Symmetry Breaking Constraints for SAT Solving*

Martin Struk[1] *and* Mikoláš Janota[2]

[1] *Charles University, Faculty of Mathematics and Physics, Ke Karlovu 3. 121 16 Praha 2, Czech Republic*
[2] *Czech Technical University in Prague, Jugoslávských partyzánů 1580/3, 160 00 Praha 6-Dejvice, Czech Republic*

Abstract

Symmetries in Boolean satisfiability (SAT) problems often cause redundant exploration of equivalent regions of the search space, significantly degrading solver performance. Symmetry breaking constraints (SBCs) address this issue by eliminating symmetric solutions while preserving satisfiability. In this work, we study the generation of SBCs from known symmetries for a given SAT formula. We present a tool for constructing SBCs using patterns, a compact representation introduced in previous work. The proposed approach generates clauses using selected sets of permutations while allowing control over the depth of generating constraints, providing a trade-off between the strength of symmetry breaking and the size of the resulting formula. Our main focus is an experimental evaluation of how different sets of permutations influence the effectiveness of symmetry breaking. The experiments are conducted on problems naturally representable as binary matrices, but more complex problems may be considered. We evaluate both solving performance and the reduction in the number of symmetric solutions using state-of-the-art SAT solvers.

Keywords

boolean satisfiability, symmetry breaking, constraint generation, permutation groups

1. Introduction

The main result of this work is a tool which, for an input SAT formula, generates symmetry-breaking clauses based on a chosen set of permutations and the depth of generating constraints. With this tool, we aim to experimentally study whether some specific sets of permutations are more effective than others.

We assume that the symmetry type of a given SAT formula is known beforehand, which differs from tools like *BreakID* [1] or *Satsuma* [2], which detect symmetry groups before generating symmetry-breaking clauses. These tools are mentioned in this context because we also aim to investigate a claim that originates from the corresponding works. Specifically, symmetry detection results in a set of permutations which is also a set of generators of the underlying symmetry group. According to these papers, such a set is very effective for breaking symmetries, although it is not clear why. We therefore aim to experimentally examine, for example, whether the property of the set of permutations being a generating set has an impact on the effectiveness of symmetry breaking in general.

Below we provide a summary of the main contributions of this work.

- generalization of the use of patterns by Codish and Janota from graphs to matrix models
- implementation of a tool for generation of symmetry breaking constraints
- systematic evaluation on different types of sets of permutations

2. Background and Main Ideas of the Algorithm

To construct the symmetry-breaking clauses themselves, we first consider the most commonly used approach, namely the *Lex-Leader* constraints [5][8]. The main idea is to select a single

* *Joint Proceedings of the Workshops and Doctoral Consortium of the 42nd International Conference on Logic Programming (ICLP 2026), co-located with the Federated Logic Conference (FLoC 2026), July 18–19 and 24–25, 2026, Lisbon, Portugal.*



© 2026 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

CEUR Workshop Proceedings ceur-ws.org ISSN 1613-0073
published 2026-11-03

CEUR-WS.org/Vol-4311/RCRA_regular_02.pdf

canonical representative from each equivalence class of symmetric solutions. Consequently, only one solution from each class remains feasible and needs to be explored by the SAT solver.

The construction begins by fixing a total ordering on the variables

$$x_1 < x_2 < \cdots < x_n. \tag{1}$$

This ordering naturally induces a lexicographic ordering on complete assignments. For every assignment x and every permutation π representing a symmetry of the formula, we then introduce the constraint

$$x \leq_{lex} \pi(x). \tag{2}$$

The selected canonical representative is therefore the lexicographically smallest assignment within its equivalence class. As a consequence, it is the only assignment in the class satisfying all added constraints, while all other symmetric assignments are eliminated. In principle, this achieves complete symmetry break.

In practice, however, complete symmetry break is often computationally prohibitive due to the large number of generated constraints and the overhead associated with enforcing a full lexicographic ordering. There are potentially $n!$ i.e. superpolynomial number of constraints generated using this method [7]. For this reason, our work focuses primarily on partial symmetry breaks, where constraints are generated only for a selected subset of permutations. As a result, only a subset of all possible *Lex-Leader* constraints is introduced, providing a trade-off between the strength of symmetry reduction and the complexity of the resulting formula. It is worth noting, however, that a properly chosen subset of *Lex-Leader* constraints may achieve complete symmetry break.

Throughout this work, we assume the standard lexicographic ordering in which $0 \leq_{lex} 1$ and variables appearing earlier in the vector of variables are considered more significant than variables appearing later.

Our primary focus is on problems whose solutions can be naturally represented as binary matrices [3]. While all presented techniques can be generalized in a straightforward manner to higher-dimensional structures (for example by flattening), such generalizations typically lead to a substantial increase in the number of variables, symmetries, and consequently in the overall computational complexity of the problem.

For an $m \times n$ matrix A, let $vec(A)$ denote the vector of length mn obtained by concatenating all rows of A. For two $m \times n$ matrices A and B, we define $A \leq B$ if and only if $vec(A) \leq_{lex} vec(B)$. Rather than working directly with already constructed solution matrices, our goal is to enforce the existence of lexicographically minimal solutions during solving. Therefore, instead of binary values, the matrices under consideration contain propositional variables. Each assignment of these variables then determines a concrete binary matrix. The ordering relation defined above consequently induces constraints on the variable assignments, specifying which assignments satisfy the required lexicographic conditions.

We now describe in greater detail how the lexicographic constraints $\leq_{lex}$ are encoded into SAT clauses. For this purpose, we use the notion of *patterns*, introduced in [4] for graphs. We generalize this notion to matrices. Tseitin transformation is then used to translate patterns into CNF [6].

The main idea is that, instead of constructing the constraint $x \leq_{lex} \pi(x)$ directly, we iteratively construct the complementary condition $\pi(x) <_{lex} x$, which is subsequently negated. The iterative construction proceeds by fixing, in each step, a position in the vectors where they differ for the first time. More precisely, we select the first position at which the variable in $\pi(x)$ is strictly smaller than the corresponding variable in x. This immediately implies that all preceding positions in the two vectors must be equal.

These properties allow us to derive the corresponding pattern representation. In this way, we obtain a pattern describing a set of assignments sharing the same structural properties with respect

to the considered permutation. By generating *Lex-Leader* clauses for such a pattern, all represented assignments are covered simultaneously, leaving only the lexicographically smallest representative feasible.

The following example illustrates the construction in detail.

We consider a problem derived from the pigeonhole principle. Given p pigeons and h holes, the goal is to assign pigeons to holes such that each hole contains at most one pigeon. Although this formulation appears trivial, it is widely used as a benchmark for symmetry breaking techniques, since instances with $p > h$ (i.e., UNSAT cases) can be particularly challenging for SAT solvers in the absence of symmetry breaking constraints.

As a concrete example, we consider the case $p = 3$ and $h = 2$. This can be represented as a 3×2 matrix A. here the variable at position (i,j) indicates whether pigeon i is assigned to hole j (value *1*) or not (value *0*).

In this setting, both pigeons and holes are symmetric, and therefore arbitrary permutations of rows and columns yield equivalent solutions. As an example, consider applying the same permutation to both rows and columns, namely swapping the first and second elements. This results in the matrix B.

$$A := \begin{pmatrix} x_1 & x_2 \\ x_3 & x_4 \\ x_5 & x_6 \end{pmatrix}, B := \begin{pmatrix} x_4 & x_3 \\ x_2 & x_1 \\ x_6 & x_5 \end{pmatrix} \tag{3}$$

We now define the vectorizations of these matrices as $a := vec(A) = \begin{pmatrix} x_1 & x_2 & x_3 & x_4 & x_5 & x_6 \end{pmatrix}$, $b := vec(B) = \begin{pmatrix} x_4 & x_3 & x_2 & x_1 & x_6 & x_5 \end{pmatrix}$. Now we will look at each step, where we fix a specific position where the vectors differ for the first time.

- Position 1: This gives us only a single constraint $x_4 < x_1$, which means that $x_4 = 0$ and $x_1 = 1$. We then substitute these constraints into a and we get the pattern $pat_1(\pi) = \begin{pmatrix} 1 & x_2 & x_3 & 0 & x_5 & x_6 \end{pmatrix}$. Here the index *1* denotes the position where the vectors differ for the first time and π denotes the permutation we applied to A in order to get B. The pattern $pat_1(\pi)$ concisely represents all solutions of the constraint $(b =)\ \pi(a) <_{lex} a$. Therefore we cover 2^4 different solutions.
- Position 2: In this case we get two constraints which are $x_4 = x_1$ and $x_3 < x_2$. The pattern we receive after substituting the constraints into a is therefore $pat_2(\pi) = \begin{pmatrix} i & 1 & 0 & i & x_5 & x_6 \end{pmatrix}$ where $i := x_4\ (= x_1)$. Therefore 2^3 different solutions are covered.
- Position 3: Here the constraints are $x_4 = x_1, x_2 = x_3$ *and* $x_2 < x_3$. We can immediately notice that the second and third constraint contradict each other, therefore the pattern $pat_3(\pi)$ is empty in this case.
- Position 4: The idea is hopefully clear by now so we will only list the patterns for the remaining cases. The pattern $pat_4(\pi)$ is empty again because of constraints $x_4 = x_1$ and $x_1 < x_4$.
- Position 5: $pat_5(\pi) = \begin{pmatrix} i & j & j & i & 1 & 0 \end{pmatrix}$ where $i := x_4\ (= x_1)$ and $i := x_3\ (= x_2)$. Therefore 2^2 different solutions are covered.
- Position 6: $pat_6(\pi)$ is empty because of constraints $x_6 = x_5$ and $x_5 < x_6$.

With the procedure from the previous example in place, the idea of the overall algorithm becomes straightforward. For a given problem with specific symmetries, we first select a set of permutations, which we then iterate over. For each permutation, we apply a procedure analogous to the one described in the previous example. The resulting clauses are negated, yielding constraints of the form $x \leq_{lex} \pi(x)$ for all assignments covered by the corresponding pattern.

In the case where permutations are applied separately to rows and columns, as in the previous example, we may choose different sets of row and column permutations and iterate over all possible pairs formed from these sets. Alternatively, permutations can also be applied directly to the vector obtained by vectorizing the matrix representation.

3. Experiments and Results

In this section, we first briefly describe the considered benchmark problems.

The first benchmark is the previously mentioned problem derived from the pigeonhole principle (php). The objective is to place p pigeons into h holes such that each hole contains at most one pigeon. As already discussed, the UNSAT instances of this problem are among the most commonly used benchmarks in the evaluation of symmetry breaking techniques, since they are notoriously difficult for SAT solvers without additional symmetry handling.

The second benchmark is the well-known n-queens problem. In this setting, the input consists of a single parameter n, and the goal is to place exactly n queens on an $n \times n$ chessboard such that no two queens attack each other. This benchmark is particularly interesting for us because symmetric solutions arise from rotations and reflections of the entire chessboard. Consequently, the problem exhibits a structurally different type of symmetry compared to the previously considered matrix permutation symmetries.

The third benchmark is the row-column sums problem (rcsums). The input consists of two lists of values, denoted by $[r_1, \ldots, r_m]$ and $[c_1, \ldots, c_n]$. Each value r_i specifies the required number of ones in the i-th row of the matrix, while each value c_j specifies the required number of ones in the j-th column. Therefore, the number of values r_i determines the number of rows, and the number of values c_j determines the number of columns. A necessary condition is that the total sum of all values r_i equals the total sum of all values c_j. Symmetries occur between rows having the same value r_i, and analogously between columns having the same value c_j. The benchmark instances are generated randomly. We use the parameter *rc* to specify the number of generated values in each list, and the parameter *rcrand* to determine the range from which the individual values r_i and c_j are generated, namely integers from *0* to *rcrand*. Finally, either one additional row or one additional column is added to ensure that the equality of total row and column sums holds. As a result, the generated matrix always has either one more row than columns or one fewer row than columns.

The fourth benchmark problem is the Latin squares problem (latinsq). The input consists of a single parameter n, which we later denote by *lsq* in order to distinguish it from the parameter used in the n-queens problem. The objective is to fill an $n \times n$ matrix with n distinct symbols such that each symbol appears exactly once in every row and exactly once in every column. In our setting, the symbols are represented simply by the integers *1,…,n*. Since our framework operates on binary variables, the Latin squares problem constitutes the first genuinely three-dimensional benchmark considered in this work. Nevertheless, it can be transformed into a two-dimensional representation by flattening the structure. Concretely, the matrices corresponding to the individual symbols are stacked vertically. Instead of an $n \times n \times n$ three-dimensional matrix, we therefore obtain a two-dimensional matrix of size $n^2 \times n$. This problem contains symmetries between rows, columns, and symbols. In the flattened representation, however, care must be taken when applying the corresponding permutations. Column permutations remain unchanged, while row permutations must be applied independently within each $n \times n$ submatrix representing a particular symbol. Similarly, symbol permutations correspond to permutations of the independent $n \times n$ submatrices associated with different symbols.

The fifth and final benchmark is the social golfer problem (golf). The input is given by three parameters g, p, and w, representing respectively the number of groups, the number of golfers in each group, and the number of weeks. The goal is to determine whether it is possible to divide all gp golfers into g groups of size p such that, during each of the w weeks, every golfer participates in exactly one group and no pair of golfers meets in the same group more than once throughout all weeks. This benchmark is again naturally three-dimensional and therefore requires flattening into a two-dimensional representation compatible with our framework. Symmetries arise between all three dimensions, namely groups, weeks, and golfers. Structurally, these symmetries are analogous to those present in the Latin squares problem, with the only difference being that the dimensions are not necessarily equal.

We now describe the contents of Table 1. Better structured plots for several of the experiments are also included below the table.

The table consists of eight columns in total, where the first three columns describe the benchmark instance itself. The first column contains the problem name, the second column specifies the corresponding parameters, and the third column indicates the selected set of permutations. Rows with an empty set of permutations entry correspond to the original benchmark instances without symmetry breaking and serve as a baseline for comparison.

We additionally describe the different permutation sets used in the experiments. The available options are *gen2*, *neighbors*, *nongen*, *randtrans*, *transpositions*, *dihedral_gen*, and *sums_neighbors*. One additional option, not included in Table 1, is *all_perm*. As the name suggests, this configuration considers all possible permutations. As expected, this approach is computationally infeasible for most non-trivial instances, since clause generation becomes excessively expensive and is therefore not practically useful. The set *gen2* consists of two permutations that act as generators of the full symmetry group. For example, over the domain *{1,...,5},* these may take the form of a transposition such as *(1 2)* together with a cyclic permutation *(1 2 3 4 5)*. The set *neighbors* contains transpositions of adjacent elements. For the domain *{1,...,5}*, this corresponds to *(1 2), (2 3), (3 4), (4 5)*. This set also forms a generating set of the symmetric group. The set *nongen* consists of transpositions that do not form a generating set. This is ensured simply by omitting one value from the set of elements that may occur in the selected transpositions. The set *randtrans* contains randomly selected transpositions, which may or may not generate the full symmetry group. The set *transpositions* includes all possible transpositions over the given domain. While it provides full coverage of pairwise swaps, it is typically significantly more expensive computationally. Finally, the sets *dihedral_gen* and *sums_neighbors* are designed specifically for certain benchmark problems. The *dihedral_gen* set is primarily used for the n-queens problem and corresponds to the dihedral group of the chessboard, i.e., rotations and reflections. The *sums_neighbors* set is an adaptation of the *neighbors* construction for the rcsums problem; it applies adjacent-type permutations only within subsets of rows or columns that share the same prescribed sum value.

The remaining five columns contain the experimental results. These columns are divided according to different values of the parameter k. During clause generation, the value k determines how deeply the algorithm explores the variable vectors represented by the patterns. More precisely, k specifies the furthest position at which the vectors are allowed to differ. For example, when $k = 4$, the algorithm proceeds up to the step where the vectors differ at the fourth position and does not continue further, regardless of the total vector length.

Instead of absolute values, the columns are labeled as fractions of the maximal possible value of k. This maximum depends on the specific benchmark instance and typically corresponds to traversing the entire vector. Consequently, the first column roughly represents exploration of the first fifth of the vector, the second column two fifths, and so forth.

Intuitively, constraints generated from deeper positions in the vector are expected to become less effective in terms of symmetry reduction. Variables appearing later in the lexicographic ordering carry lower significance, and many clauses corresponding to deeper positions may already be implicitly covered by constraints generated earlier in the vector. As a result, extending the traversal depth generally yields diminishing returns with respect to the number of eliminated symmetric solutions.

Most table entries contain three values. The first value represents the solving time of the model counter *d4* [9], measured in seconds. The second value denotes the number of solutions for the given benchmark instance. The final value is the preprocessing time required by our tool to generate the symmetry breaking clauses, again measured in seconds. All reported times are rounded to one decimal place.

In some cases, the benchmark instances are too difficult for the solver to finish within the allotted time limit. Such entries are marked as *MTE* ("Maximum Time Exceeded"). During testing, the timeout limit was set to 3600 seconds.

The benchmark instances were additionally solved using the conventional SAT solver *Kissat* [10]. These values appear only in cases where *d4* exceeded the timeout limit while *Kissat* successfully solved the instance. Therefore, if an additional value appears after the *MTE* marker, it represents the runtime of *Kissat*.

Finally, note that for rows corresponding to benchmark instances without symmetry breaking, the columns associated with different values of k are merged together. This is natural, since the parameter k is irrelevant when no symmetry breaking constraints are generated, which can be viewed as corresponding implicitly to the case $k = 0$.

At first glance at the results, we observe a clear improvement both in solving time and in the number of solutions after applying symmetry breaking, which is of course a positive but expected outcome. In most cases, the time required for clause generation is also significantly smaller than the reduction in solving time achieved by applying symmetry breaking. However, this is not always guaranteed. For instance, in the case of the golf instance with parameters g5_p4_w4, the SAT solver *Kissat* solves the problem the fastest without any symmetry breaking. This can occur in situations where the introduction of a large number of auxiliary variables increases the search space of the SAT solver more than the symmetry breaking constraints reduce it.

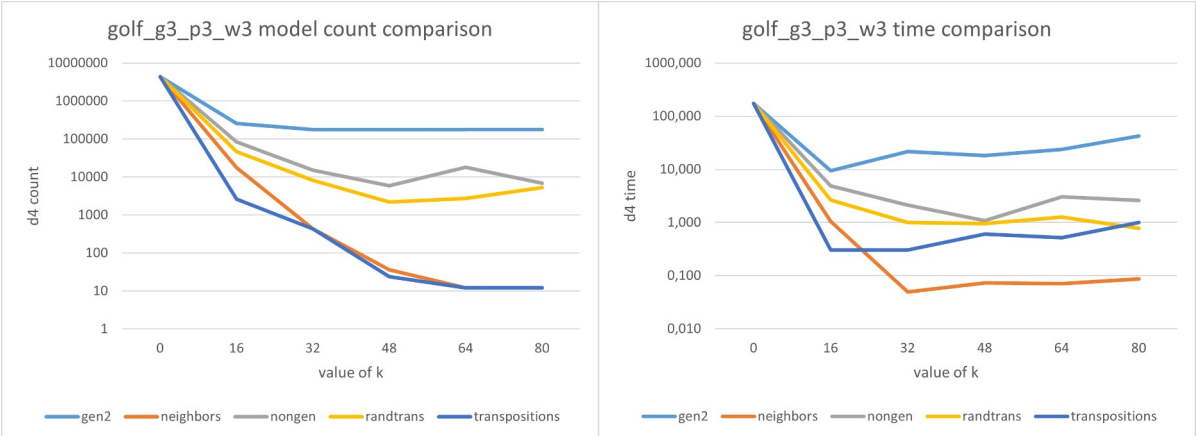
When comparing the generating sets of permutations *gen2*, *neighbors*, and *transpositions*, we observe that simply using a generating set of permutations is not sufficient for effective symmetry breaking. The *gen2* set reduces both the number of solutions and the solving time significantly less than either *neighbors* or *transpositions*. At the same time, however, we also see that using a much larger set of permutations does not necessarily lead to better performance.

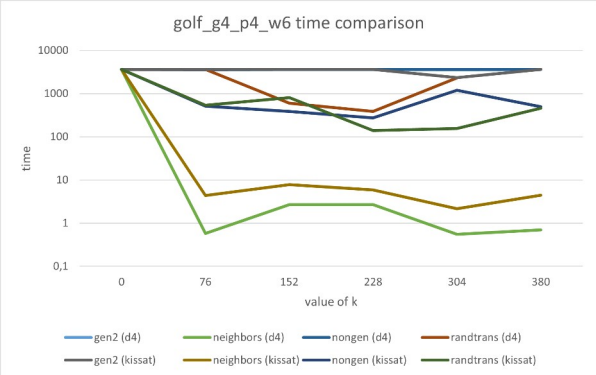
Although *transpositions* contains substantially more elements than *neighbors*, the overall difference in results is often marginal. A notable difference appears in the clause generation time, which is significantly higher for *transpositions* and is one of the reasons why this configuration could not be applied to all benchmark instances. The only case in Table 1 where *transpositions* performs better than *neighbors* in terms of the number of remaining solutions is the latinsq instance with $lsq = 5$, where *transpositions* yields *5* solutions compared to *9* solutions for *neighbors*. However, this improvement comes at the cost of a substantially higher solving time.

Finally, we compare the results for *nongen*, *randtrans*, and *neighbors*. All three configurations consist solely of transpositions and contain the same number of elements. The only difference is therefore whether the given set of permutations forms a generating set or not.

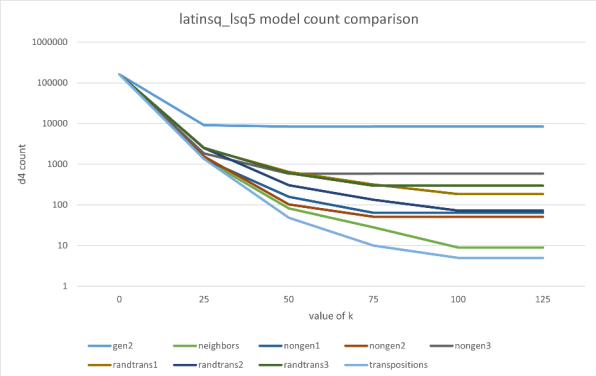
Problem			Time to solve (s); number of solutions; time to gen (s) by values of k				
Name	**Parameters**	**Set of permutations**	**1/5 of max**	**2/5 of max**	**3/5 of max**	**4/5 of max**	**max k**
golf	g3_p3_w3	gen2	9.3; 255168; 0	21.7; 176272; 0	18.1; 176214; 0	23.9; 176212; 0	42.5; 176212; 0.1
golf	g3_p3_w3	neighbors	1; 17712; 0	0; 432; 0	0.1; 36; 0.1	0.1; 12; 0.1	0.1; 12; 0.1
golf	g3_p3_w3	nongen	4.9; 82968; 0	2.1; 14976; 0.1	1.1; 5818; 0.1	3; 18058; 0.1	2.6; 6916; 0.1
golf	g3_p3_w3	randtrans	2.7; 46656; 0	1; 8091; 0	1; 2175; 0.1	1.3; 2724; 0.1	0.8; 5242; 0.2
golf	g3_p3_w3	transpositions	0.3; 2592; 0.1	0.3; 432; 0.3	0.6; 24; 0.6	0.5; 12; 0.9	1; 12; 1.1
golf	g3_p3_w3		173.1; 4354560; 0				
golf	g4_p4_w6	gen2	MTE	MTE	MTE	MTE; 2342.8	MTE
golf	g4_p4_w6	neighbors	0.6; 0; 0.5	2.7; 0; 1.8	2.7; 0; 3.8	0.6; 0; 7.6	0.7; 0; 11.3
golf	g4_p4_w6	nongen	MTE; 510	MTE; 384.7	MTE; 271.7	MTE; 1190.8	MTE; 496.3
golf	g4_p4_w6	randtrans	MTE; 540.2	MTE; 799.7	596.3; 0; 4	383.9; 0; 7.5	2301.2; 0; 11.4
golf	g4_p4_w6		MTE				
golf	g5_p4_w4	gen2	MTE; 5	MTE; 5.4	MTE; 2.8	MTE; 3.1	MTE; 3
golf	g5_p4_w4	neighbors	MTE; 4.9	MTE; 5	MTE; 2.7	MTE; 5.5	MTE; 5.2
golf	g5_p4_w4		MTE; 1.9				
latinsq	lsq5	gen2	0.2; 9084; 0	0.5; 8396; 0	0.7; 8390; 0	1.3; 8390; 0.1	0.9; 8390; 0.2
latinsq	lsq5	neighbors	0.1; 1344; 0	0.1; 82; 0.1	0.1; 28; 0.2	0.2; 9; 0.3	0.2; 9; 0.4
latinsq	lsq5	nongen1	0.3; 1344; 0	0.1; 158; 0.1	0.1; 64; 0.2	0.2; 64; 0.3	0.2; 64; 0.4
latinsq	lsq5	nongen2	0.1; 1560; 0	0.1; 102; 0.1	0.2; 51; 0.2	0.2; 51; 0.3	0.2; 51; 0.5
latinsq	lsq5	nongen3	0.4; 1842; 0.1	0.9; 589; 0.1	1.3; 585; 0.2	1.2; 585; 0.4	1.6; 585; 0.5
latinsq	lsq5	randtrans1	0.1; 2472; 0	0.2; 644; 0.1	0.3; 316; 0.2	0.2; 184; 0.3	0.4; 184; 0.4
latinsq	lsq5	randtrans2	0.1; 2496; 0	0.1; 304; 0.1	0.3; 134; 0.2	0.3; 72; 0.3	0.4; 72; 0.4
latinsq	lsq5	randtrans3	0.2; 2548; 0.1	0.3; 604; 0.1	0.6; 297; 0.2	0.6; 297; 0.3	1; 297; 0.4
latinsq	lsq5	transpositions	0.3; 1344; 0.4	1.1; 48; 1.2	2.7; 10; 2.3	6.9; 5; 3.8	3.5; 5; 6.3
latinsq	lsq5		4.2; 161280; 0				
latinsq	lsq6	gen2	1411.1; 40100544; 0	861.7; 40069124; 0.1	2850; 40068967; 0.2	2822.9; 40068955; 0.4	2877.1; 40068955; 0.5
latinsq	lsq6	neighbors	6.1; 179712; 0.1	0.4; 6680; 0.3	0.3; 2300; 0.8	0.3; 612; 1.2	0.4; 612; 1.8
latinsq	lsq6	nongen	24.3; 889660; 0.2	5.6; 125748; 0.5	15.6; 209948; 0.8	17.4; 358175; 1.3	12.3; 112505; 1.8
latinsq	lsq6	randtrans	24.7; 744000; 0.1	26.4; 586420; 0.3	11.4; 346540; 0.8	23.1; 250172; 1.2	32.4; 331459; 1.8
latinsq	lsq6		MTE; 0				
php	pig21_hol20	gen2	MTE	MTE	MTE	MTE	MTE
php	pig21_hol20	neighbors	MTE	MTE	0.5; 0; 0.6	0.1; 0; 0.9	0.1; 0; 1.3
php	pig21_hol20		MTE				
php	pig31_hol30	gen2	MTE	MTE	MTE	MTE	MTE
php	pig31_hol30	neighbors	MTE	MTE	MTE; 2.6	0,2; 0; 4,9	0,1; 0; 8,3
php	pig31_hol30		MTE				
nqueens	n10	dihedral_gen	0.4; 255; 0	0.3; 251; 0	0.5; 251; 0	0.5; 251; 0	0.4; 251; 0
nqueens	n10		0.5; 724; 0				
nqueens	n12	dihedral_gen	4.4; 5193; 0	6.7; 5174; 0	7.4; 5173; 0	4.4; 5173; 0	8.6; 5173; 0
nqueens	n12		5.7; 14200; 0				
rcsums	rc10_rcrand2	sums_neighbors	0.7; 237828; 0	0.1; 11679; 0	0.1; 370; 0	0.1; 99; 0	0.1; 87; 0.1
rcsums	rc10_rcrand2		2.5; 3106080; 0				
rcsums	rc15_rcrand16	sums_neighbors	0.1; 0; 0	0.1; 0; 0	0.1; 0; 0	0.1; 0; 0	0; 0; 0
rcsums	rc15_rcrand16		0; 0; 0				

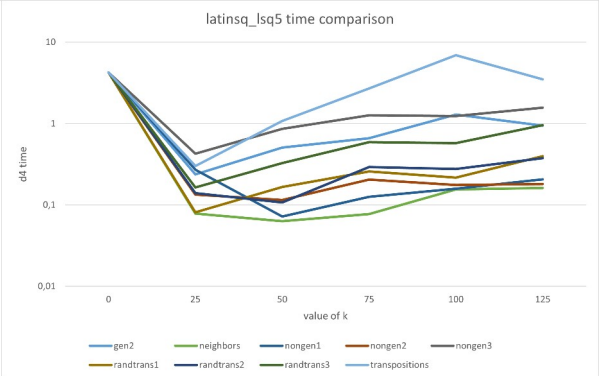
Table 1: Results of experiments

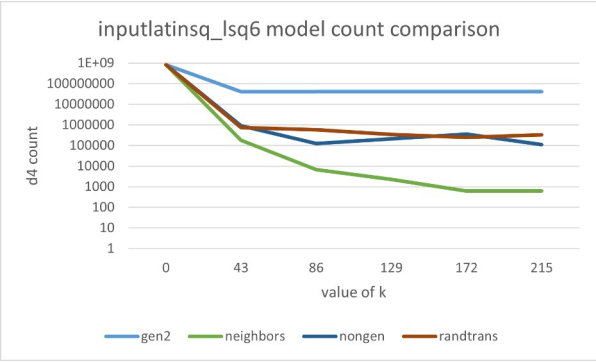


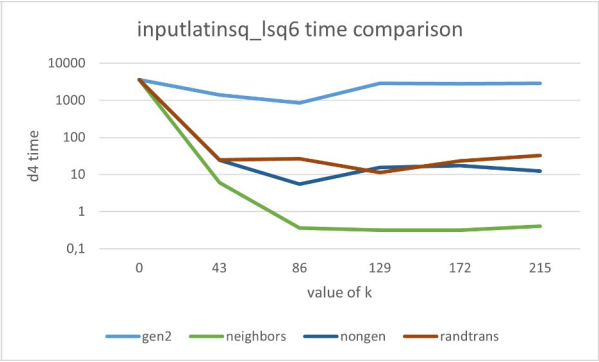

golf_g4_p4_w6 time comparison
time
value of k
gen2 (d4)
neighbors (d4)
nongen (d4)
randtrans (d4)
gen2 (kissat)
neighbors (kissat)
nongen (kissat)
randtrans (kissat)

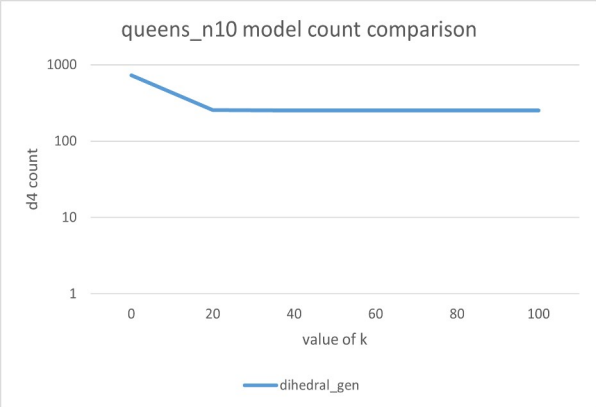

inputgolf_g5_p4_w4 time comparison
kissat time
value of k
gen2
neighbors

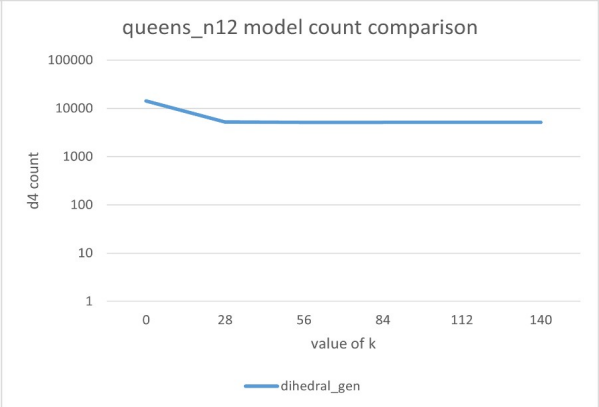

latinsq_lsq5 model count comparison
d4 count
value of k
gen2
neighbors
nongen1
nongen2
nongen3
randtrans1
randtrans2
randtrans3
transpositions

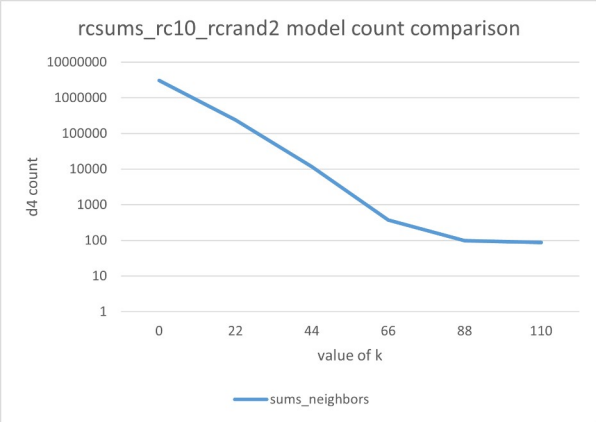

latinsq_lsq5 time comparison
d4 time
value of k
gen2
neighbors
nongen1
nongen2
nongen3
randtrans1
randtrans2
randtrans3
transpositions


inputlatinsq_lsq6 model count comparison
d4 count
value of k
gen2
neighbors
nongen
randtrans


inputlatinsq_lsq6 time comparison
d4 time
value of k
gen2
neighbors
nongen
randtrans


queens_n10 model count comparison
d4 count
value of k
dihedral_gen


queens_n12 model count comparison
d4 count
value of k
dihedral_gen



This comparison should therefore provide an indication of whether or not this property contributes to the reduction of symmetric solutions and to the resulting solver performance, reflecting the experimental aim stated in Section 1. We must distinguish two cases in the experimental setup. For the latinsq instance with *lsq = 5*, there are three distinct configurations for both *nongen* and *randtrans*. In these cases, a single fixed set of permutations of the given type is used for all values of k within a single row. For the remaining benchmark problems, however, *nongen* and *randtrans* are regenerated independently for each table entry. As a consequence, it may happen that for a fixed row the result for higher k is worse than for lower k, since the underlying set of permutations changes between runs. This effect does not occur in the fixed-permutation case.

Overall, when comparing these three configurations, it is immediately apparent that *neighbors* is by far the most effective. The differences between *nongen* and *randtrans* are less pronounced; however, *randtrans*, which may or may not form a generating set, appears to perform better more frequently. This suggests that being a generating set may have a positive impact on the effectiveness of symmetry breaking.

Nevertheless, the observed variance in the results reveals several interesting effects that warrant further investigation. These findings suggest that the behavior of *randtrans* instances depends in meaningful ways on whether they form generating sets, motivating a more systematic evaluation of this distinction in future work.

Acknowledgements

This short paper is part of the master's thesis work of Martin Struk supervised by Mikoláš Janota.

Declaration on Generative AI

During the preparation of this work, the author(s) used ChatGPT in order to: Grammar and spelling check, Paraphrase and reword, Text Translation. After using these tool(s)/service(s), the author(s) reviewed and edited the content as needed and take(s) full responsibility for the publication's content.

References

[1] Devriendt, Jo; Bogaerts, Bart. BreakID: Static Symmetry Breaking for ASP (System Description). 2016. Available from arXiv: 1608.08447 [cs.AI].

[2] Anders, Markus; Brenner, Sofia; Rattan, Gaurav. satsuma: Structure-based Symmetry Breaking in SAT. CoRR. 2024, vol. abs/2406.13557. Available from DOI: 10.48550/ARXIV.2406.13557.

[3] Flener, Pierre; Frisch, Alan; Hnich, Brahim; Kiziltan, Zeynep; Miguel, Ian; Pearson, Justin; Walsh, Toby. Breaking Row and Column Symmetries in Matrix Models. In: 2002. ISBN 978-3-540-44120-5. Available from DOI: 10.1007/3-540-46135-3_31.

[4] Codish, Michael; Janota, Mikoláš. Breaking Symmetries from a Set-Covering Perspective. 2025. Available from arXiv: 2502.10056 [cs.LO].

[5] Gent, Ian P.; Petrie, Karen E.; Puget, Jean-François. Symmetry in Constraint Programming. In: Rossi, Francesca; Beek, Peter van; Walsh, Toby (ed.). Handbook of Constraint Programming. Elsevier, 2006, vol. 2, p. 329–376. Foundations of Artificial Intelligence. Available from DOI: 10.1016/S1574-6526(06)80014-3.

[6] Tseitin, G. S. On the complexity of derivations in the propositional calculus. Studies in Constructive Mathematics and Mathematical Logic. 1968.

[7] Haken, Armin. The Intractability of Resolution. Theor. Comput. Sci. 1985, vol. 39, p. 297–308.

[8] Rossi, Francesca. Handbook of constraint programming. Ed. Peter, Van Beek; Rossi, Francesca; Van Beek, Peter; Walsh, Toby. Elsevier Science & Technology, 2014. Foundations of Artificial Intelligence.

[9] Lagniez, Jean-Marie; Marquis, Pierre. An Improved Decision-DNNF Compiler. In: Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI-17. 2017, p. 667–673. Available from DOI: 10.24963/ijcai.2017/93.

[10] Biere, Armin; Faller, Tobias; Fazekas, Katalin; Fleury, Mathias; Froleyks, Nils; Pollitt, Florian. CaDiCaL, Gimsatul, IsaSAT and Kissat Entering the SAT Competition 2024. In: Heule, Marijn; Iser, Markus; Järvisalo, Matti; Suda, Martin (ed.). Proc. of SAT Competition 2024 – Solver, Benchmark and Proof Checker Descriptions. University of Helsinki, 2024, vol. B-2024-1, p. 8–10. Department of Computer Science Report Series B. Available from: http://hdl.handle.net/10138/584822.

A. Online Resources

The complete set of results from experiments conducted for this paper can be found here: https://people.ciirc.cvut.cz/~janotmik/pats/mc_3600_50000_t4.html

More detailed and better structured plots of the data can be also found here: http://people.ciirc.cvut.cz/~janotmik/rcra26