

Dynamic Symmetry Breaking Restarted

Daniel S. Heller and Meinolf Sellmann

Brown University
Department of Computer Science
115 Waterman Street, P.O. Box 1910
Providence, RI 02912
dheller, sello@cs.brown.edu

Symmetry breaking by dominance detection (SBDD) [4, 6, 1, 12], 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 [7], while the first provably polynomial-time dominance checkers for specific types of value symmetry were devised in [15]. This work was later extended to tackle any kind of value symmetry in polynomial time [13]. Based on these results, for specific “piecewise” symmetric problems, [14] 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 been 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 [9]. One way to circumvent this problematic situation is to randomize the solver and to restart the search when a run takes too long [10]. We show how symmetry no-goods can be used in restarted methods and introduce practical enhancements of SSB for its application in restarted solvers.

1 Symmetry No-Goods and Restarts

Unfortunately, due to space restrictions, we cannot review SSB here. For definitions and a detailed description of the method, we must therefore refer the reader to [14]. 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 a whole set 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.

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, too. However, for symmetry-nogoods we can do more.

2 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 possibly cause effective symmetry-based filtering at a given search node. Like this, we hope to save many of the expensive calls to SSB-based domain filtering. 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 will discuss special methods to improve the performance of sibling-based filtering in Section 3.

2.1 A Simple Pretest

We start by introducing a very simple pretest before a full-fledged ancestor-based filtering call is being made. What we need to identify are simple conditions under which a previously expanded node α (as usual, α is identified with the partial assignment that leads to the node) cannot “almost dominate” the current search node β . To make this more precise, with “almost” we mean to say that one more assignment to β could result to a successful dominance relation with α , which is a necessary condition for SSB filtering to have any effect.

First, we observe that β must contain at least as many variable assignments as α minus 1. 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 can look a little bit closer at the two assignments α and β and see whether α is close to dominating β . 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 Q_l , we determine how many variables in each value partition are taking a value in it under assignment γ , thus computing a signature for each partition of mutually symmetric values: $sign_\gamma(Q_l) := (|\{X \in P_k \mid \gamma(v) \in Q_l\}|)_{k \leq r}$ for all $1 \leq l \leq s$.

Lemma 1. *Given assignments α and β such that α dominates β , we have that, for all $1 \leq l \leq s$, it holds that $sign_\alpha(Q_l) \leq sign_\beta(Q_l)$ (whereby with \leq we denote the component-wise comparison of the two tuples).*

Proof. Let $l \in \{1, \dots, s\}$. Since α dominates β , we have that, for all $v \in Q_l$, it holds $sign_\alpha(v) \leq sign_\beta(w)$ for some value $w \in Q_l$ that is the unique matching partner of v . Consequently, it holds that $sign_\alpha(Q_l) = \sum_{v \in Q_l} sign_\alpha(v) \leq \sum_{w \in Q_l} sign_\beta(w) = sign_\beta(Q_l)$. \square

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 $sign_\alpha(Q_l) \leq sign_\beta(Q_l) + e_k$,

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.

2.2 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 α and β .¹ Clearly, as was already noted in [14], this means that at least another $p - 1$ variable assignments need to be added to β before filtering can become effective. By adding this information to no-good α , and by keeping track of the depth of the current search node β , 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 situation: For no-good α , 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 α 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 α , and this information can be used even when backtracking up from the current position. 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 α can be effective. Consequently, we will not call for ancestor-based filtering wrt α until we reach depth $\max\{d + r + q - 1, d + p - 1\}$ in the search tree.

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 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 α . And in the second case, we can count how many more assignments are necessary before each value class signature in α can have become lower or equal to the signatures in the current partial assignment.

3 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

¹ Note that, due to the special way that SBDD unifies no-goods, $p > 1$ is only possible for no-goods generated in earlier runs.

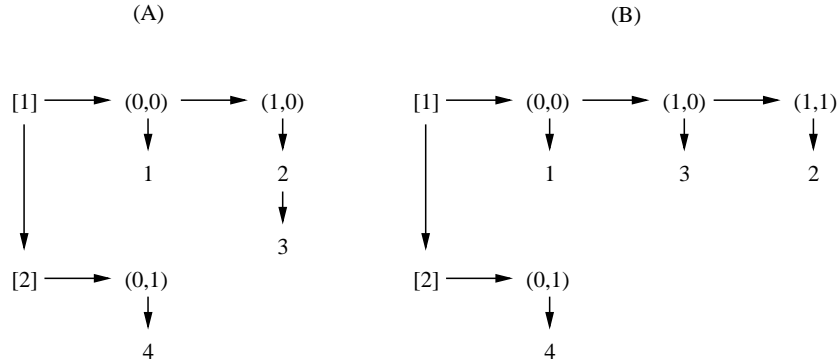


Fig. 1. An efficient data structure supporting sibling-based filtering incrementally. The leftmost column depicts value partitions $[1] = \{1, 2, 3\}$ and $[2] = \{4\}$. For both partitions, horizontally it follows a sorted list of signatures (over two variable partitions in this example) that are each 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. Value 2 in the left assignment for instance has signature $(1,0)$ and shares it with value 3.

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 [14]: 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 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 1 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 1 (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 1 (B) shows the data structure after another variable has been instantiated by adding $(X_4, 2)$ to our assignment. We see that the data structure can easily be adapted by updating the signature of value 2.

4 Conclusions

We introduced two practical algorithmic enhancements of structural symmetry breaking: delayed ancestor-based filtering and incremental sibling-based filtering. Especially the first is designed for the application in restarted solvers where it reduces the computational efforts when using symmetry no-goods from previous restarts.

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 AAEECC-6*, 99–110, 1988.
3. J. Crawford, M. Ginsberg, E. Luks, A. Roy. Symmetry-breaking predicates for search problems. *Proceedings of KR'96*, 149–159, 1996.
4. T. Fahle, S. Schamberger, M. Sellmann. Symmetry Breaking. *Proceedings of CP'01*, 93–107, 2001.
5. 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.
6. F. Focacci and M. Milano. Global cut framework for removing symmetries. *Proceedings of CP'01*, 77–92, 2001.
7. I. Gent, W. Harvey, T. Kelsey, S. Linton. Generic SBDD using computational group theory. *Proceedings of CP'03*, 333–347, 2003.
8. I. Gent and B. Smith. Symmetry breaking in constraint programming. *Proceedings of ECAI'00*, 599–603, 2000.
9. C.P. Gomes, B. Selman, N. Crato. Heavy-Tailed Distributions in Combinatorial Search. *Proceedings of CP'97*, 121–135, 1997.
10. Dynamic Restart Policies. H. Kautz, E. Horvitz, Y. Ruan, C. Gomes, B. Selman. *Proceedings of AAAI'02*, 674–682, 2002.
11. S. Prestwich and A. Roli. Symmetry Breaking and Local Search Spaces. *Proceedings of CPAIOR'05*, 273–287, 2005.
12. J.-F. Puget. Symmetry breaking revisited. *Proceedings of CP'02*, 446–461, 2002.
13. C. Roney-Dougal, I. Gent, T. Kelsey, S. Linton. Tractable symmetry breaking using restricted search trees. *Proceedings of ECAI'04*, 211–215, 2004.
14. M. Sellmann and P. Van Hentenryck. Structural Symmetry Breaking. *Proceedings of IJCAI'05*, 298–303, 2005.
15. P. Van Hentenryck, P. Flener, J. Pearson, M. Agren. Tractable symmetry breaking for CSPs with interchangeable values. *Proceedings of IJCAI'03*, 277–282, 2003.