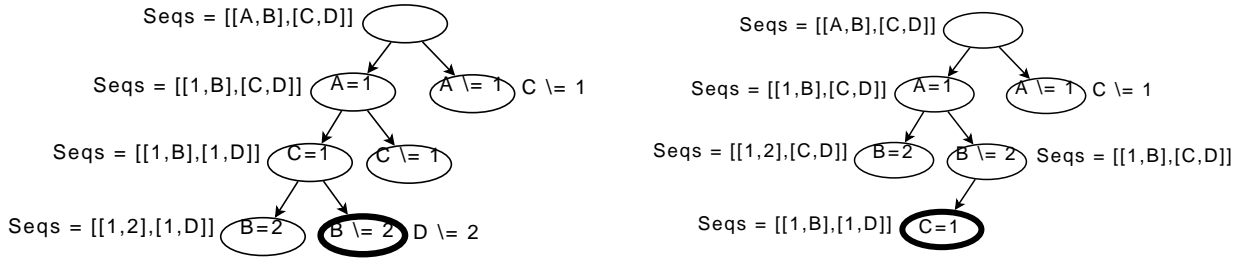


Figure 3: Impact of the choice of search heuristic on completeness. The highlighted nodes represent the same constraint store, but LDSB’s pruning depends on the search order.

n' . This is because, upon backtracking from node labeled $x = d$, the SBDS method posts the conditional constraint $f(\alpha) \Rightarrow \neg f(x = d)$. Since at node n' f is active, we know that $f(\alpha') = \alpha'$, which implies $f(\alpha)$ is now entailed and, thus, the condition in the SBDS constraint is now true.

As a result, for every active symmetry f at node n' , we can correctly post $\neg f(x = d)$ for every parent node of n' labeled $x \neq d$. Note that this is true regardless of whether n' is the left or right node of its parent. Furthermore, this also means that it is correct to compute the set of active symmetries before or after posting the unconditional constraints for the node. In fact, since the set of symmetries detected at each point might vary (due to propagation), one could iterate the process until no new symmetries are detected. We have tried computing the prunings after in addition to before posting $x \neq d$, but found that it did not improve efficiency.

5 Implementation

This section describes in more detail how we have implemented the LDSB method in ECLⁱPS^e. Consider a CSP (X, D, C) with (a) a set of interchangeable variables $W \subseteq X$, (b) a set of interchangeable values $M \subseteq D$, (c) two sequences of simultaneous interchangeable variables $\langle x_1, \dots, x_n \rangle, \langle y_1, \dots, y_n \rangle$ where $x_i, y_i \in X, 1 \leq i \leq n$, and (d) two sequences of simultaneously interchangeable values $\langle d_1, \dots, d_n \rangle, \langle l_1, \dots, l_n \rangle$ where $d_i, l_i \in D, 1 \leq i \leq n$.

Our LDSB implementation represents these symmetries as a Prolog list whose elements are (a) the Prolog term $\text{varI}(\text{ListW})$, where ListW is the list of program variables corresponding to W , (b) $\text{valI}(\text{ListM})$, where ListM is the list of values in M , (c) $\text{simVarI}([\text{X1}, \dots, \text{Xn}], [\text{Y1}, \dots, \text{Yn}])$ where Xi, Yi are the corresponding program variables, and (d) $\text{simValI}([\text{d1}, \dots, \text{dn}], [\text{l1}, \dots, \text{ln}])$ where di, li are the simultaneously interchangeable values.

Extending the example to have more than one set of interchangeable variables or values is straightforward; it is a matter of having more than one varI and valI terms. Extending the example to have more than two sequences of simultaneously interchangeable variables or values is also straightforward; it amounts to having more elements in the list argument of simVarI and simValI .

The kernel of the implementation is similar to that of GAP-SBDS, and currently consists of:

```

search(Vars,Symmetries) :-
  choose(Vars,Var,Val,RestVars),
  ( update_symmetries(Var, Val, Symmetries, NewSymmetries),
    Var = Val,
    search(RestVars,NewSymmetries)
  );
  compute_prunings(Var, Val, Symmetries, Prunings),
  Var \= Val,
  prune(Prunings),
  search(Vars,Symmetries)
).

```

where update_symmetries updates the arguments of varI and valI , by simply removing Var and Val from them, if present. Traversal of these lists is fast in practice. Note that symVarI is not modified by it, although the variables in symVarI (and varI) might become ground via propagation.

The predicate compute_prunings operates on the symmetries as they appeared before the decision point, as follows. It looks for the first $\text{varI}(W)$ where Var appears. If it finds it, it adds $Y \neq \text{Val}$ to Prunings for every other Y in W . It also looks for the first $\text{valI}(M)$ where Val appears. If it finds it, it adds $X \neq d$ for every other d in M . For simultaneous variable interchangeability, it accesses a variable attribute that indicates which symVarI s the Var is in and, for each such $\text{symVarI}(\text{Seqs})$, in which sequences S of Seqs the Var is in, and its position. It then compares each such S with every other sequence in Seqs . Having direct access from Var to the list of symmetries is crucial for performance. This is achieved by creating the attribute for each variable before the search begins.

For every new pruning added to Prunings , a recursive call to compute_prunings is performed to determine, as described in Section 4.5, possible new prunings achieved by composition, until no new prunings are obtained. Finally, predicate prune takes a list of elements of the form $Y \neq \text{Val}$ and posts them.

There are several promising variations to the current implementation. One is to represent simultaneous variable (or value) interchangeability pairwise – i.e., with each pair of sequences in Seqs represented separately. While this results in a quadratic representation, it allows permanently inactive pairs to be deleted from the list. Additionally, when two variables in the same posi-

tion of their sequence are bound to the same value, they can be removed from the sequences. This also eliminates the potential incompleteness of simultaneous value interchangeability.

Another variation is not to remove variables from the `varI` lists as they become ground. This would make updates faster but lookups slower (since the lists will be longer). Finally, one could represent each variable by an index rather than by the variable itself. This would allow us to preserve the attribute information to be preserved when variables are bound, at the cost of some indirection.

6 Experimental Results

This section presents the experimental results obtained on a set of benchmarks when searching without symmetry breaking, with LDSB, with GAP-SBDS, and with GAP-SBDD using two common search heuristics (input-order and first-fail). The experiments were conducted using the IC constraint solver of ECLⁱPS^e version 5.10.98. For the GAP methods, we used GAP version 4.4.10. The experiments were run on a machine with an Intel Core2 1.86GHz CPU with 1GB of memory, under Linux kernel version 2.6.22. Note that we do not compare with static symmetry breaking because we are focused on methods that do not interfere with the search heuristic.

6.1 Benchmarks

Our models of these problems are available at www.csse.monash.edu.au/~cmears/.

Balanced Incomplete Block Design: with parameters (v, b, k, r, λ) , where the task is to arrange v objects into b blocks such that each block has exactly k objects, each object is in exactly r blocks, and every pair of objects occurs together in λ blocks. The objects are interchangeable and the blocks are interchangeable, giving $v!b!$ symmetries, all represented in LDSB.

Social Golfers: with parameters (w, g, p) , where the task is to arrange gp golfers every week into g groups of p players over w weeks such that no two players are in the same group twice. The weeks are interchangeable, the players are interchangeable and within each week, the groups are interchangeable giving $w!(gp)!(g!)^w$ symmetries, all represented in LDSB.

Graceful Graph: with parameters (m, n) , where the edges (a, b) of the graph $K_m \times P_n$ are labeled by $|a - b|$, and no two edges have the same label. The corresponding vertices in each clique are simultaneously interchangeable, the order of the cliques is reversible, and the values are reversible, giving $4m!$ symmetries, all represented in LDSB.

Latin Square as introduced in Example 2.1, with $6(n!)^3$ symmetries, $2(n!)^3$ of which are represented in LDSB.

Magic Square: where an $N \times N$ square is filled with numbers 1 to N^2 such that all numbers are used and all rows, columns and two diagonals have the same sum. We use the symmetries of the square and the reversible

value symmetries, giving 16 symmetries, all represented in LDSB.

$N \times N$ Queens: where an $N \times N$ chessboard is coloured with N colours, so that a pair of queens in any two squares of the same colour do not attack each other. The symmetries are those of the chessboard, plus the colours are interchangeable, giving $8n!$ symmetries, all represented in LDSB.

N-Queens: where N queens are placed on a chessboard such that no two queens attack each other. This model has one variable per queen whose value is the queen's row. The symmetries are those of the chessboard, giving 8 symmetries, only 4 represented in LDSB.

Queens (bool): which uses a Boolean matrix model for the N-Queens problem described above, with 8 symmetries, all represented in LDSB.

Steiner Triples: where the task is to find $\frac{n(n-1)}{6}$ triples of distinct integers from 1 to n , such that any pair of triples has at most one element in common. The triples are interchangeable and the values are interchangeable, giving $n! \frac{(n(n-1))!}{6}$, all represented in LDSB.

6.2 Discussion

The experimental results for the four systems are shown in Table 1 (input-order search heuristic) and Table 2 (first-fail). Note that, for set variables, we replaced first-fail (since it is not supported by the set solver) by reversed input-order. For each benchmark and method the tables show the number of solutions found, the number of inner nodes from the decision tree explored, and the time taken in seconds. A “TO” entry indicates that the search timed out at the 300 second limit; “MO” indicates that memory was exhausted within the time limit. For each category, the best result is shown in bold. The last column in each table provides the percentage of the time spent by LDSB in executing the `update_symmetries` and `compute_prunings` predicates. A blank entry indicates that the LDSB time was too small to give a meaningful percentage. In each table, the upper half shows results to find the first solution and the lower half results for all solutions. Note that for N-queens there are two separate entries: “queens” uses all symmetries for SBDS/SBDD while “queens-s” uses only the 4 symmetries that LDSB can handle.

The results for first solution show that, while in some cases no symmetry breaking is the best method, LDSB is never more than 25% slower and in many cases is faster. In the most extreme case – where no solution exists – no symmetry breaking performs very badly. When compared to the other symmetry breaking methods, LDSB is uniformly faster (and often orders of magnitude faster) due to its lower overhead. LDSB also has a low memory overhead; when the symmetry group is extremely large, LDSB avoids the memory problems suffered by the GAP-based methods of having to deal with very large data structures.

For all solutions, LDSB is faster than the other three methods in all but three benchmark instances

Problem	Solutions				Branches				Time (s)				LDSB O'head
	None	SBDD	SBDS	LDSB	None	SBDD	SBDS	LDSB	None	SBDD	SBDS	LDSB	
bibd [7, 7, 3, 3, 1]	1	1	1	1	63	19	19	26	0.01	0.16	0.14	0.01	
bibd [12, 12, 6, 6, 4]	TO	0	TO	0	TO	254	TO	6635	TO	196.54	TO	5.52	6.4
bibd [13, 13, 6, 6, 4]	TO	TO	MO	0	TO	TO	MO	8659	TO	TO	MO	8.45	6.5
golf [3, 4, 3]	1	1	1	1	85	32	32	60	0.02	0.58	16.84	0.02	
golf [4, 4, 3]	1	1	1	1	2420	83	83	448	1.38	7.66	175.88	0.22	11.6
golf [4, 4, 4]	1	1	1	1	38	38	38	38	0.02	0.12	1.03	0.02	
graceful [3, 3]	1	1	1	1	452	199	199	199	0.52	0.61	0.55	0.4	1.7
graceful [4, 2]	1	1	1	1	805	343	343	343	1.23	1.06	1.02	0.71	1.2
latin [20]	1	1	MO	1	400	274	MO	274	0.96	3.44	MO	1.0	
latin [25]	1	TO	MO	1	635	TO	MO	463	2.75	TO	MO	2.89	0.1
latin [30]	1	1	TO	1	900	684	TO	684	4.59	21.78	TO	4.79	0.1
magicsquare [4]	1	1	1	1	32	20	20	20	0.0	0.02	0.02	0.0	
magicsquare [5]	1	1	1	1	6059	5497	5497	5497	0.49	8.94	5.18	0.61	7.0
nn_queens [7]	1	1	1	1	72	27	27	27	0.0	0.08	0.08	0.01	
nn_queens [8]	TO	0	0	0	TO	104013	104007	207054	TO	236.57	192.0	17.99	12.5
queens [20]	1	1	1	1	49746	37330	37330	37330	2.29	35.08	25.41	3.0	9.4
queens [22]	1	TO	1	1	356964	TO	269369	269369	18.93	TO	211.89	23.54	8.7
queens [24]	1	1	1	1	84875	63790	63790	63790	4.9	76.42	56.56	6.04	8.4
queens-s [20]	1	1	1	1	49746	37330	37330	37330	2.29	35.48	25.73	3.0	9.4
queens-s [22]	1	TO	1	1	356964	TO	269369	269369	18.93	TO	209.08	23.54	8.7
queens-s [24]	1	1	1	1	84875	63790	63790	63790	4.9	73.68	55.82	6.04	8.4
queens_bool [20]	1	1	1	1	221412	16316	16316	16316	2.62	127.06	53.2	2.65	8.4
queens_bool [22]	1	TO	TO	1	1847255	TO	TO	122683	21.34	TO	TO	20.95	8.6
queens_bool [24]	1	1	1	1	316472	18540	18540	18540	3.48	233.42	101.39	3.65	9.5
steiner [7]	1	1	1	1	20	14	14	16	0.0	0.06	0.05	0.01	
steiner [8]	TO	0	0	0	TO	66	60	1126	TO	0.93	1.41	0.38	13.7
steiner [9]	1	1	1	1	4545	66	65	523	1.54	4.01	7.34	0.23	11.1
					↑ First solution ↑ - ↓ All solutions ↓								
bibd [7, 7, 3, 3, 1]	151200	1	1	2	2747085	19	19	27	113.01	0.18	0.18	0.02	
bibd [12, 12, 6, 6, 4]	TO	0	TO	0	TO	254	TO	6635	TO	194.13	TO	5.54	6.7
bibd [13, 13, 6, 6, 4]	TO	TO	MO	0	TO	TO	MO	8659	TO	TO	MO	8.41	6.6
golf [3, 4, 3]	TO	4	4	336	TO	88	78	11050	TO	4.19	110.67	3.78	13.8
golf [4, 4, 3]	TO	3	TO	492	TO	154	TO	84467	TO	15.99	TO	41.62	11.2
golf [4, 4, 4]	TO	1	TO	12	TO	56	TO	2564	TO	50.85	TO	1.5	11.2
graceful [3, 3]	TO	284	284	380	TO	14818	14795	16704	TO	36.02	28.05	21.72	1.1
graceful [4, 2]	TO	15	15	30	TO	5230	5207	5923	TO	17.47	15.95	12.51	1.1
latin [4]	576	2	2	4	5152	11	9	10	0.15	0.05	0.08	0.01	
latin [5]	161280	2	2	56	1832945	21	18	117	67.07	0.41	1.27	0.07	
latin [6]	TO	12	12	9408	TO	129	122	17139	TO	17.43	136.44	10.1	5.7
magicsquare [3]	8	1	1	1	118	17	8	8	0.01	0.02	0.01	0.0	
magicsquare [4]	7040	880	880	880	490586	57411	57358	57358	31.31	58.65	31.18	5.97	9.3
nn_queens [7]	TO	1	1	4	TO	607	602	1079	TO	0.93	0.79	0.09	
nn_queens [8]	TO	0	0	0	TO	104013	104007	207054	TO	236.15	191.09	17.93	12.4
queens [12]	14200	1787	1787	3975	248010	31079	31070	55142	5.78	20.69	12.92	3.3	12.9
queens [13]	73712	9233	9233	20028	1303089	159151	159139	279679	31.56	129.78	72.34	16.82	11.4
queens [14]	365596	45752	TO	99883	7089362	875178	TO	1554840	180.05	228.58	TO	97.86	12.3
queens-s [12]	14200	3570	3570	3975	248010	54137	54129	55142	5.78	35.19	20.57	3.3	12.9
queens-s [13]	73712	18462	18462	20028	1303089	277770	277749	279679	31.56	215.87	115.71	16.82	11.4
queens-s [14]	365596	TO	TO	99883	7089362	TO	TO	1554840	180.05	TO	TO	97.86	12.3
queens_bool [12]	14200	1787	1787	12894	1861227	29419	29415	105370	14.14	76.07	33.41	11.49	9.3
queens_bool [13]	73712	TO	9233	66648	10518918	TO	145460	539244	75.6	TO	189.54	61.25	8.9
steiner [7]	TO	1	1	2	TO	30	26	88	TO	0.17	0.15	0.03	
steiner [8]	TO	0	0	0	TO	66	60	1126	TO	0.95	1.41	0.37	13.7
steiner [9]	TO	1	1	36	TO	314	290	43904	TO	13.57	26.34	19.98	12.8

Table 1: Experimental Results (input order). 300 second timeout.

(golf[4,4,3], steiner[9], and latin[6], the first two due to difficulties with set representation, while the last one is due to incompleteness), even though GAP-SBDS and GAP-SBDD often find fewer solutions and explore a smaller search tree than LDSB due to LDSB's incompleteness. Note that LDSB is considerably faster than the other methods for N-Queens, even though some of the symmetries are not represented in it. The no symmetry breaking method is often unable to find all solutions within the time limit.

7 Conclusion

Handling symmetries when solving constraint satisfaction problems can improve the efficiency of the search, and it is sometimes vital for practicality. We describe LDSB, a lightweight method for breaking symmetries during search that is able to represent many common

symmetries in a compact form that can be manipulated to detect and eliminate symmetric sections of the search tree. Also, the method respects the choices made by the search heuristic and does not require complex group theory computations to be effective. Our results show that its overhead is low enough to be used without fear of an explosion in memory use or running time.

Although we have done some optimisation of the representation and its implementation, there is still room for improvement. The method can be extended to handle variable-value symmetries by transforming them into variable symmetries. Given that good results have been obtained without doing complete pruning, further experiments may show that even better results may come by considering only a subset of a problem's symmetries.

Problem	Solutions				Branches				Time (s)				LDSB O'head
	None	SBDD	SBDS	LDSB	None	SBDD	SBDS	LDSB	None	SBDD	SBDS	LDSB	
bibd [7, 7, 3, 3, 1]	1	1	1	1	63	19	19	26	0.02	0.16	0.15	0.02	
bibd [12, 12, 6, 6, 4]	TO	0	TO	0	TO	254	TO	6635	TO	193.71	TO	6.24	6.1
bibd [13, 13, 6, 6, 4]	TO	TO	MO	0	TO	TO	MO	8659	TO	TO	MO	9.07	5.9
golf [3, 4, 3]	1	1	1	1	85	32	32	60	0.03	0.67	17.05	0.02	
golf [4, 4, 3]	1	1	1	1	2420	83	83	448	1.48	8.63	177.46	0.23	11.1
golf [4, 4, 4]	1	1	1	1	38	38	38	38	0.02	0.11	1.1	0.02	
graceful [3, 3]	1	1	1	1	1714	720	720	720	1.88	1.57	1.43	1.0	1.2
graceful [4, 2]	1	1	1	1	927	195	195	195	1.4	0.62	0.59	0.45	1.0
graceful [5, 2]	TO	TO	TO	1	TO	TO	TO	57334	TO	TO	TO	292.41	0.5
latin [20]	1	TO	MO	1	403	TO	MO	309	1.07	TO	MO	1.11	0.1
latin [25]	1	1	MO	1	625	491	MO	491	2.53	10.49	MO	2.62	0.0
latin [30]	1	TO	TO	1	908	TO	TO	742	5.21	TO	TO	5.36	0.1
magicsquare [4]	1	1	1	1	32	19	19	19	0.0	0.02	0.01	0.0	
magicsquare [5]	1	1	1	1	960	758	758	763	0.08	0.88	0.73	0.1	
magicsquare [6]	1	1	1	1	93429	72747	72747	71652	7.49	187.2	119.61	9.08	7.2
nn_queens [7]	1	1	1	1	53	16	16	16	0.0	0.05	0.07	0.0	
nn_queens [8]	TO	0	0	0	TO	76079	76073	149579	TO	162.72	127.17	19.09	8.4
queens [20]	1	1	1	1	71	43	43	43	0.01	0.06	0.05	0.0	
queens [22]	1	1	1	1	23	18	18	18	0.01	0.03	0.03	0.01	
queens [24]	1	1	1	1	28	23	23	23	0.0	0.03	0.03	0.0	
queens-s [20]	1	1	1	1	71	43	43	43	0.01	0.03	0.05	0.0	
queens-s [22]	1	1	1	1	23	18	18	18	0.01	0.02	0.03	0.01	
queens-s [24]	1	1	1	1	28	23	23	23	0.0	0.04	0.03	0.0	
queens.bool [20]	1	1	1	1	134098	16316	16316	16316	4.92	130.98	55.78	5.15	4.5
queens.bool [22]	1	TO	TO	1	1062027	TO	TO	122683	40.25	TO	TO	41.53	4.4
queens.bool [24]	1	1	1	1	168353	18540	18540	18540	6.9	236.89	105.88	7.22	4.8
steiner [7]	1	1	1	1	20	14	14	16	0.01	0.06	0.06	0.0	
steiner [8]	TO	0	0	0	TO	66	60	1126	TO	1.01	1.39	0.37	14.7
steiner [9]	1	1	1	1	4545	66	65	523	1.71	3.56	7.07	0.23	11.1
↑ First solution ↑ - ↓ All solutions ↓													
bibd [7, 7, 3, 3, 1]	151200	1	1	2	2157049	19	19	27	117.6	0.18	0.18	0.03	
bibd [12, 12, 6, 6, 4]	TO	0	TO	0	TO	254	TO	6635	TO	194.17	TO	5.87	6.0
bibd [13, 13, 6, 6, 4]	TO	TO	MO	0	TO	TO	MO	8659	TO	TO	MO	9.12	6.0
golf [3, 4, 3]	TO	4	4	336	TO	88	78	11050	TO	4.62	111.32	3.85	13.1
golf [4, 4, 3]	TO	3	TO	492	TO	154	TO	84467	TO	18.0	TO	42.35	10.7
golf [4, 4, 4]	TO	1	TO	12	TO	56	TO	2564	TO	58.9	TO	1.55	10.5
graceful [3, 3]	TO	284	284	380	TO	40143	48106	59732	TO	84.97	73.6	64.76	1.0
graceful [4, 2]	TO	15	15	30	TO	2278	2255	3093	TO	7.87	7.1	5.92	1.1
latin [4]	576	2	2	4	4744	9	7	10	0.16	0.06	0.06	0.01	
latin [5]	161280	2	2	56	1703565	20	17	117	70.34	0.22	0.61	0.07	
latin [6]	TO	12	12	9402	TO	117	110	17102	TO	8.2	118.26	10.19	5.6
magicsquare [3]	8	1	1	1	115	17	8	8	0.01	0.03	0.01	0.01	
magicsquare [4]	7040	880	880	880	417747	30961	27172	27235	23.5	30.14	14.81	2.85	9.4
nn_queens [7]	TO	1	1	4	TO	437	432	863	TO	0.66	0.58	0.11	11.1
nn_queens [8]	TO	0	0	0	TO	76079	76073	149579	TO	162.04	127.24	19.13	8.3
queens [12]	14200	1787	1787	3975	218841	20683	20673	37098	5.4	13.3	8.64	2.42	11.4
queens [13]	73712	9233	9233	20028	1133851	104308	104295	184052	28.52	81.28	46.97	12.19	10.9
queens [14]	365596	45752	45752	99883	6075251	552180	552180	992027	160.26	145.84	263.67	67.88	11.1
queens-s [12]	14200	3570	3570	3975	218841	36145	34445	37098	5.4	21.70	13.16	2.42	11.4
queens-s [13]	73712	18462	18462	20028	1133851	182571	173088	184052	28.52	134.68	71.35	12.19	10.9
queens-s [14]	365596	TO	TO	99883	6075251	TO	TO	992027	160.26	TO	TO	67.88	11.1
queens.bool [12]	14200	1787	1787	12894	1077180	29419	29415	105370	23.52	78.59	36.0	21.18	5.3
queens.bool [13]	73712	TO	9233	66648	5533223	TO	145460	539244	121.92	TO	202.64	106.79	5.1
steiner [7]	TO	1	1	2	TO	30	26	88	TO	0.16	0.15	0.02	
steiner [8]	TO	0	0	0	TO	66	60	1126	TO	1.02	1.38	0.36	14.4
steiner [9]	TO	1	1	36	TO	314	290	43904	TO	12.98	25.76	19.64	13.6

Table 2: Experimental Results (first fail). 300 second timeout.

References

- [Backofen and Will, 1999] Rolf Backofen and Sebastian Will. Excluding symmetries in constraint-based search. In *CP*, pages 73–87, 1999.
- [Cohen *et al.*, 2005] David Cohen, Peter Jeavons, Christopher Jefferson, Karen E. Petrie, and Barbara M. Smith. Symmetry definitions for constraint satisfaction problems. In Peter van Beek, editor, *LNCS*, volume 3709, pages 17–31, 2005.
- [Fahle *et al.*, 2001] Torsten Fahle, Stefan Schamberger, and Meinolf Sellmann. Symmetry breaking. In *CP*, pages 93–107, 2001.
- [Flener *et al.*, 2006] Pierre Flener, Justin Pearson, Meinolf Sellmann, and Pascal Van Hentenryck. Static and dynamic structural symmetry breaking. In *CP*, pages 695–699, 2006.
- [Focacci and Milano, 2001] Filippo Focacci and Michela Milano. Global cut framework for removing symmetries. In *CP*, pages 77–92, 2001.
- [Gent and Smith, 2000] I. P. Gent and B. M. Smith. Symmetry breaking in constraint programming. In *ECAI 2000 14th European Conference on Artificial Intelligence*, 2000.
- [Gent *et al.*, 2002] Ian P. Gent, Warwick Harvey, and Tom Kelsey. Groups and constraints: Symmetry breaking during search. In Pascal van Hentenryck, editor, *LNCS*, volume 2470, pages 415–430, 2002.
- [McDonald, 2001] Iain McDonald. Optimum symmetry breaking in CSPs using group theory. In *Proc. CP'01*,

page 771, 2001.

- [Mears *et al.*, 2008] Christopher Mears, Maria Garcia de la Banda, Mark Wallace, and Bart Demoen. A novel approach for detecting symmetries in CSP models. In *CPAIOR*, 2008.
- [Petrie and Smith, 2003] K. E. Petrie and B. M. Smith. Symmetry breaking in graceful graphs. In *Proc. CP'03*, pages 930–934, 2003.
- [Puget, 2002] Jean-Francois Puget. Symmetry breaking revisited. In Pascal van Hentenryck, editor, *LNCS*, volume 2470, pages 446–461, 2002.
- [Van Hentenryck *et al.*, 2003] P. Van Hentenryck, P. Flener, J. Pearson, and M. Agren. Tractable symmetry breaking for CSPs with interchangeable values. In *Proc. IJCAI'03*, 2003.