Noname manuscript No.
(will be inserted by the editor)

Lightweight Dynamic Symmetry Breaking

**Christopher Mears · Maria Garcia de la Banda ·
Bart Demoen · Mark Wallace**

the date of receipt and acceptance should be inserted later

Abstract Symmetries in constraint problems present an opportunity for reducing search. This paper presents *Lightweight Dynamic Symmetry Breaking*, an automatic symmetry breaking method that is efficient enough to be used as a default, since it never yields a major slowdown while often giving major performance improvements. This is achieved by automatically exploiting certain kinds of symmetry that are common, can be compactly represented, easily and efficiently processed, automatically detected, and lead to large reductions in search. Moreover, the method is easy to implement and integrate in any constraint system. Experimental results show the method is competitive with the best symmetry breaking methods without risking poor performance.

1 Introduction

Many constraint problems have *symmetries* — that is, permutations of their variable/value pairs that map solutions to solutions and non-solutions to non-solutions [2]. Constraint problems that contain symmetries can be solved faster by using symmetry breaking techniques which avoid the exploration of symmetric parts of the search space when solving the problems. This is correct because if the explored parts led to a solution, the symmetric search space is guaranteed to contain only symmetric solutions, which can be automatically generated without search. If the explored parts instead led to failure, the symmetric search space is guaranteed to also lead to failure.

There are two main approaches to symmetric breaking: *static* and *dynamic*. Static symmetry breaking alters the original problem by adding new constraints that ensure the search will find only a single representative of each group of symmetric solutions. In contrast, dynamic symmetry breaking leaves the original problem unaltered and, instead, alters the search procedure itself to exclude symmetric regions.

Christopher Mears · Maria Garcia de la Banda · Mark Wallace
E-mail: {chris.mears / maria.garciadelabanda / mark.wallace}@monash.edu
Monash University, Australia

Bart Demoen
E-mail: bart.demoen@cs.kuleuven.be
KU Leuven, Belgium

Given the advantages of symmetry breaking and the significant amount of research in this area, one would expect most current constraint solving systems to embed symmetry breaking methods. This is however not the case, perhaps because both static and dynamic approaches can result in considerable slowdowns, particularly when only looking for the first solution. It is therefore not clear to the uninitiated (and sometimes even to the expert) constraint programmer exactly how to make use of symmetry breaking for their program. The wrong choice of method may give even worse performance than a naive search; conversely, to avoid symmetry breaking altogether may be to miss an opportunity to drastically improve search time. Of course, one might try several symmetry breaking methods and compare their performance, but this can require significant effort — especially if the methods require the development of problem-specific constraints or checks.

To fill this gap we have designed *Lightweight Dynamic Symmetry Breaking*[1] (*LDSB*), an *automatic* symmetry breaking system that has *reliable performance*. In other words, it is a system that automatically uses the symmetries given by the user to solve a problem in such a way that it often achieves good speedups and never yields a major slowdown (relative to solving the problem without symmetry breaking). We do this by targeting symmetries that are *common* in constraint problems, can be *compactly* represented, and can be *easily* and *efficiently* processed (as breaking them does not require complex computational group theory computations). Note that this means we are willing to give up completeness (i.e., breaking all symmetries), if that yields better performance. However, the method is complete for 3 out of the 4 kinds of symmetries that it targets and, importantly, all four kinds of symmetries can be automatically detected [19]. Moreover, the approach is easy to implement and to integrate into any constraint system. In particular, we report on the results of two implementations of LDSB — one each for the ECL^{i}PS^e [32] and Gecode [9] constraint programming platforms (our LDSB implementation is already part of the ECL^{i}PS^e distribution and will be part of the next Gecode release).

Our empirical results show that LDSB has little overhead for problems where symmetries do not significantly reduce search, and performs competitively with other symmetry breaking methods when they do. In particular, LDSB is almost always faster than the dynamic methods we compared with, and is competitive even with static methods that are generated by hand for specific problems. 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.

The outline of the paper is as follows. Section 2 provides the necessary background definitions. Section 3 discusses related work. Section 4 describes in detail the internal syntax used by LDSB to represent symmetries. Section 5 describes the symmetry breaking algorithm of LDSB, together with proofs of its correctness and (some) completeness. Section 6 discusses the implementation of LDSB. Section 7 gives an experimental evaluation of the method. Finally, Section 8 presents our conclusions.

2 Background

Constraint Satisfaction Problems (CSPs)

Let $vars(O)$ and $vals(O)$ denote the set of variables and values of any syntactical object O, respectively, and $|S|$ the cardinality of set S. A constraint satisfaction problem (CSP) is a tuple (X,D,C) where X is a set of variables, D a function that maps each variable in X to

[1] This paper is based on work that appeared in [20].

its domain (a finite set of values), and C a set of constraints s.t. $vars(C) \subseteq X$. For brevity, we say $D = \{a_1, \cdots, a_n\}$ when all variables in X have the same domain $\{a_1, \cdots, a_n\}$, and denote a set of consecutive integers $\{i, i+1, \cdots, j\}$ by $i..j$. A *literal* of $P = (X, D, C)$ is of the form $x = d$ where $x \in X$ and $d \in D(x)$. We denote the set of all literals of P by $lit(P)$. An *assignment of* P is a subset of $lit(P)$. Where the identity of P is clear, we omit the "of P" part. An *assignment over* set of variables V is one that contains one literal per variable in V. A constraint $c \in C$ is a set of assignments over $vars(c) \subseteq X$. Assignment A over $vars(c)$ is *allowed* by c if $A \in c$. Assignment A over $V \subseteq X$ *satisfies* constraint c if $vars(c) \subseteq V$ and the projection of assignment A over $vars(c)$ — computed as $\{x = d | (x = d) \in A, x \in vars(c)\}$ — is allowed by c. A *solution* of P is an assignment over X that satisfies every $c \in C$.

Example 1 The Latin Square problem requires one to find an $N \times N$ matrix of $1..N$ values, each occurring once in each row and column. A CSP for $N = 3$ has 9 integer variables $\{x_{ij} | i, j \in 1..3\}$, one per cell in row i and column j (see Figure 1). Each variable has domain $1..3$ and C has 18 disequalities ensuring values occur exactly once in each row and exactly once in each column. Formally, the CSP can be defined as $P = (X, D, C)$ where:

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

Assignment $\{x_{11} = 1,\ x_{21} = 3\}$ is allowed by constraint $x_{11} \neq x_{21}$, while assignment $\{x_{11} = 1,\ x_{21} = 1\}$ is disallowed by it. Assignment $\{x_{11} = 1,\ x_{21} = 3,\ x_{31} = 1\}$ satisfies $x_{11} \neq x_{21}$ but does not satisfy $x_{11} \neq x_{31}$. The right-hand side of Figure 1 shows two possible solutions of P. □

Constraint satisfaction problems can be transformed into optimisation problems by adding an optimisation function to maximise or minimise certain criteria. While for simplicity we focus on satisfaction problems, the results could easily be extended to optimisation problems simply by, for example, representing the optimisation function as an additional constraint.

Symmetries in CSPs

A *solution symmetry* σ of $P = (X, D, C)$ is a permutation on $lit(P)$ that induces a permutation ρ_σ on assignments that maps solutions to solutions [2]. Two important kinds of solution symmetries are induced by permuting either the variables in X or the values in each $D(x)$. A permutation s on the variables in X induces a permutation σ_s on $lit(P)$ by defining $\sigma_s(x = d) = (s(x) = d)$. A *variable symmetry* is a permutation of variables whose induced literal permutation is a solution symmetry [24]. A set S of value permutations s_x, one for the values in each $D(x), x \in X$, induces a permutation σ_S of literals by defining $\sigma_S(x = d) = (x = s_x(d))$. A *value symmetry* is a set of value permutations whose induced literal permutation is a solution symmetry [24]. Like most other papers in the area, this paper

x_{11}	x_{12}	x_{13}
x_{21}	x_{22}	x_{23}
x_{31}	x_{32}	x_{33}

1	3	2
2	1	3
3	2	1

3	1	2
1	2	3
2	3	1

Fig. 1: Variables in the Latin Square of size 3 and two solutions.

only discusses value symmetries where the values are treated equivalently across variables, i.e., where $s_i = s_j$ for all $s_i, s_j \in S$ and therefore a single s_i serves as a representative of S. We will use $\langle xd_1,\dots,xd_n\rangle \rightarrow \langle xd'_1,\dots,xd'_n\rangle$, where $\{xd_1,\dots,xd_n\} = \{xd'_1,\dots,xd'_n\}$ are sets of either variables in X or values in D, to denote the permutation that maps each xd_i to xd'_i. A *variable-value* symmetry is any solution symmetry that is not a variable or a value symmetry.

In an abuse of notation for the sake of brevity, we will sometimes use a permutation on variables (or values) to mean its induced permutation on literals. Likewise, we may treat a set of permutations on variables (or values) as the set of induced permutations on literals.

Variable symmetries in CSPs often appear in the form of *interchangeable variables* [17], i.e., a set of variables W s.t. any permutation of W is a variable symmetry of the CSP. Similarly, value symmetries often appear in the form of *interchangeable values* [17], i.e., a set of values W, s.t. any permutation of W is a value symmetry of the CSP. If the CSP contains more than one set of interchangeable variables or values, then it is said to have *piecewise* interchangeable variables or values [17]. Note that these sets must be, by definition of interchangeability, disjoint.

Example 2 The Latin square CSP of Example 1 has, among others, value symmetries that swap any two values (e.g., $\langle 1,2,3\rangle \rightarrow \langle 2,1,3\rangle$) and, therefore, the values 1..3 are interchangeable. It also has variable symmetries for any two rows and any two columns (e.g., $\langle x_{11},x_{21},x_{31},x_{12},x_{22},x_{32},x_{13},x_{23},x_{33}\rangle \rightarrow \langle x_{12},x_{22},x_{32},x_{11},x_{21},x_{31},x_{13},x_{23},x_{33}\rangle$ which gives the symmetric solutions in Figure 1), and variable-value symmetries swapping rows — or columns — with values (e.g. $\sigma(x_{ij} = k) = (x_{ik} = j), \forall i,j,k \in 1..3$). Note that this variable-value symmetry cannot be obtained by composing any of the variable and value symmetries in the CSP. □

Permutation Groups

A permutation group is a set of permutations that is closed under composition and inverses. The solution symmetries of a CSP P form a permutation group, where each element is a permutation on the set of literals $lit(P)$. Given permutations $\{f_1,f_2,\dots,f_n\}$ on $lit(P)$, we denote by $[f_1,f_2,\dots,f_n]$ the closure under composition and inverses of $\{f_1,f_2,\dots,f_n\}$. For a permutation group G, if $[f_1,f_2,\dots,f_n] = G$ then we say that $\{f_1,f_2,\dots,f_n\}$ *generates* G, and we call $\{f_1,f_2,\dots,f_n\}$ a *generating set* of G. By extension, given $\{S_1,S_2,\dots,S_n\}$ where each S_i is a set of permutations on $lit(P)$, we denote by $[S_1,S_2,\dots,S_n]$ the closure under composition and inverses of $\bigcup_{i=1}^{n} S_i$.

For the following, let P be a CSP and let G be a permutation group on $lit(P)$. The *orbit* $Orbit(\ell,G)$ of literal $\ell \in lit(P)$ under group G is defined as $\{f(\ell) \mid f \in G\}$. The *stabiliser* $Stab(\ell,G)$ of literal $\ell \in lit(P)$ under permutation group G is defined as $\{f \in G \mid f(\ell) = \ell\}$. Note that $Stab(\ell,G)$ forms a permutation group on $lit(P)$. The *pointwise* stabiliser $PointStab(L,G)$ under G of a set $L \subseteq lit(P)$ is defined as $\{f \in G \mid \forall \ell \in L.f(\ell) = \ell\}$. Note that $PointStab(L,G)$ also forms a permutation group on $lit(P)$ and that if $L = \{\ell_1,\ell_2,\dots,\ell_n\}$, then $PointStab(L,G)$ can also be computed as $Stab(\ell_1,Stab(\ell_2,\dots,Stab(\ell_n,G)))$. This is important for LDSB because it means that pointwise stabilisers can be computed incrementally. The *setwise* stabiliser $SetStab(L,G)$ under G of a set L of literals is defined as $\{f \in G \mid \forall \ell \in L.f(\ell) \in L\}$. As discussed later, setwise stabilisers identify *active* symmetries, i.e., symmetries that leave an assignment unchanged. Note that the pointwise stabiliser of set L is a subset of the setwise stabiliser of L. Importantly, while LDSB computes pointwise stabilisers rather than setwise stabilisers, there are many situations in which the two are identical.

Given a CSP P and a finite set W of variables, we use $\mathscr{S}_{W,P}$ to denote the set containing every permutation σ_s on $lit(P)$ induced by a permutation s on W and defined as follows:

$$\sigma_s = \begin{cases} \sigma_s(x = d) = (s(x) = d) & \text{if } x \in W \\ \sigma_s(\ell) = \ell & \text{otherwise} \end{cases}$$

Similarly, $\mathscr{S}_{W,P}$ can be defined for a set W of values with

$$\sigma_s = \begin{cases} \sigma_s(x = d) = (x = s(d)) & \text{if } d \in W \\ \sigma_s(\ell) = \ell & \text{otherwise} \end{cases}$$

Example 3 The complete set Σ of solution symmetries for a Latin square CSP P of size N (see Example 1) forms a group of cardinality $6(N!)^3$ since there are $N!$ permutations on the rows, $N!$ permutations on the columns, $N!$ permutations on the value, and 6 permutations among the row, column and value dimensions. For the case in which $N = 3$ the cardinality of the group is $6(3!)^3 = 1296$.

The set $\mathscr{S}_{\{1,2,3\},P}$ contains all permutations on $lit(P)$ induced by permutations of values 1, 2 and 3, and is a subset of Σ (assuming $N \geq 3$).

The set $G = Stab(x_{11} = 1, \Sigma)$ forms a group and contains all permutations that do not move the literal $x_{11} = 1$; namely, permutations among rows 2 and 3, columns 2 and 3, values 2 and 3, interchanges of variable and value dimensions, and their compositions. Thus, the cardinality of G is 48. The orbit of $x_{12} = 1$ by G is the set $Orbit(x_{12} = 1, G) = \{x_{11} = 2, x_{11} = 3, x_{12} = 1, x_{13} = 1, x_{21} = 1, x_{31} = 1\}$.

The pointwise stabiliser of $\{x_{11} = 1, x_{11} = 2\}$ in Σ contains the permutations among rows 2 and 3, columns 2 and 3, the two variable dimensions and their compositions. Thus, it has cardinality $2 \times 2 \times 2 = 8$. The setwise stabiliser of $\{x_{11} = 1, x_{11} = 2\}$ in Σ contains the same permutations as the pointwise stabiliser plus the permutation that swaps values 1 and 2 (and compositions), and has cardinality $8 \times 2 = 16$. □

3 Related Work

Static Symmetry Breaking

Both static and dynamic symmetry breaking methods select a single representative of the search space and ensure its symmetric images are not explored. Static methods achieve this by adding to the problem constraints that select a representative solution before the search starts.

While different static methods [6,23,17] have different properties, the more efficient ones tend to handle only restricted kinds of symmetries: for example, the method of Law et al. [17] can only handle the value symmetries in the Latin square problem. Further, the breaking of all symmetries together might not be correct, i.e., one cannot just specify different symmetries separately and expect the conjunction of their generated constraints to still have representative solutions. For example, column symmetry in Latin Squares can be broken by a lexicographic ordering of the columns, while row symmetry can be broken by an anti-lexicographic ordering of the rows. However, the Latin Square problem has no solution in which columns are ordered lexicographically and rows anti-lexicographically [5]: these constraints rule out all feasible solutions to the problem. While some methods are correct for particular kinds of symmetries, e.g., the method of Law et al. [17] is correct for interchangeable variables and values, Crawford et al. [3] have shown that, in general, correct static symmetry-breaking constraints that can be automatically generated lead to an exponential number of constraints.

Finally, static approaches can incur significant slowdowns [23] whenever there is a *disagreement* between the static constraints added and the search strategy, i.e., when the representative is not among the solutions that the search would have found first, or when the search strategy yields poor propagation for the static constraints. It is possible to add static constraints that do not disagree with the search if a static search strategy (i.e., a static variable and value ordering) can be chosen in advance. Unfortunately, the best search strategy for a given problem is not always static and, even if it were and the constraint programmer was able to detect this, the task of defining static symmetry breaking constraints that do not disagree with the search strategy can be quite difficult requiring the programmer to have specialised symmetry knowledge. In particular, we are not aware of any *general* automatic method to generate static symmetry constraints that is reliably faster than dynamic symmetry methods. While this is possible for piecewise interchangeable variables and values [6], currently it is not possible in general.

Dynamic Symmetry Breaking

Dynamic approaches, such as SBDD [8,4] and SBDS [1,12], alter the search to exclude regions symmetric to those previously explored. SBDS achieves this by adding constraints at each search node upon backtracking to eliminate symmetric variable assignments. It can be seen as an instance of the *$\mathscr{S}$-excluding* search tree method of Backofen and Will [1], which was independently developed by Gent and Smith [12]. SBDS solves $P = (X,D,C)$ with set of solution symmetries Σ by performing a depth-first search of a tree (see Figure 2) whose nodes have either zero or two children, with the left and right children (if any) labelled by $x = d$ and $x \neq d$, respectively, for some $x \in X, d \in D(x)$. The search proceeds as usual until it backtracks to the right child of a node n. Let A be the assignment *labeling* node n (i.e., the set of literals labeling the left branches of the path that goes from the root to n). Once the region under the left child has been explored, we know that every possible assignment that includes $A \cup \{x = d\}$ has been examined. Therefore, we can exclude any assignment that is symmetric to it. SBDS achieves this by posting on the right child of n, for each solution symmetry $\sigma \in \Sigma$, the conditional constraint

$$A \wedge x \neq d \wedge \sigma(A) \Rightarrow \neg\sigma(x = d)$$

where the left hand side of the $\Rightarrow$ indicates the condition that must hold for the right hand side to be posted. In an abuse of notation, we interpret A as the conjunction of its literals and denote by σ both the solution symmetry and its induced permutation ρ_σ on assignments. If constraints can be posted locally to a subtree, SBDS simply posts $\sigma(A) \Rightarrow \neg\sigma(x = d)$ since both A and $x \neq d$ hold in that subtree.

While SBDS finds representatives of every solution and constructs minimal search trees with respect to symmetry [26] (that is, it never reaches a node that is symmetrical to a previously explored node), it has four sources of inefficiency. The first is the posting of *vacuous* constraints (that is, constraints that have already been posted) due to symmetries for which $\sigma(x = d) = (x = d)$, since their conditional constraint will again add $x \neq d$. The second is the posting of *multiple* constraints due to distinct symmetries that yield identical

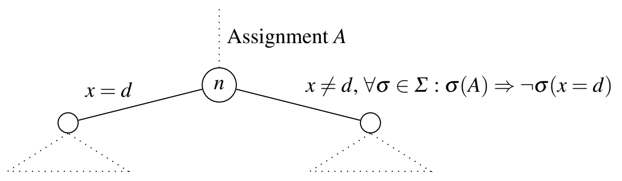

Assignment A
n
$x = d$
$x \neq d, \forall\sigma \in \Sigma : \sigma(A) \Rightarrow \neg\sigma(x = d)$

Fig. 2: Overview of SBDS decision point.

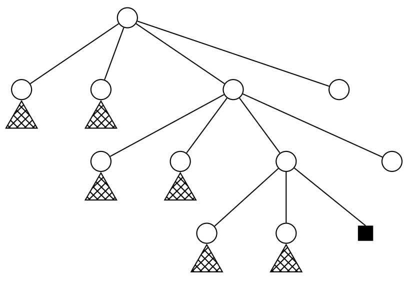


Fig. 3: Overview of SBDD search tree.

conditional constraints. The third is the processing of *broken* symmetries, i.e., those for which we already know that $\sigma(A)$ can never hold because it is inconsistent with $A \wedge x \neq d$. The fourth and last source of inefficiency is the need for the implementation to process every symmetry. This is a problem for large sets and it also imposes a burden on the user, who has to provide a function describing the effect of each symmetry.

In contrast to SBDS, SBDD exploits symmetries by detecting whether the current search node is symmetric to (or *dominated* by) a node already explored and, if so, not exploring it. To perform dominance checks, SBDD maintains a record of the search nodes that have been fully explored. If the search uses restarts [14], then this record may include search nodes from a previous run. For a traditional depth-first search without restarts, the fully explored nodes are those explored before the current node. Note that not every search node needs to be stored since, once an entire sub-tree has been explored, its nodes can be fully represented by the root of the sub-tree. Therefore, the number of search nodes that need to be stored is only polynomial in the number of variables and values of the problem. Figure 3 illustrates this point by displaying an SBDD search tree, where the square indicates the current node while the triangles show the fully explored nodes that might dominate the current node. Poor performance in SBDD occurs when the overhead introduced by the dominance checks (which is typically quite significant) is not compensated by the savings obtained by the associated prunings.

Both SBDS and SBDD can correctly and completely break all symmetries in a problem. However, they also require all these symmetries, or a symmetry dominance function, to be specified by the user. Since, in general, this is impractical, GAP-SBDS [10] and GAP-SBDD [11] use the computational group theory system GAP [28] to automatically compose and process the symmetries. As a result, users only need to provide a set of generators for the symmetry group, rather than all the symmetries. Further, GAP avoids the posting of vacuous and multiple constraints, and helps with the processing of broken symmetries. However, breaking every single symmetry can introduce very significant overheads at each search node. If these overheads are not compensated by significant savings, large slowdowns can occur.

Partial Symmetry Breaking

A *shortcut* form of SBDS is presented by Gent and Smith [12] by means of a graph colouring problem. The idea is to reduce overhead by only using active symmetries (i.e., those for which $\sigma(A) = A$), since then the constraints can be posted unconditionally (i.e., we can post $\neg\sigma(x = d)$ rather than $\sigma(A) \Rightarrow \neg\sigma(x = d)$). In particular, consider the graph colouring problem used by Gent and Smith [12], which is modelled as CSP $(X, 1..m, C)$, where X has one integer variable for each vertex, m is the maximum permitted number of colours, and the constraints in C ensure that adjacent vertices (i.e., variables) have different colours (i.e., values). Apart from symmetries that may occur in the input graph itself, the above CSP only has interchangeable value symmetries (for the set of colours). Assume the search has reached node n with labeling assignment A, and it is about to backtrack from branch $x_i = d$ to $x_i \neq d$. The search now needs to determine which symmetries satisfy $\sigma(A) = A$ and, to avoid posting vacuous constraints, $\sigma(x = d) \neq (x = d)$. Such σ exists only if d is an unused colour and $\sigma(x = d) = (x = e)$, where e is another unused colour. In addition, there must be a σ for each unused colour that moves d to that colour, because the problem contains all symmetries among the values. The effect is that when an unused colour leads to failure, no other unused colour should be tried in its place. The above chain of reasoning led Gent and Smith [12] to a simple rule for the graph colouring that is easy to implement and, in this case, completely breaks all value symmetries. Note that the method is as correct as SBDS and SBDD but, in general, it sacrifices completeness to increase efficiency. Unfortunately, it requires effort on behalf of the programmer to derive the reduced shortcut rule. Further, the reasoning that leads to a simple rule for this problem does not transfer to all problems.

The shortcut method is an example of *partial symmetry breaking* [18], which uses the observation that symmetry breaking methods such as SBDS can be sometimes made more efficient by considering only a subset of a problem's symmetries. This will be the case whenever the overhead introduced by processing the symmetries and posting the constraints is not compensated by the associated reductions in search space. McDonald and Smith [18] experimentally showed on the alien tiles and social golfers benchmarks that using a subset of the problems' symmetries with SBDS is beneficial. They also showed that the size of the chosen subset is not the only important factor, since the choice of which symmetries are included also affects performance, with symmetries that prune search near the root being the most effective. While the authors give an algorithm to choose the best subset for a given problem, it has two drawbacks: it can take a long time to run for only a small benefit and, like static symmetry breaking methods, it requires a fixed search strategy.

Another partial symmetry breaking method is STAB [25], a dynamic symmetry breaking method that, as the shortcut method, only breaks active symmetries (that is, set stabilisers). The main difference is in the kind of symmetry breaking constraints added: STAB adds lexicographical constraints, rather than the conditional constraints added by SBDS methods. To do so, STAB needs to compute a graph isomorphism at each search node. Specialised versions for 2D matrix problems (where rows and columns can be permuted) are presented in [25] and evaluated for several instances of the BIBD benchmark.

The Symmetry Breaking by Nonstationary Optimisation method [22] operates by searching for violated lexicographical constraints rather than posting them. At each search node, it does a local search for a symmetry whose corresponding lexicographical constraint would be violated had it been posted. The method is partial (because the time spent searching at each node is limited) and it can be combined with static lexicographical constraints.

Other methods can be seen as partial symmetry breaking methods because they can only break specific kinds of symmetry — for example, Van Hentenryck et al. [30] give a tractable and complete method for breaking interchangeable value symmetries, and Roney-Dougal et al. [26] describe a complete and efficient method to break arbitrary value symmetries. One of the most popular of these methods is *structural symmetry breaking (SSB)*, which can be seen a partial symmetry breaking form of SBDD, since it can only be used to break sets of piecewise interchangeable variables and piecewise interchangeable values [6]. This restriction allows SSB to define an algorithm for dominance checking that is complete for the kinds of symmetries handled, i.e., it breaks all symmetries in permutation group $[S_{X_1,P},\ldots,S_{X_n,P},S_{D_1,P},\ldots,S_{D_m,P}]$, where $\{X_1,\ldots,X_n\}$ are disjoint sets of variables, and $\{D_1,\ldots,D_m\}$ are disjoint sets of values. To achieve this SSB defines the concept of value signature, i.e., a tuple that counts how many variables from each set of interchangeable variables take that value. Then, assignment A dominates assignment B if the signature under A of every value a can be matched to the signature under B of a symmetrically-equivalent value b, such that the signature of each a is component-wise less than or equal to the signature of its matching b. In order to check for dominance, the method constructs a bipartite graph with two vertices (one on the left, and another on the right) for each value, and an edge between value a on the left and value b on the right if a and b are in the same interchangeable set of values and the signature under A of a is component-wise less than the signature under B of b. Assignment A dominates assignment B if this graph contains a perfect matching.

Interestingly, this result is used to *filter* nodes, i.e., to reduce the variables' domains such that the dominated nodes are not created, rather than to detect that they are dominated after the fact. If the filtering step is performed at every node, then SSB is complete w.r.t. piecewise variable and variable interchangeability, that is, it does not construct dominated nodes. However, the filtering step is expensive — $O(nm^{3.5}+n^2m^2)$, for m values and n variables — and the method works only for these two kinds of symmetries. There is also a static version of SSB [17], where a polynomial number of constraints are able to break all of the piecewise interchangeable variable and value symmetries. Structural symmetry breaking has also been extended to break wreath symmetries on values [7].

Our approach, LDSB, is a partial dynamic symmetry breaking method that can be seen as either an extension of the shortcut method but which relieves the programmer of any burden by automating the task of processing and composing symmetries, or as a restriction of the STAB method that does not require the computation of a graph isomorphism at each node. This is achieved by focusing on four common kinds of symmetries that can be broken using very efficient algorithms. In particular, we provide two very simple and efficient implementations of these algorithms (one for ECL^{i}PS^e and one for Gecode). There are several important points to note. Firstly, for three out of the four kinds of symmetries, our algorithms can be proved to be complete despite the simplicity of the algorithms. Secondly, LDSB does not truly compose symmetries, but rather applies them repeatedly to literals. This is important for efficiency: while composing symmetries can lead to exponentially large sets of symmetries, LDSB computes at most a quadratic ($|X|\times|D|$) number of literals in (X,D,C) to obtain the same result. Thirdly, all this is achieved by using only very simple computations based on results from group theory. Finally, as we will see in the experimental evaluation, all these characteristics make LDSB a system that it is not only easy to implement (and, thus, add to any constraint system), but also easy to use and efficient enough to be used as a default.

4 Representing symmetries in LDSB

This Section introduces the internal syntax used by LDSB to represent symmetries. This representation is the basis of LDSB's symmetry breaking algorithms and it gives insight into the kinds of symmetries for which LDSB is most effective. Note that while users are currently asked to directly represent the symmetries of the problem in this syntax, it is straightforward to implement a simple interface that translates any other syntax (such as the common one used by GAP) into it.

4.1 The LDSB Patterns and their Syntax

Given a CSP $P = (X,D,C)$, the most general form for any variable symmetry s is $\langle x_1,\ldots,x_n\rangle \rightarrow \langle x'_1,\ldots,x'_n\rangle$, where $\{x_1,\ldots,x_n\} = \{x'_1,\ldots,x'_n\} = X$, and $s(x_i) = x'_i$. Since the inverse of each symmetry is also a symmetry, then $\langle x'_1,\ldots,x'_n\rangle \rightarrow \langle x_1,\ldots,x_n\rangle$ is also a symmetry. This motivates LDSB's syntax which represents s and its inverse simply as $\{\langle x_1,\ldots,x_n\rangle, \langle x'_1,\ldots,x'_n\rangle\}$. This representation is the same as Cauchy's standard two-line notation for permutations, except that we consider it to represent both the permutation and its inverse.

Naturally, the order in which the two tuples (which we will refer as sequences) appear in this set is not significant. This is the first of the four LDSB patterns introduced in this section, and it will be referred to as the *interchangeable variable sequences* pattern. Since any value symmetry can be similarly represented using sequences of values (instead of sequences of variables), the second LDSB pattern is the equivalent *interchangeable value sequences* pattern.

Note that, for simplicity, we will assume no variable (or value) appears more than once in an LDSB sequence. This does not cause any loss of generality since, by definition of symmetry, a variable (value) can only be mapped to one variable (value). Thus, only one occurrence is needed.

While all variable and value symmetries can be represented using one of the above two patterns, many symmetries or CSPs have special properties that allow them to be represented more compactly. As we will see, it is for these symmetries that LDSB is able to prune symmetric solutions more efficiently and to detect and prune more compositions of the symmetries. Let us discuss these properties and their consequences in terms of our syntax.

Omit self mappings: The first symmetry property is to map a certain subset of the elements (be it variables or values) to themselves. For these kinds of symmetries, LDSB uses a shorthand where the sequences omit those elements that are mapped to themselves. Note that, when denoting any symmetry using this shorthand, the elements in the two sequences are the same.

Example 4 For the Latin Square CSP of Example 1, one could use fifteen instances of our interchangeable sequences patterns to represent all value and all variable symmetries in the CSP. The following five instances represent all the value symmetries:

$\{\langle 1,2\rangle, \langle 2,1\rangle\}$
$\{\langle 1,3\rangle, \langle 3,1\rangle\}$
$\{\langle 2,3\rangle, \langle 3,2\rangle\}$
$\{\langle 1,2,3\rangle, \langle 2,3,1\rangle\}$
$\{\langle 1,2,3\rangle, \langle 3,1,2\rangle\}$

Note that values that are mapped to themselves are omitted from the sequences, and that LDSB does not explicitly represent the identity symmetry. The following five instances represent all the column symmetries:

$\{\langle x_{11},x_{21},x_{31},x_{12},x_{22},x_{32}\rangle,\langle x_{12},x_{22},x_{32},x_{11},x_{21},x_{31}\rangle\}$
$\{\langle x_{11},x_{21},x_{31},x_{13},x_{23},x_{33}\rangle,\langle x_{13},x_{23},x_{33},x_{11},x_{21},x_{31}\rangle\}$
$\{\langle x_{12},x_{22},x_{32},x_{13},x_{23},x_{33}\rangle,\langle x_{13},x_{23},x_{33},x_{12},x_{22},x_{32}\rangle\}$
$\{\langle x_{11},x_{21},x_{31},x_{12},x_{22},x_{32},x_{13},x_{23},x_{33}\rangle,\langle x_{13},x_{23},x_{33},x_{11},x_{21},x_{31},x_{12},x_{22},x_{32}\rangle\}$
$\{\langle x_{11},x_{21},x_{31},x_{12},x_{22},x_{32},x_{13},x_{23},x_{33}\rangle,\langle x_{12},x_{22},x_{32},x_{13},x_{23},x_{33},x_{11},x_{21},x_{31}\rangle\}$

while the next five instances represent all the row symmetries:

$\{\langle x_{11},x_{12},x_{13},x_{21},x_{22},x_{23}\rangle,\langle x_{21},x_{22},x_{23},x_{11},x_{12},x_{13}\rangle\}$
$\{\langle x_{11},x_{12},x_{13},x_{31},x_{32},x_{33}\rangle,\langle x_{31},x_{32},x_{33},x_{11},x_{12},x_{13}\rangle\}$
$\{\langle x_{21},x_{22},x_{23},x_{31},x_{32},x_{33}\rangle,\langle x_{31},x_{32},x_{33},x_{21},x_{22},x_{23}\rangle\}$
$\{\langle x_{11},x_{12},x_{13},x_{21},x_{22},x_{23},x_{31},x_{32},x_{33}\rangle,\langle x_{31},x_{32},x_{33},x_{11},x_{12},x_{13},x_{21},x_{22},x_{23}\rangle\}$
$\{\langle x_{11},x_{12},x_{13},x_{21},x_{22},x_{23},x_{31},x_{32},x_{33}\rangle,\langle x_{21},x_{22},x_{23},x_{31},x_{32},x_{33},x_{11},x_{12},x_{13}\rangle\}$

The astute reader will note that the last instances in each group of five are in fact the inverse of the second last and, therefore, redundant. □

Simplify involutions: The second symmetry property is to be equal to its inverse (in the language of permutations, these are termed "involutions"). For such symmetries we can split their elements into three disjoint sets *Id*, El_1 and El_2, where *Id* contains all the elements that are mapped to themselves, while all elements in El_1 are mapped to elements in El_2 and vice versa (and, thus, have the same cardinality $|El_1| = |El_2|$). LDSB represents involutions as a set with two sequences, one for the elements in El_1 and one for the elements in El_2.

Example 5 Consider the fifteen pattern instances given in Example 4 for the Latin Square CSP of size 3. If we use the above property (and also eliminate the three instances previously identified as redundant), we can simplify the instances obtaining the following ones:

$\{\langle 1\rangle,\langle 2\rangle\}$
$\{\langle 1\rangle,\langle 3\rangle\}$
$\{\langle 2\rangle,\langle 3\rangle\}$
$\{\langle 1,2,3\rangle,\langle 2,3,1\rangle\}$
$\{\langle x_{11},x_{21},x_{31}\rangle,\langle x_{12},x_{22},x_{32}\rangle\}$
$\{\langle x_{11},x_{21},x_{31}\rangle,\langle x_{13},x_{23},x_{33}\rangle\}$
$\{\langle x_{12},x_{22},x_{32}\rangle,\langle x_{13},x_{23},x_{33}\rangle\}$
$\{\langle x_{11},x_{21},x_{31},x_{12},x_{22},x_{32},x_{13},x_{23},x_{33}\rangle,\langle x_{13},x_{23},x_{33},x_{11},x_{21},x_{31},x_{12},x_{22},x_{32}\rangle\}$
$\{\langle x_{11},x_{12},x_{13}\rangle,\langle x_{21},x_{22},x_{23}\rangle\}$
$\{\langle x_{21},x_{22},x_{23}\rangle,\langle x_{31},x_{32},x_{33}\rangle\}$
$\{\langle x_{11},x_{12},x_{13}\rangle,\langle x_{31},x_{32},x_{33}\rangle\}$
$\{\langle x_{11},x_{12},x_{13},x_{21},x_{22},x_{23},x_{31},x_{32},x_{33}\rangle,\langle x_{31},x_{32},x_{33},x_{11},x_{12},x_{13},x_{21},x_{22},x_{23}\rangle\}$

□

Set representation for full permutation sets: The third property is that all permutations of a particular set of sequences of elements (be it variables or values) are symmetries of the CSP. Such group of symmetries can be represented very compactly in LDSB as the set of sequences. Note that this set represents not only any permutation of two elements in the set but also their compositions and, thus, the full permutation set.

Example 6 Consider the twelve pattern instances given in Example 5 for the Latin Square CSP of size 3. If we use the above property we can simplify the twelve instances into the following three. One for value permutations:

$$\{\langle 1\rangle, \langle 2\rangle, \langle 3\rangle\}$$

One for column permutations:

$$\{\langle x_{11}, x_{21}, x_{31}\rangle, \langle x_{12}, x_{22}, x_{32}\rangle, \langle x_{13}, x_{23}, x_{33}\rangle\}$$

And one for row permutations:

$$\{\langle x_{11}, x_{12}, x_{13}\rangle, \langle x_{21}, x_{22}, x_{23}\rangle, \langle x_{31}, x_{32}, x_{33}\rangle\} \quad \square$$

Sequences of length one: If all sequences in an instance of our interchangeable sequences pattern contain only one variable (value), then we can simplify this instance by using a set of variables (values) rather than a set of tuples of variables (values). This special version of the sequence pattern will be referred to as the *interchangeable variables* pattern, and the equivalent one for values will be referred to as the *interchangeable values* pattern. These two, together with our two general interchangeable sequences patterns, form the four patterns used by LDSB. Note that instances of all four patterns can be detected automatically [19, 31].

LDSB syntax: Based on the above discussion, let us now formally define the syntax used by LDSB to represent symmetries. Given a CSP $P = (X, D, C)$, an LDSB *pattern instance* W must be an instance of either the interchangeable variables (values) pattern, or the interchangeable variable (value) sequences pattern. In the former case, W must be a set of variables (values) such that $W \subseteq X$ ($W \subseteq D$). In the latter case, W must be a set of variable (value) sequences such that the following five properties apply. Let $seq^i(p)$ denote the element (variable or value) in the p^{th} position of seq^i. First, $vars(W) \subseteq X$ ($vals(W) \subseteq D$). Second, all sequences must have the same length k. Third, all elements in a sequence must be different, i.e., $\forall i. |vars(seq^i)| = k$ ($|vals(seq^i)| = k$). Fourth, no two sequences seq^i and seq^j, where $i \neq j$, can have the same element in a given position, i.e., $\forall p. seq^i(p) \neq seq^j(p)$. And finally, every pair of sequences seq^i and seq^j must be either disjoint, i.e., $vars(seq^i) \cap vars(seq^j) = \emptyset$ ($vals(seq^i) \cap vals(seq^j) = \emptyset$), or have the same set of elements, i.e., $vars(seq^i) = vars(seq^j)$ ($vals(seq^i) = vals(seq^j)$).

Example 7 LDSB represents the symmetries of the Latin Square CSP of size 3 by means of the following three pattern instances:

$$\{1, 2, 3\}$$
$$\{\langle x_{11}, x_{21}, x_{31}\rangle, \langle x_{12}, x_{22}, x_{32}\rangle, \langle x_{13}, x_{23}, x_{33}\rangle\}$$
$$\{\langle x_{11}, x_{12}, x_{13}\rangle, \langle x_{21}, x_{22}, x_{23}\rangle, \langle x_{31}, x_{32}, x_{33}\rangle\}$$

where the first one is an instance of the interchangeable values pattern, and the last two are instances of the interchangeable variable sequences pattern.

For a Latin Square of size N, LDSB can represent and break all these symmetries with complexity $O((L+1)N^2)$, where L is the number of literals eliminated due to symmetry (see Section 5 for a more detailed complexity result). $\square$

As we will see later, LDSB symmetry-breaking algorithms also break some (not all) compositions of these symmetries (see section 5 for details).

The internal syntax presented above is a slight departure from standard notation, but is a better match for LDSB. Consider a problem that builds a 4×3 matrix of variables [[A,B,C],[D,E,F],[G,H,I],[J,K,L]], where every permutation of the rows is a variable symmetry. Our notation for these symmetries is simply $\{\langle A,B,C\rangle, \langle D,E,F\rangle, \langle G,H,I\rangle, \langle J,K,L\rangle\}$. A representation using using individual symmetries in standard cycle notation would be the three symmetries $(AD)(BE)(CF)$, $(AG)(BH)(CI)$, and $(AJ)(BK)(CL)$. The disadvantage of this form is that the symmetry interchanging, say, row DEF with row JKL is not immediately apparent (in this form it requires applying the first symmetry and then the third) making it more difficult to process by our system.

4.2 The symmetries represented by LDSB patterns

Let us now discuss the symmetries represented by each pattern instance. Consider a CSP $P = (X,D,C)$ that has a pattern instance W and let $\mathscr{G}_{W,P}$ denote the permutation group on $lit(P)$ represented by W. If W is an instance of the interchangeable variables (values) pattern, then $\mathscr{G}_{W,P}$ is identical to $\mathscr{S}_{W,P}$. If $W = \{seq^1, \ldots, seq^m\}$ is an instance of the interchangeable value sequences pattern, then $\mathscr{G}_{W,P}$ is induced by the set $\{s_{i,j} \mid i,j \in 1..m\}$ of permutations on values, where:

$$s_{i,j} = \begin{cases} s_{i,j}(seq^i(p)) = seq^j(p) & \forall p \in 1..k \\ s_{i,j}(seq^j(p)) = seq^i(p) & \forall p \in 1..k \text{ if } vals(seq^i) \cap vals(seq^j) = \emptyset \\ s_{i,j}(d) = d & \text{for all } d \in D \setminus (vals(seq^i) \cup vals(seq^j)) \end{cases}$$

and k is the length of every sequence in W. Note that, given the five properties of the instances of interchangeable value sequences pattern introduced in Section 4.1, each $s_{i,j}$ is a permutation on some $D(x)$ and $\{s_{i,j} \mid i,j \in 1..m\}$ forms a permutation group. The set $\mathscr{G}_{W,P}$ can be similarly defined when W is an instance of the interchangeable variable sequences pattern.

Then, given a CSP P with set $Patts = \{W^1, \ldots, W^m\}$ of pattern instances, $Patts$ represents any permutation in the permutation group $[\mathscr{G}_{W^1,P}, \ldots, \mathscr{G}_{W^m,P}]$, where $\mathscr{G}_{W^i,P}$ is the permutation group represented by pattern instance W^i.

Note that any variable (and value) permutation can be represented in LDSB using an instance of one of the four patterns introduced above. This already distinguishes LDSB from other incomplete methods, such as SSB, that can only break permutations represented by instances of the interchangeable variables and values patterns (and not with those represented by the more general interchangeable sequences patterns).

Example 8 The variable symmetry group resulting from permuting columns in the Latin square problem cannot be broken by SSB since, to do so, it must be represented in terms of sets of variable permutations. This is not possible since after a permutation the variables in each column must remain in the same order. The value symmetry group generated by the symmetry that rotates values $1 \to 2 \to 3 \to 4 \to 1$ cannot be represented by SSB either, while it is represented in LDSB by the instance of the interchangeable value sequences pattern $\{\langle 1,2,3,4\rangle, \langle 2,3,4,1\rangle\}$. □

However, LDSB is most effective when the pattern instance W is a compact representation of $\mathscr{G}_{W,P}$. That is, when there is considerable distance between the amount of space needed to store W and the number of symmetries represented by it. This is because, as we will see later, the symmetry breaking algorithms will then be able to break a large number of symmetries by traversing a small data structure. If W is an instance of the interchangeable variable or value pattern, LDSB uses $O(W)$ storage space to represent $|\mathscr{G}_{W,P}| = |\mathscr{S}_{W,P}| = |W|!$ symmetries. Thus, in this case the bigger the W the more compact the representation is. If W is an instance of the interchangeable variable or value sequence pattern, LDSB uses $O(|W| * k)$ storage space, where k is the length of the sequences in W, to represent at most $|W|!$ symmetries and at least $|W|$, each of which takes $O(k)$ to store. The former occurs whenever every permutation of W results in a distinct value symmetry; for example, if all symmetries are involutions and therefore all sequences in W are disjoint. The latter occurs whenever the only permutations of sequences that lead to distinct value symmetries are those that map each sequence to one other sequence. This case arises, for instance, when every two sequences in W share at least one element in different positions; let us illustrate this with an example.

Example 9 Consider the symmetries of the Latin Square CSP of size 4 given in Example 7. For the instance of the interchangeable values pattern $\{1,2,3,4\}$, LDSB only needs to store 4 values to represent $4! = 24$ symmetries, while if we had to store each of these 24 symmetries we would need $24 \times 4 = 96$ values (since storing each symmetry would require the 4 images of the 4 original values). This is a considerable saving (from 96 down to 4). Consider now the case of the instance of the interchangeable variable sequences pattern that swaps columns and which, as mentioned in the previous example, it can be compactly represented by $\{\langle x_{11},x_{21},x_{31},x_{41}\rangle, \langle x_{12},x_{22},x_{32},x_{42}\rangle, \langle x_{13},x_{23},x_{33},x_{43}\rangle, \langle x_{14},x_{24},x_{34},x_{44}\rangle\}$.
LDSB needs to store 16 variables, to represent $4! = 24$ symmetries (those obtained by any permutation of the columns). If we had to store each of these 24 symmetries we would need $24 \times 16 = 384$ variables (since storing each symmetry would require the images of the 16 variables). Again, this is a considerable saving (from 384 down to 16). □

As we will see later, it is common for interchangeable variable and value sequences to be disjoint (e.g., all benchmarks used in our experiments have disjoint sequences) and, therefore, for the representation to be compact leading to very effective pruning by the symmetry breaking search algorithms in LDSB.

Interestingly, other types of symmetries previously presented in the literature for specific kinds of problems also result in a compact representation. For example, wreath symmetries [7] occur in CSPs that not only have piecewise value interchangeability for sets of values $W^1,\ldots,W^m$, but also have symmetries that map values from one set to values to the other. This is simply represented in LDSB by the sets $W^1,\ldots,W^m$ (each of which is an instance of our interchangeable values pattern) and a single set $\{seqW^1,\ldots,seqW^m\}$, where each $seqW^i$ is a sequence containing all elements in W^i in some order (which is an instance of our interchangeable value sequences pattern).

Note also that LDSB is a partial symmetry breaking method since it can only break instances of the above symmetry patterns and, therefore, it cannot break variable-value symmetries other than those obtained by the composition of variable and value symmetries. Furthermore, as we will see later, LDSB can only guarantee completeness of symmetry breaking for instances of three out of its four patterns.

Algorithm 1: `search`(X,*Patts*)

```
input : Set $X$ of search variables; Set Patts of pattern instances
1  if all variables in $X$ are fixed then stop;
2  ;
3  Select variable $x \in X$ and value $d \in D(x)$;
4  Create choice point
5      Left branch:
6          Assert $x = d$;
7          search($X$,update($x = d$,Patts));
8      Right branch:
9          $L \leftarrow$ some_orbit($x = d$,Patts);
10         foreach literal $l \in L$ do Assert $\neg(l)$;;
11         search($X$,Patts);
```

5 Breaking symmetries during search

The symmetry breaking algorithm of LDSB is similar to that of SBDS, with the main differences being (a) the kinds of symmetries used, (b) the way these symmetries are represented and processed, and (c) the fact that only symmetries that are active at a search node n are used to compute the symmetry breaking constraints at n (recall that symmetry σ is active at n if $\sigma(A) = A$, where A is the assignment labelling n, i.e., σ is active if it is a set stabiliser of A).

The search starts with a call to `search`(X,*Patts*), described in Algorithm 1, where X is the set of variables to be labelled and *Patts* is a set of pattern instances representing symmetries known to be active at the root node (i.e., all symmetries in the problem, since the assignment labelling the root node is empty). If all variables in X are fixed (i.e., their domains are known to have a single value), the search is complete. Otherwise, the search chooses the next non-fixed variable x and value d, according to the labeling strategy chosen by the user, and it explores the $x = d$ branch as usual. The only difference is that, before doing that, the search might need to update *Patts*, since posting $x = d$ might have caused some symmetries in the pattern instances to become non-active (or even broken). Upon backtracking, the search posts the negation of (some of) the literals that, according to *Patts*, are symmetric to $x = d$, i.e., (some of) those in the orbit of $x = d$ for each of the pattern instances (which it always includes $x = d$ itself). In summary, the only difference with the usual search is that we update the set of symmetries when exploring the left-hand side, and we post the negation of the orbit when exploring the right-hand side. Note that Algorithm 1 does not impose restrictions on the selection of x or d and, therefore, any variable or value search strategy can be used.

The way in which each instance pattern $W \in$ *Patts* is updated and the extra pruning achieved depends on the pattern associated to W, as indicated by Algorithms 2 and 3, respectively. Note that the algorithms are not particularly surprising in themselves, since they are the obvious result of applying group theory to the particular symmetry patterns being processed. What we find surprising (and is the basis of LDSB effectiveness) is the fact that such simple sequence of steps is sufficient to completely break any instance of three out of the four patterns (as proved in the following sections).

5.1 Correctness and Completeness for Interchangeable Variables

Let us first focus on interchangeable variables. Updating a set W of interchangeable variables for literal $x = d$ simply requires us to eliminate x from W, since any variable symmetry σ

Algorithm 2: `update`($x = d$, *Patts*)

```
input : Set $Patts$ of pattern instances before literal $x = d$ is posted
output: Set of pattern instances after literal $x = d$ is posted
foreach symmetry pattern $W \in Patts$ do
    if $W$ is a set of interchangeable variables then $W \leftarrow W \setminus \{x\}$;
    ;
    else if $W$ is a set of interchangeable values then $W \leftarrow W \setminus \{d\}$;
    ;
    else if $W$ is a set of interchangeable value sequences then
        $W \leftarrow W \setminus \{sq | d \in sq\}$;
    else if $W$ is a set of interchangeable variable sequences then
        do nothing;
    end
end
return $Patts$;
```

Algorithm 3: `some_orbit`($x = d$, *Patts*)

```
input : Set $Patts$ of pattern instances
output: Some subset $L$ of the literals in the orbit of $x = d$ according to $Patts$
$L \leftarrow \{x = d\}$;
foreach pattern instance $W \in Patts$ do
    if $W$ is a set of interchangeable variables AND $x \in W$ then
        foreach $w \in W \setminus \{x\}$ do $L \leftarrow L \cup \{w = d\}$;
        ;
    if $W$ is a set of interchangeable values AND $d \in W$ then
        foreach $e \in W \setminus \{d\}$ do $L \leftarrow L \cup \{x = e\}$;
        ;
    if $W$ is a set of interchangeable value sequences AND $\exists seq^1 \in W, seq^1[i] = d$ then
        foreach $seq^2 \in W \setminus \{seq^1\}$ do $L \leftarrow L \cup \{x = seq^2[i]\}$;
        ;
    if $W$ is a set of interchangeable variable sequences AND $\exists seq^1 \in W, seq^1[i] = x$ then
        foreach $seq^2 \in W \setminus \{seq^1\}$ do
            if active($seq^1, seq^2$) then $L \leftarrow L \cup \{seq^2[i] = d\}$;
            ;
        end
    end
end
return $L$;
```

that maps x to a different variable w in W is not active for node $x = d$. This is because the assignment A labelling this node contains literal $x = d$ and since, by construction, w cannot appear in A, we have that $\sigma(A) \neq A$. Computing (some of) the set of literals symmetric to $x = d$ is achieved by simply computing all literals $w = d$ such that w is in W and is different from x.

Example 10 Consider a CSP $(\{x_1, x_2, x_3, x_4, y_1, y_2\}, 1..4, C)$, where W is the set of interchangeable variables $\{x_1, x_2, x_3, x_4\}$. The left-hand side of Figure 4 shows the effect of the search on W and on the symmetry breaking constraints generated (in bold). Importantly, W only changes when the search explores a literal whose variable is in W. Also, note that once $x_1 = 1$ is posted, the variable symmetry $s =< x_1, x_3 > \leftrightarrow < x_2, x_4 >$ is non-active, since it maps x_1 to x_2; that is why s is not represented by the updated W at the node labeled by assignment $\{y_1 = 1, x_1 = 1\}$, i.e., by $W_{\{y_1=1, x_1=1\}} = \{x_2, x_3, x_4\}$. While s is again active at node $x_2 = 1$, the fact that s is not represented by $W_{\{y_1=1, x_1=1, y_2=2, x_2=1\}} = \{x_3, x_4\}$ either, is

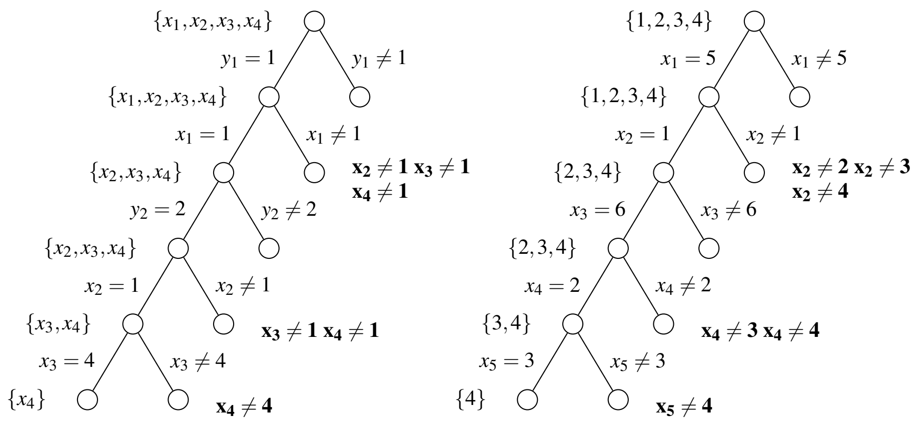


Fig. 4: Evolution of interchangeable variable and interchangeable value patterns. Instances are shown as sets, and symmetry breaking constraints in bold.

not a problem, since $W_{\{y_1=1,x_1=1,y_2=2,x_2=1\}}$ does represent symmetry $< x_3 > \leftrightarrow < x_4 >$ which achieves as much pruning as s at this point. □

Consider a CSP $P = (X,D,C)$ that has set of interchangeable variables $W \subseteq X$, and an LDSB search that has reached node n labelled by assignment A, with current $Patts = \{W_A\}$ and branches $x = d$ and $x \neq d$. In this context, we can prove that the search performed by LDSB is correct for a single set of interchangeable variables W, i.e., that the set of literals pruned from the search is a subset of (in fact, identical to) the set formed by the literals in the orbit of $x = d$ according to permutation group $SetStab(A, \mathscr{G}_{W,P})$, which contains all symmetries in $\mathscr{G}_{W,P}$ that are active at n. Formally:

Theorem 1 *(Correctness for interchangeable variables)*
For a CSP P, let W be an instance of the interchangeable variables pattern for P, and W_A the pattern instance computed by LDSB from W at assignment A. Then, the set returned by `some_orbit`*($x = d, \{W_A\}$) is equal to that obtained by Orbit($x = d, SetStab(A, \mathscr{G}_{W,P})$).*

Proof Since W is an instance of the interchangeable variables pattern, W is a set of variables in P and $\mathscr{G}_{W,P} = \mathscr{S}_{W,P}$. Therefore,

$$SetStab(A, \mathscr{G}_{W,P}) = SetStab(A, \mathscr{S}_{W,P})$$

By definition of set stabiliser, every element of $SetStab(A, \mathscr{S}_{W,P})$ moves literals in A to other literals in A. Therefore, $SetStab(A, \mathscr{S}_{W,P})$ can be generated by the composition of permutations on $W \cap vars(A)$ that stabilise A, and permutations on $W \setminus vars(A)$ (all of which stabilise A), which gives:

$$SetStab(A, \mathscr{S}_{W,P}) = [SetStab(A, \mathscr{S}_{W \cap vars(A),P}), \mathscr{S}_{W \setminus vars(A),P}]$$

By definition of search we have $x \notin vars(A)$ and, thus, $x = d$ cannot be moved by any permutation in $SetStab(A, \mathscr{S}_{W \cap vars(A),P})$. Therefore, the orbit of $x = d$ is not affected by these permutations, which gives:

$$Orbit(x = d, SetStab(A, [\mathscr{S}_{W \cap vars(A),P}), \mathscr{S}_{W \setminus vars(A),P}]) = Orbit(x = d, \mathscr{S}_{W \setminus vars(A),P})$$

Algorithm 4: `active`(seq^1, seq^2)

```
input : Sequences $seq^1$ and $seq^2$ of interchangeable variables
output: true if $seq^1 \leftrightarrow seq^2$ is known to be active, false otherwise
1 for $j \leftarrow 1$ to $|seq^1|$ do
2     if fixed($seq^1[j]$) and fixed($seq^2[j]$) and $seq^1[j] = seq^2[j]$ then
3         do nothing;
4     else if not fixed($seq^1[j]$) and not fixed($seq^2[j]$) then
5         do nothing;
6     else return false ;
7     ;
8 end
9 return true;
```

By Algorithm 2, we have $W \setminus vars(A) = W_A$. Thus, $Orbit(x = d, SetStab(A, \mathscr{S}_{W,P}))$ is equal to $Orbit(x = d, \mathscr{S}_{W_A,P})$ which is exactly the set `some_orbit`$(x = d, \{W_A\})$ computed by Algorithm 3.

The fact that `some_orbit`$(x = d, \{W_A\})$ is equal to (rather than a subset of) $Orbit(x = d, SetStab(A, \mathscr{G}_{W,P}))$ not only proves correctness but is also useful to prove our completeness theorem. This theorem is based on the completeness of SBDS and basically states that the pruning effect of each conditional constraint

$$\sigma(A) \Rightarrow \neg\sigma(x = d)$$

posted by SBDS for any $\sigma \in \mathscr{S}_{W,P}$ is also achieved in LDSB.

Theorem 2 *(Completeness for interchangeable variables)*
For a CSP P, let W be an instance of the interchangeable variables pattern for P, W_A the pattern instance computed by LDSB from W at assignment A and C_A the constraint:

$$A \wedge x \neq d \wedge \{\neg lit | lit \in \texttt{some\_orbit}(x = d, \{W_A\})\}$$

Then, for any σ in $\mathscr{G}_{W,P}$, the constraint $\sigma(A) \Rightarrow \neg\sigma(x = d)$ is entailed by C_A.

Proof Since W is an instance of the interchangeable variables pattern, W is a set of variables in P and $\mathscr{G}_{W,P} = \mathscr{S}_{W,P}$. We will prove that the SBDS constraint is entailed by C_A by showing that either $\sigma(A)$ is inconsistent with C_A or $\neg\sigma(x = d)$ is entailed by C_A. In particular, if $x \notin W$ then $\neg\sigma(x = d)$ is equal to $x \neq d$, which is trivially entailed by C_A (directly appears in it). Assume, to the contrary, that $x \in W$. Define $vars_d(A) = \{v | (v = d) \in A\}$.

- Suppose $\sigma \in SetStab(vars_d(A), \mathscr{G}_{W,P})$. By definition of search, $x \notin vars(A)$ and, thus, $x \notin vars_d(A)$. Since σ is a set stabiliser, $\sigma(x) \notin vars_d(A)$. If $\sigma(x) \in vars(A) \setminus vars_d(A)$, then $\neg\sigma(x = d)$ is trivially entailed by C_A (since $\sigma(x)$ is already assigned to a different value by A). Otherwise, $\sigma(x) \in W \setminus vars(A)$ and, by Theorem 1, $\sigma(x = d)$ is in `some_orbit`$(x = d, \{W_A\})$ and therefore its negation is trivially entailed by C_A (it directly appears in C_A).
- Suppose, to the contrary, that $\sigma \notin SetStab(vars_d(A), \mathscr{G}_{W,P})$. Then there must exist a variable $v \in vars_d(A)$ s.t. $\sigma(v) \notin vars_d(A)$. If $\sigma(v) \in vars(A) \setminus vars_d(A)$, then for some $d' \neq d$, $(v = d) \in A$ and $\sigma(v = d') \in A$, which means $\sigma(A)$ is inconsistent with A. If, to the contrary, $\sigma(v) \notin vars(A)$, then $\sigma(v = d)$ is in `some_orbit`$(x = d, \{W_A\})$ and therefore its negation is in C_A. Thus, we again have $\sigma(A)$ inconsistent with C_A.

It is easy to extend the above two theorems for the case of piecewise interchangeable variables.

Theorem 3 *(Correctness for piecewise interchangeable variables)*
For a CSP P, let $W^1,\ldots,W^m$ be disjoint instances of the interchangeable variables pattern for P, and $W_A^1\ldots,W_A^m$ the pattern instances computed by LDSB from $W^1,\ldots,W^m$, respectively, at assignment A. Then, the set returned by `some_orbit`*($x=d,\{W_A^1,\ldots,W_A^m\}$) is equal to that obtained by $Orbit(x=d, SetStab(A,[\mathscr{G}_{W^1,P},\ldots,\mathscr{G}_{W^m,P}]))$.*

Proof We prove this theorem by reducing to the case of a single set of interchangeable variables. Since $W^1,\ldots,W^m$ are disjoint instances of the interchangeable variables pattern, they are disjoint sets of variables in P. Let W^i be the set containing x. Only the permutation group $\mathscr{G}_{W^i,P}$ contains permutations that affect the orbit of $x=d$. Therefore:

$$Orbit(x=d, SetStab(A,[\mathscr{G}_{W^1,P},\ldots,\mathscr{G}_{W^m,P}])) = Orbit(x=d, SetStab(A,\mathscr{G}_{W^i,P}))$$

By Algorithm 3, only W_A^i is used to add literals to the computed orbit. Thus:

$$\texttt{some\_orbit}(x=d,\{W_A^1,\ldots,W_A^m\}) = \texttt{some\_orbit}(x=d,\{W_A^i\},)$$

Finally, by Theorem 1, this is equal to $Orbit(x=d, SetStab(A,\mathscr{G}_{W^i,P}))$.

Theorem 4 *(Completeness for piecewise interchangeable variables)*
For a CSP P, let $W^1,\ldots,W^m$ be disjoint instances of the interchangeable variables pattern for P, $W_A^1,\ldots,W_A^m$ the pattern instances computed by LDSB from $W^1,\ldots,W^m$, respectively, at assignment A, and C_A the constraint:

$$A \wedge x \neq d \wedge \{\neg lit | lit \in \textit{some\_orbit}(x=d,\{W_A^1,\ldots,W_A^m\})\}$$

Then, for any σ in $[\mathscr{G}_{W^1,P},\ldots,\mathscr{G}_{W^m,P}]$, the constraint $\sigma(A) \Rightarrow \neg\sigma(x=d)$ is entailed by C_A.

Proof Since $W^1,\ldots,W^m$ are disjoint instances of the interchangeable variables pattern, they are disjoint sets of variables in P. If $x \notin W^1 \cup \ldots \cup W^m$ then $\neg\sigma(x=d)$ is equal to $x \neq d$, which is trivially entailed by C_A (directly appears in it). Assume, to the contrary, that $x \in W^1 \cup \ldots \cup W^m$. Since the sets are disjoint, x must appear in a single W^i. This case can again be reduced to a single set of interchangeable variables, with an equivalent proof to that of Theorem 2.

The above theorem proves that LDSB is complete for piecewise interchangeable variables, just as SBDS, SBDD and SSB.

5.2 Correctness and Completeness for Interchangeable Values

Let us now focus on interchangeable values. Similarly to the case of interchangeable variables, updating a set W of interchangeable values for literal $x=d$ simply requires us to eliminate d from W, while the orbit consists of all literals $x=e$ such that value e is in W and is different from d.

Example 11 Consider the CSP $(X, 1..10, C)$ where $W = 1..4$ are interchangeable values. The right-hand side of Figure 4 shows the effect of the search on W and on the symmetry breaking constraints generated (in bold). As for the case of variables, W only changes if the explored literal involves a value in W. □

Theorem 5 *(Correctness for interchangeable values)*
For a CSP P, let W be an instance of the interchangeable variables pattern for P, and W_A the pattern instance computed by LDSB from W at assignment A. Then, the set returned `some_orbit`*$(x = d, \{W_A\})$ is equal to that obtained by $Orbit(x = d, SetStab(A, \mathscr{G}_{W,P}))$.*

Proof Since W is an instance of the interchangeable values pattern, W is a set of values in P and $\mathscr{G}_{W,P} = \mathscr{S}_{W,P}$. Therefore,

$$SetStab(A, \mathscr{G}_{W,P}) = SetStab(A, \mathscr{S}_{W,P})$$

As for the case of variables, by definition of the set stabiliser every element of $SetStab(A, \mathscr{S}_{W,P})$ moves literals in A to other literals in A. Therefore, $SetStab(A, \mathscr{S}_{W,P})$ can be generated by the composition of permutations on $W \cap vals(A)$ that stabilise A, and permutations on $W \setminus vals(A)$ (all of which stabilise A), which gives:

$$SetStab(A, \mathscr{S}_{W,P}) = [SetStab(A, \mathscr{S}_{W \cap vals(A),P}), \mathscr{S}_{W \setminus vals(A),P}]$$

For the case of values, any $\sigma \in SetStab(A, \mathscr{S}_{W \cap vals(A),P})$ must be the identity permutation (i.e., $SetStab(A, \mathscr{S}_{W \cap vals(A),P}) = PointStab(A, \mathscr{S}_{W \cap vals(A),P})$), since such σ can only move the value in any literal $(y = e) \in A$ to itself. Therefore,

$$SetStab(A, \mathscr{S}_{W,P}) = \mathscr{S}_{W \setminus vals(A),P}$$

and, consequently,

$$Orbit(x = d, SetStab(A, \mathscr{S}_{W,P})) = Orbit(x = d, \mathscr{S}_{W \setminus vals(A),P})$$

As before, by Algorithm 2 we have $W \setminus vals(A) = W_A$ and, thus, $Orbit(x = d, SetStab(A, \mathscr{S}_{W,P}))$ is equal to $Orbit(x = d, \mathscr{S}_{W_A,P})$, which is exactly the set `some_orbit`$(x = d, \{W_A\})$ computed by Algorithm 3.

Theorem 6 *(Completeness for interchangeable values)*
For a CSP P, let W be an instance of the interchangeable values pattern for P, W_A the pattern instance computed by LDSB from W at assignment A and C_A the constraint:

$$A \wedge x \neq d \wedge \{\neg lit | lit \in \texttt{some\_orbit}(x = d, \{W_A\})\}$$

Then, for any σ in $\mathscr{G}_{W,P}$, the constraint $\sigma(A) \Rightarrow \neg\sigma(x = d)$ is entailed by C_A.

Proof As shown in the proof of Theorem 5 above, W is a set of values in P, $\mathscr{G}_{W,P} = \mathscr{S}_{W,P}$, and for any $\sigma \in \mathscr{S}_{W,P}$, $\sigma(A)$ can only be consistent with A if $\sigma \in \mathscr{S}_{W \setminus vals(A),P}$. In this case, Theorem 5 also proves that $\sigma(x = d)$ belongs to the set computed by `some_orbit`$(x = d, \{W_A\})$ and, thus, its negation is trivially entailed by C_A (it directly appears in C_A).

Similar correctness and completeness results to those obtained for piecewise interchangeable variables can be obtained for piecewise interchangeable values. The main difference (in the proofs) occurs in the completeness result since, in the case of value interchangeability, once a symmetry σ becomes non-active it is actually broken and, thus, cannot later be reactivated.

Theorem 7 *(Correctness for piecewise interchangeable values)*
For a CSP P, let $W^1, \ldots, W^m$ be disjoint instances of the interchangeable values pattern for P, and $W_A^1 \ldots, W_A^m$ the pattern instances computed by LDSB from $W^1, \ldots, W^m$, respectively, at assignment A. Then, the set returned by `some_orbit`*$(x = d, \{W_A^1, \ldots, W_A^m\})$ is equal to that obtained by $Orbit(x = d, SetStab(A, [\mathscr{G}_{W^1,P}, \ldots, \mathscr{G}_{W^m,P}]))$.*

Proof The proof is identical to that of Theorem 5, once we have reduced to the case of a single set of interchangeable values, using a similar reasoning to that used in the proof of Theorem 3.

Theorem 8 *(Completeness for piecewise interchangeable values)*
For a CSP P, let $W^1,\ldots,W^m$ be disjoint instances of the interchangeable values pattern for P, $W_A^1\ldots,W_A^m$ the pattern instances computed by LDSB from $W^1,\ldots,W^m$, respectively, at assignment A, and C_A the constraint:

$$A \wedge x \neq d \wedge \{\neg lit | lit \in \texttt{some\_orbit}(x = d, \{W_A^1,\ldots,W_A^m\})\}$$

Then, for any σ in $[\mathscr{G}_{W^1,P},\ldots,\mathscr{G}_{W^m,P}]$, the constraint $\sigma(A) \Rightarrow \neg\sigma(x = d)$ is entailed by C_A.

Proof Since $W^1,\ldots,W^m$ are disjoint instances of the interchangeable values pattern, they are disjoint sets of values in P and $\mathscr{G}_{W^i,P} = \mathscr{S}_{W^i,P}, \forall i \in 1..m$. Thus, $[\mathscr{G}_{W^1,P},\ldots,\mathscr{G}_{W^m,P}] = [\mathscr{S}_{W^1,P},\ldots,\mathscr{S}_{W^m,P}]$. For any $\sigma \in [\mathscr{S}_{W^1,P},\ldots,\mathscr{S}_{W^m,P}]$, $\sigma(A)$ can only be consistent with A if $\sigma \in [\mathscr{S}_{W^1\setminus vals(A),P},\ldots,\mathscr{S}_{W^m\setminus vals(A),P}]$. For such σ, Theorem 7 proves that $\sigma(x = d)$ belongs to the set computed by $\texttt{some\_orbit}(x = d, \{W_A^1,\ldots,W_A^m\})$ and, therefore, its negation is trivially entailed by C_A (it appears in C_A).

The above theorem proves that LDSB is as complete as SBDS, SBDD and SSB for piecewise interchangeable variables.

5.3 Correctness and Completeness for Interchangeable Value Sequences

Updating a set W of interchangeable value sequences for literal $x = d$ simply requires us to eliminate any sequence sq in W that contains value d since, as for the case of interchangeable values, symmetries containing d are broken and therefore permanently inactive. The set of symmetric literals that can be pruned consists of all $x = e$ such that for every sequence $seq^1 \in W$ where $seq^1(i) = d$ (i.e., d appears in position i) there is a sequence $seq^2 \in W$ different from seq^1 where $seq^2(i) = e$.

Example 12 Consider the CSP (X,D,C) where $D = 1..6$ and there is a symmetry σ that reflects the values, $\sigma(x = i) = (x = 7 - i)$. This symmetry can be represented by interchangeable value sequences $\{\langle 1,2,3\rangle,\langle 6,5,4\rangle\}$. If the search branches at the root on $x = 2$ (for some $x \in X$), then on the left branch the sequence $\langle 1,2,3\rangle$ is removed from the interchangeable set, while on the right branch $x \neq 2$ LDSB adds $x \neq 5$, since values 2 and 5 appear in the same position in different sequences. □

Correctness results can also be obtained for this case, though correctness is based on a subset relationship not on equality (as it is not always equal).

Theorem 9 *(Correctness for interchangeable value sequences)*
For a CSP P, let $W = \{seq^1,\ldots,seq^m\}$ be a set of m interchangeable value sequences, $\mathscr{G}_{W,P}$ be the group of symmetries represented by W, and W_A the pattern instance computed by LDSB at assignment A. The set $\texttt{some\_orbit}(x = d, \{W_A\})$ computed by LDSB is a subset of the set obtained by $Orbit(x = d, SetStab(A, \mathscr{G}_{W,P}))$.

Proof By Algorithm 2, $W_A \subset W$ and no sequence in W_A contains a value in $vals(A)$. Therefore, any $\sigma \in \mathscr{G}_{W_A,P}$ is also in $\mathscr{G}_{W,P}$ and stabilises A. By Algorithm 3, for any literal $\ell \in$ `some_orbit`$(x = d, \{W_A\})$ we have $\ell = (x = e)$ for some $e \neq d$. Also by Algorithm 3, there must exist a sequence $seq^i \in W_A$ such that $seq^i(p) = d$, and a different sequence seq^j where $seq^j(p) = e$. The symmetry σ_{ij}, which maps the values in seq^i to those in seq^j, is in $\mathscr{G}_{W_A,P}$ (and therefore in $\mathscr{G}_{W,P}$) and since, as shown above, all elements of $\mathscr{G}_{W_A,P}$ stabilise A, σ_{ij} is also in $SetStab(A, \mathscr{G}_{W,P})$.

Therefore, $\ell = \sigma_{ij}(x = d)$ must be in $Orbit(x = d, SetStab(A, \mathscr{G}_{W,P}))$.

The above theorem can be easily extended to sets of interchangeable value sequences. Note that, in this case the sets might not be disjoint.

Theorem 10 *(Correctness for sets of interchangeable value sequences)*
For a CSP P, let $W^1, \ldots, W^m$ be instances of the interchangeable value sequences pattern for P, and $W_A^1 \ldots, W_A^m$ the pattern instances computed by LDSB from $W^1, \ldots, W^m$, respectively, at assignment A. Then, the set returned by `some_orbit`$(x = d, \{W_A^1, \ldots, W_A^m\})$ is a subset of that obtained by $Orbit(x = d, SetStab(A, [\mathscr{G}_{W^1,P}, \ldots, \mathscr{G}_{W^m,P}]))$.

Proof Straightforward given Theorem 9 and the fact that, by Algorithm 3,

$$\texttt{some\_orbit}(x = d, \{W_A^1, \ldots, W_A^m\})) = \bigcup_{i \in 1..m} \texttt{some\_orbit}(x = d, \{W_A^i\}))$$

The completeness results for interchangeable values extend straightforwardly to the case of *disjoint* interchangeable value sequences by replacing $W \setminus vals(A)$ with $W \setminus \{seq \mid seq \in W \wedge seq \cap vals(A) \neq \emptyset\}$. However, this is not the case for non-disjoint sequences (one can easily build counter examples for this case).

5.4 Correctness for Interchangeable Variable Sequences

Unlike the other patterns, updating a set W of interchangeable variable sequences requires no work in LDSB, i.e., W_A is equal to W for every assignment A. As a result, some of the symmetries represented by W_A might be non-active or even broken for A. This complicates the pruning when backtracking from $x = d$ since, as shown in Algorithm 3, we now need to check whether the symmetries being considered are active or not when computing the orbit of $x = d$. In particular, for every sequence $seq^1 \in W$ where $seq^1(i) = x$, the algorithm finds any other $seq^2 \in W$ such that symmetry $seq^1 \leftrightarrow seq^2$ is active and adds literal $w = d$ to the orbit L, where $seq^2(i) = w$. This process will generate vacuous literals whenever two sequences contain the same variable in the same position, and multiple $y = d$ literals whenever more than one symmetry maps x to y. The reason for LDSB not to eliminate non-active symmetries from its representation, is that it might need them to detect reactivated symmetries even if, as shown in Figure 5, some reactivated symmetries might still be lost.

Theorem 11 *(Correctness for interchangeable variable sequences)*
For a CSP P, let $W = \{seq^1, \ldots, seq^m\}$ be a set of m interchangeable variable sequences and $\mathscr{G}_{W,P}$ be the group of symmetries represented by W. The set `some_orbit`$(x = d, \{W\})$ computed by LDSB for assignment A is a subset of the set obtained by $Orbit(x = d, SetStab(A, \mathscr{G}_{W,P}))$.

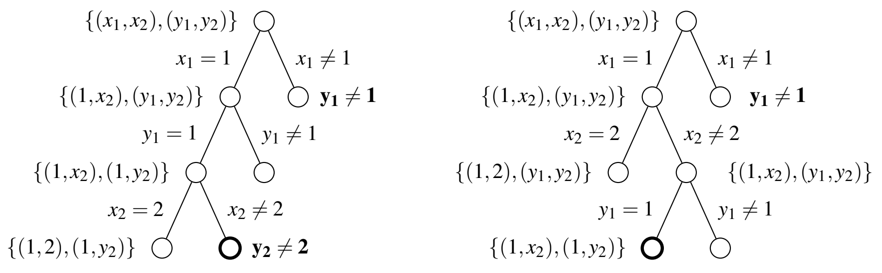


Fig. 5: Impact of the choice of search heuristic on completeness. While the highlighted nodes have the same assignment, LDSB's pruning is different.

Proof Let us define $\mathscr{G}_{W_A,P} \subseteq \mathscr{G}_{W,P}$ as the set of symmetries σ_{ij} that maps the variables in seq^i to those in seq^j, and for which $\texttt{active}(seq^i, seq^j)$ is true for assignment A. By Algorithm 4, every such $\sigma_{ij} \in \mathscr{G}_{W_A,P}$ stabilises A. By Algorithm 3, for any literal $\ell \in \texttt{some\_orbit}(x = d, \{W\})$, we have $\ell = (y = d)$ for some y distinct from x. Also, by Algorithm 3, there must exist a sequence $seq^i \in W$ such that x is in position p, and a different sequence seq^j where y also occurs at position p, such that $\sigma_{ij} \in \mathscr{G}_{W_A,P}$ (and therefore in $\mathscr{G}_{W,P}$) and since, as shown above, all elements of $\mathscr{G}_{W_A,P}$ stabilise A, σ_{ij} is also in $SetStab(A, \mathscr{G}_{W,P})$.

Therefore, $\ell = \sigma_{ij}(x = d)$ must be in $Orbit(x = d, SetStab(A, \mathscr{G}_{W,P}))$.

As for values, the above theorem can be easily extended to sets of interchangeable variable sequences.

Theorem 12 *(Correctness for sets of interchangeable variable sequences)*
For a CSP P, let $W^1, \ldots, W^m$ be instances of the interchangeable variable sequences pattern for P, and $W_A^1 \ldots, W_A^m$ the pattern instances computed by LDSB from $W^1, \ldots, W^m$, respectively, at assignment A. Then, the set returned by $\texttt{some\_orbit}(x = d, \{W_A^1, \ldots, W_A^m\})$ is a subset of that obtained by $Orbit(x = d, SetStab(A, [\mathscr{G}_{W^1,P}, \ldots, \mathscr{G}_{W^m,P}]))$.

Proof Straightforward given Theorem 11 and the fact that, by Algorithm 3,

$$\texttt{some\_orbit}(x = d, \{W_A^1, \ldots, W_A^m\}) = \bigcup_{i \in 1..m} \texttt{some\_orbit}(x = d, \{W_A^i\})$$

Note that for interchangeable variable sequences, LDSB may be incomplete when a symmetry is not active at assignment A but becomes active at a later search node (as shown in Figure 5).

5.5 Composing Symmetries

While the set L of literals returned by $\texttt{some\_orbit}(x = d, Patts)$ can be safely pruned from the search, we may be able to prune more literals if we consider not only the symmetries explicitly represented by each of the pattern instances in *Patts*, but also those obtained by composing these symmetries. As shown by our experimental results, this can be vital for some problems. Our implementation uses Algorithm 5 to find some of these extra literals by applying the symmetries represented in *Patts* not only to the original literal $x = d$, but also to any new symmetrical literal found. To achieve this, the algorithm builds a Queue of new literals to be processed (initially set to $x = d$). Each literal in Queue is then retrieved and (some of) the literals in its orbit computed. Any such literal that has not been computed

Algorithm 5: `composed_orbit`$(x = d, Patts)$

```
input  : Set $Patts$ of pattern instances before literal $x = d$ is posted
output: Set $L$ of literals symmetric to $x = d$ according to $Patts$
1  Queue $\leftarrow [x = d]$;
2  $L \leftarrow \{x = d\}$;
3  while Queue is not empty do
4      $l \leftarrow$ serve(Queue);
5      foreach literal $sl \in$ some_orbit($l$,$Patts$) do
6          if $sl \notin L$ then
7              $L \leftarrow L \cup \{sl\}$;
8              append(Queue, $sl$);
9          end
10     end
11 end
```

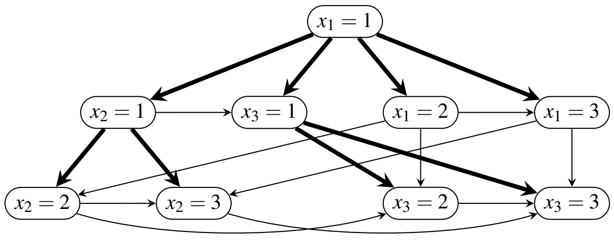


Fig. 6: Graph of symmetric relationship between literals in Example 13. Two literals are joined if there is an active symmetry that maps one to the other. Thick, straight edges show the relationships used to build set L.

before, is not only added to L but also appended to Queue to be processed. The process finishes when no literal can be retrieved from Queue. This process can be imagined as a breadth-first traversal of a graph, where each literal is a vertex and there is an edge from literal a to literal b if $\sigma(a) = b$ for some active symmetry σ.

Example 13 Consider a CSP $(\{\{x_1,x_2,x_3\},1..3,C)$ whose symmetries are represented by $Patts = \{\{x_1,x_2,x_3\},1..3\}$. Let us assume the search first tries $x_1 = 1$ and then backtracks. Figure 6 shows the graph-traversal view of the call to `composed_orbit`$(x_1 = 1, Patts)$, where the thick edges indicate new literals that are appended to Queue. The call first processes literal $x_1 = 1$ and calls `some_orbit`$(x_1 = 1, Patts)$. This returns the set $\{x_2 = 1, x_3 = 1, x_1 = 2, x_1 = 3\}$, where the first two literals are due to the variable symmetry and the last two are due to the value symmetry. All four literals are added to L and appended to Queue. Next, literal $x_2 = 1$ is processed and `some_orbit`$(x_2 = 1, Patts)$ returns the set $\{x_1 = 1, x_2 = 2, x_2 = 3, x_3 = 1\}$, where only the two middle literals are new and, thus, added to L and Queue. Next, literal $x_3 = 1$ is processed and `some_orbit`$(x_3 = 1, Patts,)$ returns the set $\{x_1 = 1, x_2 = 1, x_3 = 2, x_3 = 3\}$, where only the last two literals are new and, thus, added to L and Queue. At this point, L contains all literals in the problem and, consequently, the processing of every literal left in Queue does not add new literals to L or Queue. Thus, execution soon finishes with all literals pruned on the right branch $x_1 \neq 1$. □

Note that this algorithm does not truly compose symmetries, but rather applies symmetries repeatedly to literals. This difference is important for efficiency: while composing even a small number of symmetries can lead to exponentially large sets of symmetries, this algorithm computes at most a quadratic ($|X| \times |D|$) number of literals. However, the algorithm can also miss some symmetries. This is because LDSB first computes the stabilisers of each symmetry separately and then composes their elements. This can result in the loss of some opportunities, since for given permutation groups G_1 and G_2, $[\mathit{SetStab}(A,G_1),\mathit{SetStab}(A,G_2)]$ might be a proper subset of (rather than equal to) $\mathit{SetStab}(A,[G_1,G_2])$. This is illustrated in the following example.

Example 14 Consider the CSP $P = (\{x_1,x_2,x_3\},1..3,C)$, where all variables are interchangeable and all values are interchangeable. Let $G_1 = \mathscr{S}_{\{x_1,x_2,x_3\},P}$, $G_2 = \mathscr{S}_{\{1,2,3\},P}$ and assignment $A = \{x_1 = 1, x_2 = 2\}$. Then $\mathit{SetStab}(A,G_1)$ and $\mathit{SetStab}(A,G_2)$ contain only the identity, and therefore $[\mathit{SetStab}(A,G_1), \mathit{SetStab}(A,G_2)]$ also contains only the identity. However, $\mathit{SetStab}(A,[G_1,G_2])$ contains the permutation that simultaneously swaps x_1 with x_2, and value 1 with value 2.

Note that this result can indeed affect the orbit. For instance, in this case $\mathit{Orbit}(x_3 = 1,[\mathit{SetStab}(A,G_1),\mathit{SetStab}(A,G_2)])$ contains only $x_3 = 1$ itself, whereas $\mathit{Orbit}(x_3 = 1, \mathit{SetStab}(A,[G_1,G_2]))$ also contains $x_3 = 2$. □

Theorem 13 *(Correctness of LDSB's composition)*
For a CSP P, let $\mathscr{G}_{W_A^1,P},\ldots,\mathscr{G}_{W_A^m,P}$ be the groups represented by each of the pattern instances in $\mathit{Patts}_A = \{W_A^1,\ldots,W_A^m\}$, respectively, computed by LDSB at assignment A. Then the set of literals L_A = $\texttt{composed\_orbit}(x = d, \mathit{Patts}_A)$ is a subset of the set $\mathit{Orbit}(x = d,[\mathit{SetStab}(A,\mathscr{G}_{W_A^1,P}),\ldots,\mathit{SetStab}(A,\mathscr{G}_{W_A^m,P})])$.

Proof Follows directly from Theorems 3, 7, 9 and 11, and the fact that, as defined in Algorithm 5, a call to $\texttt{composed\_orbit}(x = d, \mathit{Patts}_A)$ repeatedly applies the symmetries represented by each pattern instance in Patts_A to each literal. This repeated application is exactly the definition of composition.

Note that if $W^1,\ldots,W^m$ represents either piecewise interchangeable variables, piecewise interchangeable values, or disjoint piecewise interchangeable value sequence patterns, (for which LDSB computes exactly the setwise stabilisers of A), then the subset relation in Theorem 13 is instead equality.

With composition, at the right branch $x \neq d$ of each search node A, LDSB must do $O(|\mathit{Patts}_A| \times |L_A \cup \{x = d\}|)$ work, where $|\mathit{Patts}_A|$ is the sum of the sizes of all symmetry pattern instances in Patts_A and $|L_A|$ is the set of literals computed by $\texttt{some\_orbit}(x = d, \mathit{Patts}_A)$, i.e., the set of literals that, together with $x = d$, will be eliminated.

5.6 Correctness of LDSB

We are now in a position to state the correctness of LDSB.

Theorem 14 *(Correctness of LDSB)*
For a CSP P, let $\mathit{Patts} = \{W^1,\ldots,W^m\}$ be a set of pattern instances for P, $W_A^1 \ldots, W_A^m$ the pattern instances computed by LDSB from $W^1,\ldots,W^m$, respectively, at any given assignment A, and $\mathscr{G}_{W_A^1,P},\ldots,\mathscr{G}_{W_A^m,P}$ be the groups represented by each of the associated pattern instances. The call to $\texttt{search}(X,\mathit{Patts})$ will prune at every right branch $x = d$ of the node associated to A a subset of the literals in $\mathit{Orbit}(x = d,[\mathit{SetStab}(A,\mathscr{G}_{W_A^1,P}),\ldots,\mathit{SetStab}(A,\mathscr{G}_{W_A^m,P}])$.

Proof Follows directly from the previous correctness theorems.

6 Implementation

LDSB has been implemented as a module for the ECL^{i}PS^e [32] constraint programming platform and for the Gecode [9] constraint solving system. The only requirement for a programmer to use LDSB is to specify a set of symmetries of the problem (see appendix B for a brief description of the user interface).

The core of the search ECL^{i}PS^e (and the Gecode) implementation closely follows Algorithm 1 via the ECL^{i}PS^e predicates `update/2` and `composed_orbit/2`. These two predicates implement the associated calls to `update`$(x = d, Patts)$ and `composed_orbit`$(x = d, Patts)$ in the algorithm.

Updating the symmetries The ECL^{i}PS^e predicate `update(X, D)` implements the call to `update`$(x = d, Patts)$, representing the variable `x` by the ECL^{i}PS^e variable `X` and the value `d` by an ECL^{i}PS^e atom which is the value of the input variable `D`. To achieve this, predicate `update(X, D)` first removes `X`, from the list of variables where it occurs, in each of the instances of the interchangeable variable pattern. Next, it removes `D` from the list of values where it occurs, in each of the instances of the interchangeable value pattern. And finally, it removes every sequence of values containing `D` from every instance of the interchangeable value sequence pattern. These pattern instances are immediately accessible via the *attribute* of `X`. Attributes are given before the search begins to every variable X appearing in *Patts*, and allow access to a lookup table that indicates the instances in which X appears and X's position in any sequence of any instance of the interchangeable variable sequence pattern. This makes traversal of pattern instances fast in practice.

Pruning symmetric literals The ECL^{i}PS^e predicate `composed_orbit(X, D, L)` implements the call to `composed_orbit`$(x = d, Patts)$, returning the list of literals to be pruned as the value of the output argument `L`. To achieve this, `composed_orbit(X, D, L)` first looks for each instance `W` of the interchangeable variable pattern that contains variable `X`, adding `Y` $\neq$ `D` to `L` for every other `Y` in `W`. Next, it looks for each instance `M` of the interchangeable value pattern that contains value `D`, adding `X` $\neq$ `V` to `L`, for every other `V` in `M`. Thirdly, it looks for each instance `Seqs` of the interchangeable value sequence pattern containing a sequence of values `S` in which `D` appears. For each such `Seqs` and `S`, it finds the position `P` in `S` in which `D` appears and, for each other sequence of values `S2` in `Seqs`, it looks up the `P`th value, `d2`, adding `X` $\neq$ `d2` to `L`.

Finally, for instances of the interchangeable variable sequence pattern, it accesses the lookup table to see in which instance variable `X` appears and, for each such instance `Seqs`, in which sequence `S` of `Seqs` the variable `X` appears, and in which position, `P`. It then compares each such `S` with every other sequence `S2` in `Seqs`. If the variables fixed in `S` exactly match those fixed in `S2`, and if the `P`th variable in `S2` is `Y`, then it adds `Y` $\neq$ `d` to `L`. Having direct access from `X` to the list of instances in which `X` appears, is crucial for performance.

Every literal added to `L` is also added to the queue of literals to be processed (if it has not previously been processed) to determine possible new prunings achieved by composition. Finally, predicate `prune` takes a list of elements of the form `Y` $\neq$ `Val` and posts them.

In Gecode we add the update and pruning steps to the system's own depth-first search engine, in similar locations to the ECL^{i}PS^e implementation.

There are several possible variations to the current implementation. One is to represent interchangeable variable (or value) sequences pairwise — i.e., represent each pair of sequences in `Seqs` separately. While this results in a quadratic number of extra symmetries, it allows us to permanently delete non-active pairs from the list. Additionally, when two variables in the same position of their respective sequences are bound to the same value, the variables can be removed from the sequences. Another variation is not to remove variables from the interchangeable variable lists as they become fixed. This would make updates faster but lookups slower (since the lists will be longer).

Supplementary Table 6 shows the results of a comparison between LDSB implemented using a pairwise representation of interchangeable variable sequences, and using the usual aggregated representation. The number shown is the time taken to find the first solution or all solutions, under the given variable ordering. In most cases there is no difference between the two representations; in the case of the balanced incomplete block design problem (`bibd`) there is a slight advantage to the pairwise representation, while in the Latin square problem there is a large penalty for it (due to its quadratic form). Since the pairwise form gives only a rare, small benefit and can cause a large penalty, we decided to exclude it from further experiments.

Handling Set Variables Symmetries involving set variables can be handled in the same way as for integer variables, with three modifications.

First, when branching at a search node on variable x and value d, the left and right children are $d \in x$ and $d \notin x$ (instead of $x = d$ and $x \neq d$) and we continue branching on variable x until it is fixed before we choose a different variable.

Second, we change the definition of "active" for sequences of set variables (see Algorithm 4). Instead of checking that each pair of corresponding variables has either both variables fixed to the same value or both variables unfixed, we simply check that the domains of the two variables are the same.

Third, we modify how interchangeable value symmetries are updated. When we first branch on a variable x, we record the state of any interchangeable value symmetry. As branches are made, we record each value used on a left branch. Once x has become fixed to a set E, we split any recorded interchangeable value symmetry on values V into two interchangeable value symmetries, one on $V \cap E$ and another on $V \setminus E$.

7 Experimental Comparison

The aim of this section is to experimentally evaluate LDSB's performance in terms of speed by comparing the results obtained when using LDSB, against those obtained when using either other symmetry breaking implementations or no symmetry breaking at all. To achieve this we decided to carry out comparisons in two different systems: ECL^{i}PS^e and Gecode. This is because while the former allows us to compare LDSB against a broad number of systems, (including GAP-SBDS and GAP-SBDD, which are only available in ECL^{i}PS^e), the latter allows us to perform a fairer comparison with static symmetry breaking systems for which fast constraint propagation is critical.

The experiments on the traditional benchmarks were run on a quad-core Intel Core i7-920 2.67GHz computer with a limit of 2GB of memory, using Linux kernel version 2.6.32. The other experiments were run on a cluster of quad-core Intel Xeon E5310 1.66GHz CPUs with a limit of 1GB of memory, using Linux kernel version 2.6.18.

7.1 Experimental Comparison using ECL^{i}PS^e

Let us first report on the results obtained when solving a set of satisfaction and a set of optimisation benchmarks using the IC constraint solver in ECL^{i}PS^e version 6.0 #188, using no symmetry breaking (**none** in the tables), GAP-SBDS (**GS**), GAP-SBDD (**GD**), a generic and incomplete form of SBDS (**IS**), structural symmetry breaking (**SSB**), a generic form of static symmetry breaking (**Stat**), a version of LDSB with symmetry composition disabled (**LNC**), and a version of LDSB with symmetry composition enabled (**LDSB**). Note that we have only included methods that are automatic, that is, they only require the user to give them the problem and its symmetries. This means we have left systems that might be faster than those above. For example, while SBDD can be used more efficiently without GAP, to do so the user must provide a problem-specific dominance check.

GS, **GD** and **IS** are the GAP-SBDS, GAP-SBDD and SBDS implementations, respectively, included in ECL^{i}PS^e. The first two are given as input all symmetries of the problems and break them all using of the GAP system (version 4.4.12). The latter receives as input symmetries functions derived from the symmetry pattern instances given to LDSB as follows. Since having one function for each symmetry in the group would result into far too many functions [21], we converted each set of interchangeable items (whether variables, values or sequences of same) into a set of symmetry functions, one per transposition of items in the set. To keep to a practical number of symmetry functions, we also decided not to compose symmetries.

Stat has been implemented by us and automatically generates static symmetry breaking constraints from the input symmetries as follows. Interchangeable variable symmetries are broken by ordering the variables with $\leq$ constraints; interchangeable value symmetries are broken by value precedence constraints [16]; and interchangeable variable sequence symmetries are broken by lexicographical $\leq$ constraints. (In the case of two-dimensional matrices with interchangeable rows and columns, these constraints are equivalent to double-lex.) For the benchmarks that use set variables, **Stat** channels the set variables into boolean variables and posts the symmetry breaking constraints on those booleans. None of the generated constraints are simplified by considering the constraints of the problem. The symmetries provided to **Stat** as input are the same as for LDSB, except for instances of the interchangeable value sequence pattern, since these cannot be broken by **Stat**.

Since no implementation of SSB is available, we have implemented **SSB** in ECL^{i}PS^e as a dynamic form of SSB that follows the description in [7], i.e., it performs ancestor- and sibling-based filtering, rather than plain dominance checks. The symmetries provided to **SSB** as input are all the symmetries it can break, i.e., all pairwise variable and value interchangeable symmetries (which in LDSB are represented by the instances of the variable and value interchangeable patterns). Note that these include all the symmetries in the optimisation problems, but only a few of those in the traditional problems (only the Latin Square and $n \times n$-queens problems have interchangeable values, and no problem has interchangeable variables since we did not attempt to apply this method to the benchmarks that use set variables).

The models for all benchmarks (discussed below) will be available on the *Constraints* journal's editor's page (currently `http://www.crt.umontreal.ca/~pesant/Constraints/constraints.html`). Each benchmark instance was solved using both a naive variable order and the first-fail heuristic, which assigns the variable with the smallest domain size first. For set variables, the "domain size" is the number of values whose membership in the set is not yet determined.

7.1.1 Results for Traditional Symmetric Problems

The first set of benchmarks includes constraint satisfaction and optimisation problems often used for symmetry detection/breaking. We will call this set the "traditional" benchmarks and it includes the following problems:

- **bibd**(v,k,λ): the balanced incomplete block design problem, where v objects are to be arranged into $b = \frac{v(v-1)\lambda}{k(k-1)}$ blocks of exactly k objects such that each object is in exactly $r = \frac{\lambda(v-1)}{k-1}$ blocks, and every two objects occur together in exactly λ blocks. Our model has a boolean variable x_{ij} for each pair of object i and block j, indicating whether object i is in block j. Both the objects and the blocks are interchangeable, corresponding to two instances of the interchangeable variable sequences pattern. This results in $v!b!$ symmetries, all of which are represented in LDSB. The **Stat** method posts lex constraints on the rows and columns of the matrix of boolean variables. The **SSB** method cannot handle the symmetries in this problem.
- **golf**(w,g,p): the social golfers problem, where gp golfers are to be arranged each week into g groups of p players over w weeks, with no two players appearing in the same group twice. Our model has a set variable for each group in each week, whose value is the set of players in that group. The weeks are interchangeable, the players are too and, within each week, the groups are interchangeable giving $w!(gp)!(g!)^w$ symmetries, all in LDSB. The **Stat** method channels the set variables to boolean variables, and posts lex constraints to order the weeks, lex constraints to order the players, and separate lex constraints for each week to order the groups in that each week.
- **graceful**(m,n): the graceful graph labeling problem, where the vertices V of a graph (V,E) are to be labelled with distinct integers from 0 to $|E|$ such that the derived labels on the edges are all different. The label on an edge (a,b) is derived as the absolute difference of the labels on vertices a and b. We use as input graphs of the form $K_m \times P_n$. Our model has an integer variable for each vertex in the graph, and an auxiliary integer variable for each edge; we search only on the vertex variables. The corresponding vertices in each clique are sequence interchangeable, the order of the cliques is reversible (represented via sequence interchangeable values), and the values are too, giving $4m!$ symmetries, all in LDSB. The **Stat** method posts lex constraints to order the corresponding n elements within the cliques, and a lex constraint to order the first clique before the last clique. The **SSB** method cannot handle the symmetries in this problem.
- **latin**(n): The Latin square problem as described previously, with $2(n!)^3$ of the $6(n!)^3$ symmetries represented in LDSB via instances of the value interchangeability pattern for all values, and of the interchangeable variable sequences for both rows and columns. The interchanges between the values and the row/column dimensions cannot be represented by LDSB, because they are variable-value symmetries. The **Stat** method posts lex constraints on the rows and columns of the matrix of boolean variables, and orders the values using precedence constraints. The **SSB** method cannot handle the symmetries in this problem.
- **magicsquare**(n): the $n \times n$ magic square problem, where every integer from 1 to n^2 is placed on an $n \times n$ square such that all rows, columns and the two diagonals have the same sum. Each cell of the square is represented by an integer variable. We use the horizontal, vertical and diagonal reflections of the square and the reversible value symmetries (an instance of the interchangeable value sequences pattern), all of which are representable in LDSB. The **Stat** method posts one lex constraint for each of the

horizontal, the vertical and the diagonal reflections. The **SSB** method cannot handle the symmetries in this problem.

– **nn_queens** (n): the $n \times n$-queens problem, where an $n \times n$ chessboard is coloured with n colours, so that a pair of queens placed in any two squares of the same colour would not attack each other. The symmetries are the horizontal, vertical and diagonal reflections of the square, plus the colours are interchangeable, giving $8n!$ symmetries, all in LDSB. The **Stat** method posts a lex constraint to break the horizontal reflections, and orders the values using precedence constraints. The **SSB** method can handle only the interchangeable values.
  We also consider an optimisation version of this problem, where the task is to find the smallest number of colours required to satisfy the problem constraints.
– **queens**(n): the n-queens problem, where n queens are placed on a chessboard such that no two queens attack each other. Our model has one variable q_i per column, whose value indicates the row of the queen in that column. The symmetries are the horizontal, vertical and diagonal reflections of the chessboard (8 symmetries in total), with only the horizontal and vertical reflections (4 symmetries) represented in LDSB. The interchanging of the values and variables cannot be represented in LDSB. The **Stat** method posts a lex constraint to break the reflection on the variables. The **SSB** method cannot handle the symmetries in this problem.
– **queens_bool**(n): same as above but with a Boolean matrix model. All 8 symmetries are in LDSB. As for the **magicsquare** benchmark, the **Stat** method posts one lex constraint for each of the horizontal reflection, the vertical reflection and the diagonal reflection. The **SSB** method cannot handle the symmetries in this problem.
– **steel**(n): the Steel Mill slab design problem, which is to allocate orders for steel blocks to slabs. The objective is to minimize the wasted space on the slabs. The value n indicates that the first n orders from the data set in CSPLib [13] are used. The slabs (values) are interchangeable. The **Stat** method orders the values using precedence constraints.
– **steiner**(n): the Steiner triple problem, which is to find $\frac{n(n-1)}{6}$ triples of distinct integers from 1 to n, with any two triples having at most one element in common. We have one set variable per triple. The triples (variables) are interchangeable, as are the values, giving $n!\frac{n(n-1)}{6}!$ symmetries, all in LDSB. As for the social golfers, the **Stat** method channels the set variables to boolean variables, and posts lex constraints to order the triples and lex constraints to order the values in the triples.

The experimental results obtained in ECL^{i}PS^e for the traditional benchmarks are shown in Tables 1, 2, 3 and 4. For each benchmark and method, the tables show the time taken in seconds to find the first/all solutions using the first-fail and input-order variable ordering heuristics. An entry with symbol "-" indicates that the search timed out at the 600 second limit, "M" indicates that memory was exhausted, and "NA" indicates we did not apply the method to the given problem. For each category, the best result and those within 10% (or 0.1 seconds) of the best are shown in bold. The instances differ between tables because for some combinations of search strategy and problem, no method was able to finish within the time limit. Note that the main reason to show the results for all solutions is tradition, since one rarely needs to compute all solutions of a satisfaction problem. While the results do give some idea of how the method would perform when having to traverse the complete search space, this can be better evaluated by using optimisation problems, as we do in more depth in the next section.

The results for traditional benchmarks show that although in some cases no symmetry breaking is the best method (to find the first solution), LDSB is at worst 25% slower (or

	none	GS	GD	IS	Stat.	SSB	LNC	LDSB
bibd 7,3,5	-	M	-	2.0	**0.4**	-	0.7	0.9
bibd 7,3,6	-	M	-	4.6	**0.5**	-	1.4	1.6
golf 5,5,4	-	M	-	28.8	131.6	NA	**16.5**	**17.2**
golf 5,6,4	120.0	M	-	22.0	-	NA	**12.8**	**12.8**
graceful 3,4	149.6	204.1	227.6	**59.1**	135.2	-	**61.4**	**60.4**
graceful 3,5	-	-	-	-	-	-	-	-
graceful 5,2	-	333.2	338.4	**121.6**	164.2	-	**115.5**	**119.4**
latin 40	**7.3**	-	-	14.0	M	438.4	**7.8**	**7.7**
magicsquare 5	**0.3**	1.0	1.2	**0.3**	**0.3**	2.0	**0.3**	**0.3**
magicsquare 6	**2.6**	100.9	143.3	3.0	4.0	-	3.0	3.0
nn_queens 7	**0.3**	0.5	0.4	**0.3**	0.4	**0.3**	**0.3**	**0.3**
nn_queens 8	-	112.2	131.6	**7.0**	202.6	643.4	**6.9**	**7.2**
queens 150	**30.4**	-	-	**32.2**	**31.4**	-	**33.3**	**32.1**
queens 200	**19.3**	-	-	**20.5**	**20.1**	-	**20.7**	**20.7**
queens 300	**0.6**	15.7	-	0.7	**0.6**	3.4	**0.6**	**0.6**
queens_bool 26	**1.4**	55.5	135.5	4.1	2.2	-	1.5	**1.5**
queens_bool 28	**8.8**	496.2	-	29.7	15.5	-	**9.5**	**9.5**
queens_bool 30	**172.4**	-	-	599.2	302.1	-	193.2	198.8
steiner 9	0.9	3.6	1.9	0.4	0.9	NA	**0.4**	**0.4**
steiner 15	0.4	-	-	0.7	-	NA	**0.3**	**0.3**

Table 1: Traditional Benchmark Results, first-fail variable ordering, first solution. 600 second timeout.

	none	GS	GD	IS	Stat.	SSB	LNC	LDSB
bibd 7,3,5	-	M	-	336.6	**117.3**	-	137.0	159.6
bibd 7,3,6	-	M	-	-	-	-	-	-
graceful 4,2	215.4	7.1	8.0	9.4	9.7	-	8.6	**2.7**
latin 6	-	48.4	**3.5**	4.5	4.7	-	4.7	4.9
magicsquare 3	**0.3**	0.4	0.4	**0.3**	**0.3**	**0.3**	**0.3**	**0.3**
magicsquare 4	8.2	13.5	27.3	1.4	2.4	-	1.3	**0.9**
nn_queens 7	147.0	0.9	1.0	**0.3**	1.0	1.8	**0.3**	**0.3**
nn_queens 8	-	112.4	133.1	**7.1**	203.0	648.2	**6.9**	**7.0**
queens 13	10.3	43.2	68.9	6.9	7.4	-	6.3	**4.5**
queens 14	56.4	243.7	405.8	35.3	40.4	-	32.1	**23.8**
queens 15	337.0	-	-	211.9	240.3	-	197.3	**138.2**
queens_bool 13	28.7	168.0	377.5	**16.3**	23.0	-	27.7	28.2
queens_bool 14	167.4	-	-	**87.8**	130.4	-	156.0	167.2
queens_bool 15	-	-	-	**517.2**	-	-	-	-
steiner 9	-	11.7	6.0	12.1	**3.0**	NA	8.9	8.9
nn_queens 7	**0.3**	1.5	1.0	**0.3**	0.9	0.7	**0.3**	**0.3**
nn_queens 8	-	-	264.8	51.9	413.1	-	**7.5**	**7.7**
steel 36	-	-	20.3	11.8	**1.3**	199.9	1.6	1.7
steel 37	**0.4**	-	1.8	0.6	1.5	1.0	**0.4**	**0.4**
steel 38	-	-	21.5	12.6	**1.5**	226.9	**1.6**	1.7

Table 2: Traditional Benchmark Results, first-fail variable ordering. Time to find all solutions for satisfaction problems and prove optimality for optimisation problems (listed in italics). 600 second timeout.

within one second) and in many cases is faster, while not using symmetry breaking can lead to very bad performance. For most problems, **SSB** imposes an overhead that is not repaid by its reduction in search. When compared to the complete symmetry breaking methods, LDSB is uniformly faster than GAP-SBDS (and often orders of magnitude faster) due to its lower overhead. The same happens for GAP-SBDD with the exception of three cases. LDSB also has a low memory overhead compared to the GAP-based methods (data not shown) which have to deal with large data structures when the symmetry group is large. LDSB performs worst when eliminated symmetries become reactivated (**bibd** and all solutions for **queens_bool**). For **bibd** the static symmetry breaking performs best, and for **queens_bool** the incomplete SBDS method or no symmetry breaking perform best. It is clear from the

	none	GS	GD	IS	Stat.	SSB	LNC	LDSB
bibd 7,3,5	-	M	-	2.0	**0.4**	-	0.7	0.8
bibd 7,3,6	-	M	-	4.5	**0.5**	-	1.2	1.6
golf 5,5,4	-	M	-	29.3	129.7	NA	**16.3**	**16.4**
golf 5,6,4	119.5	M	-	21.8	-	NA	**12.5**	**12.8**
graceful 3,4	**4.8**	15.9	18.4	**4.6**	**4.8**	30.1	**4.7**	**4.7**
graceful 3,5	290.8	-	-	**235.3**	**230.7**	-	**231.4**	**243.5**
graceful 5,2	-	-	-	**580.2**	**564.0**	-	**561.4**	**552.7**
latin 40	**6.5**	-	34.5	13.1	M	11.5	**7.1**	**6.8**
magicsquare 5	**0.4**	4.8	8.0	**0.5**	0.5	31.8	**0.5**	**0.5**
magicsquare 6	-	-	-	-	-	-	-	-
nn_queens 7	**0.3**	0.5	0.5	**0.3**	0.4	**0.3**	**0.3**	**0.3**
nn_queens 8	-	169.6	188.5	**6.9**	253.0	-	**7.0**	**7.0**
queens 26	**1.9**	49.0	60.4	2.2	2.2	341.8	2.3	2.4
queens 28	**13.8**	411.2	502.3	16.2	15.7	-	16.8	17.3
queens 30	**255.1**	-	-	302.7	289.8	-	309.8	318.2
queens_bool 26	**0.8**	55.0	135.0	3.6	1.7	-	1.0	1.0
queens_bool 28	**4.6**	491.6	-	25.1	11.0	-	5.2	5.3
queens_bool 30	**85.9**	-	-	507.6	212.0	-	97.6	100.0
steiner 9	0.9	3.5	1.9	0.4	0.9	NA	**0.4**	**0.4**
steiner 15	0.4	-	-	0.7	-	NA	**0.3**	**0.3**

Table 3: Traditional Benchmark Results, input-order variable ordering, first solution. 600 second timeout.

	none	GS	GD	IS	Stat.	SSB	LNC	LDSB
bibd 7,3,5	-	M	-	333.4	**111.8**	-	129.6	160.8
bibd 7,3,6	-	M	-	-	-	-	-	-
graceful 4,2	206.2	15.0	16.7	17.7	16.8	-	16.4	**5.2**
latin 6	-	56.2	6.3	**4.5**	**4.8**	-	**4.9**	**5.0**
magicsquare 3	**0.3**	0.4	0.4	**0.3**	**0.3**	0.3	**0.3**	**0.3**
magicsquare 4	10.4	27.4	49.8	2.8	3.5	-	2.6	**1.4**
nn_queens 7	132.0	1.2	1.2	**0.3**	1.2	2.0	**0.3**	**0.3**
nn_queens 8	-	167.8	190.7	**6.9**	255.3	-	**6.8**	**7.0**
queens 13	10.8	65.5	105.0	8.1	8.6	-	7.2	**6.0**
queens 14	62.9	388.2	658.5	45.6	48.5	-	41.0	**34.8**
queens 15	376.5	-	-	273.4	290.6	-	245.4	**202.2**
queens_bool 13	18.5	159.8	374.1	**11.8**	18.4	-	18.7	19.2
queens_bool 14	101.3	-	-	**63.2**	105.0	-	106.9	109.5
queens_bool 15	614.3	-	-	**369.6**	609.5	-	639.8	656.0
steiner 9	-	11.8	6.1	12.0	**3.1**	NA	8.7	8.7
nn_queens 7	**0.3**	3.6	1.6	0.4	0.9	2.5	**0.3**	**0.3**
nn_queens 8	-	-	-	199.2	-	-	**23.6**	**24.5**
steel 36	344.2	35.4	4.7	1.8	1.3	13.6	**0.6**	**0.6**
steel 37	-	-	5.0	1.6	1.5	10.2	**0.6**	**0.6**
steel 38	402.2	-	5.2	2.0	1.5	15.6	**0.6**	**0.6**

Table 4: Traditional Benchmark Results, input-order variable ordering. Time to find all solutions for satisfaction problems and prove optimality for optimisation problems (listed in italics). 600 second timeout.

results that LDSB is the most consistent performer and it often returns best or near best times.

The behaviour of the **Stat** method is interesting to note, especially on the **steiner** problem, which uses set variables. For this problem the static constraints are double-lex on the matrix of channeled boolean variables, which is known to perform well. However, **Stat** performs poorly on this problem because the static constraints disagree with the search order. The exception is in Table 4, where the variable order agrees with the constraints and, because the whole search tree is examined, the value order is irrelevant – in this case **Stat** is the clear winner.

Note that for **bibd**, the specialised version of the STAB method for 2D matrix problems with row and column symmetry, was reported to be 15 times faster than static symmetry breaking for finding all solutions to BIBD problems [25]. We have tried LDSB on the same problems as in [25], and found that it is approximately 2 times slower than static symmetry breaking, making STAB clearly the best method for this problem. However, the evaluated method is specialised for this kind of problems and it is therefore not clear whether it would perform well on others.

Prestwich et al. [22] give experimental results to show that SBNO can break more symmetry than STAB on the **bibd** problem, but they do not attempt to compare running times. SBNO also performs well on the **steiner** and **golf** problems, but using an integer model instead of a set model.

7.1.2 Results for optimisation problems

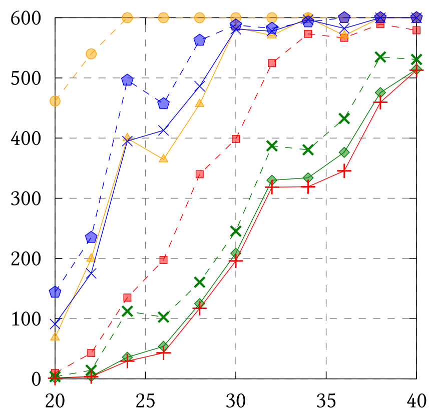
The second set of benchmarks, referred to as the “optimisation” benchmarks, are instances randomly generated (following [17]) for the following two optimisation problems.

- **Graph Colouring**: where the task is to find the chromatic number of a graph. The instances are randomly generated (see [17]) to have partitions of symmetric variables and values: n nodes are partitioned so that each partition (of maximum size 8) is either an independent set or a complete graph, and each subgraph with nodes in two partitions is either an independent or a complete bipartite graph. Instances are divided into classes according to (1) how the partition sizes are distributed (“uniform” or “biased”), (2) the number of nodes, and (3) the proportion q of partitions that are complete subgraphs. There are 20 instances of each class. Colours are interchangeable (giving $n!$), as are nodes within each partition. Both symmetry groups are given to LDSB as instances of the interchangeable values and the interchangeable variables patterns, respectively. The **Stat** method uses $\leq$ constraints on the sets of interchangeable variables and precedence constraints on the values. The **SSB** method can handle all of the symmetries in this problem.
- **Concert Hall Scheduling**: choose among n applications specifying a period and an offered price to use k identical concert halls, to maximise profit. The instances are randomly generated following [17]: k is set to 8 and applications are generated in partitions of uniformly distributed size, with all applications in a partition being identical. There are 20 instances for each size. Concert halls are interchangeable ($k!$ symmetries) and applications within each partition are too. Again, both symmetry groups are given to LDSB as instances of the interchangeable values and the interchangeable variables patterns, respectively. The **Stat** method uses $\leq$ constraints on the sets of interchangeable variables and precedence constraints on the values. The **SSB** method can handle all of the symmetries in this problem.

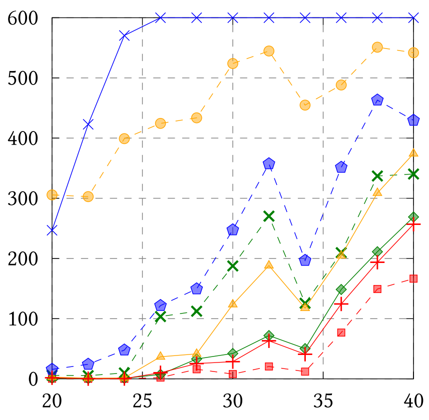
Note that these benchmarks serve as a worst-case comparison for LDSB, since they are best suited to SSB and static methods (the problems inherently have only piecewise value interchangeability, and the instances are constructed to have piecewise variable interchangeability).

Figures 7a to 7c show a representative subset of the results for the concert hall optimisation and the graph colouring instances.[2] Note that we run both benchmarks with the first-fail and the input-order variable ordering heuristics. We only show three representatives, because the behaviour of the methods is similar across the heuristics. Each data point in the figures

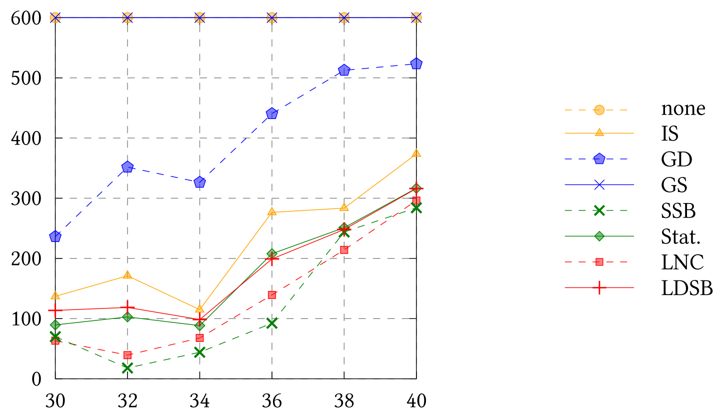
[2] Appendix D reproduces these figures in a larger size, and includes the number of instances solved for each point.



(a) Concert hall scheduling, first-fail.



(b) Graph colouring, biased distribution, $q = 0.75$, input order.



(c) Graph colouring, uniform distribution, $q = 1$, input order.

Fig. 7: Dynamic symmetry breaking results.

represents 20 instances, the horizontal axis shows instance size (offers or nodes), and the vertical axis shows the mean time in seconds taken to solve the set of instances to optimality (making a total of 220 out of the complete 440 concert hall instances in Figure 7a and 340 of the 2520 graph colouring instances in Figures 7b and 7c). Any instance that did not solve within the time-limit of ten minutes is considered to have taken ten minutes for the purpose of calculating the mean.

The three plots shown are representative of the complete results. The **none**, **GS** and **GD** methods perform poorly; in particular, **GS** often exhausts memory and **none** often times out. **LDSB**, **Stat** and **SSB** typically perform the best, but none of them is the clear winner. **LNC** also performs very well, but is more inconsistent than the others.

Overall, out of the combined total of 2960 optimisation instances, **Stat** solved 2629, **LDSB** solved 2625, **LNC** solved 2584, **SSB** solved 2480, **IS** solved 2312, **GD** solved 1951, **none** solved 986 and **GS** solved 299. This shows (a) that having composition is advantageous (41 instances out of 2625), and (b) that there is a significant difference between the two best (**Stat** and **LDSB**, which are only 4 out of 2629 instances apart) and the rest (145 instances of difference with the closest competitor **SSB**).

7.2 Experimental Comparison using Gecode

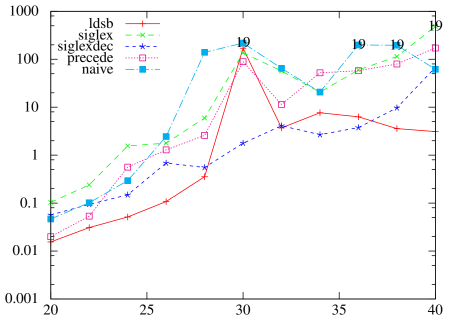
We have tested in Gecode the satisfaction problems used in the ECL^{i}PS^e experiments, to compare LDSB and the automatically-generated static constraints under a faster constraint engine. The results are shown in Table 5. Note that the instances are different because Gecode is able to quickly solve larger problems than ECL^{i}PS^e.

	Input order		First fail	
	Stat.	LDSB	Stat.	LDSB
bibd 7,3,8	**0.0**	0.2	**0.0**	0.2
bibd 8,4,5	**1.5**	1.8	**1.5**	1.8
bibd 8,4,6	**0.1**	0.1	**0.1**	0.1
graceful 3,5	**4.7**	**5.0**	81.9	**17.0**
graceful 4,3	**9.7**	**10.4**	7.8	**4.5**
graceful 6,2	-	-	-	**101.1**
latin 40	**0.4**	2.6	10.8	**2.7**
latin 45	**0.8**	4.6	51.1	**4.9**
latin 50	**1.4**	7.9	465.2	**8.4**
magic_square 5	**0.0**	0.0	**0.0**	0.0
magic_square 6	-	-	**0.3**	0.4
nn_queens 8	**1.3**	**1.4**	**0.8**	**0.9**
nn_queens 9	-	-	**562.7**	**594.1**
queens 30	**24.9**	31.3	**0.0**	0.0
queens 600	-	-	**30.4**	35.5
queens 1000	-	-	**0.2**	**0.2**
queens_bool 30	**56.6**	64.5	**56.6**	64.5
queens_bool 31	**11.3**	12.8	**11.3**	12.8
queens_bool 32	**76.5**	87.3	**74.7**	87.1

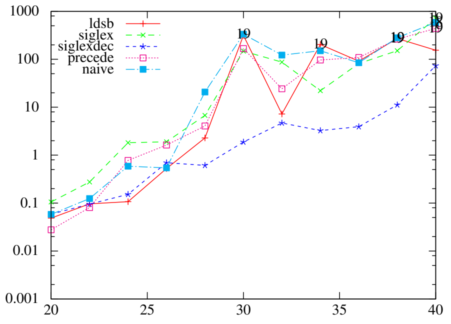
Table 5: Satisfaction Benchmark Results in Gecode, first solution. 600 second timeout.

We can observe that for the **bibd**, **magic_square**, **nn_queens**, **queens** and **queens_bool** problems **LDSB** is usually slower than **Stat**, but not significantly. For the **graceful** and **latin** problems, the results depend on the variable order. When the search order agrees with the static constraints, **LDSB** is several times slower than **Stat**, particularly on smaller instances where the difference can be up to an order of magnitude. When the variables are assigned in an unpredictable order (under the first-fail heuristic), **LDSB** is several times faster that **Stat**, particularly on larger instances where the difference can be one or two orders of magnitude.

As mentioned before, we have performed a second set of experiments that compares LDSB with the best-performing static symmetry breaking constraints, all implemented in Gecode. In particular, we compare with the value precedence (**precede**) constraint for breaking value interchangeability [16] and with the SigLex constraint (**siglex**) for breaking variable and value interchangeability [17]. We also test the descending-partition-size variation of SigLex (**siglexdec**). Although structural symmetry breaking combined with restarts has proved effective [15], we do not compare with this method because we wish to vary only the symmetry breaking method, rather than the fundamental search method (restarts). Finally, we compare with a semi-naive static symmetry breaking method (**naive**), where the symmetric variables are ordered, and the variables are channeled into a matrix of boolean variables, whose columns (corresponding to the integer values) are lex-ordered. We have recreated the experiments of Law et al. [17] which use the optimisation benchmarks described above.

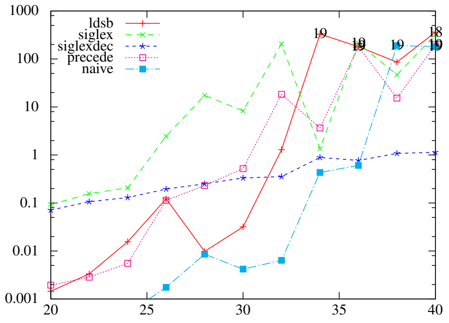


(a) First fail breaking ties by number of constraints.

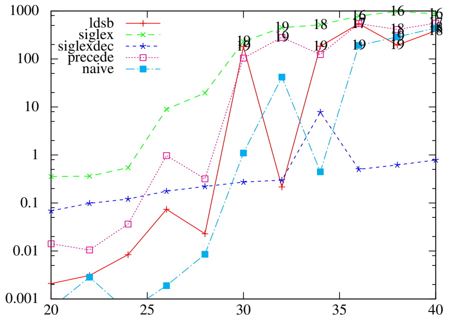


(b) First fail breaking ties by variable index.

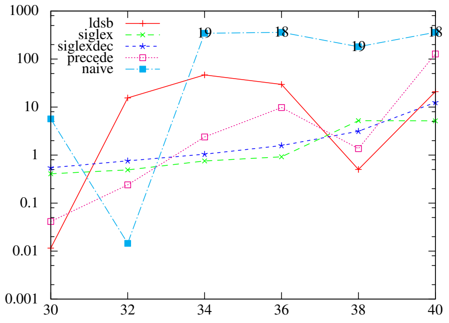
Fig. 8: Static symmetry breaking on concert hall scheduling.



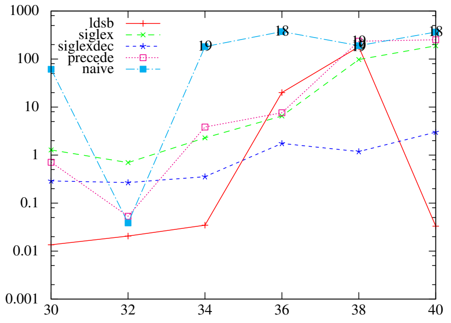
(a) Biased distribution, $q = 0.75$



(b) Biased distribution, $q = 1$.



(c) Uniform distribution, $q = 0.75$.



(d) uniform distribution, $q = 1$

Fig. 9: Static symmetry breaking on graph colouring.

Each benchmark instance was run with a time limit of one hour. Variables were chosen by minimum domain, and the smallest value was tried first.

Figures 8a–9d show the results from a representative selection of the experiments. Each data point represents 20 instances, the horizontal axis shows instance size, and the vertical axis shows the logarithmic mean time in seconds taken to solve the set of instances to optimality. Any instance that did not solve within the time-limit of one hour is considered to have taken one hour for the purpose of calculating the mean. The number plotted on each point is the number of instances solved (omitted if all 20).

Figures 8a and 8b show the results for concert hall scheduling and, in particular, the effect of altering the variable selection strategy. In both figures, the search chooses the variable with the smallest domain first, but ties are broken differently. The plots show that changing the tie-breaking order affects only LDSB. This is because the constraints added by the static methods cause the domains of many variables to be reduced before search, and so there are fewer ties to be broken at variable selection. The plots also show that for LDSB, breaking ties by selecting the variable involved in the most constraints improves performance. This is true for all of the experiments and, therefore, in the following figures only results for this tie-breaking method are shown.

Figures 9a and 9b show the results for the graph colouring problem with the biased distribution. This case is the best for **siglexdec**, which shows very consistent performance across the instance sizes. All other methods vary greatly, and LDSB is generally competitive with those. Finally, Figures 9c and 9d show results for the graph colouring problem with the uniform distribution. Here **siglex** and **siglexdec** are the most consistent performers, while LDSB is again competitive with the other methods.

Overall, out of the combined total of 2960 optimisation instances, **siglexdec** solved 2954, **precede** solved 2933, **siglex** solved 2920, **naive** solved 2907 and **LDSB** solved 2905. Clearly, **siglexdec** is the most consistent performer in these experiments, and is especially impressive in the case of the biased-distribution graph colouring problems, to which it is most suited. Interestingly, **LDSB** is competitive with the static methods (it solves only 49 instances less than the best method out of 2954), despite its generality. However, it is not clear how much of the difference between **LDSB** and the static methods is due to symmetry breaking's effect on the variable selection strategy, rather than to the symmetry breaking method itself; it is possible that the propagation caused by the constraints of the static methods accidentally leads the search towards, or away from, the optimal solution.

8 Conclusion

This paper has presented Lightweight Dynamic Symmetry Breaking (LDSB), an automatic symmetry breaking method that is efficient enough to be used as a default, since it never yields a major slowdown.

To achieve this LDSB uses an internal representation based on four symmetry patterns. While these patterns can represent all variable and value symmetries, LDSB is most effective for those symmetries for which these patterns are very compact. Importantly, LDSB does not guarantee to break all the symmetries in the group which can be generated from the symmetries explicitly represented. This allows LDSB to define very efficient breaking algorithms that are easy to implement and are also complete for some of the most common symmetry patterns. This mix of compactness, efficiency and high degree of completeness, makes LDSB highly effective.

This paper experimentally investigates the performance of LDSB's symmetry breaking against a very wide range of alternative approaches. The results strongly support the claim that LDSB is an extremely good default symmetry-breaking method, even compared with more complete or more specialised approaches.

– As compared with no symmetry breaking, on problem instances where symmetry breaking does not bring any useful pruning of the search tree, LDSB incurs a very low over-

head. For example in finding first solutions to large Latin Square problems, which have many symmetries which do not prune the search for the first solution, the overhead is less than 5%.
– Its consistency in all of the benchmarks indicates that a programmer can use LDSB by default, without worrying that its running time might be inordinately high as can be the case with some complete methods.
– As compared with the most powerful dynamic symmetry-breaking methods (GAP-SBDS and GAP-SBDD), LDSB is typically faster and, for hard problems, more scalable. Out of some 80 problem instances GAP-SBDD is faster only twice, in finding all solutions to the Steiner problem, where LDSB's incompleteness is maximised, and even on this problem LDSB is faster than GAP-SBDS.
– Our implementation of the partial symmetry-breaking method (SSB) that breaks piecewise interchangeable variable and value symmetries, is only able to solve about 10% of the problem instances, and is never faster than LDSB.
– A generic form of static symmetry breaking (implemented by us) is only able to solve slightly more than half the problem instances. On these benchmarks it is generally much slower than LDSB, and its best result is only 2% faster.
– LDSB is compared against the fastest static symmetry-breaking methods implemented on problem instances specially designed to reveal their optimal performance.
  – For concert hall scheduling, LDSB arguably outperforms all the static methods except **siglexdec**.
  – For a class of graph colouring problems, artificially constructed to have partitions which are either independent sets of complete graphs, LDSB performs similarly to three out of four models having special symmetry-breaking constraints, but is outperformed by a model with a tailored global constraint known as *siglexdec*.
– LDSB is implemented, like any dynamic symmetry breaking technique, in a way that can take advantage of any (static or dynamic) search order. The benefit of this flexibility is illustrated in the concert hall scheduling example of Figures 8a and 8b.

In short, LDSB is easy to use, the overhead induced by its symmetry breaking search is low, and it is almost always repaid handsomely by the savings gained. However, LDSB cannot always outperform specialised static symmetry breaking constraints when conditions suit the latter (i.e., a small set of effective constraints can be found that do not conflict with the search order). Further, note we did not compare LDSB with non-GAP implementations of SBDS and SBDD which are typically faster but require the user to provide extra information.

The results show that LDSB has achieved the following goals:

– It is automatic and does not require the user to provide long or complex input, such as exhaustive lists of symmetries, symmetry-breaking constraints or dominance checking functions
– It breaks commonly occurring symmetries efficiently
– It reduces the time taken to solve problems in almost all cases
– It avoids the runtime overhead and implementation complexity of computational group computations
– It is available in the widely used constraint programming systems ECL^{i}PS^e and Gecode, and is easy to add to other solvers such as Lazy-FD.

Acknowledgements

We wish to acknowledge Monash University for the use of their Nimrod software in this work. The Nimrod project has been funded by the Australian Research Council and a number of Australian Government agencies, and was initially developed by the Distributed Systems Technology CRC. This research was partly sponsored by the Australian Research Council grant DP110102258, and by the Research Foundation Flanders (FWO) through projects *WOG: Declarative Methods in Computer Science* and *G.0221.07*.

References

1. Rolf Backofen and Sebastian Will. Excluding symmetries in constraint-based search. In Joxan Jaffar, editor, *CP*, volume 1713 of *Lecture Notes in Computer Science*, pages 73–87. Springer, 1999.
2. David A. Cohen, Peter Jeavons, Christopher Jefferson, Karen E. Petrie, and Barbara M. Smith. Symmetry definitions for constraint satisfaction problems. *Constraints*, 11(2-3):115–137, 2006.
3. James M. Crawford, Matthew L. Ginsberg, Eugene M. Luks, and Amitabha Roy. Symmetry-breaking predicates for search problems. In Luigia Carlucci Aiello, Jon Doyle, and Stuart C. Shapiro, editors, *KR*, pages 148–159. Morgan Kaufmann, 1996.
4. Torsten Fahle, Stefan Schamberger, and Meinolf Sellmann. Symmetry breaking. In Walsh [33], pages 93–107.
5. Pierre Flener, Alan M. Frisch, Brahim Hnich, Zeynep Kiziltan, Ian Miguel, Justin Pearson, and Toby Walsh. Breaking row and column symmetries in matrix models. In Van Hentenryck [29], pages 462–476.
6. Pierre Flener, Justin Pearson, and Meinolf Sellmann. Static and dynamic structural symmetry breaking. *Ann. Math. Artif. Intell.*, 57(1):37–57, 2009.
7. Pierre Flener, Justin Pearson, Meinolf Sellmann, Pascal Van Hentenryck, and Magnus Ågren. Dynamic structural symmetry breaking for constraint satisfaction problems. *Constraints*, 14(4):506–538, 2009.
8. Filippo Focacci and Michela Milano. Global cut framework for removing symmetries. In Walsh [33], pages 77–92.
9. Gecode Team. Gecode: Generic constraint development environment, 2006. Available from `http://www.gecode.org`.
10. Ian P. Gent, Warwick Harvey, and Tom Kelsey. Groups and constraints: Symmetry breaking during search. In Van Hentenryck [29], pages 415–430.
11. Ian P. Gent, Warwick Harvey, Tom Kelsey, and Steve Linton. Generic SBDD using computational group theory. In Rossi [27], pages 333–347.
12. Ian P. Gent and Barbara M. Smith. Symmetry breaking in constraint programming. In Werner Horn, editor, *ECAI*, pages 599–603. IOS Press, 2000.
13. I.P. Gent and T. Walsh. CSPLib: a benchmark library for constraints. Technical report, Technical report APES-09-1999, 1999. Available from http://csplib.cs.strath.ac.uk/.
14. Carla P. Gomes, Bart Selman, Nuno Crato, and Henry A. Kautz. Heavy-tailed phenomena in satisfiability and constraint satisfaction problems. *J. Autom. Reasoning*, 24(1/2):67–100, 2000.
15. Daniel S. Heller, Aurojit Panda, Meinolf Sellmann, and Justin Yip. Model restarts for structural symmetry breaking. In Peter J. Stuckey, editor, *CP*, volume 5202 of *Lecture Notes in Computer Science*, pages 539–544. Springer, 2008.
16. Y. Law and J. Lee. Symmetry breaking constraints for value symmetries in constraint satisfaction. *Constraints*, 11(2-3):221–267, 2006.
17. Y. C. Law, J. H. M. Lee, Toby Walsh, and J. Y. K. Yip. Breaking symmetry of interchangeable variables and values. In Christian Bessiere, editor, *CP*, volume 4741 of *Lecture Notes in Computer Science*, pages 423–437. Springer, 2007.
18. Iain McDonald and Barbara M. Smith. Partial symmetry breaking. In Van Hentenryck [29], pages 431–445.
19. Christopher Mears, Maria Garcia de la Banda, Mark Wallace, and Bart Demoen. A novel approach for detecting symmetries in CSP models. In Laurent Perron and Michael A. Trick, editors, *CPAIOR*, volume 5015 of *Lecture Notes in Computer Science*, pages 158–172. Springer, 2008.
20. Christopher Mears, Maria Garcia de la Banda, Bart Demoen, and Mark Wallace. Lightweight dynamic symmetry breaking. In *SymCon'08: The Eighth International Workshop on Symmetry in Constraint Satisfaction Problems*, 2008.

21. Karen E. Petrie and Barbara M. Smith. Symmetry breaking in graceful graphs. In Rossi [27], pages 930–934.
22. Steve D. Prestwich, Brahim Hnich, Helmut Simonis, Roberto Rossi, and S. Armagan Tarim. Partial symmetry breaking by local search in the group. *Constraints*, 17:148–171, 2012.
23. Jean-Francois Puget. On the satisfiability of symmetrical constrained satisfaction problems. In *Proceedings of the 7th International Symposium on Methodologies for Intelligent Systems*, ISMIS '93, pages 350–361, London, UK, UK, 1993. Springer-Verlag.
24. Jean-Francois Puget. Symmetry breaking revisited. In Van Hentenryck [29], pages 446–461.
25. Jean-Francois Puget. Symmetry breaking using stabilizers. In Rossi [27], pages 585–599.
26. Colva M. Roney-Dougal, Ian P. Gent, Tom Kelsey, and Steve Linton. Tractable symmetry breaking using restricted search trees. In Ramon López de Mántaras and Lorenza Saitta, editors, *ECAI*, pages 211–215. IOS Press, 2004.
27. Francesca Rossi, editor. *Principles and Practice of Constraint Programming - CP 2003, 9th International Conference, CP 2003, Kinsale, Ireland, September 29 - October 3, 2003, Proceedings*, volume 2833 of *Lecture Notes in Computer Science*. Springer, 2003.
28. The GAP Group. *GAP – Groups, Algorithms, and Programming, Version 4.4.9*, 2006.
29. Pascal Van Hentenryck, editor. *Principles and Practice of Constraint Programming - CP 2002, 8th International Conference, CP 2002, Ithaca, NY, USA, September 9-13, 2002, Proceedings*, volume 2470 of *Lecture Notes in Computer Science*. Springer, 2002.
30. Pascal Van Hentenryck, Pierre Flener, Justin Pearson, and Magnus Ågren. Tractable symmetry breaking for CSPs with interchangeable values. In Georg Gottlob and Toby Walsh, editors, *IJCAI*, pages 277–284. Morgan Kaufmann, 2003.
31. Pascal Van Hentenryck, Pierre Flener, Justin Pearson, and Magnus Ågren. Compositional derivation of symmetries for constraint satisfaction. In Jean-Daniel Zucker and Lorenza Saitta, editors, *SARA*, volume 3607 of *Lecture Notes in Computer Science*, pages 234–247. Springer, 2005.
32. M. G. Wallace, S. Novello, and J. Schimpf. ECLiPSe : A platform for constraint logic programming. *ICL Systems Journal*, 12(1), 1997.
33. Toby Walsh, editor. *Principles and Practice of Constraint Programming - CP 2001, 7th International Conference, CP 2001, Paphos, Cyprus, November 26 - December 1, 2001, Proceedings*, volume 2239 of *Lecture Notes in Computer Science*. Springer, 2001.

A Supplementary Table

Problem	Aggregated	Pairwise
bibd [7, 35, 15, 3, 5] (first_fail) (all)	261.8	242.1
bibd [7, 35, 15, 3, 5] (first_fail) (first)	1.0	1.4
bibd [7, 35, 15, 3, 5] (input_order) (all)	250.4	230.7
bibd [7, 35, 15, 3, 5] (input_order) (first)	0.9	1.3
bibd [7, 42, 18, 3, 6] (first_fail) (first)	2.3	3.0
bibd [7, 42, 18, 3, 6] (input_order) (first)	2.2	2.9
golf [5, 5, 4] (first_fail) (first)	29.0	29.4
golf [5, 5, 4] (input_order) (first)	28.9	29.2
golf [5, 6, 4] (first_fail) (first)	21.6	22.0
golf [5, 6, 4] (input_order) (first)	21.4	21.4
graceful [3, 4] (first_fail) (first)	90.7	94.9
graceful [3, 4] (input_order) (first)	6.7	6.7
graceful [3, 5] (input_order) (first)	372.5	364.7
graceful [4, 2] (first_fail) (all)	3.6	3.7
graceful [4, 2] (input_order) (all)	7.4	7.5
graceful [5, 2] (first_fail) (first)	178.7	182.7
latin 6 (first_fail) (all)	6.7	6.9
latin 6 (input_order) (all)	6.8	6.9
latin 40 (first_fail) (first)	11.6	29.6
latin 40 (input_order) (first)	10.3	27.0
latin 60 (first_fail) (first)	57.5	182.0
latin 60 (input_order) (first)	51.2	175.9

Table 6: Comparison of pairwise sequences versus aggregated sequences.

Table 6 shows the results of a comparison between LDSB implemented using a pairwise representation of interchangeable variable sequences, and using the usual aggregated representation. The number shown is the time taken to find the first solution or all solutions, under the given variable ordering.

B User Interface

LDSB offers the user two main predicates: `ldsb_initialise` to declare the symmetries and `ldsb_indomain` to label variables during search. Let us first introduce them with an example and then discuss them in more detail.

Example 15 Consider the CSP $(\{x_1,x_2,x_3\}, 1..3, \mathit{all\_different}(\{x_1,x_2,x_3\}))$ which has interchangeable variables and interchangeable values. This problem can be easily specified in ECL^{i}PS^e with LDSB as follows (LDSB-specific parts are shown in italics).

```
1  :- lib(ic).
2  :- lib(ldsb).
3  problem :-
4    Vars = [](X1,X2,X3),
5    Vars #:: 1..3,
6    alldifferent(Vars),
7    ldsb_initialise(Vars, [ variable_interchange([X1,X2,X3]),
8                            value_interchange([1,2,3]) ]),
9    search(Vars, 0, input_order, ldsb_indomain, complete, []).
```

Lines 1 and 2 load the *ic* solver library and LDSB's library, respectively. The problem variables are declared on line 4 and given initial domains on line 5. On line 6 the only constraint in the problem is posted. On lines 7 and 8 the search variables and symmetries are given to the LDSB initialisation routine. Finally, on line 9 the solver's search routine is called with LDSB's *ldsb_indomain* predicate, which is used to instantiate the variables. □

Declaring symmetries The instances of the symmetry patterns that can be declared by `ldsb_initialise` are the four discussed in Section 4:

- `variable_interchange(Vars)`
- `value_interchange(Vals)`
- `sequence_variable_interchange(ListVars)` (for variable sequences)
- `sequence_value_interchange(ListVals)` (for value sequences)

where `Vars` is a list of variables, `Vals` a list of values, `ListVars` a list of lists of variables, and `ListVals` a list of lists of values. The ECL^{i}PS^e syntax for instances of the variable sequence interchange pattern is illustrated in appendix C. Common instances, such as variable/value interchangeability acting on all problem variables/values, can be specified with special keywords. For example, the initialisation in example 15 above can be replaced with:

```
ldsb_initialise(Vars, [variable_interchange, value_interchange])
```

The `ldsb_initialise` routine attaches attributes to the variables that appear in the declared LDSB pattern instances (if any). The attributes contain the state of each pattern instance (i.e., the value of W used in Algorithms 2 and 3), and for each variable its index in the matrix `Vars` of decision variables, and its position in any instance of the interchangeable variable sequence pattern. This stored information is used later to process the symmetries during search.

Breaking symmetries during search Two alternative predicates are provided to support symmetry breaking during search by LDSB:

- `ldsb_indomain`
- `ldsb_try`

The first predicate, `ldsb_indomain`, instantiates a variable to a value, trying the values in ascending order, like the usual `indomain` predicate in constraint programming. The difference is that `ldsb_indomain` additionally carries out pruning based on the declared symmetries, as described in section 5 above. To perform the search without symmetry breaking, the call to the search procedure would instead be:

```
search(Vars, 0, input_order, indomain, complete, []).
```

The other predicate, `ldsb_try`, takes a variable and a value as arguments, and instantiates the variable to the value. On backtracking, it removes the value from the variable's domain, and carries out the pruning based on the declared symmetries. The `ldsb_try` routine reads, uses, and updates the pattern instances recorded in the variables' attributes. This predicate is used in the implementation of `ldsb_indomain` (as shown in appendix C) but it is additionally made available to end users in case they want to program an application-specific value choice routine.

Example 16 The problem of Example 15 can also be expressed in Gecode. For brevity, we show only the relevant parts instead of the whole program (again, LDSB-specific portions are in italics).

```
1  class Problem : public Space {
2  public:
3      IntVarArray xs;
4      Problem() : xs(*this, 6, 0, 1) {
5          Matrix<IntVarArray> m(xs, 2, 3);
6          Symmetries syms;
7          syms << rows_interchange(m);
8          branch(*this, xs, INT_VAR_NONE, INT_VAL_MIN, syms);
9      }
10 }
```

Line 1 includes the LDSB library. The problem variable array is declared on line 4 and its six variables are created and given initial domains on line 5. On line 6 we declare a matrix-wrapper around the flat array to give meaning to rows and columns. The set that will contain the pattern instances is created on line 7 and the single pattern instance is constructed and added to the set on line 8. Finally, the set of symmetries is passed to the branching function on line 9. □

Note that the LDSB version of the `branch` function accepts the same options as Gecode's own branch method, allowing any variable and value ordering to be used with LDSB.

C Latin Square solved with LDSB in ECL^{i}PS^e

Example 17 The Latin Square problem finds an $N \times N$ matrix of $1..N$ values, each occurring once in each row and column. A CSP for $N = 3$ has 9 integer variables $\{x_{ij} | i, j \in 1..3\}$, one per cell in row i and column j (see Figure 1). This problem can be specified for any size N in ECL^{i}PS^e with LDSB as follows (LDSB-specific parts are shown in italics).

```
:- lib(ic).
:- lib(ldsb).

latin(N, Board) :-
        dim(Board, [N,N]),              % [ECLiPSe array syntax]
        Board #:: 1..N,
        constrain(N,Board),
        ldsb_initialise(Board, [rows_interchange,
                                columns_interchange,
                                values_interchange]),
        term_variables(Board, Vars), search(Vars).

search([]).
search([X|Xs]) :- ldsb_indomain(X) , search(Xs).

constrain(N,Board) :- constrain2(1, N, Board).
constrain2(I, N, _Board) :- I > N.
constrain2(I, N, Board) :- I =< N,
        Row is Board[I, 1..N],    % [Row is a list of variables]
        alldifferent(Row),
        Col is Board[1..N, I],    % [Col is a list of variables]
        alldifferent(Col),
        I1 is I+1, constrain2(I1, N, Board).
```

We write the search manually to illustrate the explicit use of the `ldsb_indomain` predicate. □

Although the following implementation of `ldsb_indomain` is part of the ECL^{i}PS^e LDSB library, it is included here to show the relationship between `ldsb_indomain` and `ldsb_try`.

```
% Tries values for X in ascending order.
ldsb_indomain(X) :- nonvar(X), !.
ldsb_indomain(X) :-
        ic:is_solver_var(X),
        get_min(X,V),
        ldsb_try(X, V),
        ldsb_indomain(X).
```

D Supplementary Figures

The following figures are larger versions of those in Figures 7a to 7c. The number plotted on top of each point is the number, out of 20, of instances solved for that problem size. If all instances are solved, the number is omitted.

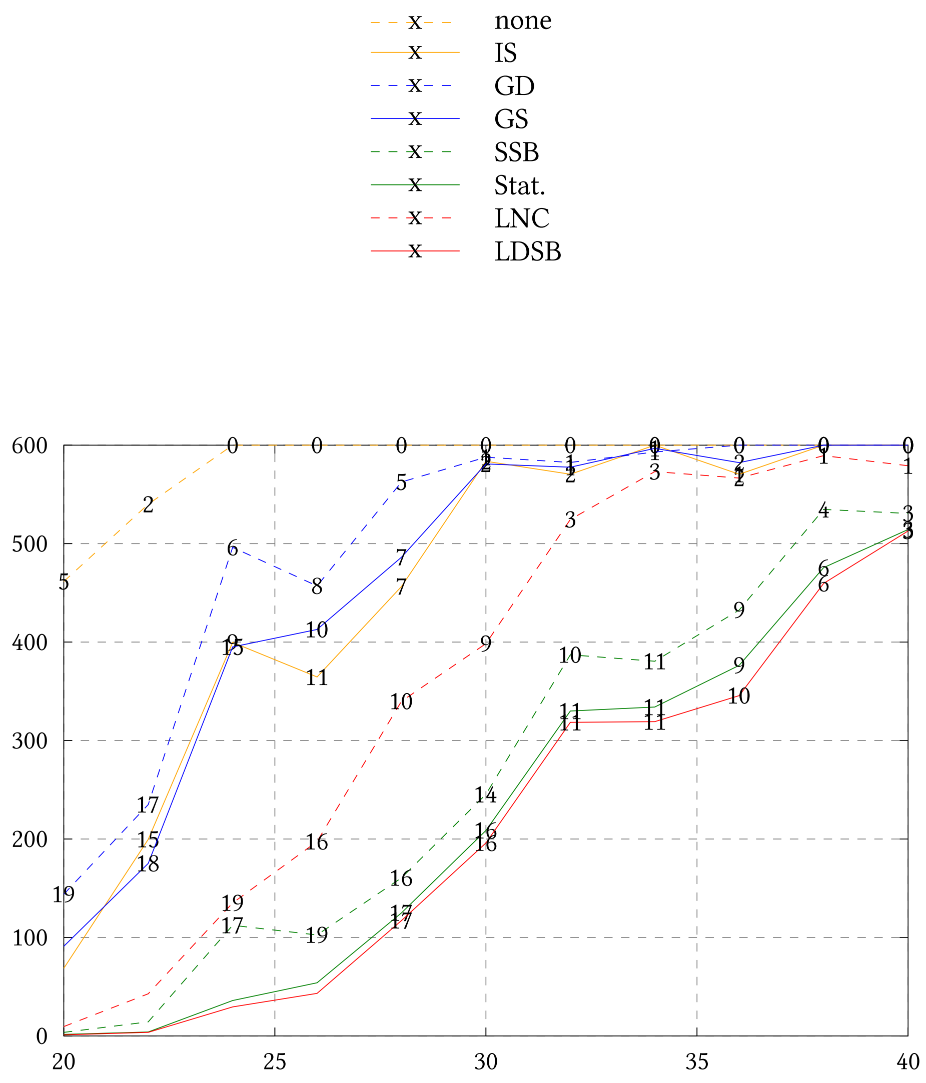


Fig. 10: Concert hall scheduling, first-fail.

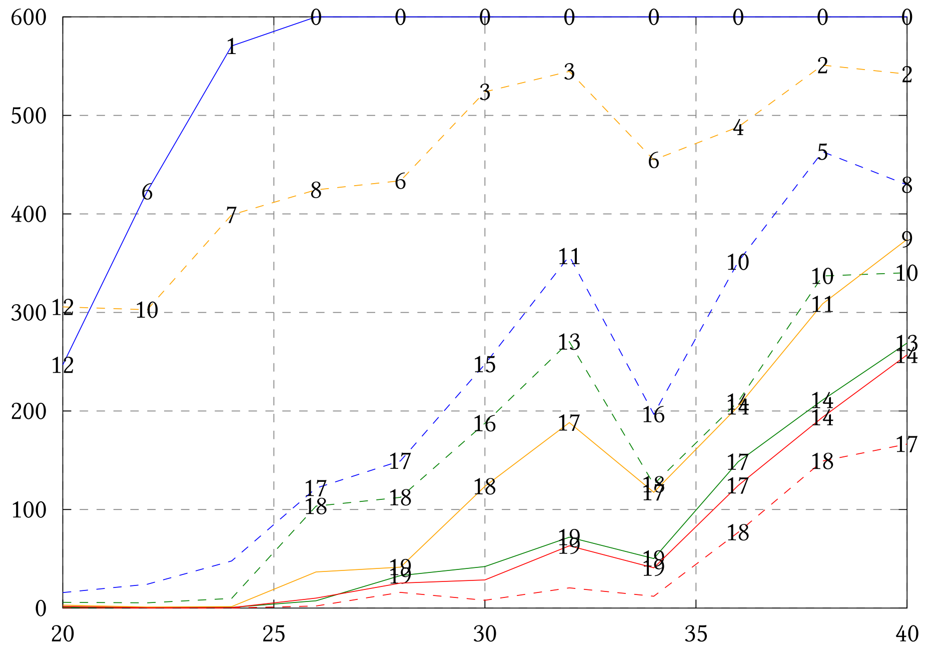


Fig. 11: Graph colouring, biased distribution, $q = 0.75$, input order.

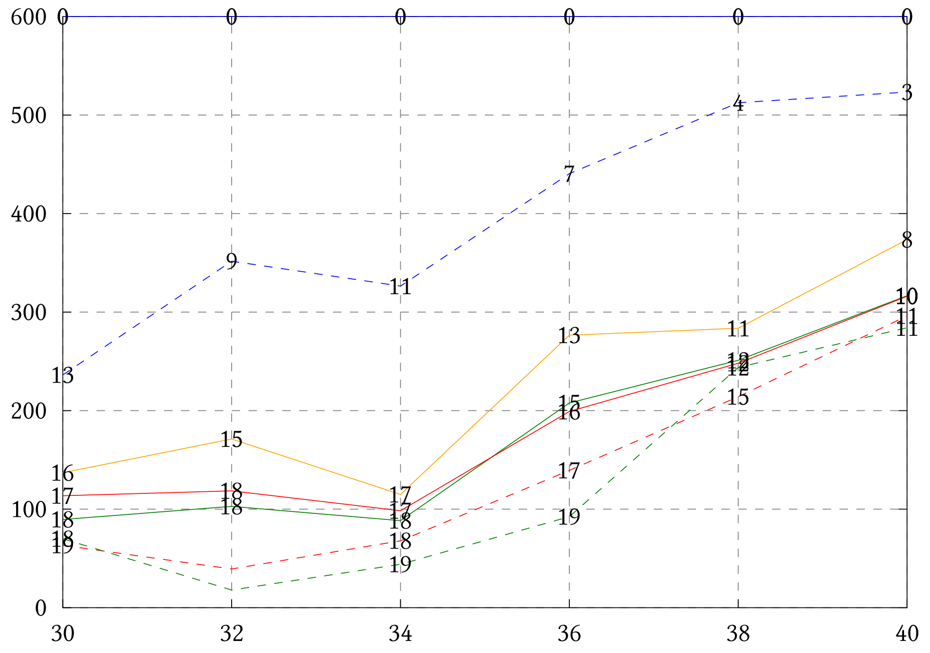


Fig. 12: Graph colouring, uniform distribution, $q = 1$, input order.