

# Dynamic Symmetry Breaking Restarted

Daniel S. Heller and Meinolf Sellmann

Brown University, Department of Computer Science  
P.O. Box 1910, Providence, RI 02912, U.S.A.  
dheller,sello@cs.brown.edu

## Abstract

Recently, structural symmetry breaking (SSB), a new technique for breaking all piecewise variable and value symmetry in constraint satisfaction problems (CSPs), was introduced. As of today, it is unclear whether the heavy symmetry filtering that SSB performs is at all worthwhile. This paper has two aims: First, we assess the feasibility of SSB. To this end, we introduce the first random benchmark generator that produces CSP instances with piecewise symmetric variables and values of constrainedness. It allows us to evaluate SSB on different regions of constrainedness. Secondly, we study how symmetry breaking and restarts interact. We propose practical enhancements of SSB that allow us to re-use symmetry no-goods in subsequent restarts efficiently. With those enhancements, we find that symmetry breaking can actually benefit from restarts. However, the improvements to be gained by restarting are far smaller than those that can be obtained for methods that break only some symmetries or none at all. Surprisingly, we find that a combination of restarts and breaking value symmetry only can be competitive with, or even be superior to, complete symmetry breaking.

**Keywords:** structural symmetry breaking, dynamic symmetry breaking, piecewise symmetry

## 1 Introduction

Symmetry breaking has received considerable and increasing interest in past years. It is widely accepted now that symmetries can cause significant problems to systematic solvers that unnecessarily explore redundant parts of the search tree. Methods to avoid this undesirable behavior range from adapting ordering heuristics [2], adding static constraints to the problem [4; 6], adding constraints during search [9], and filtering values based on a symmetric dominance analysis when comparing the current search node with those that were previously expanded [5; 7; 1; 13].

Especially the latter technique, known as symmetry breaking by dominance detection (SBDD), has proven to excel on problems that contain large symmetry groups. The core task of SBDD is the dominance detection algorithm. The first automated dominance detection algorithms were based

on group theory [8], while the first provably polynomial-time dominance checkers for specific types of value symmetry were devised in [16]. This work was later extended to tackle any kind of value symmetry in polynomial time [14]. Based on these results, for specific “piecewise” symmetric problems, [15] showed that breaking variable and value symmetry can be broken simultaneously in polynomial time. The method was named structural symmetry breaking (SSB) and is based on the structural abstraction of a given partial assignment of values to variables.

Compared with other symmetry breaking techniques, the big advantage of dynamic symmetry breaking is that it can accommodate dynamic variable and value orderings. Dynamic orderings have shown to be vastly superior to static orderings in many different types of constraint satisfaction problems. However, robust heuristics for the selection of variables and values are hard to come by. For the task of variable selection, a bias towards variables with smaller domains often works comparably well, but there always remains a fair probability that we hit instances on which a solver gets trapped in extremely long runs. Particularly, heavy-tailed runtime distributions have been reported [10]. One way to circumvent this problematic situation is to randomize the solver and to restart the search when a run takes too long [11]. While dynamic symmetry breaking and restarts are orthogonal techniques in that they can be applied independently from one another, their interplay has not been studied yet.

In this contribution, we wish to investigate questions regarding dynamic symmetry breaking in general and SSB in particular. While the existing theoretical worst-case analysis of SSB shows that we can guarantee symmetry-free search trees in polynomial time, it remains unclear so far whether we can also implement the method so that it performs well in practice. We introduce practical enhancements of SSB such as delayed ancestor-based filtering and incremental data structures for sibling-based filtering. We then apply the method in combination with one-shot and restarted solvers. To conduct this study, we introduce the first randomized test suite for symmetry breaking experiments. It allows us to evaluate the method over an entire region of constrainedness rather than on isolated benchmark instances of unknown level of constrainedness only.

The paper is organized as follows: In the following section, we briefly review structural symmetry breaking. We discuss how symmetry breaking by dominance detection can be ex-

exploited as a no-good store between restarts in Section 3. Then, in Sections 4 and 5 we devise efficient mechanisms to speed-up structural symmetry breaking in practice by introducing delayed ancestor-based filtering and by devising an incremental data structure for sibling-based filtering. The technical core of the paper concludes in Section 6 where we present extensive numerical results on the effect of our enhancements as well as the interplay of symmetry breaking and restarts.

## 2 Background

Recently, a new technique was developed that, for the first time, allows us to simultaneously break value and variable symmetry in CSPs [15]. This technique, named *structural symmetry breaking (SSB)*, is based on the quantitative abstraction of a constraint program that contains sets of pairwise symmetric variables and values. Before we study implementation issues later, we now give a high-level description of SSB.

### Definition 1

- A Constraint Satisfaction Problem (CSP) is a tuple  $(Z, V, D, C)$  where  $Z = \{X_1, \dots, X_n\}$  is a finite set of variables,  $V = \{v_1, \dots, v_m\}$  is a set of values,  $D = \{D_1, \dots, D_n\}$  is a set of finite domains where each  $D_i \in D$  is the set of possible instantiations to variable  $X_i$ , and  $C = \{c_1, \dots, c_p\}$  is a finite set of constraints where each  $c_i \in C$  is defined on a subset of the variables in  $Z$  and specifying their valid combinations. We say that the CSP has scalar variables iff for all  $D_i \in D$  it holds that  $D_i \subseteq V$ . Throughout this paper, we will consider scalar CSPs.
- Given a CSP with scalar variables, an assignment  $A$  is a set of pairs  $(X, v) \in Z \times V$  such that  $(X, v), (X, w) \in A$  implies  $v = w$ . An assignment of cardinality  $n$  is called complete, otherwise it is called partial. A complete assignment satisfying all constraints is called a solution.

### Definition 2

- Given a set  $S$  and a set of sets  $P = \{P_1, \dots, P_r\}$  such that  $\bigcup_i P_i = S$  and the  $P_i$  are pairwise non-overlapping, we say that  $P$  is a partition of  $S$ , and we write  $S = \sum_i P_i$ .
- Given a set  $S$  and a partition  $S = \sum_i P_i$ , a bijection  $\pi : S \mapsto S$  such that  $\pi(P_i) = P_i$  (where  $\pi(P_i) = \{\pi(s) \mid s \in P_i\}$ ) is called a piecewise permutation over  $S = \sum_i P_i$ .

The type of symmetry that the method can tackle efficiently is defined as follows:

### Definition 3

- Given a CSP  $(Z, V, D, C)$ , and partitions  $Z = \sum_{k \leq r} P_k$ ,  $V = \sum_{l \leq s} Q_l$ , we say that the CSP has piecewise variable and value symmetry iff all variables within each  $P_k$  and all values within each  $Q_l$  are considered as interchangeable [3].
- Given two assignments  $A$  and  $B$  on a piecewise symmetric CSP, we say that  $A$  dominates  $B$  iff there exist piecewise permutations  $\pi$  over  $Z = \sum_{k \leq r} P_k$  and  $\alpha$

over  $V = \sum_{l \leq s} Q_l$  such that for all  $(X, v) \in A$  it holds that  $(\pi(X), \alpha(v)) \in B$ .

- Given two arbitrary assignments  $A$  and  $B$  for a piecewise symmetric CSP, we call the problem of determining whether  $A$  dominates  $B$  the Dominance Detection Problem.

## 2.1 SSB for Dominance Checking

The core idea to devising an efficient dominance checker for piecewise symmetric CSPs lies in the definition of *signatures* of values under an assignment.

### Definition 4

- Given a partial assignment  $A$ , for all values  $v$ , we define

$$\text{sign}_A(v) := (|\{X_i \in P_k \mid (X_i, v) \in A\}|)_{k \leq r},$$

where  $k$  indexes the different variable partitions  $\sum_{k \leq r} P_k$ . That is, the signature of  $v$  under  $A$  is the tuple that counts, for each variable partition, by how many variables in the partition the value is taken in  $A$ .

- We say that a value  $v$  in an assignment  $A$  dominates a value  $w$  in assignment  $B$  iff  $v$  and  $w$  belong to the same value-symmetry class and  $\text{sign}_A(v) \leq \text{sign}_B(w)$ .<sup>1</sup>
- We say that a value  $v$  in an assignment  $A$  is structurally equivalent to a value  $w$  in assignment  $B$  iff  $v$  and  $w$  belong to the same value-symmetry class and  $\text{sign}_A(v) = \text{sign}_B(w)$ .

The following result, from [15], connects dominance relations and partial assignments.

### Lemma 1

An assignment  $A$  dominates another assignment  $B$  in a piecewise symmetric CSP iff there exists a piecewise permutation  $\alpha$  over  $\sum_{l \leq s} Q_l$  such that  $v$  in  $A$  dominates  $\alpha(v)$  in  $B$  for all  $v \in V$ .

This lemma allows us to check dominance between assignments  $A$  and  $B$ : We set up a bipartite graph where, for each value  $v$ , there is one node on the left and one on the right. An edge connects two nodes with associated values  $v$  and  $w$  from the same value partition iff  $\text{sign}_A(v) \leq \text{sign}_B(w)$ . Then,  $A$  dominates  $B$  iff the bipartite graph contains a perfect matching.

## 2.2 SSB for Filtering

Based on the dominance checking algorithm, we can now filter values from domains iff setting the respective variable to some value would lead to a symmetric choice point. Since symmetry-based filtering *anticipates* when variable assignments will result in symmetric configurations, within SSB we have to distinguish two different types of filtering: *ancestor-based filtering* where we compare extensions to the current partial assignment with previously fully expanded search nodes, and *sibling-based filtering* where we compare extensions to the current partial assignments with other such extensions.

<sup>1</sup>Where the  $\leq$ -relation on vectors is defined as the usual component-wise comparison, i.e.:  $x \leq y$  iff  $x_i \leq y_i \forall i$ .

The latter is very easy to handle as sibling-symmetry can only be caused by value symmetry in the problem.<sup>2</sup> Consequently, we can break all sibling symmetry simply by choosing only one arbitrary value out of each group of values within the same value partition that have the same signature. Ancestor-based filtering on the other hand can be performed by considering almost successful dominance checks: When the bipartite graph that we set-up contains an almost perfect matching where just one more edge is missing to complete it, we can quickly identify such *critical edges* and check whether one more variable assignment would cause the critical edge to be added to the graph. In this case, we have found a *critical variable assignment* that can and should be avoided by removing the respective value from the variable’s domain.

This concludes our brief review of SSB. For a more detailed description of the method and a worst-case asymptotic runtime analysis, we refer the reader to [15].

### 3 Symmetry No-goods and Restarts

Branching decisions are critical to the efficiency of systematic search, and truly robust heuristics for choosing branching variables are not known. Empirical analysis of runtime distributions with randomized choices of branching variables has revealed that there is a substantial chance of very long runs [10]. At the same time, there is also a good probability that a random selection of variables will result in a short run. Consequently, it has been suggested to simply restart backtracking solvers when a run takes too long. This method of restarted randomized searches has been one key element of the latest generations of outstandingly powerful DPLL-based SAT-solvers.

Restarts work already quite well when everything is forgotten between two runs. In SAT, however, no-good learning algorithms augment the SAT-formula during search so as to improve the performance of unit propagation within DPLL. The no-goods learned implicitly store information on which parts of the search space have already been investigated. Moreover, they also contain information on which parts have not been investigated yet but cannot contain solutions as search would fail for the same reasons as it did earlier. Within one run of a backtracking solver, the first information is obviously obsolete as the systematic search already guarantees that the same part of the search space will not be investigated twice. However, the information becomes interesting when restarts are being used.

With respect to symmetry breaking, SSB (as a special form of SBDD) stores the most general previously fully expanded search nodes as a list of no-goods. In contrast to ordinary no-goods, an SBDD no-good implicitly represents an equivalence class of no-goods (namely the set of all its symmetric variants), and it is the algorithmic task of the dominance checker to see whether this set contains a no-good that is relevant with respect to the current search node. In this view, SBDD resembles the task of performing inference in predicate logic where we also need the help of a unification algorithm to find an applicable rule or fact as represented implicitly by all-quantified rules and facts in the formula.

<sup>2</sup>This assumes that all siblings are generated by branching on the same variable.

What is interesting to note is that SBDD no-goods also keep a record of those parts of the search space that have already been searched through. In that regard, it is of interest to store them (or at least the most powerful ones) between restarts. There is a trade-off, however: No-goods will only be beneficial if the method that prevents us from exploring the same part of the search space more than once does not impose a greater computational cost than what the exploration would cost anyway. One simple thing that we can do is to remove those no-goods from the list that have very little impact anyway because they only represent a small part of the search space. This is an idea that is commonly used in SAT. However, for symmetry-nogoods we can do more.

### 4 Delayed Ancestor-based Filtering

We introduce delayed symmetry filtering. The core idea here is to apply an inexpensive inference mechanism that quickly identifies which no-goods cannot cause effective symmetry-based filtering at a given search node. The aim here is to save many of the expensive calls to SSB-based domain filtering as described in the previous section. Note that no-goods are only used for ancestor-based filtering, which is why this idea will only be applied for this type of symmetry filtering. We discuss special methods to improve the performance of sibling-based filtering in Section 5.

#### 4.1 A Simple Pretest

To cut down on full-fledged ancestor-based filtering calls, we introduce a simple pretest. What we need to identify are simple conditions under which a previously expanded node  $\alpha$  (as usual,  $\alpha$  is identified with the partial assignment that leads to the node) cannot “almost dominate” the current search node  $\beta$ . Precisely,  $\alpha$  “almost dominates”  $\beta$  if one more assignment to  $\beta$  could result in a successful dominance relation with  $\alpha$ . This is a necessary condition for SSB filtering to have any effect

First, we observe that  $\beta$  must contain exactly one less variable assignments as  $\alpha$ . This is a trivial condition which is always true in a one-shot tree search as all no-goods stored by SBDD were taken from search nodes at the same or lower depth as that of the current node. However, for no-goods stored in earlier restarts, this test can quickly reveal that ancestor-based filtering will not be effective.

Only if the above condition holds, we perform one more test before applying the full-fledged filtering call: we look more closely at the two assignments  $\alpha$  and  $\beta$  and see whether  $\alpha$  is close to dominating  $\beta$ . Before looking at each value individually, determining all their signatures and which ones dominates which, we can do the same on the level of value classes: For each value partition, we determine how many variables in each value partition are taking a value in it under assignment  $\gamma$ , thus computing a signature for each partition of mutually symmetric values:

$$\text{sign}_\gamma(Q_l) := (|\{X \in P_k \mid \gamma(v) \in Q_l\}|)_{k \leq r} \quad \forall 1 \leq l \leq s.$$

Now, from the description of SSB dominance checking in Section 2.1, we can infer:

#### Lemma 2

Given assignments  $\alpha$  and  $\beta$  such that  $\alpha$  dominates  $\beta$ , we have that, for all  $1 \leq l \leq s$ ,  $\text{sign}_\alpha(Q_l) \leq \text{sign}_\beta(Q_l)$  (whereby  $\leq$  denotes the component-wise comparison of the two tuples).

**Proof:** Let  $l \in \{1, \dots, s\}$ . Since  $\alpha$  dominates  $\beta$ , we have that, for all  $v \in Q_l$ ,  $\text{sign}_\alpha(v) \leq \text{sign}_\beta(w)$  for some value  $w \in Q_l$  that is the unique matching partner of  $v$ . Consequently,

$$\text{sign}_\alpha(Q_l) = \sum_{v \in Q_l} \text{sign}_\alpha(v) \leq \sum_{w \in Q_l} \text{sign}_\beta(w) = \text{sign}_\beta(Q_l).$$

■

Thus, SSB filtering can only be effective if the inequality holds for all but at most one value partition  $l$ , and if for that partition we have that  $\text{sign}_\alpha(Q_l) \leq \text{sign}_\beta(Q_l) + e_k$ , where  $e_k$  is the unit vector with a 1 in position  $1 \leq k \leq r$ . Only if this condition holds, we finally apply ancestor-based filtering.

Note that our simple pretest can be conducted much faster than a full-fledged filtering call: it runs in time linear in the size of the given assignments (which is in  $O(n)$ ) whereas ancestor-based filtering wrt each ancestor requires time  $O(m^{2.5} + mn)$  for a CSP with  $n$  variables and  $m$  values.

#### 4.2 Deterministic Lower Bounds

Assume that a call to the ancestor-based filtering procedure reveals that we are at least  $p$  edges short of finding a perfect matching in the value dominance graph that was set-up for assignments  $\alpha$  and  $\beta$ .<sup>3</sup> Clearly, as was already noted in [15], this means that at least another  $p - 1$  variable assignments need to be added to  $\beta$  before filtering can become effective. By adding this information to no-good  $\alpha$ , and by keeping track of the depth of the current search node  $\beta$ , we can avoid many useless filtering calls. What is interesting is that we cannot only propagate this information when diving deeper into the tree, but also upon backtracking.

Consider the following example: For no-good  $\alpha$ , the check against the current search node in depth  $d$  results in a maximum matching with 4 edges missing to be perfect. Then, at depth  $d + 4 - 1 = d + 3$ , we call for ancestor filtering wrt  $\alpha$  again and find that there are still 4 edges missing. Clearly, this means that none of the last 3 branching decisions has brought us any closer to a successful dominance relation with  $\alpha$ , and this information can be used even when backtracking up from the current position as illustrated in Figure 1 (A): At depth  $d + 2$ , for example, we know, even without conducting the filtering call, that the maximum matching must have 4 edges missing. Which implies that, when diving deeper into the tree from depth  $d + 2$ , ancestor-based filtering cannot be effective until we reach depth  $d + 2 + 4 - 1 = d + 5$ .

More generally, if at depth  $d + p - 1$  we find a maximum matching with  $q \leq p$  edges missing to perfection, when backtracking up to depth  $d + r - 1$  for some  $r < p$ , we are sure that there are at least  $\max\{r + q, p\} - 1$  more variable assignments necessary before filtering wrt ancestor  $\alpha$  can be effective. Consequently, we will not call for ancestor-based filtering wrt  $\alpha$  until we reach depth  $\max\{d + r + q - 1, d + p - 1\}$  in the search tree (see Figure 1 (B)).

Note that the above procedure also works if we never get to perform the full filtering procedure at depth  $d + p - 1$  because our pretest fails: If the first condition fails, instead of using the number of missing perfect matching edges, we can

<sup>3</sup>The method by which SBDD unifies no-goods ensures that  $p > 1$  is only possible for no-goods generated in earlier runs.

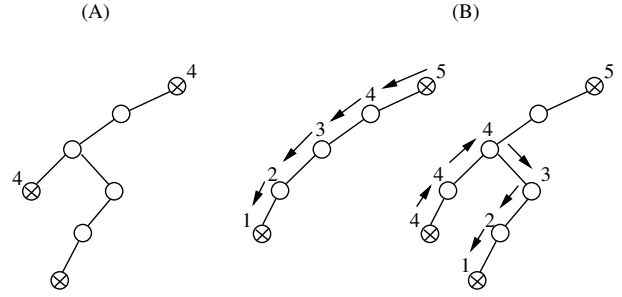


Figure 1: The figures show part of the search tree. The root node is considered to have depth  $d$ . At this node, we find that at least 4 (A) or 5 (B) more assignments are necessary before the respective ancestor could dominate the current partial assignment. We dive deeper into the tree without conducting filtering (hollow nodes) wrt the given ancestor until we reach depth  $d + 3$  (A) or  $d + 4$  (B) where filtering could be effective. In both cases, the call to the filtering algorithm reveals that at least 4 more variable assignments are necessary before dominance can occur. Consequently, the same holds for all ancestors of the respective nodes. By propagating this information up in the tree (see B), filtering is delayed further when branching off from intermediary nodes.

simply count the number of variable assignments still needed before at least as many variables are assigned in the current search node as are in  $\alpha$ . And in the second case, we can count how many more assignments are necessary before each value class signature in  $\alpha$  can have become lower or equal to the signatures in the current partial assignment.

#### 5 Incremental Sibling-based Filtering

Sibling-based filtering requires that we compute the sets of mutually symmetric values that have the same signature under the current partial assignment. Rather than recomputing the signatures of all values and regrouping the values after a branching step has added another variable assignment, we use an incremental data structure for this purpose so as to conduct this type of symmetry related inference as efficiently as possible.

First, let us describe the idea of sparse signatures that are needed to guarantee the worst-case complexity as given in [15]: Instead of writing down entire signatures, for each value we maintain a sparse list that only contains the non-zero entries of a signature, together with the information to which variable partition an entry in the sparse list belongs. To set up this sparse representation from a new partial assignment, we first order the variable instantiations in a given partial assignment according to the partition that the corresponding variable belongs to. This can be done in time linear in the number of variable partitions. In this order, we then scan through the partial assignment and set up the sparse signatures simply by adding one to the last entry if the current variable belongs to the same partition as the last, and by introducing a new non-zero entry if the variable belongs to a new partition.

For the current search node, we group values in the same value partition and with the same signature in the following

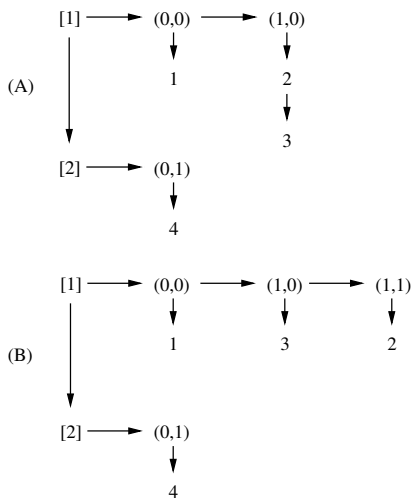


Figure 2: An efficient data structure supporting sibling-based filtering incrementally. The leftmost column depicts value partitions  $[1] = \{1, 2, 3\}$  and  $[2] = \{4\}$ . For both partitions, horizontally it follows a sorted list of signatures that are each associated with all the values underneath them in the current partial assignment. Even though signatures are actually stored in sparse format, we show them explicitly to improve the readability.

data structure. It consists of an array of lists, one for each value partition. Each such list contains, in lexicographic order, the different signatures within the respective value partition. Associated with each signature is yet another list of values in the partition that have the signature, whereby each value holds a pointer to the signature. Note that this data structure allows us to perform sibling based-filtering extremely efficiently. Given a variable, the different values that we need to consider when branching are only the first values in each list of each signature in each value partition.

Now, when branching by assigning a value to some variable, we update the data structure incrementally. Note that the value assigned is the first in its list of values with the same signature. Moreover, the value holds a pointer to its signature. Therefore, we can compute its new signature incrementally, and since the signatures within the value partition are ordered lexicographically, we can also find out quickly to which signature the value needs to be added, whereby we create a new list of values if the value’s new signature is not yet in our list. Finally, we remove the value from the list of values for its old signature and add it to the list of values for its new signature, while updating the value’s pointer to its own signature.

We illustrate the data structure in Figure 2 on the following example. Assume we are given four variables  $X_1, \dots, X_4$ , whereby the first two and the last two are symmetric. Assume further that the variables can take four values  $1, \dots, 4$ , whereby the first three are symmetric. Figure 2 (A) shows our incremental data structure for the partial assignment  $\{(X_1, 2), (X_2, 3), (X_3, 4)\}$ . We see that we can easily pick non-symmetric values simply by choosing the first representative for each signature. In our example, those are the values 1, 2, and 4. Figure 2 (B) shows the data structure after another variable has been instantiated by adding  $(X_4, 2)$  to

	UNBIASED		BIASED			
	15		15		30	
VPC	AllDiff	GCC	AllDiff	GCC	AllDiff	GCC
2	100	100	100	100	100	98-100
3	100	100	100	100	100	52-100
4	100	98-100	100	92	84-96	14-80
5	100	100	88	66	52-82	0-54
6	100	98-100	68	18-32	26-76	0-50
7	94	96-100	26	4-18	6-40	0-50
8	90	88-94	18	0-6	0-16	0-34
9	84	92	0	0-2	0	0-32
10	48	58	0	0	0	0-22
11	16	14	0	0	0	0-22
12	4	0	0	0	0	0

Table 1: Percentages of feasible solutions in the different benchmark sets for different numbers of values per constraint (VPC). We give ranges where even the best algorithm hit the time limit of 600 seconds.

our assignment. We see that the data structure can easily be adapted by updating the signature of value 2.

## 6 Experimental Results

With all the practical enhancements that we introduced in the previous section, we now wish to test structural symmetry breaking in practice. To this end, we need an appropriate benchmark set. Surprisingly enough, despite the large body of work on symmetry breaking, so far we are still lacking a benchmark set which allows us to experiment with symmetry breaking techniques over different regions of constrainedness. Standard benchmarks in the literature are graceful graphs, n-queens, balanced incomplete block designs, or the infamous social golfers, none of which give us a reasonable insight regarding the constrainedness of problem instances. Consequently, until today we lack a comparison of different symmetry breaking techniques over the entire region of constrainedness.

### 6.1 The Benchmark

We introduce a very simple benchmark generator that produces surprisingly hard instances of piecewise symmetric CSPs: Given a number of variables  $n$  and values  $m$ , as well as the number of variables  $n_c$  and the number of values  $m_c$  per constraint, we generate a given number of global cardinality constraints (GCC) over a set of  $n_c$  randomly chosen variables and  $m_c$  randomly chosen constraints, whereby we enforce that all variables in the constraint together take each chosen value exactly once. We vary the basic concept in the following ways:

- We add one more GCC over all variables and values that enforces that each value be chosen at most 2 times. Alternatively, we add an AllDifferent constraint over all variables and values.
- We draw variables and values uniformly or with a bias such that components with higher indices are more likely to be chosen than those with lower indices.

Tables 1 summarize the properties of our benchmark sets. We consider problems with 15 variables and values and 30

Algo	FO	SO	NO	FR	SR	NR
Sym-Level	Full	Sibl.	None	Full	Sibl.	None
Restarted	no	no	no	yes	yes	yes

Table 2: Overview of the different algorithm variants: Full refers to the variant where we call for ancestor and sibling-based filtering at every search node. Sibling refers to breaking value symmetry only by performing just sibling-based filtering. As a convention, when pretests or delayed filtering are switched off, we add '-P' or '-D' to the name of the algorithm.

variables and values. The number of variables per constraint is fixed at 12 while the number of values per constraint runs from 2 to 12, thus giving us a range of differently constrained instances. Table 1 shows the percentage of feasible instances out of 50 randomly generated ones. In addition, we vary the constraint over all variables and values (GCC or AllDifferent), and we select variables either uniformly or in a biased fashion, while values are always selected uniformly.

## 6.2 The Contestants

We implemented structural symmetry breaking as explained in the previous sections, whereby we can choose to

- run full SSB filtering for each search node,
- run sibling-based filtering only, or
- perform no symmetry breaking at all.

Additionally, when running full SSB, we can switch both the use of lower bounds and the simple pretest for delayed ancestor-based filtering on or off. The branching variable is determined dynamically by a min-domain heuristic. We can also choose to run the solver with a restart heuristic whereby we choose a linear increase in the fail limits starting with as little as 100. In case of the restarted method, the branching variable is chosen according to a min-domain heuristic over a random subset of 20% of the variables. Table 2 summarizes the settings and names the different contestants that we let compete against one another.

All experiments in this paper were conducted on a 2 GHz AMD Athlon 64 Processor 3000+ CPU with 512 MByte main memory running Linux 2.6.10. Our code was written in C++ and compiled by the Gnu g++ compiler version 3.3.5 with optimization flag -O6. As our constraint solver, we used Ilog Concert 2.2 on top of Ilog Solver 6.2.

## 6.3 The Influence of Pretests and Delayed Filtering

First, we evaluated the effects of the practical improvements that we introduced in this paper. Figure 3 shows the impact of pretesting and delayed filtering on two benchmark sets with very different characteristics. We interrupted method FR after 600 CPU seconds and compare its runtime against FR-PD when the latter executes the same search tree. While all instances in the 15 variable benchmark were solved to completion within the time limit, only about 40% could be solved in the 30 variable benchmark which explains the ruggedness of the curve.

As was to be expected, in both cases we see a beneficiary impact of our techniques to reduce the filtering efforts incurred by SSB: On the 15 variable benchmark, we see that

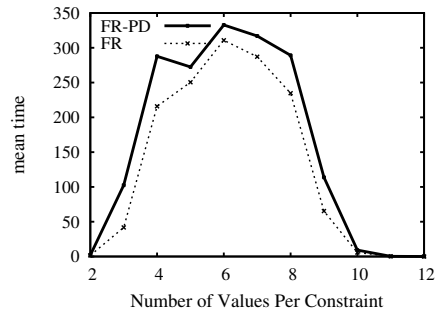
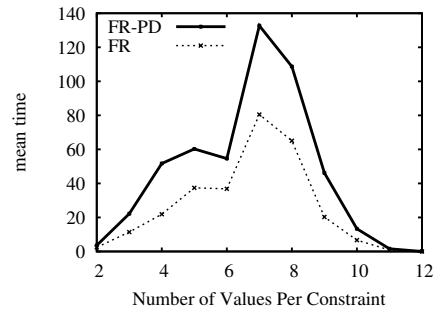


Figure 3: The impact of pretesting and delayed filtering on 50 instances with 15 variables and values, biased variable selection, and global GCC (top) and 30 variables and values, biased variable selection, and global AllDiff (bottom). We compare runtimes (on identical instances) by algorithms FR and FR-PD.

both pretest and delayed filtering save us up to 9000 full symmetry checks, which results in a speed-up of 3 for the overall method. On the 30 variable benchmark, we see an even greater impact: For 4 values per constraint we save more than 175,000 full symmetry checks which results in a speedup of more than 10. As we will see in Section 6.5, these improvements are key to making use of symmetry no-goods in restarted search methods.

## 6.4 The Impact of Symmetry Breaking

The main objective of this investigation is to study the practicability of SSB. Clearly, symmetry breaking only ever pays off when the work that we have to put in to detect symmetry does not exceed the work that we save in this way. Figure 4 shows three algorithms running three different levels of symmetry breaking in a one-shot deterministic run on instances with 15 variables and values and biased variable selection. Experiments were interrupted after 600 CPU seconds.

We see clearly that SSB (FO) leads to remarkable improvements over a standard CP approach that is unaware of symmetry altogether (NO). This holds particularly in the critically and over-constrained regions, whereby ignoring symmetry breaking in part or altogether can be the better option in the very under-constrained region, as we see in the experiment with uniform value selection and global GCC.

Especially the benchmark with biased variable selection and global AllDifferent constraint shows that SSB, despite its huge computational costs, is very worthwhile and can lead to speed-ups of orders of magnitude. To investigate the effect of

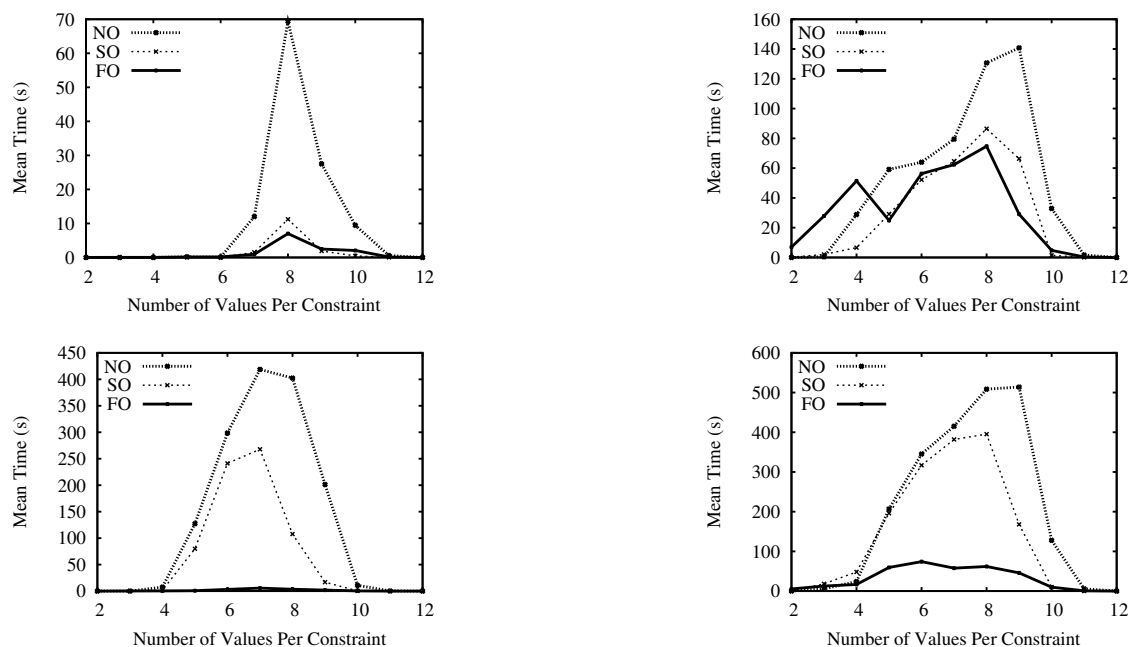


Figure 4: The figures give mean times in seconds of non-restarted algorithms on 50 instances with 15 variables and values, 12 variables per constraint, unbiased (top) or biased (bottom) variable selection, and AllDifferent (left) or GCC (right) as constraint over all variables and values.

symmetry breaking further, in Table 3 we show the number of fails and the time spent per choice point for FO, SO, and NO on two very different benchmark sets with 15 variables and values and 12 uniformly chosen variables per constraint, as well as 30 variables and values and 12 non-uniformly chosen variables per constraint (both have the AllDifferent as global constraint). The table reveals the dramatic change in the characteristic of our algorithm that is introduced by symmetry breaking: NO and SO investigate hundredthousands and millions of search nodes while spending a miniscule amount of time per search node. FO on the other hand spends far more time in every choice point but therefore greatly reduces the number of nodes visited.

### 6.5 The Impact of Restarts

After seeing that symmetry breaking is beneficial even at the tremendous costs that SSB filtering incurs, we are curious to see how restarts affect the landscape. We show the performance of the restarted algorithms in Figure 5 on the benchmark sets with 15 variables and in Table 4 on the benchmark sets with 30 variables.

The comparison of Figure 5 with Figure 4 shows that the algorithm that is unaware of symmetries can benefit greatly from restarts. In the biased/AllDifferent case, for example, NR is more than 10 times faster than NO. The erratic curves for method NR depict an increase in the variance of the running time. Note that this variance is actually not introduced by the restarts but inherent to a method that can get very unlucky by getting stuck exploring symmetric parts of the search space over and over and over again. The fact that this was not visible in Figure 4 is due to the time limit (that we needed to impose to conduct our experiments within reason-

able time) which artificially decreases the variance of the slow algorithm NO. Table 4 also shows very clearly that NR performs far better than NO.

We get a similar picture when comparing the performance of SO and SR. In the set with 15 variables and values, 12 non-uniformly chosen variables per constraint, and one global AllDifferent constraint, for example, SR is about 25 times faster than SO. What is interesting to note is that restarts can actually be counter-productive in the over-constrained region. We suspect that this has to do with the linear increase of fail-limits that we chose for our experiments. Clearly, the shortest proof-tree showing that no solution exists can be quite large for a method that does not eliminate symmetry, and so it naturally takes a lot of restarts to get to that point.

When we perform full SSB filtering, we see that restarts do not help on the benchmark with 15 variables and values. Only as we tackle large and very hard problems, FR starts to outperform FO as can be seen in Table 4 when we consider instances with a global GCC.

This leads to a surprising conclusion: Just breaking value symmetry in combination with restarts is in many cases competitive with full SSB! Compare FR and SR on global AllDifferent instances in Figure 5, or on global GCC instances in Table 4, for example. Of course, SSB is still the clear winner in the critical region on our large benchmark set with 30 variables, but the good performance of restarted sibling-filtering is still astonishing. It is reasonable to conjecture that restarts can actually help reducing the adversary effects of variable symmetry, thus making light-weight value filtering a serious competitor. We believe that this could explain the common assessment that symmetry breaking does not pay off in local search methods [12]: Local search methods naturally explore

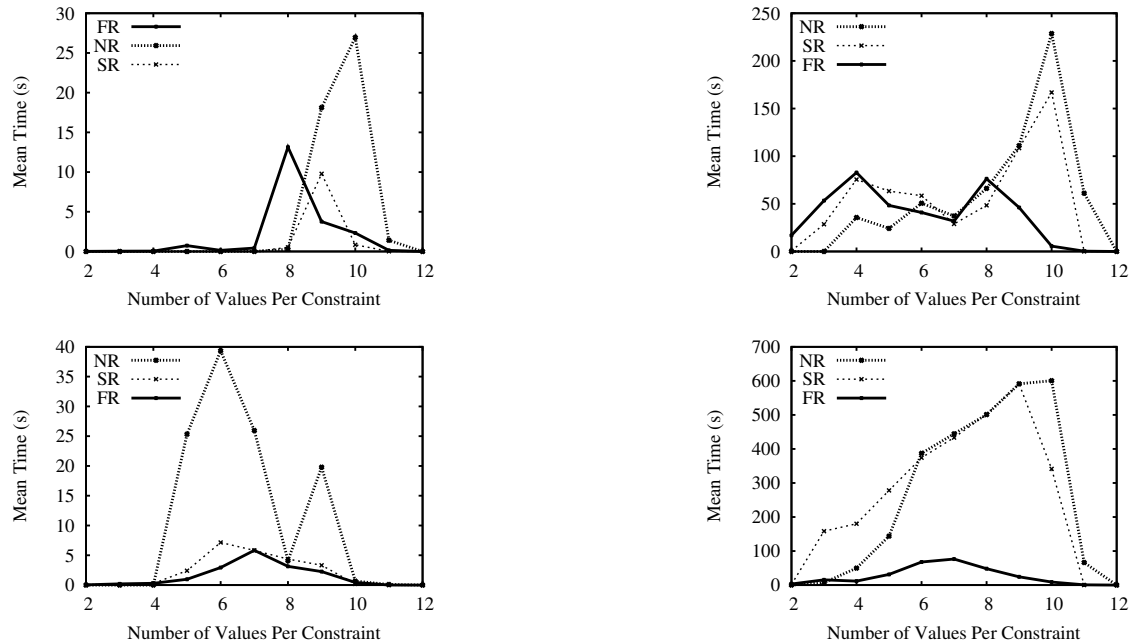


Figure 5: The figures give mean times in seconds of restarted algorithms on 50 instances with 15 variables and values, 12 variables per constraint, unbiased (top) or biased (bottom) variable selection, and AllDifferent (left) or GCC (right) as constraint over all variables and values.

wider parts of the search region. Moreover, the choice how solutions are represented often removes value symmetry directly. When we also take into account that local search is usually applied on underconstrained problems only, then the assessment that symmetry breaking is not worthwhile in local search is not so surprising anymore.

## 7 Conclusions

We provided practical enhancements of structural symmetry breaking and tested them on the first randomized benchmark set for symmetry breaking experiments. We showed that our practical enhancements lead to substantial speedups that are crucial to make restarts for SSB worthwhile. However, even with these enhancements, restarted methods that break only value symmetry are often almost as competitive as full symmetry breaking.

## References

- [1] N. Barnier and P. Brisset. Solving the Kirkman’s schoolgirl problem in a few seconds. *Proceedings of CP’02*, 477–491, 2002.
- [2] C. Brown, L. Finkelstein, P. Purdom Jr. Backtrack searching in the presence of symmetry. *Proceedings of AAECC-6*, 99–110, 1988.
- [3] D.A. Cohen, P. Jeavons, C. Jefferson, K.E. Petrie, B.M. Smith. Symmetry Definitions for Constraint Satisfaction Problems. *Constraints*, 11(2–3): 115–137, 2006.
- [4] J. Crawford, M. Ginsberg, E. Luks, A. Roy. Symmetry-breaking predicates for search problems. *Proceedings of KR’96*, 149–159, 1996.
- [5] T. Fahle, S. Schamberger, M. Sellmann. Symmetry Breaking. *Proceedings of CP’01*, 93–107, 2001.
- [6] P. Flener, A. Frisch, B. Hnich, Z. Kiziltan, I. Miguel, J. Pearson, T. Walsh. Breaking row and column symmetries in matrix models. *Proceedings of CP’02*, 462–476, 2002.
- [7] F. Focacci and M. Milano. Global cut framework for removing symmetries. *Proceedings of CP’01*, 77–92, 2001.
- [8] I. Gent, W. Harvey, T. Kelsey, S. Linton. Generic SBDD using computational group theory. *Proceedings of CP’03*, 333–347, 2003.
- [9] I. Gent and B. Smith. Symmetry breaking in constraint programming. *Proceedings of ECAI’00*, 599–603, 2000.
- [10] C.P. Gomes, B. Selman, N. Crato. Heavy-Tailed Distributions in Combinatorial Search. *Proceedings of CP’97*, 121–135, 1997.
- [11] Dynamic Restart Policies. H. Kautz, E. Horvitz, Y. Ruan, C. Gomes, B. Selman. *Proceedings of AAAI’02*, 674–682, 2002.
- [12] S. Prestwich and A. Roli. Symmetry Breaking and Local Search Spaces. *Proceedings of CPAIOR’05*, 273–287, 2005.
- [13] J.-F. Puget. Symmetry breaking revisited. *Proceedings of CP’02*, 446–461, 2002.
- [14] C. Roney-Dougal, I. Gent, T. Kelsey, S. Linton. Tractable symmetry breaking using restricted search trees. *Proceedings of ECAI’04*, 211–215, 2004.
- [15] M. Sellmann and P. Van Hentenryck. Structural Symmetry Breaking. *Proceedings of IJCAI’05*, 298–303, 2005.
- [16] P. Van Hentenryck, P. Flener, J. Pearson, M. Agren. Tractable symmetry breaking for CSPs with interchangeable values. *Proceedings of IJCAI’03*, 277–282, 2003.

*Dynamic Symmetry Breaking Restarted*

	15, unbiased, AllDifferent			30, biased, AllDifferent		
	FO	SO	NO	FO	SO	NO
2	1.1K - 14	120 - 15	67 - 15	17K - 151	42 - 547	17 - 723K
3	2.4K - 17	87 - 23	113 - 16	11K - 1.7K	46 - 17K	19 - 3.7M
4	2.4K - 19	83 - 41	53 - 19	35K - 4.9K	51 - 435K	21 - 6.9M
5	3.6K - 38	39 - 716	24 - 892	37K - 7.9K	57 - 4.3M	26 - 14M
6	5.1K - 32	43 - 368	30 - 2708	32K - 11K	65 - 7.6M	31 - 19M
7	6.6K - 140	44 - 35K	30 - 401K	29K - 11K	74 - 6.5M	38 - 15M
8	9.4K - 720	48 - 233K	33 - 2.08M	28K - 6.9K	84 - 2.6M	49 - 13M
9	7.6K - 329	52 - 37K	36 - 766K	25K - 2.9K	104 - 215K	63 - 9.6M
10	6.4K - 319	61 - 7600	40 - 239K	16K - 402	131 - 5.6K	78 - 5.7M
11	3.4K - 43	76 - 132	46 - 12.5K	4.5K - 42	112 - 90	90 - 349K
12	X - 0	X - 0	X - 0	X - 0	X - 0	X - 0

Table 3: Times per choice point in micro seconds and number of search nodes (averages over 50 instances per data point).

	AllDifferent						GCC					
	One-Shot			Restarted			One-Shot			Restarted		
	FO	SO	NO	FR	SR	NR	FO	SO	NO	FR	SR	NR
2	100	100	98	100	100	100	46	62	86	78	100	100
3	100	100	90	100	100	100	36	36	78	46	100	100
4	86	98	76	76	92	100	41	40	55	41	88	47
5	60	72	40	56	56	88	30	30	24	28	54	24
6	52	23	4	54	18	51	14	10	2	12	8	10
7	61	41	2	65	20	9	14	4	2	10	12	6
8	80	84	0	82	52	0	4	0	0	14	0	0
9	96	100	0	98	76	0	24	2	0	34	0	0
10	100	100	36	100	100	0	26	10	0	76	2	82
11	100	100	96	100	100	100	52	50	48	70	100	100
12	100	100	100	100	100	100	100	100	100	100	100	100

Table 4: Histogramm for the benchmark sets with 30 variables and values, 12 variables per constraint, and biased variable selection. The first column gives the number of values per constraint, the numbers in the table the percentage of instances solved within 600 seconds (50 instances per data point).