Satsuma: Structure-based Symmetry Breaking in SAT

MARKUS ANDERS, RPTU Kaiserslautern-Landau, Germany
SOFIA BRENNER, Charles University, Czech Republic
GAURAV RATTAN, University of Twente, The Netherlands

Symmetry reduction is crucial for solving many interesting SAT instances in practice. Numerous approaches have been proposed, which try to strike a balance between symmetry reduction and computational overhead. Arguably the most readily applicable method is the computation of static symmetry breaking constraints: a constraint restricting the search-space to non-symmetrical solutions is added to a given SAT instance. A distinct advantage of static symmetry breaking is that the SAT solver itself is not modified. A disadvantage is that the strength of symmetry reduction is usually limited. In order to boost symmetry reduction, the state-of-the-art tool BreakID (Devriendt et al., 2016) pioneered the identification and tailored breaking of a particular substructure of symmetries, the so-called row interchangeability groups.

In this paper, we propose a new symmetry breaking tool called satsuma. The core principle of our tool is to exploit more diverse but frequently occurring symmetry structures. This is enabled by new practical detection algorithms for row interchangeability, row-column symmetry, Johnson symmetry, and various combinations. Based on the resulting structural description, we then produce symmetry breaking constraints.

We provide benchmarks testing the effectiveness of our new implementation in conjunction with the state-of-the-art SAT solver Kissat. To this end, we compare satsuma, BreakID, and using no symmetry breaking. We find that satsuma successfully speeds up Kissat on last year's SAT competition instances. Compared to BreakID, we observe significantly better breaking performance on instances with Johnson symmetry, and lower computational overhead across all tested families.

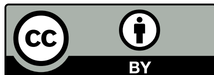
JAIR Associate Editor: Chu-Min Li

JAIR Reference Format:
Markus Anders, Sofia Brenner, and Gaurav Rattan. 2026. Satsuma: Structure-based Symmetry Breaking in SAT. *Journal of Artificial Intelligence Research* 85, Article 6 (January 2026), 31 pages. DOI: 10.1613/jair.1.18744

1 Introduction

Symmetries are present in many interesting SAT instances, ranging from hard combinatorial problems to circuit design. Making use of symmetry is paramount in order to efficiently solve many of these instances. Practical approaches for symmetry reduction must always strike a balance between the computational overhead incurred and the strength of the symmetry reduction. Two decades of research have led to many approaches to tackle this problem (Aloul, Markov, et al. 2003; Crawford et al. 1996; Devriendt, Bogaerts, and Bruynooghe 2017; Devriendt, Bogaerts, Bruynooghe, and Denecker 2016; Devriendt, Bogaerts, Cat, et al. 2012; Junttila, Karppa, et al. 2020; Kirchweger and Szeider 2021; Metin et al. 2018; Sabharwal 2009). At one end of the spectrum, isomorph-free generation techniques (Junttila, Karppa, et al. 2020; Kirchweger and Szeider 2021) apply sophisticated algorithms in conjunction with the solver, such that a solver only explores asymmetric branches of the search. While these techniques are successful in solving hard combinatorial instances (e.g., (Kirchweger, Scheucher, et al. 2022)), this comes at the price of substantial overhead: both in terms of computational cost as well as interfering with

Authors' Contact Information: Markus Anders, ORCID: 0009-0004-5992-8433, anders@cs.uni-kl.de, RPTU Kaiserslautern-Landau, Kaiserslautern, Germany; Sofia Brenner, ORCID: 0009-0006-8512-2569, sofia@kam.mff.cuni.cz, Charles University, Prague, Czech Republic; Gaurav Rattan, ORCID: 0000-0002-5095-860X, g.rattan@utwente.nl, University of Twente, Enschede, The Netherlands.



This work is licensed under a Creative Commons Attribution International 4.0 License.

© 2026 Copyright held by the owner/author(s).
DOI: 10.1613/jair.1.18744

the other strategies employed by solvers. Hence, one must be sure that the symmetry reduction is worth the additional overhead. It therefore seems impractical to turn these techniques "on-by-default".

Arguably at the other end are tools producing *static symmetry breaking constraints* (Aloul, Markov, et al. 2003; Crawford et al. 1996; Devriendt, Bogaerts, Bruynooghe, and Denecker 2016). These tools add additional clauses and variables to a given instance, with the aim of reducing the number of symmetric branches explored by the solver. While the symmetry reduction is usually not as strong as for dynamic techniques, such constraints can be computed comparatively cheaply. More importantly, a distinct advantage of static symmetry breaking constraints is that the SAT solver itself is not modified, and hence there is a complete separation of concerns. State-of-the-art static symmetry breaking tools are successfully applied as an "on-by-default" technique (Bogaerts, Nordström, et al. 2023; Devriendt, Bogaerts, Bruynooghe, and Denecker 2016). Static symmetry breaking is not only used in SAT, but also in various other areas of constraint programming (Audemard et al. 2007; Devriendt and Bogaerts 2016; Gent et al. 2006; Pfetsch and Rehn 2019).

In order to improve symmetry reduction, a rather recent development in static symmetry breaking is to detect and make use of so-called *row interchangeability* subgroups (Devriendt, Bogaerts, Bruynooghe, and Denecker 2016; Pfetsch and Rehn 2019). In SAT, this feature was introduced by the state-of-the-art symmetry breaking tool BreakID (Devriendt, Bogaerts, Bruynooghe, and Denecker 2016), but it is also used in symmetry breaking in mixed integer programming (Pfetsch and Rehn 2019). Row interchangeability groups stem from a natural modeling of the variables as a matrix in which all *rows* are interchangeable by a symmetry. The idea is to first identify these row interchangeability groups, and then produce tailored symmetry breaking constraints. The current generation of tools identifies row interchangeability by hoping for and exploiting a particular structure in the generators of the symmetry group. However, the method is not guaranteed to work and sometimes incurs significant overhead (Anders, Schweitzer, and Soos 2023). Despite this, the gain in symmetry reduction seems to be worth the trade-off (Devriendt, Bogaerts, Bruynooghe, and Denecker 2016).

In the realm of constraint programming, symmetry breaking constraints for more structures have been considered: for example, *row-column symmetry* (Flener, Frisch, Hnich, Kiziltan, et al. 2002) is a natural extension of row interchangeability, where both the rows and columns are interchangeable. These symmetries are common in combinatorics, scheduling, or assignment problems (Flener, Frisch, Hnich, Kiziltan, et al. 2002; Flener, Frisch, Hnich, Kızıltan, et al. 2001), such as the well-known pigeonhole principle. While traditional complete symmetry breaking constraints are unlikely to be efficiently computable for these structures (Flener, Frisch, Hnich, Kiziltan, et al. 2002), different practical constraints are well-studied in the literature for *manual* breaking of symmetries (Flener, Frisch, Hnich, Kiziltan, et al. 2002; Katsirelos et al. 2010). Another area in which symmetry breaking has been studied in detail is graph generation (Codish, Gange, et al. 2016; Codish, Miller, et al. 2019; Kirchweger and Szeider 2021). A typical problem in this area is to determine the existence of a graph with a specific property. Symmetries in these problems often simply correspond to *isomorphic graphs*. Even though this is rarely mentioned explicitly in symmetry breaking literature, such symmetries can be described by so-called *Johnson groups* (Babai 2016; Babai et al. 1987).

In *automated* symmetry breaking, making use of such results requires us to *identify* the appropriate structures first. However, generalizing the existing identification strategies of contemporary tools to more elaborate structures seems elusive.

1.1 Contribution

We present a new algorithm for the generation of symmetry breaking constraints, and a prototype implementation called satsuma. The overarching goal is to achieve a better trade-off for symmetry breaking as a general-purpose SAT preprocessing step. This means, we want to compute *more effective* breaking constraints in *less time*.

Our contributions to achieve this are two-fold. First, we explore whether the approach of "identifying and exploiting specific group structures" can be pushed further, to achieve better breaking performance. Second, we implement various algorithmic efficiency improvements.

New Techniques. We place the identification of specific symmetry groups at the very heart of SATSUMA. The approach is enabled by our main contribution, a new class of practical detection algorithms. In particular, we provide algorithms identifying row interchangeability (Section 3.1), row-column symmetry (Section 3.2), and Johnson symmetry (Section 3.3). Furthermore, we detect certain *combinations* of the above groups, as well as groups which are *similar* to the above groups, building essentially a *structural description* of the group. Symmetry breaking constraints are then chosen based on the type of detected structure: for each detected structure, we determine a set of carefully chosen symmetries, for which conventional symmetry breaking constraints are produced.

Our detection algorithms are all based on the highly efficient *individualization-refinement* framework, as is commonly used in practical graph isomorphism algorithms (McKay and Piperno 2014). Our detection algorithms are all heuristics, in that identification of a particular group cannot be guaranteed. However, the success of the heuristics provably depends only on a well-studied graph property (see Section 2).

These algorithms can be applied *without* computing the symmetries of the formula first: they are purely graph-based. We exploit this by first running our tailored detection algorithms, and then only apply general-purpose symmetry detection on parts not yet identified.

Efficiency Improvements. The second focus of SATSUMA is to reduce the computational overhead incurred by symmetry breaking. This is achieved in four main ways.

1. We use a compact graph encoding that is adapted from a classic encoding introduced in (Aloul, Ramani, et al. 2003).
2. Before symmetry breaking is applied, we use "symmetry-preserving" SAT preprocessing techniques, as suggested in (Anders 2022). Specifically, we apply the unit and pure literal rules.
3. We use the graph isomorphism solver DEJAVU (Anders and Schweitzer 2021) for general-purpose symmetry detection.
4. After symmetries have been detected, most algorithms in the SATSUMA implementation are carefully designed to run in quasi-linear time in the encoding size of the generating set and the input formula, as suggested in (Anders, Schweitzer, and Soos 2023).

Benchmarks. We provide benchmarks testing the effectiveness of the resulting implementation in conjunction with the state-of-the-art SAT solver KISSAT (Biere et al. 2024). Specifically, we compare the use of no symmetry breaking, SATSUMA, and BREAKID on a range of SAT instance families. In our benchmarks, we observe that SATSUMA in combination with KISSAT

1. is the fastest configuration on SAT competition instances, in particular improving the average time over KISSAT by 12% and improving the PAR2 score by 13%,
2. severely outperforms the other configurations on instances with Johnson symmetries,
3. and incurs less computational overhead than BREAKID across *all* tested benchmark families. In particular, we observe a 77% reduction on the SAT competition instances and better asymptotic scaling on some benchmark families.

In an ablation study, we confirm that our new special detection algorithms are crucial in achieving the above results.

In summary, we introduce the new symmetry breaking tool SATSUMA that is based on new algorithmic ideas. Our benchmarks provide evidence that it significantly outperforms the previous state-of-the-art in symmetry breaking, and is able to speed up state-of-the-art SAT solvers in a general-purpose setting.

While the core techniques remain the ones presented in the extended abstract (Anders, Brenner, et al. 2024a), we added multiple examples illustrating the symmetries that are detected, an extensive benchmark section (see Section 5) as well as several improvements in the graph encoding (see Section 2.2) and in the implementation itself (see Section 4).

2 Preliminaries

We begin by introducing important preliminaries. First, we discuss Boolean formulas and their symmetries. We continue by modeling symmetries of formulas using graphs. Then, we introduce important tools in order to detect symmetries, namely the individualization-refinement framework. Lastly, we discuss particular group structures that can be found in Boolean formulas.

2.1 Satisfiability and Symmetry

SAT.. In this paper, a Boolean formula F in *conjunctive normal form* (CNF) is denoted by

$$F = \{\{l_{1,1}, \cdots, l_{1,k_1}\}, \cdots, \{l_{m,1}, \cdots, l_{m,k_m}\}\}.$$

Each element $C \in F$ is called a *clause*, whereas a clause itself consists of a set of *literals*. A literal is either a variable v or its negation $\neg v$. We sometimes write $\overline{v}$ in place of $\neg v$ for brevity. We write $\mathrm{Var}(F) \coloneqq \{v_1, \dots, v_n\}$ for the set of *variables* of F and use $\mathrm{Lit}(F)$ for its literals. Note that by our definition, a CNF does not contain duplicate clauses. This assumption is important for many of the statements in this paper. Our implementation removes duplicate clauses in one of its first steps (see Section 4).

A *symmetry*, or *automorphism*, of F is a permutation $\varphi\colon \mathrm{Lit}(F) \to \mathrm{Lit}(F)$ satisfying the following properties.

(1) It maps F to itself, i.e., $F^\varphi = F$, where F^φ means applying φ element-wise to the literals in each clause.
(2) For all $l \in \mathrm{Lit}(F)$ it holds that $\neg\varphi(l) = \varphi(\neg l)$.

We define the *support* of φ as

$$\mathrm{supp}(\varphi) = \{l \in \mathrm{Lit}(F)\colon l^\varphi \neq l\},$$

i.e., the set of all literals moved by φ. The set of all symmetries of F is $\mathrm{Aut}(F)$. Observe that it is easy to efficiently *test* whether a given permutation φ is indeed an automorphism of F: for each clause $C \in F$, we check whether $C^\varphi \in F$ holds.

An *assignment* of F is a function $\theta\colon \mathrm{Var}(F) \to \{0, 1\}$. We define the evaluation of F under θ in the usual way, i.e., either $F[\theta] = 1$ or $F[\theta] = 0$ holds. A formula F is *satisfiable* if there exists an assignment θ with $F[\theta] = 1$, and *unsatisfiable* otherwise. Next, we formally define how assignments and automorphisms are composed. Given an assignment θ of F, an automorphism $\varphi \in \mathrm{Aut}(F)$ and $v \in \mathrm{Lit}(F)$, we define $\theta^\varphi(v) \coloneqq \theta(v')$ if $\varphi(v) = v'$ for $v' \in \mathrm{Var}(F)$ and $\theta^\varphi(v) \coloneqq \neg\theta(v')$ if $\varphi(v) = \neg v'$ for $v' \in \mathrm{Var}(F)$, where naturally $\neg 0 = 1$ and $\neg 1 = 0$ hold. It follows readily that for $\varphi \in \mathrm{Aut}(F)$, we have

$$F[\theta] = F^\varphi[\theta^\varphi] = F[\theta^\varphi].$$

Symmetry Breaking Constraints. All symmetry breaking constraints in this paper are so-called *lex-leader* constraints. Such constraints may falsify solutions, but always guarantee that if a solution exists, a solution that is the lexicographic leader within its class of symmetric assignments is preserved in the resulting formula.

Let $\prec$ denote a total order of $\mathrm{Var}(F)$. We order an assignment θ according to $\prec$, yielding a $\{0, 1\}$-string. We can then order assignments θ, θ' of F lexicographically by comparing their corresponding strings, denoted by $\prec_{\mathrm{lex}}$. Given an automorphism φ of F, it suffices to evaluate F on those assignments θ for which $\theta^\varphi \preceq_{\mathrm{lex}} \theta$ holds, since $F[\theta^\varphi] = F[\theta]$. In particular, we may add a *lex-leader constraint* $\mathrm{LL}^\prec_\varphi$ to F, which ensures that $\theta^\varphi \preceq_{\mathrm{lex}} \theta$ holds. It is easy to see that F is satisfiable if and only if $F \bigwedge_{\varphi \in \mathrm{Aut}(F)} \mathrm{LL}^\prec_\varphi$ is satisfiable (Sakallah 2021). Clearly, since the symmetric group has $n!$ elements, always encoding this full lex-leader constraint is infeasible in practice.

Lex-leader constraints can be efficiently encoded as a CNF formula, and different encodings have been studied in detail (Aloul, Markov, et al. 2003; Devriendt, Bogaerts, Bruynooghe, and Denecker 2016). The practical encoding we use is the same that BreakID uses. This encoding is discussed in detail in (Devriendt, Bogaerts, Bruynooghe, and Denecker 2016). The main focus of this paper lies in *efficiently* determining a *favorable variable order* and a *set of generators*, for which then standard lex-leader constraints are constructed (see Section 4).

2.2 Model Graphs

In this section, we model the input formulas as graphs, such that the automorphism group of the formula is tightly linked to the automorphism group of the graph encoding.

Graphs. An undirected graph $G = (V, E)$ consists of a vertex set V and an edge relation $E \subseteq \binom{V}{2}$. We refer to the set of vertices of G as $V(G)$, and to the set of edges as $E(G)$. A *vertex coloring* π of G is a map $\pi\colon V(G) \to [k]$ for a finite set of *colors* $[k]$. We call (G, π) a *vertex-colored* graph. The *color class* C of a color c consists of all vertices of G with color c, i.e., $C \coloneqq \pi^{-1}(c)$ holds. Naturally, the color classes form a partition of $V(G)$, the *color partition* corresponding to π.

A bijection $\varphi : V(G) \to V(G)$ is called an *automorphism* or *symmetry* of (G, π), whenever

$$(G, \pi)^\varphi = (G^\varphi, \pi^\varphi) = (G, \pi)$$

holds. Here, G^φ denotes the graph with vertex set $V(G)$ and edges $\{u^\varphi, v^\varphi\}$ whenever $\{u, v\}$ is an edge of G (where v^φ simply denotes the image of v under φ). The coloring π^φ is given by $\pi^\varphi(v) = \pi(v^\varphi)$ for every $v \in V(G)$. The set of all automorphisms, or, the *automorphism group*, of (G, π) is denoted by $\mathrm{Aut}(G, \pi)$.

For a given CNF formula F, we want to compute the symmetries of F by modeling it as a graph whose symmetries are precisely the symmetries of F. Let us formalize this notion. We call a graph $G(F)$ a *model graph* for a CNF formula F, if and only if

(1) $\mathrm{Lit}(F) \subseteq V(G(F))$,
(2) the groups $\mathrm{Aut}(G(F))$ and $\mathrm{Aut}(F)$ are isomorphic, and $\mathrm{Aut}(G(F))|_{\mathrm{Lit}(F)} = \mathrm{Aut}(F)$

hold.

We begin with the following basic construction for a model graph $G_{\text{basic}}(F)$. The vertex set V consists of the literals and clauses of F. There are edges connecting the literals of a common variable to each other. Clauses are connected to the literals they contain. Formally, let

$$E \coloneqq \{\{v, \neg v\} : v \in \mathrm{Var}(F)\} \cup \{\{C, l\} : l \in C,\ C \in F\}.$$

Define a coloring π by setting $\pi(l) \coloneqq 1$ for all literals $l \in \mathrm{Lit}(F)$ and $\pi(C) \coloneqq 2$ for all clauses $C \in F$. It is well-known that the automorphisms of $G_{\text{basic}}(F)$ restricted to $\mathrm{Lit}(F)$ are precisely the automorphisms of F (Sakallah 2021).

In practice, we apply optimizations to the encoding, which are described below.

Binary Clauses as Edges. Let us first consider an optimization for *binary clauses*. Our optimization for binary clauses is a classic encoding described in (Aloul, Ramani, et al. 2003), the so-called *MIN3C encoding*. It should be mentioned that (Aloul, Ramani, et al. 2003) describes further optimizations of this encoding. However, the final encoding as used in practice in (Aloul, Ramani, et al. 2003) may cause spurious symmetry that requires additional handling. Here, spurious symmetry means that there may be symmetries of the graph which are *not* symmetries of the CNF.

We now describe the MIN3C encoding of (Aloul, Ramani, et al. 2003). The goal is to reduce the size of the graph by modeling binary clauses as edges instead of vertices.

If a variable v occurs in a binary clause, we add a *variable vertex* $\dot{v}$ and connect it to its literal vertices v and $\overline{v}$. We color all variable vertices with color 3. We remove the edge connecting v and $\overline{v}$. Finally, for each binary clause

Fig. 1. Reduction of binary clause vertices in the model graph of the CNF formula $\{\{a, b\}, \{\overline{a}, c\}, \{\overline{b}, \overline{c}\}, \{\overline{a}, \overline{c}\}\}$.

Fig. 2. Reduction of ternary clause vertices in the model graph of the CNF formula $\{\{a, b, c\}, \{a, b, \overline{c}\}\}$. The tuple $\{a, b\}$ is used for the reduction. Note that it occurs strictly more often than all other tuples occurring in the two clauses.

C with $C = \{v, l\}$ or $C = \{\overline{v}, l\}$, we remove the clause vertex C and its incident edges, and add the edge connecting the literal vertices v (or $\overline{v}$) and l directly. We denote the resulting graph by $G_{\text{bin}}(F)$. By simply choosing between the smaller of the two graphs G_{basic} and G_{bin}, we obtain the MIN3C encoding as described in (Aloul, Ramani, et al. 2003). Figure 1 illustrates the original model graph G_{basic} compared to the optimized encoding G_{bin}.

Tuple Vertices for Ternary Clauses. We now describe a further optimization of this model graph that reduces the number of vertices used for *ternary clauses*. While reducing vertices representing ternary clauses is inherently more challenging, we may remove some of them in specific cases. We do so by counting whether a particular pair of literals l_1, l_2 occurs multiple times in ternary clauses. If so, we create and use a gadget that represents this pair of literals.

Let

$$\text{Count}_2(l_1, l_2) := |\{C \mid C \in F, |C| = 3, \{l_1, l_2\} \subset C\}|$$

denote the number of times l_1 and l_2 appear together in ternary clauses of F.

Consider a ternary clause $\{l_1, l_2, l_3\} \in F$. Assume that for the tuple $\{l_1, l_2\}$ it holds that

$$\text{Count}_2(l_1, l_2) > \text{Count}_2(l_1, l_3) \text{ and } \text{Count}_2(l_1, l_2) > \text{Count}_2(l_2, l_3). \tag{1}$$

We then apply the following reduction. If they have not been introduced before, we create a *tuple representation vertex* $(l_1, l_2)^{\uparrow}$ and a *tuple connection vertex* $(l_1, l_2)^{\downarrow}$. The intuition here is that the connection vertex establishes which literals are part of the tuple. The representation vertex represents *all ternary clauses* in which l_1 and l_2 are the tuple with the highest occurrence count.

The representation vertex $(l_1, l_2)^{\uparrow}$ is connected to the connection vertex $(l_1, l_2)^{\downarrow}$. Tuple representation vertices receive color 4, and tuple connection vertices receive color 5. (Recall the literal vertices are colored with color 1, clause vertices with color 2, and variable vertices with 3.) The connection vertex $(l_1, l_2)^{\downarrow}$ is connected to the vertices representing l_1 and l_2, respectively. The representation vertex is connected to the vertex representing l_3. The vertex representing C is removed from the graph. Figure 2 illustrates the reduction.

We apply the above reduction exhaustively. Observe that if we can replace at least three clauses containing (l_1, l_2) by the tuple construction for (l_1, l_2), we reduce the total number of vertices of the graph. In practice, we only replace tuples that occur at least 16 times, and only apply the reduction if there are at least 2048 ternary clauses. This is a simple measure to mitigate cases in which computing the graph reduction takes longer than

the expected speed-up in symmetry detection. To simplify our arguments below, let us however assume that we apply the reduction whenever possible.

Combining the binary and ternary optimizations, we denote our final model graph as $G_{\text{opt}}(F)$. We prove correctness of the construction.

Lemma 2.1. *For any CNF formula F, the graph $G_{opt}(F)$ is a model graph for F.*

Proof. We show that the graph $G_{\text{opt}}(F)$ satisfies the definition of a model graph. The first condition is clearly satisfied: the initial vertex set consists of literals and clauses of F, and we only remove clause vertices during our construction of $G_{\text{opt}}(F)$.

For the second condition, suppose that φ is an automorphism of F. In the following, we extend φ to obtain an automorphism ψ of $G_{\text{bin}}(F)$.

Note that we either remove all binary clauses or none of them: this depends on whether we choose $G_{\text{basic}}(F)$ is $G_{\text{opt}}(F)$ as the encoding of binary clauses. Hence, if in our construction we removed a clause vertex C, we would have also removed C^φ. We let ψ map the remaining *clause vertices* according to how φ maps clauses to clauses, that is, if $C \in V(G_{\text{opt}}(F))$ then C is mapped to C^φ.

For the reduction of ternary clause, observe that

$$\begin{aligned}&\{\!\{\text{Count}_2(l_1,l_2),\text{Count}_2(l_1,l_3),\text{Count}_2(l_2,l_3)\}\!\} = \\ &\{\!\{\text{Count}_2(l_1^\varphi,l_2^\varphi),\text{Count}_2(l_1^\varphi,l_3^\varphi),\text{Count}_2(l_2^\varphi,l_3^\varphi)\}\!\}\end{aligned} \tag{2}$$

holds. By construction, the tuple l and l' is uniquely determined by the count (see Equation 1). We can conclude that if we constructed $(l,l')^\uparrow$ and $(l,l')^\downarrow$, then we must have also constructed $(l^\varphi,l'^\varphi)^\uparrow$ and $(l^\varphi,l'^\varphi)^\downarrow$. We may thus define ψ to map $(l_1,l_2)^\uparrow$ to $(l_1^\varphi,l_2^\varphi)^\uparrow$, and $(l_1,l_2)^\downarrow$ to $(l_1^\varphi,l_2^\varphi)^\downarrow$. Furthermore, if a variable vertex $\dot{v}$ was introduced, then ψ maps the variable vertex $\dot{v}$ to $\dot{v^\varphi}$.

Observe that ψ defines a color-preserving permutation of $G_{\text{opt}}(F)$. It remains to be shown that ψ preserves the edge relation of $G_{\text{opt}}(F)$. For the edges connecting to remaining clause vertices C with $|C| \neq 3$ this follows immediately. Since we either replace all binary clauses by edges or none of them, for edges $\{l_1,l_2\}$ representing binary clauses, $\{l_1^\varphi,l_2^\varphi\} \in E(G_{\text{opt}}(F))$ follows immediately. Since we either introduce variable vertices for all literals or none of them, it follows immediately that the edges connecting literals to literals, or literals to variable vertices are preserved.

It remains to be shown that the edges related to ternary clauses are preserved. From Equation 2 and our ternary reduction we can conclude the following: if l is connected to $(l,l')^\uparrow$, then l^φ is connected to $(l^\varphi,l'^\varphi)^\uparrow$. Clearly, $(l,l')^\uparrow$ connects to $(l,l')^\downarrow$, and conversely $(l^\varphi,l'^\varphi)^\uparrow$ connects to $(l^\varphi,l'^\varphi)^\downarrow$. If $(l,l')^\downarrow$ connects to l'', then $(l^\varphi,l'^\varphi)^\downarrow$ connects to l''^φ. Thus, the edge relation is preserved and ψ is an automorphism of G_{opt}.

Conversely, suppose that ψ is an automorphism of $G_{\text{opt}}(F)$. We argue that $\varphi = \psi|_{\text{Lit}(F)}$ is a symmetry of F.

Since ψ is color-preserving, the following subsets of the vertex set are stable under ψ: literal vertices, variable vertices, clause vertices, tuple representation vertices and tuple connection vertices.

We first argue that the symmetries of our model graph respect negation. That is, we argue that if $l = l^\varphi$ holds, then $\neg l = (\neg l)^\varphi$ holds. That is, for every literal l and an automorphism φ of model graph, it holds that $(\neg l)^\varphi = \neg(l^\varphi)$. If we used the binary clause construction with variable vertices $\dot{v}$, this follows immediately from the fact that l and $\neg l$ are only connected to a single vertex of color 3 (i.e., $\dot{v}$). If we did not introduce variable vertices, there is an edge connecting l and $\neg l$. However, since in this case binary clauses are represented by clause vertices, this means that l (and $\neg l$) have a unique neighbor of color 1. Hence, if l is mapped to l^φ, $(\neg l)^\varphi = \neg l^\varphi$ follows.

It remains to be shown that the symmetries of the model graph respect the clauses of the formula. That is, for each $C \in F$, $C^\varphi \in F$ holds. For clauses C with $|C| \neq 3$ we can immediately conclude that $C^\varphi \in F$ holds: this is either ensured by clause vertices, or the edges representing binary clauses.

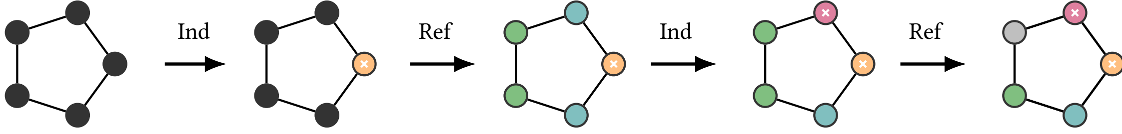


Fig. 3. An illustration of the IR process. Individualization steps break symmetries or similarities (nodes marked with a cross are individualized). Refinement steps propagate this information.

For ternary clauses C, we observe the following. The first possibility is that there exists a clause vertex v for such a clause C. Since ψ is an automorphism of the model graph, the clause C' represented by the vertex v^{ψ} must consist of literals $\{l^{\varphi} : l \in C\}$. Hence φ maps every such clause C to another such clause C' of F, as desired.

The second possibility is when the vertex for C was removed: Then, w.l.o.g, $C = \{l_1, l_2, l_3\}$ is represented by an edge connecting a literal vertex l_3 to a vertex $(l_1, l_2)^{\downarrow}$. From the construction, it follows that if $\psi((l_1, l_2)^{\downarrow}) = (l_1{}', l_2{}')^{\downarrow}$, then $\varphi(l_1) = \psi(l_1) = l_1{}'$ and $\varphi(l_2) = \psi(l_2) = l_2{}'$ hold. Furthermore, let $\psi(l_3) = l_3'$ denote the vertex to which l_3 is mapped. Since ψ is an automorphism, $(l_1{}', l_2{}')^{\downarrow}$ must be connected to l_3'. Due to our construction, this means that $\{l_1', l_2', l_3'\} \in F$. Since $C^{\varphi} = \{l_1', l_2', l_3'\} \in F$, the claim follows. □

2.3 Graphs and Symmetries

Permutation Groups. The symmetries of a CNF formula naturally form a permutation group. Hence, we recall some basic notions of permutation group theory. A more detailed account can be found in (Seress 2003).

Let Ω be a nonempty finite set. Let $\mathrm{Sym}(\Omega)$ denote the *symmetric group* on Ω, i.e., the group of permutations of Ω, and set $\mathrm{Sym}(n) \coloneqq \mathrm{Sym}([n])$. A *permutation group* is a subgroup Γ of $\mathrm{Sym}(\Omega)$, that is, Γ is closed under the composition of maps. We write $\Gamma \leq \mathrm{Sym}(\Omega)$. In this situation, we also say that Γ *acts on* Ω.

For $g \in \Gamma$ and $\omega \in \Omega$, we write ω^g for the image of ω under the permutation g. Furthermore, we write $\omega^{\Gamma} = \{\omega^g : g \in \Gamma\}$ for the *orbit* of ω under Γ. In other words, ω^{Γ} consists of all points in Ω that can be reached from ω by applying elements of Γ. It is easily verified that the property of being in the same orbit under Γ defines an equivalence relation on the elements of Ω. The partition of Ω into the orbits of Γ is called the *orbit partition*.

For $\omega \in \Omega$, let $\Gamma_{\omega} \coloneqq \{g \in \Gamma : \omega^g = \omega\}$ denote the *stabilizer* of ω in Γ. In other words, Γ_{ω} consists of those elements in the group Γ that map ω to itself. It can be shown that the stabilizer Γ_{ω} is a subgroup of Γ.

Example 2.2. Let Γ be the subset of $\mathrm{Sym}(4)$ consisting of the permutations $g \coloneqq (1, 3)$ (that is, the permutation swapping 1 and 3) and the identity id. This set is closed under multiplication, so Γ is a subgroup of $\mathrm{Sym}(4)$. For $\omega = 1$, we have $\omega^g = 3$ and $\omega^{\Gamma} = \{1, 3\}$. Similarly, we obtain $2^{\Gamma} = \{2\}$ and $4^{\Gamma} = \{4\}$, and hence the orbit partition of $\Omega = [4]$ is $\{\{1, 3\}, \{2\}, \{4\}\}$. We have $\Gamma_4 = \Gamma$ and $\Gamma_1 = \{\mathrm{id}\}$.

The *direct product* of permutation groups Γ_1 and Γ_2 which act on domains Ω_1 and Ω_2, respectively, is the Cartesian product $\Gamma_1 \times \Gamma_2$, endowed with a component-wise multiplication. It naturally acts component-wise on $\Omega_1 \times \Omega_2$.

Individualization-Refinement. A central ingredient in our algorithms is the so-called *individualization-refinement* (IR) paradigm. The IR paradigm is the central technique in all state-of-the-art symmetry detection algorithms (Anders and Schweitzer 2021; Darga et al. 2004; Junttila and Kaski 2011; McKay and Piperno 2014), and highly engineered implementations are available. The paradigm mainly consists of the *individualization* technique, paired with the so-called *color refinement algorithm*. In this paragraph, we focus on a high-level explanation of the routine. A detailed account can be found in (Anders 2024; McKay and Piperno 2014).

The central idea of IR (see Figure 3 for an illustration) is the following: given a vertex-colored graph (G, π) and a vertex $v \in V(G)$, the vertex v is *individualized*. Basically this means that it obtains a new color. The routine

then proceeds with a so-called *color refinement*: in each step, every vertex of G obtains a new color, based on its former color together with the colors of its neighbors in G. In this step, two vertices are assigned the same color if they had the same color previously and the multisets of their neighbors' colors are identical. This recoloring procedure is repeated until the corresponding color partition stabilizes. The final coloring π' is then returned. We use the notation $\pi' = \mathrm{IR}((G,\pi),v)$ to denote this process. The call $\mathrm{IR}((G,\pi),v)$ can be computed in time $O(|E(G)|\log|V(G)|)$. In (Anders 2024), a precise description of the involved algorithms is given.

The coloring π' is a *refinement* of π in the sense that vertices with the same color in π' already had the same color in π. In other words, a color c of π is either preserved in π', or partitioned into several other colors $c_1,\ldots,c_n$. For $i \in [n]$, we call the sets

$$\{u \in V(G)\colon \pi(u) = c,\ \pi'(u) = c_i\}$$

the *fragments* of c in π'. The second crucial observation is that vertices in the same orbit under the stabilizer $\mathrm{Aut}(G,\pi)_v$ obtain the same color in π'. However, it is possible that the color partition of π' is *coarser* than the orbit partition in the sense that the vertices of multiple orbits might obtain the same color in π'.

Clearly, this process can be applied inductively to individualize multiple vertices. It is also possible to pass the empty sequence ε to IR, i.e., to run only the color refinement procedure. Arguing as above, the resulting color partition is guaranteed to be at least as coarse as the orbit partition of $\mathrm{Aut}(G,\pi)$ (i.e., the stabilizer of the empty sequence).

The next lemma summarizes the properties of IR to which we refer throughout the paper:

Lemma 2.3. *Given a vertex-colored graph (G,π) and a vertex $v \in V(G)$, the refined coloring $\pi' = \mathrm{IR}((G,\pi),v)$ has the following properties.*

(1) The coloring π' is a refinement *of π: for $u,w \in V(G)$ with $\pi'(u) = \pi'(w)$, we have $\pi(u) = \pi(w)$.*
(2) The color partition of π' is at least as coarse *as the orbit partition of $\Gamma = \mathrm{Aut}(G,\pi)_v$: vertices $u,w \in V(G)$ with $\pi'(u) \neq \pi'(w)$ lie in different orbits of Γ, i.e., we have $w \notin u^\Gamma$.*
(3) The colors of π' are isomorphism-invariant*: for every $\varphi \in \mathrm{Sym}(V(G))$, it holds that $\mathrm{IR}((G^\varphi,\pi^\varphi),v^\varphi) = \mathrm{IR}((G,\pi),v)^\varphi$. In particular, if $\varphi \in \mathrm{Aut}(G,\pi)$ holds, then*

$$\mathrm{IR}((G,\pi),v)^\varphi = \mathrm{IR}((G^\varphi,\pi^\varphi),v^\varphi) = \mathrm{IR}((G,\pi),v^\varphi)$$

holds.

These properties follow almost immediately from the definition of IR, and we refer to (McKay and Piperno 2014) for a treatment of the topic. We also mention that usually, as opposed to the description above, IR is defined for *sequences* of vertices instead of single vertices.

We now recall the notion of *Tinhofer graphs* (Arvind et al. 2017). In view of the second part of Lemma 2.3, these are precisely the graphs for which the two partitions coincide.

Definition 2.4 (Tinhofer Graph (Arvind et al. 2017; Tinhofer 1991))**.** A graph G is called Tinhofer if for all $v \in V(G)$, the orbit partition of $\Gamma \coloneqq \mathrm{Aut}(G,\pi)_v$ coincides with the color partition of $\pi' \coloneqq \mathrm{IR}((G,\pi),v)$ and the same applies recursively to the colored graph (G,π') (this corresponds to individualizing multiple vertices of G). Formally, the first property means that for all $u,w \in V(G)$, we have $w \in u^\Gamma$ if and only if $\pi'(u) = \pi'(w)$.

In particular, IR works well on Tinhofer graphs: practical graph isomorphism solvers are guaranteed to run in polynomial-time. Indeed, general-purpose practical symmetry detection algorithms only work well *because* most practical graphs seem to be Tinhofer.

2.4 Symmetry Structures in SAT

The idea of our tool is to detect certain symmetry structures that are subsequently exploited. In this section, we describe the main structures detected by the tool. The description of the detection algorithms is the subject of Section 3.

Throughout, let F be a SAT formula. As a first step, consider the *disjoint direct decomposition* of the symmetries $\mathrm{Aut}(F)$: this is a partition $\mathrm{Lit}(F) = L_1 \cup \cdots \cup L_k$ of $\mathrm{Lit}(F)$ for which there exists a decomposition $\mathrm{Aut}(F) = A_1 \times \cdots \times A_k$ into a direct product of subgroups such that, for every $i \in [k]$, the automorphisms in A_i only move the literals in L_i. As usual, $\cup$ denotes the disjoint union of sets. A disjoint direct decomposition naturally decomposes the symmetry breaking problem, and it suffices to treat each factor separately. In the following, we always refer to the finest such decomposition, which is unique, since, if there were two distinct decompositions, their intersection would yield a decomposition refining both, which is a contradiction We call its parts $L_1, \ldots, L_k$ the *disjoint direct factors* of F.

Lemma 2.5. *Every disjoint direct factor is a union of orbits of* $\mathrm{Aut}(F)$.

Proof. Consider a disjoint direct composition $\mathrm{Lit}(F) = L_1 \cup \ldots \cup L_k$ as above, and let $\mathrm{Aut}(F) = A_1 \times \cdots \times A_k$ be the corresponding decomposition. Without loss of generality, consider $l \in L_1$, and let l' be a literal in the orbit of l. This means that $l' = l^{\varphi}$ for some $\varphi \in \mathrm{Aut}(F)$. Writing $\varphi = \alpha_1 \ldots \alpha_k$ with $\alpha_i \in A_i$, we obtain $l' = l^{\varphi} = l^{\alpha_1}$, as $l^{\alpha_2 \ldots \alpha_k} = l$ by definition. Now $(l')^{\alpha_1^{-1}} = l^{\alpha_1 \alpha_1^{-1}} = l$. In particular, the automorphism α_1^{-1} moves l' to l (note that $\alpha_1^{-1} \in A_1$ as A_1 is a group). This implies $l' \in L_1$, as the literals in $L_2 \cup \ldots \cup L_k$ are fixed by all elements of A_1. □

Example 2.6. Consider the SAT formula $F = (x_1 \vee x_2) \wedge (x_3 \vee x_4 \vee x_5)$. It is easily seen that the variables x_1 and x_2 as well as x_3, x_4 and x_5 can be interchanged, and there is no symmetry mapping an element of $\{x_1, x_2\}$ to $\{x_3, x_4, x_5, \neg x_3, \neg x_4, \neg x_5\}$ and vice versa. Thus $\mathrm{Aut}(F)$ acts on $\mathrm{Var}(F)$ as $\mathrm{Sym}(\{x_1, x_2\}) \times \mathrm{Sym}(\{x_3, x_4, x_5\})$ (and the images of the negated literals are defined accordingly). The corresponding disjoint direct decomposition is $\mathrm{Lit}(F) = \{x_1, x_2, \neg x_1, \neg x_2\} \cup \{x_3, x_4, x_5, \neg x_3, \neg x_4, \neg x_5\}$.

Let us make a general observation regarding negation symmetry.

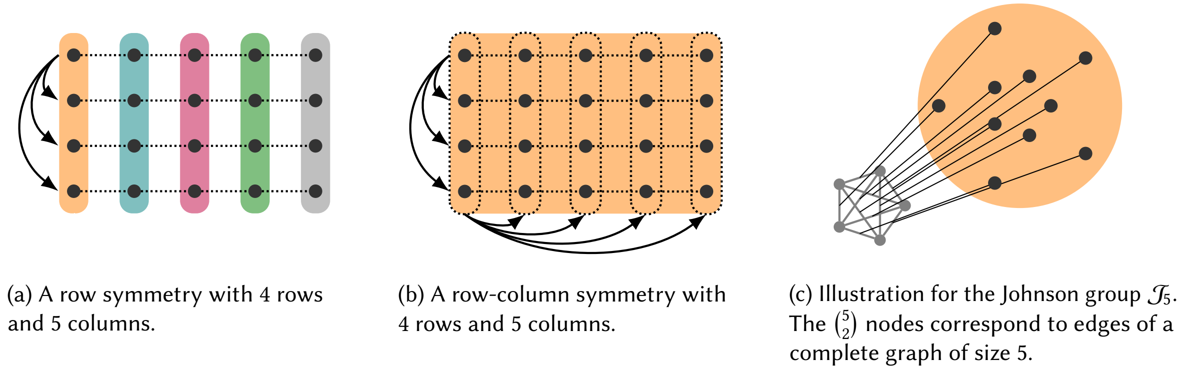
Remark 2.7. For an orbit σ of literals under $\mathrm{Aut}(F)$, also the set $\neg\sigma \coloneqq \{\neg v \colon v \in \sigma\}$ is an orbit of literals. Hence two cases can occur: either we have $\sigma = \neg\sigma$, or the orbits σ and $\neg\sigma$ are disjoint.

In order to simplify the exposition, we only consider the second scenario in the following. As factors in the disjoint direct decomposition, we detect several variants of three main kinds of symmetries, namely row symmetry, row-column symmetry, and Johnson symmetry. Let us now define these notions in the special context of CNF formulas.

Row Symmetry. Row interchangeability, or *row symmetry*, naturally occurs in the context of matrix modeling (Flener, Frisch, Hnich, Kızıltan, et al. 2001) and is already successfully exploited in automated symmetry breaking. We say that a SAT formula F *exhibits row symmetry* if there exists a disjoint direct factor $L \subseteq \mathrm{Lit}(F)$ which can be arranged in a matrix M such that $\mathrm{Aut}(F)|_L$ acts by permuting the rows of M. In addition, we require that every column of M is an orbit of $\mathrm{Aut}(F)$. See Figure 4a for an illustration. The colored boxes illustrate orbits, whereas dashed lines indicate vertices in the same row. The rows can be permuted using symmetry.

Example 2.8 (Graph coloring)**.** Let $G = (V, E)$ be an asymmetric graph, and let c be a natural number. Then the formula

$$F = \bigwedge_{v \in V} (x_{1,v} \vee \cdots \vee x_{c,v}) \wedge \bigwedge_{v \in V} \bigwedge_{1 \le i < j \le c} (\neg x_{i,v} \vee \neg x_{j,v}) \wedge \bigwedge_{\{v,w\} \in E} \bigwedge_{i=1}^{c} (\neg x_{i,v} \vee \neg x_{i,w})$$



(a) A row symmetry with 4 rows and 5 columns.

(b) A row-column symmetry with 4 rows and 5 columns.

(c) Illustration for the Johnson group $\mathcal{J}_5$. The $\binom{5}{2}$ nodes correspond to edges of a complete graph of size 5.

Fig. 4. Various group structures used throughout the paper. Colors indicate orbits of the group.

in variables $x_{k,v}$ ($k \in [c]$ and $v \in V(G)$) encodes the question whether G has a proper coloring that uses at most c colors. The first conjunction in F ensures that every vertex obtains one of the colors in $[c]$, the second that no vertex obtains more than one color, and the last that adjacent vertices do not obtain the same color. Clearly, every permutation of the colors induces a symmetry of the formula. Moreover, as G is asymmetric, it is easily seen that F has no further symmetries. Since the variables can be arranged in a matrix $X = (x_{k,v})_{k,v}$ in which every permutation of the colors corresponds to a permutation of the rows, F exhibits row symmetry.

Figure 6 illustrates the symmetries of a graph coloring instance on an asymmetric graph with 6 vertices using 4 colors.

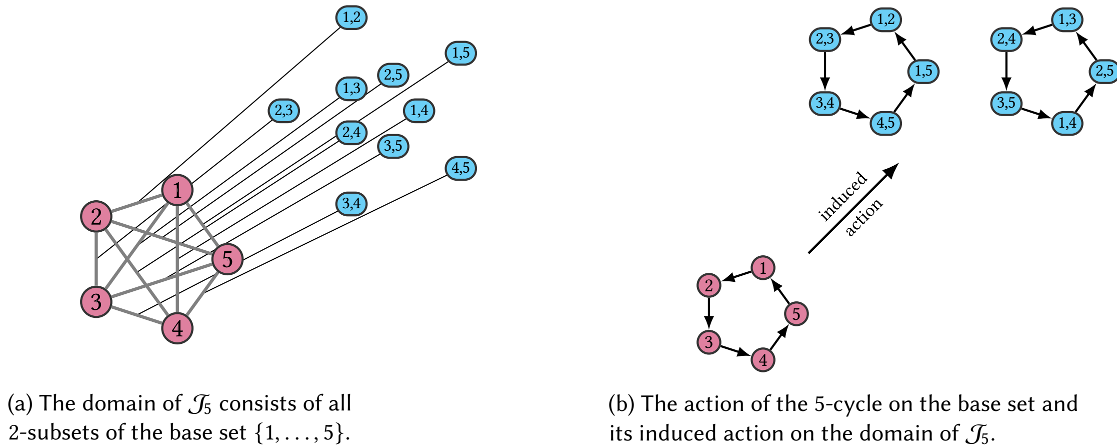
We should address a technical difference between the definition above and and the definition of row symmetry used *implicitly* by the BreakID algorithm. In our definition, a disjoint direct factor should *only* admit the action of the row symmetry group, or a particularly defined extension (see Section 3). BreakID on the other hand would accept any row symmetry *subgroup* that it detects (see (Anders, Schweitzer, and Soos 2023) for further discussion). Hence, in practice, it is to be expected that the tools disagree about detecting row symmetry. In particular, satsuma often instead identifies larger, more expressive groups, such as row-column symmetry, as is explained below. These larger groups often subsume the row symmetry detected by BreakID.

Row-column symmetry. Row-column symmetries are an extension of row interchangeability. Row-column symmetry naturally occurs whenever both the rows and columns of a matrix of variables are interchangeable. Examples can be found in scheduling, design, and combinatorial problems, as is discussed in (Flener, Frisch, Hnich, Kızıltan, et al. 2001).

For $m, n \in \mathbb{N}$, the *row-column symmetry* group is $\Gamma \coloneqq \mathrm{Sym}(n) \times \mathrm{Sym}(m)$, acting component-wise on $[n] \times [m]$. We think of $[n] \times [m]$ as an $n \times m$ matrix M, on which $(\sigma_1, \sigma_2) \in \Gamma$ acts by permuting the n rows according to σ_1 and the m columns according to σ_2.

A SAT formula F *exhibits row-column symmetry* if there exists a disjoint direct factor $L \subseteq \mathrm{Lit}(F)$ consisting of an orbit σ of $\mathrm{Aut}(F)$ and its negation $\neg\sigma$ such that the following holds: the literals in σ can be arranged in an $n \times m$-matrix M such that $\mathrm{Aut}(F)|_\sigma$ acts as a row-column symmetry group on M. See Figure 4b for an illustration.

Example 2.9 (Pigeonhole principle)**.** The primary example for a formula exhibiting row-column symmetry is the well-known pigeonhole principle with $n + 1$ pigeons and n holes. It can be encoded in the following formula



(a) The domain of $\mathcal{J}_5$ consists of all 2-subsets of the base set $\{1, \ldots, 5\}$.

(b) The action of the 5-cycle on the base set and its induced action on the domain of $\mathcal{J}_5$.

Fig. 5. An illustration of the Johnson group $\mathcal{J}_5$.

on variables x_{ij} with $i \in [n+1]$ and $j \in [n]$, where x_{ij} encodes that pigeon i goes to hole j:

$$F = \bigwedge_{i=1}^{n+1} \bigwedge_{\substack{j,k \in [n] \\ j \neq k}} (\neg x_{ij} \vee \neg x_{ik}) \wedge \bigwedge_{j=1}^{n} \bigwedge_{\substack{i,m \in [n+1] \\ i \neq m}} (\neg x_{ij} \vee \neg x_{mj}) \\ \wedge \bigwedge_{i=1}^{n+1} (x_{i1} \vee \cdots \vee x_{in}).$$

Clearly, the pigeons as well as the holes can be permuted independently, so every row and every column permutation of the variable matrix (x_{ij}) is a symmetry of the formula. Figure 6 illustrates a row-column symmetry with matching variable labels for pigeonhole principle with 5 pigeons and 4 holes.

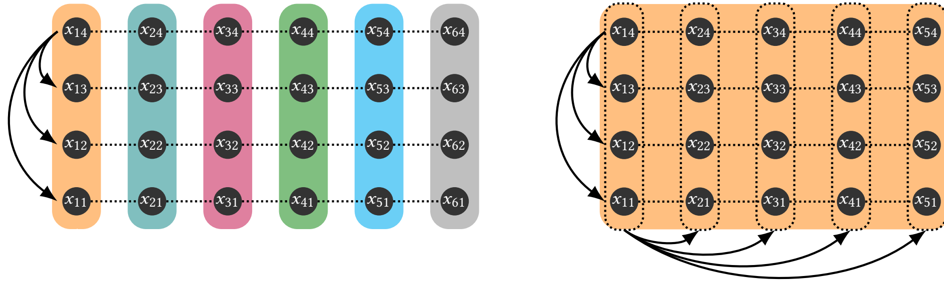
Note that the action of $\mathrm{Aut}(F)$ on σ naturally extends to a row-column symmetry action on $\neg\sigma$, by $\alpha(\neg x) = \neg\alpha(x)$ for all variables x and $\alpha \in \mathrm{Aut}(F)$. For this reason, our algorithm generates the matrix M of the literals in σ and extends this to $\neg\sigma$, see Section 3.2 for details.

Johnson symmetry. Johnson groups are naturally tied to the graph isomorphism problem. Whenever a problem asks for the existence of an undirected graph with a certain property, typically, the underlying symmetries form a Johnson group.

Observe that $\pi \in \mathrm{Sym}(n)$ induces a permutation on the domain $\binom{[n]}{2}$ of 2-subsets of $[n]$, mapping $\{a_1, a_2\}$ to $\{a_1^\pi, a_2^\pi\}$. This way, $\mathrm{Sym}(n)$ becomes a permutation group on a domain of size $\binom{|n|}{2}$, the *Johnson group* $\mathcal{J}_n$. Technically, these groups are specifically Johnson groups of arity 2. The corresponding action is called a *Johnson action.*

Example 2.10. Let $n = 4$. The symmetric group $\mathrm{Sym}(4)$ acts on the $\binom{4}{2} = 6$ subsets of size 2 of $[4]$. For instance, for $g = (1, 3)$ and $S = \{1, 4\} \in \binom{[4]}{2}$, we have $S^g = \{1^g, 4^g\} = \{3, 4\}$. In this way, we can embed $\mathrm{Sym}(4)$ as a subgroup of $\mathrm{Sym}(\binom{[4]}{2}) \cong \mathrm{Sym}(6)$.

We now define Johnson symmetries for SAT formulas. Intuitively, the variables the formula correspond to the "edges" (i.e., sets of two vertices) of a complete graph. There is a symmetric action on the "vertices" of this



(a) A row symmetry with 4 rows and 6 columns.

(b) A row-column symmetry with 4 rows and 5 columns.

Fig. 6. Row and row-column symmetry with variable labels. Colors indicate orbits of the group.

underlying graph and the variables of the formula ("edges") are permuted accordingly. See Figure 4c and Figure 5 for an illustration. Formally, a SAT formula F *exhibits a Johnson symmetry* if the following holds: there exists a disjoint direct factor $L \subseteq \mathrm{Lit}(F)$ consisting of an orbit σ of $\mathrm{Aut}(F)$ and its negation $\neg\sigma$ such that the literals in σ can be relabeled as $x_{\{i,j\}}$ for all $\{i, j\} \in \binom{[n]}{2}$ and $\mathrm{Aut}(F)|_\sigma$ acts as the Johnson group $\mathcal{J}_n$ (by permuting the index sets). Again, the action of $\mathrm{Aut}(F)$ naturally extends to $\neg\sigma$.

A standard example of formulas exhibiting a Johnson symmetry are encodings of so-called Ramsey-type problems:

Example 2.11. (Ramsey numbers) Let $k, n \in \mathbb{N}$. Is there a graph on n vertices that neither contains a r-clique nor an independent set of size s? This question can be encoded as a SAT formula as follows.

The variables are $x_{i,j}$ for $i, j \in [n]$ with $i \neq j$, which encode the edges of the graph. Every subset of vertices of size r must not be a clique, so it contains at least one non-edge:

$$\bigwedge_{\substack{U \subseteq [n] \\ |U|=r}} \left(\bigvee_{\substack{u_1,u_2 \in U \\ u_1 < u_2}} \neg x_{u_1,u_2} \right).$$

Similarly, every subset of vertices of size s must not be an independent set, meaning that it contains at least one edge:

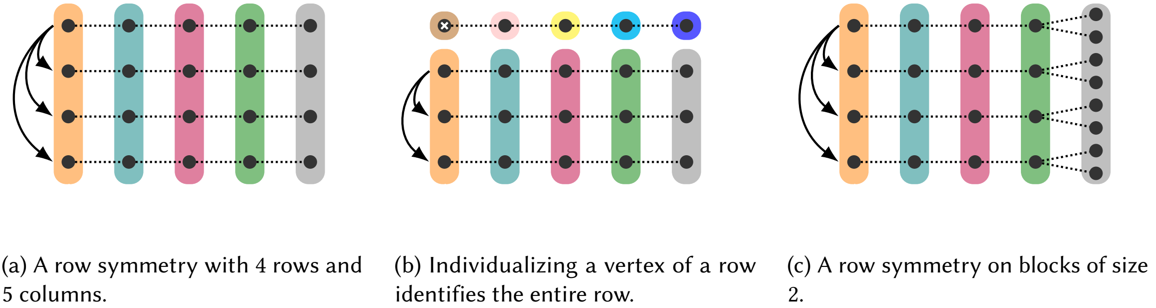
$$\bigwedge_{\substack{U \subseteq [n] \\ |U|=s}} \left(\bigvee_{\substack{u_1,u_2 \in U \\ u_1 < u_2}} x_{u_1,u_2} \right).$$

Let F be the conjunction of the two formulas. Since the labels $1, \ldots, n$ of the vertices can be permuted arbitrarily, F exhibits a Johnson symmetry. Note that if $r = s$, we have an additional negation symmetry swapping the variables and their negatives.

3 Structure Detection Algorithms

We now present our new detection algorithms. All algorithms are centered around detecting a particular structure on a model graph $G(F)$ of a given CNF formula F. Recall that any model graph $G(F)$ contains a vertex for each literal, so whenever we discuss literals may use the terms "vertex" and "literal" interchangeably.

Let us begin by listing the main design principles underlying all of our algorithms.



(a) A row symmetry with 4 rows and 5 columns.

(b) Individualizing a vertex of a row identifies the entire row.

(c) A row symmetry on blocks of size 2.

Fig. 7. Illustrations of different aspects of row symmetry.

Colors instead of Orbits. Our algorithms work on the *assumption* that the model graph $G(F)$ is Tinhofer (see Definition 2.4). The assumption gives us that we can compute orbits of stabilizers using IR. In particular, the color classes of $\pi = \mathrm{IR}(G(F), \varepsilon)$ *are* then the orbits of $\mathrm{Aut}(G(F))$.

Confirming the Structure. The input model graph might not be Tinhofer. However, each algorithm constructs a carefully chosen set of candidate permutations, which suffices to prove the existence of a certain group action. In particular, these permutations offer a certificate of correctness of the detected group. It is then checked that these permutations are indeed automorphisms of the formula F.

Color-by-color. Our detection algorithms proceed color-by-color, or orbit-by-orbit: given an orbit, the algorithms stabilize a specific set of vertices, observing the effect on the given orbit as well as other orbits. If an orbit exhibits a specific group action, then this effect will follow a particular pattern that can be easily recognized. Based on this a model of the purported structure is constructed.

3.1 Row Symmetry

We describe an algorithm to detect row symmetry. That is, we want to arrange variables in a matrix such that the action of $\mathrm{Aut}(F)$ corresponds to row permutations of this matrix.

Intuitively, the algorithm proceeds as follows: assuming that a row symmetry is present, i.e., such a matrix form can be found, the individualization step individualizes a vertex v. This way, the row of v cannot be permuted with other rows any longer. Thus, every vertex in this row is fixed, so the IR procedure will distinguish it from all other vertices in its column, which can still be permuted. Iterating this step, we can thus identify the rows of the matrix.

The algorithm starts from a candidate column σ. As candidates, the algorithm considers all color classes of an initial color partition obtained by color refinement.

First, we define an auxiliary function that transposes two pair-wise disjoint, duplicate-free lists of literals of equal length: for $l \in \mathrm{Lit}(F)$, let

$$\mathrm{transpose}_F((l_1 \dots l_k), (l'_1 \dots l'_k))_F(l) \coloneqq \begin{cases} l'_i & \text{if } l = l_i \text{ with } i \in [k] \\ l_i & \text{if } l = l'_i \text{ with } i \in [k] \\ l & \text{otherwise.} \end{cases}$$

Using this, Algorithm 1 describes our procedure to detect row symmetry.

(Description of Algorithm 1.) The algorithm applies IR for each $v \in \sigma$ (see Figure 7b). All vertices v' in other orbits which are individualized in this process, i.e., which are fixed once v is fixed, are added to the purported

Algorithm 1: Detection algorithm for row symmetry.

```
1  function DetectRowSymmetry
       Input : ➢ formula F
               ➢ candidate set σ ⊆ Lit(F) with |σ| ≥ 3
       Output : ≺ matrix with row symmetry including σ, or ⊥ if check fails
2      (G, π) ≔ G(F);
3      π′ ≔ IR((G, π), ε);
4      // construct a candidate row for each v ∈ σ
5      foreach v ∈ σ do
6          π_v ≔ IR((G, π′), v);
7          let τ be a list of literals that are singletons in π_v but not in π′;
8          sort literals in τ according to their color in π_v;
9          row[v] ≔ τ;
10     check that rows are pair-wise disjoint;
11     // verify that M exhibits row symmetry
12     foreach i ∈ {1 ... |σ| − 1} do
13         v ≔ σ[i − 1]; v′ ≔ σ[i];
14         check that transpose_F(row[v], row[v′]) is a symmetry of F;
15     return matrix M constructed from row
```

"row" of v. We then verify that every row transposition of the resulting matrix is indeed a symmetry of F. For an illustration of the procedure, see Figure 7a.

(Runtime of Algorithm 1.) Let n and m refer to the number of vertices and edges of $G(F)$. Using an efficient color refinement implementation, the IR calls and construction of rows in the algorithm can be implemented in runtime $O(|\sigma|(n+m)\log n)$. Using an efficient check, such as described in (Anders 2024), all candidate rows can be checked in time $O(n+m)$.

It should be mentioned that in practice, individualizing a vertex and applying color refinement will rarely result in a runtime of $\Omega((n+m)\log n)$.

(Correctness of Algorithm 1.) We first make the following observation for orbits of stabilizers in row interchangeability groups (compare to Figure 7).

Lemma 3.1. *Let $\Gamma = \mathrm{Sym}(n)$ be a row interchangeability group acting on $[n] \times [m]$. For every $(i,j) \in [n] \times [m]$, the orbit of $(k,l) \in [n] \times [m]$ under the stabilizer $\Gamma_{(i,j)}$ of (i,j) is given by*

$$(k,l)^{\Gamma_{(i,j)}} = \begin{cases} \{(i,l)\} & \text{if } k = i \\ ([n] \setminus \{i\}) \times \{l\} & \text{otherwise.} \end{cases}$$

Proof. Interpreting $[n] \times [m]$ as an $n \times m$-matrix M, recall that Γ acts by permuting the rows of M. In other words, the stabilizer $\Gamma_{(i,j)}$ consisting of all row permutations that fix the i-th row and permute the other rows arbitrarily. Now consider the orbit of $(k,l) \in [n] \times [m]$ under the stabilizer $\Gamma_{(i,j)}$ of (i,j). If $k = i$ then (k,l) can only be mapped to elements in the same row as $\Gamma_{(i,j)}$ fixes the i-th row of M. On the other hand, since Γ acts by permuting the rows, every element of M can only be mapped to elements in the same column, that is, (k,l) must be fixed. Similarly, for $k \neq i$, the element (k,l) can be mapped to all elements in the l-th column except for (i,l). ◻

Next, we prove that the algorithm always returns correct symmetries of F and that in case the model graph is Tinhofer, the algorithm is guaranteed to detect row interchangeability groups.

Theorem 3.2. *Let F be a SAT formula.*

(1) If Algorithm 1 returns a matrix M, every row permutation of M is a symmetry of F.
(2) If F exhibits row interchangeability with at least three rows including the orbit σ and $G(F)$ is a Tinhofer graph, Algorithm 1 detects this structure and returns a corresponding matrix of literals.

Proof. The first claim is guaranteed by the last part of Algorithm 1 which ensures that transpositions of the rows of the returned matrix M are indeed symmetries of F (Line 12). Every permutation can be written as a product of transpositions. Thus the above implies that arbitrary row permutations are symmetries of F.

Now assume that F exhibits a row symmetry with at least three rows including σ, the latter being an orbit, and $G(F)$ is Tinhofer. We argue that the algorithm successfully detects this symmetry. We remark that the orbits of $\mathrm{Aut}(G(F))$ restricted to the literals are precisely orbits of $\mathrm{Aut}(F)$. Let L be the disjoint direct factor of F containing σ and assume that the literals in L can be partitioned into a matrix M that exhibits row symmetry (see Figure 7a). Due to the assumption that $G(F)$ is Tinhofer, if the vertex v corresponding to a literal l of F is individualized, the resulting refined coloring consists of the orbits of $\mathrm{Aut}(G(F))_v$. In particular, due to Lemma 3.1, the vertices in the row of M are fixed and all other vertices are contained in orbits of size at least two since we have at least three rows (see Figure 7b). Note that since we have at least three rows, $\neg l$ must be in the row of l. Hence after executing the loop for v, $\mathrm{row}[v]$ contains precisely the vertices in the row of v. Isomorphism-invariance of the IR routine (see Lemma 2.3) ensures that for each row, the order in which symmetrical singletons are colored will be consistent in each row (see Line 8). This ensures that the rows we construct can indeed be transposed (see Line 12 onwards), and the algorithm correctly returns a corresponding matrix. □

Example 3.3. We consider the formula F from Example 2.8. Recall that the rows of the matrix correspond to distinct colors. We individualize a vertex corresponding to a variable $x_{k,v}$ for $k \in [c]$ and $v \in V(G)$. In the corresponding refined coloring, all vertices corresponding to variables $x_{k,w}$ for $w \in V(G)$ are singletons, and thus identified as the entries in the row of $x_{k,v}$.

Recursive Row Symmetry. In practice, orbits often do not exhibit a natural row symmetry. In particular, we consider the case that an orbit of size k, with a natural symmetric action, is connected to another orbit of size ck, where the symmetric action acts on larger blocks of size c (see Figure 7c).

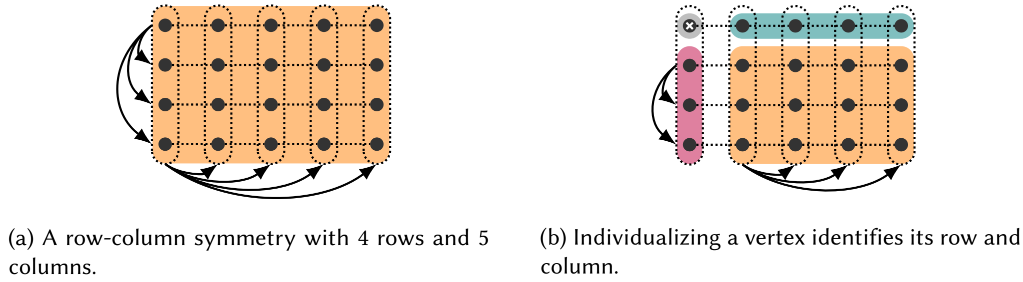
We extend our algorithm to detect this particular case as follows: in Line 7, we add non-singleton fragments of colors instead of only singleton fragments to the row. Let c be a color of π with a fragment c' in π'. We add the vertices $\pi'^{-1}(c')$ to the row, whenever $|\pi'^{-1}(c')||\sigma| = |\pi^{-1}(c)|$. This means we consider vertices of color c', whenever there is the possibility that the color c is split into $|\sigma|$ parts of size $|\pi'^{-1}(c')|$. We call $\pi'^{-1}(c')$ a *block* of its orbit.

On these blocks, we recursively call our algorithm for row symmetry. Essentially, this enables us to detect recursive row symmetry structures. That is, the recursively detected row symmetry will have columns of size c, where each block is a column of the row symmetry.

Example 3.4 (Recursive row symmetry)**.** Consider the symmetries of the formula

$$(x_1 \vee y_{1,1} \vee y_{1,2}) \wedge (x_2 \vee y_{2,1} \vee y_{2,2}) \wedge (x_3 \vee y_{3,1} \vee y_{3,2}) \wedge (y_{1,1} \vee y_{2,1} \vee y_{3,1}) \wedge (y_{1,2} \vee y_{2,2} \vee y_{3,2}).$$

Variables x_1, x_2, x_3 are fully interchangeable by symmetry. However, mapping x_i to x_j requires us to map variables $y_{i,1}, y_{i,2}$ to $y_{j,1}, y_{j,2}$. Hence, the row symmetry acts on blocks of size 2. There are 3 blocks of size 2, namely $\{y_{i,1}, y_{i,2}\}$ for $i \in \{1, 2, 3\}$. The blocks are also connected by row symmetry: exchanging $y_{i,1}$ with $y_{i,2}$, requires us to swap $y_{j,1}$ and $y_{j,2}$ for all j.



(a) A row-column symmetry with 4 rows and 5 columns.

(b) Individualizing a vertex identifies its row and column.

Fig. 8. Illustrations of different aspects of row-column symmetry. The pointwise-stabilizer of an arbitrary vertex will break the original orbit into 4 fragments, which can be used to identify row-column symmetry.

Row Symmetry in Stabilizer. As a slight extension in case that the test for row symmetry fails, we opportunistically recurse on one of the largest fragments from the first IR call and check whether it exhibits row symmetry. This extension is used in the other detection algorithms as well.

3.2 Row-Column Symmetry

Next, we describe a detection algorithm for row-column symmetry. The aim is to determine an arrangement of the variables in the input set σ in a matrix such that $\mathrm{Aut}(F)$ acts on σ by permuting the rows and columns of this matrix. Similarly to the case of row symmetry, individualizing a vertex will split the remaining vertices into three parts, corresponding to the row and column of v and a part containing the remaining vertices. By successively individualizing the vertices in the row of v, one discards the column permutations and thus every vertex can be assigned a column. Proceeding similarly for the columns allows to recover the entire matrix structure.

As discussed in Section 2.4, a disjoint direct factor exhibiting row-column symmetry consists of an orbit of literals and its negation, which is also an orbit of literals. We detect row-column symmetry only on one of these orbits, and expand the resulting automorphisms to the other one: for a permutation φ of $\mathrm{Lit}(F)$ and all $l \in \mathrm{Lit}(F)$, let

$$\mathrm{expand}_F(\varphi)(l) \coloneqq \begin{cases} \varphi(l) & \text{if } l \in \mathrm{supp}(\varphi) \\ \neg\varphi(\neg l) & \text{if } \neg l \in \mathrm{supp}(\varphi) \\ l & \text{otherwise.} \end{cases}$$

Our procedure for row-column symmetry is described in Algorithm 2.

(Description of Algorithm 2.) Given a set $\sigma \subseteq \mathrm{Lit}(F)$, we apply IR to a fixed vertex $v \in \sigma$ (see Figure 8b). Assuming that a row-column symmetry is present, this determines a purported "row" $\mathrm{row}[v]$ and "column" $\mathrm{col}[v]$ of v. The algorithm now successively individualizes the vertices in $\mathrm{row}[v]$ and $\mathrm{col}[v]$. This way, every vertex in σ is assigned a reference vertex in each of $\mathrm{row}[v]$ and $\mathrm{col}[v]$, determining its position in the purported matrix. We then verify that the matrix is well-defined and that every row and column transposition, expanded to $\neg\sigma$, is indeed a symmetry of F. For an illustration of the procedure, see Figure 8.

(Correctness of Algorithm 2.) In order to prove the correctness of Algorithm 2, we first observe the following (compare to Figure 8).

Algorithm 2: Detection algorithm for row-column symmetry.

```
1  function DetectRowColumnSymmetry
       Input : ➢ formula F
               ➢ candidate set σ ⊆ Lit(F)
      Output : ≺ candidate matrix M, or ⊥ if check fails
2      (G, π) ≔ G(F);
3      π′ ≔ IR((G, π), ε);
4      choose arbitrary v ∈ σ;
5      π_v ≔ IR((G, π), v);
6      check that σ has 4 fragments in π_v;
7      label fragments of σ in π_v not containing v as σ_1, σ_2, σ_3 in increasing size;
8      // we determine “coordinates” in matrix relative to v
9      row[v] ≔ col[v] = v ;                          // v defines a row and a column
10     foreach r ∈ σ_1 do
11         row[r] ≔ v, col[r] ≔ r ;            // r is in row of v, and defines a column
12         π_r ≔ IR((G, π′), r);
13         let τ be the fragment of σ in π_r of size |σ_2| not containing v if exists;
14         foreach t ∈ τ do col[t] ≔ r;                           // t is in column of r
15     foreach c ∈ σ_2 do
16         col[c] ≔ v, row[c] = c ;            // c is in column of v, and defines a row
17         π_c ≔ IR((G, π′), c);
18         let τ be the fragment of σ in π_c of size |σ_1| not containing v if exists;
19         foreach t ∈ τ do row[t] ≔ c;                              // t is in row of c
20     construct matrix M where M[r, c] = v′ with row[v′] = r and col[v′] = c;
21     // verify that M exhibits row-column symmetry
22     check that every vertex in σ has a unique row and a unique column label;
23     check that distinct vertices are assigned distinct label pairs;
24     check that M has pairwise disjoint rows, and pairwise disjoint columns;
25     foreach r ∈ σ_1 do check that expand_F(transpose_F(M[∗, r], M[∗, v])) is a symmetry of F;
        // M[∗, x] denotes column of x
26     foreach c ∈ σ_2 do check that expand_F(transpose_F(M[c, ∗], M[v, ∗])) is a symmetry of F ;
        // M[x, ∗] denotes row of x
27     return M
```

Lemma 3.5. *Let $\Gamma = \mathrm{Sym}(n) \times \mathrm{Sym}(m)$ be a row-column symmetry group acting on $[n] \times [m]$. For every $(i, j) \in [n] \times [m]$, the orbit of $(k, l) \in [n] \times [m]$ under the action of $\Gamma_{(i,j)}$ is given by*

$$(k,l)^{\Gamma_{(i,j)}} = \begin{cases} \{(k,l)\} & \text{if } (k,l) = (i,j) \\ \{i\} \times ([m] \setminus \{j\}) & \text{if } k = i,\ l \neq j \\ ([n] \setminus \{i\}) \times \{j\} & \text{if } k \neq i,\ l = j \\ ([n] \setminus \{i\}) \times ([m] \setminus \{j\}) & \text{otherwise.} \end{cases}$$

Proof. Again, we identify $[n] \times [m]$ with the entries of an $n \times m$-matrix M. Then Γ acts on M by permuting the rows and the columns of M. Let $\pi \in \Gamma$ be a permutation that fixes the entry (i, j). Write $\pi = (\pi_r, \pi_c)$, where π_r is a permutation of the rows and π_c a permutation of the columns of M. Then π_r fixes the i-th row and π_c fixes the j-th column of M. On the other hand, every such element of Γ fixes the entry (i, j).

Now consider the orbit of $(k, l) \in [n] \times [m]$ under the stabilizer $\Gamma_{(i,j)}$. By definition, it consists of (k, l) for $(k, l) = (i, j)$. For $k = i$ and $l \neq j$, we can map $(k, l) = (i, l)$ to all elements in the i-row, except for (i, j). Similarly, we argue if $k \neq i$ and $l = j$. Finally, if $k \neq i$ and $l \neq j$, we can map (k, l) to all vertices (k', l') with $k' \neq i$ and $l' \neq j$. This shows the claim. □

We prove that the algorithm always returns correct symmetries of F and that in case the model graph is Tinhofer, it is guaranteed to detect row-column symmetry groups.

Theorem 3.6. *Let F be a SAT formula.*

(1) If Algorithm 2 returns a matrix M of literals, every permutation of the rows or the columns of M, expanded to the negations of the literals in M, is a symmetry of F.
(2) If F exhibits a row-column symmetry with at least three rows and at least three columns on σ and $G(F)$ is a Tinhofer graph, then Algorithm 2 detects this structure and returns a corresponding matrix representation of the literals in σ.

Proof. The first claim is guaranteed by the last part of Algorithm 2 which ensures that transpositions of the rows (Line 25) and columns (Line 26) of the returned matrix M, expanded to the corresponding negated literals, are indeed symmetries of F. By suitably composing such transpositions, we obtain that every permutation of the rows or columns of M induces a symmetry of F in this way.

Now assume that $G(F)$ is Tinhofer and that F exhibits row-column symmetry with at least three rows and columns on σ. In other words, the literals in σ can be arranged in a matrix M on which $\mathrm{Aut}(F)$ acts by row and column permutations (see Figure 8a). Individualizing a fixed vertex $v \in \sigma$ causes σ to split into four fragments according to the orbits of the stabilizer $\mathrm{Aut}(F)_v$: the singleton $\{v\}$, two fragments σ_1 and σ_2 corresponding to the remainders of the row and the column of M containing v, and a fragment σ_3 containing the remaining vertices (see Lemma 3.5 and Figure 8b). Since we assume that M has at least three rows and columns, $\sigma_1, \sigma_2, \sigma_3$ are non-singletons and σ_3 is the largest fragment. Without loss of generality, let $\sigma_1 \cup \{v\}$ be the row and $\sigma_2 \cup \{v\}$ be the column of v in M. Every column of M is determined by the unique element of $\sigma_1 \cup \{v\}$ that it contains (similarly for the rows). Individualizing a vertex $r \in \sigma_1$ leads a similar split of σ into four fragments. The fragments corresponding to the row and column of r can be distinguished by observing that v lies in the same row, but not in the same column as r. For all vertices in the column of r, we store this information (Line 14). Similarly, we proceed for the columns (Line 19). After this procedure, every element of σ is assigned a row and column representative in σ_2 and σ_1 respectively, which, up to a permutation of the rows and columns, allows us to recover the matrix M. □

Example 3.7. We consider the formula F from Example 2.9. In the matrix form, every row corresponds to a pigeon and every column corresponds to a hole. Individualizing a vertex corresponding to a variable x_{ij} for $i \in [n+1]$ and $j \in [n]$, the variables in $\{x_{ik} : k \neq j\}$ and $\{x_{kj} : k \neq i\}$ can be only permuted among themselves, as well as all remaining variables. This way, we have identified which variables belong to the same pigeon (i.e., the same row) or hole (i.e., the same column) as x_{ij}. Successively individualizing the variables in $\{x_{ik} : k \neq j\}$ determines which of the remaining variables are contained in their column, i.e, correspond to the same hole. Proceeding analogously for the pigeons, every variable will be assigned a row and a column in the matrix that it belongs to, i.e., one pigeon and one hole that it corresponds to.

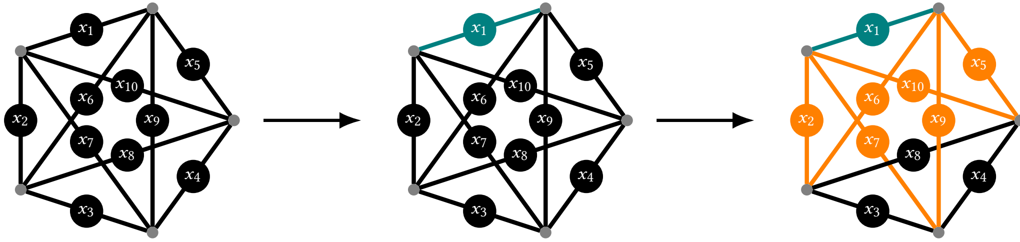


Fig. 9. Individualizing a variable x_1 in a Johnson symmetry (represented by edges in the illustration), splits the set of edges into edges incident to x_1, and edges not incident to x_1.

3.3 Johnson Symmetry

Finally, we describe a procedure to detect Johnson actions. We remark that there is a classic algorithm to detect Johnson groups (Babai et al. 1987). A difference to our heuristic is that we do not know the generators of the group, and instead apply techniques directly on a given graph.

Our aim is to identify the variables in the input set σ with the 2-subsets of $[n]$, where $|\sigma| = \binom{n}{2}$. We thus search for a bijection $b\colon \sigma \to \binom{[n]}{2}$ such that $\mathrm{Aut}(F)$ acts as the Johnson group $\mathcal{J}_n$ on σ via this bijection (see Section 2.4, Figure 4, and Figure 5). To avoid confusion, we refer to the elements of $[n]$ as *labels* and to those of $\mathrm{Lit}(F)$ as *literals* or *vertices of* $G(F)$.

(Description of Algorithm 3.) Suppose that F exhibits a Johnson symmetry on σ. As described above, there is an (unknown) bijection $b\colon \sigma \to \binom{[n]}{2}$ (see Figure 5). We maintain a list label$[v]$ for every $v \in \sigma$, to which we add $i \in [n]$ when we deduce that $i \in b(v)$. If the algorithm returns a list label, a possible bijection b is given by $b(v) = \text{label}[v]$ for all $v \in \sigma$. Note that b is only determined up to permutation of the labels, so our algorithm merely determines vertices obtaining the same label and assigns the labels consecutively.

The algorithm proceeds as follows: we apply IR to $v \in \sigma$, yielding a coloring π_v. Write $b(v) = \{i, j\}$ for some $i, j \in [n]$. The coloring π_v has three fragments: $\{v\}$, the fragment σ_v containing all $u \in \sigma$ with $|b(u) \cap \{i, j\}| = 1$, and the remaining elements (see Figure 9). We call the vertices in σ_v *adjacent* to v and collect them in ad$[v]$. Now choose $w \in \text{ad}[v]$. We can assume $b(w) = \{j, k\}$ for some $k \notin \{i, j\}$. As before, we find the vertices adjacent to w by applying IR to w. Individualizing both v and w, the resulting coloring $\pi_{v,w}$ contains exactly one further singleton consisting of $y \in \sigma$ with $b(y) = \{i, k\}$. Now $\text{ad}[v] \cap \text{ad}[w] = \{y\} \cup \{u \in \sigma\colon b(u) = \{j, r\} \text{ for some } r \notin \{i, j, k\}\}$. The vertices in $\text{ad}[v] \cap \text{ad}[w] \setminus \{y\}$ thus obtain the label j. Similarly, we determine the vertices obtaining the label i or k. After ensuring that the labels have not been considered previously, we add them to the list label for the respective vertices.

Example 3.8. Consider the formula $F = \bigwedge_{\{a,b,c\} \in S}(x_a \vee x_b \vee x_c)$ with

$$S = \{\{1, 2, 6\}, \{1, 7, 9\}, \{1, 5, 10\}, \{3, 6, 9\}, \{5, 6, 8\}, \{4, 5, 9\}, \{2, 3, 7\}, \{2, 8, 10\}\{3, 4, 8\}\}.$$

In fact, this formula encodes the (obviously silly) question whether the complete graph on 5 vertices contains a triangle. Every variable $x_1, \ldots, x_{10}$ corresponds to an edge in the complete graph. For an illustration, see Figure 9.

Using the above procedure, we would like to show that F exhibits a Johnson symmetry for $n = 5$ on $\sigma = \{x_1, \ldots, x_{10}\}$. That is, we try to find a bijection $\sigma \to \binom{[5]}{2}$. Basically, this means that we want to assign vertex labels to the edges in the complete graph, to which the variables in σ correspond. We first individualize x_1. The fragments of the resulting coloring are $\{x_1\}$, $\{x_2, x_5, x_6, x_7, x_9, x_{10}\}$ and $\{x_3, x_4, x_8\}$. In the underlying complete graph, this means that we distinguish edges that share an endvertex with the edge corresponding to x_1 from those that do not.

Algorithm 3: Detection algorithm for Johnson actions.

1 **function** `DetectJohnson`
    Input : ➢ formula F
    ➢ candidate set $\sigma \subseteq \mathrm{Lit}(F)$
    Output : ≺ bijective labeling of σ by 2-subsets of $[n]$, or $\perp$ if check fails
2 $(G, \pi) \coloneqq G(F)$, $\pi' \coloneqq \mathrm{IR}((G, \pi), \varepsilon)$, vnr = 1;
3 check that $|\sigma| \geq 28$ and $|\sigma| = \binom{n}{2}$ for some $n \in \mathbb{N}$;
4 **foreach** *vertex* $v \in \sigma$ **do** set label$[v]$ = [ ];
5 **while** *there are vertices* $v \in \sigma$ *with* $|label[v]| \leq 1$ **do**
6 $E_i \coloneqq E_j \coloneqq E_k \coloneqq \{\}$;
7 choose $v \in \sigma$ with $|\mathrm{label}[v]| \leq 1$;
8 $\pi_v \coloneqq \mathrm{IR}((G, \pi'), v)$;
9 check that number of fragments of σ in π_v is 3;
10 let σ_v be the smaller non-singleton fragment;
11 **foreach** $x \in \sigma_v$ **do** add x to ad$[v]$;
12 choose arbitrary $w \in$ ad$[v]$;
13 $\pi_w \coloneqq \mathrm{IR}((G, \pi'), w)$;
14 check that number of fragments of σ in π_w is 3;
15 let σ_w be the smaller non-singleton fragment;
16 **foreach** $x \in \sigma_w$ **do** add x to ad$[w]$;
17 $\pi_{v,w} \coloneqq \mathrm{IR}((G, \pi_v), w)$;
18 let $\{y\}$ be the unique singleton fragment of σ in $\pi_{v,w}$ different from $\{v\}$ and $\{w\}$ if existent, otherwise **return** $\perp$;
19 $\pi_y \coloneqq \mathrm{IR}((G, \pi'), y)$;
20 check that number of fragments of σ in π_y is 3;
21 let σ_y be the smaller non-singleton fragment;
22 **foreach** $x \in \sigma_y$ **do** add x to ad$[y]$;
23 add v to E_i and E_j, add w to E_j and E_k, add y to E_i and E_k;
24 **foreach** $x \in ad[v] \cap ad[y]$ *and* $x \neq w$ **do** add x to E_i;
25 **foreach** $x \in ad[v] \cap ad[w]$ *and* $x \neq y$ **do** add x to E_j;
26 **foreach** $x \in ad[w] \cap ad[y]$ *and* $x \neq v$ **do** add x to E_k;
27 **foreach** $E \in \{E_i, E_j, E_k\}$ **do**
28 **if** $\bigcap_{v \in E} label[v] = \emptyset$ **then**
29 append vnr to label$[v]$ for $v \in E$;
30 vnr += 1;
31 check that new labels were added to label in this iteration;
32 verify that label induces a bijection between σ and $\binom{[n]}{2}$;
33 **foreach** $i \in [n-1]$ **do**
34 let β denote the permutation of σ induced by the Johnson action induced by $(i, i+1) \in \mathrm{Sym}(n)$ using label;
35 check that $\mathrm{expand}_F(\beta)$ is a symmetry of F;
36 **return** *label*

Now additionally individualizing $\{x_2\}$, we can analogously distinguish edges that share an endvertex with the edge corresponding to $\{x_2\}$ with those that do not. In particular, $\{x_6\}$ becomes a singleton, and thus the edges corresponding to x_1, x_2, x_6 form a triangle in the complete graph. Hence we know that we can assign them labels of the form $\{i,j\}$, $\{j,k\}$, and $\{i,k\}$ with distinct i,j,k. Note that the values for the labels i,j,k can be chosen arbitrarily in $[n]$. Different choices correspond to relabelings of the vertices of the underlying complete graph. Continuing this way, we successively identify which variables correspond to edges sharing an endvertex. (We remark that technically, the example is slightly too small for Algorithm 3 to work as described.)

We now proceed with the formal analysis of Algorithm 3.

(Correctness of Algorithm 3.) We make the following observation on stabilizers in Johnson groups:

Lemma 3.9. *Let $n \in \mathbb{N}$ and consider the Johnson group $\Gamma \coloneqq \mathcal{J}_n$, acting on 2-subsets of $[n]$.*

(1) For $\{i,j\} \in \binom{[n]}{2}$, the orbit of $S \in \binom{[n]}{2}$ under the stabilizer $\Gamma_{\{i,j\}}$ of $\{i,j\} \in \binom{[n]}{2}$ is given by

$$S^{\Gamma_{\{i,j\}}} = \begin{cases} \{S\} & \text{if } S = \{i,j\} \\ \{T \in \binom{[n]}{2} : |T \cap \{i,j\}| = 1\} & \text{if } |S \cap \{i,j\}| = 1 \\ \{T \in \binom{[n]}{2} : T \cap \{i,j\} = \emptyset\} & \text{if } S \cap \{i,j\} = \emptyset. \end{cases}$$

(2) For $k \neq i,j$, the orbit of $S \in \binom{[n]}{2}$ under $\Gamma_{\{i,j\}} \cap \Gamma_{\{i,k\}}$ is given by

$$S^{\Gamma_{\{i,j\}} \cap \Gamma_{\{i,k\}}} = \begin{cases} \{S\} & \text{if } S \in \{\{i,j\},\{i,k\},\{j,k\}\} \\ \{\{i,r\} : r \in [n] \setminus \{i,j,k\}\} & \text{if } S = \{i,s\} \text{ with } s \in [n] \setminus \{i,j,k\} \\ \{\{j,r\} : r \in [n] \setminus \{i,j,k\}\} & \text{if } S = \{j,s\} \text{ with } s \in [n] \setminus \{i,j,k\} \\ \{\{k,r\} : r \in [n] \setminus \{i,j,k\}\} & \text{if } S = \{k,s\} \text{ with } s \in [n] \setminus \{i,j,k\} \\ \{S \in \binom{[n]}{2} : S \cap \{i,j,k\} = \emptyset\} & \text{if } S \cap \{i,j,k\} = \emptyset. \end{cases}$$

Proof.

(1) Let $S \in \binom{[n]}{2}$. If $S = \{i,j\}$, the orbit $S^{\Gamma_{\{i,j\}}}$ consists only of S by definition of the stabilizer. Now suppose that $|S \cap \{i,j\}| = 1$ holds. Without loss of generality, let $S = \{i,r\}$ for some $r \in [n] \setminus \{i,j\}$. Let $\pi \in \Gamma_{\{i,j\}}$. Either π fixes i and j, in which case we have $S^\pi = \{i,r'\}$ for some $r' \in [n] \setminus \{i,j\}$, or π interchanges i and j, in which case we have $S^\pi = \{j,r'\}$ for some $r' \in [n] \setminus \{i,j\}$. In both cases, we have $|S^\pi \cap \{i,j\}| = 1$. On the other hand, it is easy to see that for every set $T \in \binom{[n]}{2}$ with $|T \cap \{i,j\}| = 1$, there exists $\pi \in \Gamma_{\{i,j\}}$ with $S^\pi = T$. The description of $S^{\Gamma_{\{i,j\}}}$ in the case $S \cap \{i,j\} = \emptyset$ can be derived analogously.

(2) Note that an element in $\Gamma_{\{i,j\}} \cap \Gamma_{\{i,k\}}$ fixes or interchanges the labels i and j, and at the same time fixes or interchanges the labels i and k. This is only possible if it fixes all of i, j and k. The structure of the orbits then follows similarly to the first claim. □

We now prove that the algorithm always returns correct symmetries of F and that in case the model graph is Tinhofer, the algorithm is guaranteed to detect that F exhibits a Johnson symmetry on the input set σ.

Theorem 3.10. *Let F be a SAT formula.*

(1) If Algorithm 3 returns a list label *of labels in $[n]$, then for every element in $\mathcal{J}_n$, the induced permutation of σ according to label, expanded to $\neg\sigma$, is a symmetry of F.*

(2) If F exhibits a Johnson symmetry with Johnson group $\mathcal{J}_n$ with $n \geq 8$ on σ and $G(F)$ is a Tinhofer graph, then Algorithm 3 detects this structure and returns a corresponding labeling of the literals in σ by 2-subsets of $[n]$.

Proof. The last part of Algorithm 3 ensures that the Johnson action j_π induced by a transposition $\pi \coloneqq (i, i+1) \in \mathcal{J}_n$ by permuting the elements in σ according to their labels in *label* is a symmetry of F when expanded

to $\neg\sigma$. By suitably composing these transpositions, it follows that every element of $\mathcal{J}_n$ induces a symmetry of F in this way.

Now suppose that F exhibits a Johnson symmetry with Johnson group $\mathcal{J}_n$ with $n \geq 8$ (i.e., $|\sigma| \geq 28$). Furthermore, assume that $G(F)$ is Tinhofer. In particular, there is a bijection $b\colon \sigma \to \binom{[n]}{2}$ (see Figure 5). We claim that when the algorithm terminates, there is a permutation $\tau \in \mathrm{Sym}(n)$ of the label set $[n]$ such that we have $\mathrm{label}[v] = \{\tau(i), \tau(j)\}$ if $b(v) = \{i, j\}$. Note that the bijection b itself is determined only up to permutation of the labels. Again, for the sake of clarity, we refer to the elements of $[n]$ as *labels* and reserve the term *vertices* for the vertices of the graph $G(F)$.

The individualization of a vertex v with $b(v) = \{i, j\}$ (Line 8) leads to a color partition with three fragments since $G(F)$ is Tinhofer (see Lemma 3.9 and Figure 9). The smaller non-singleton fragment is $\sigma_v = \{u \in \sigma\colon |b(u) \cap \{i, j\}| = 1\}$. For this, note that $|\sigma_v| = 2(n-2)$ holds and that we have $n \geq 8$ by assumption. The list $\mathrm{ad}[v]$ (Line 11) then consists of all vertices $u \in \sigma$ with $b(u) = \{i, r\}$ or $b(u) = \{j, r\}$ with $r \in [n] \setminus \{i, j\}$.

Now let $w \in \mathrm{ad}[v]$. Up to this point, the labels i and j are interchangeable, so we may assume $b(w) = \{j, k\}$ for some $k \in [n] \setminus \{i, j\}$. We repeat the above procedure with w in place of v. In particular, $\mathrm{ad}[w]$ (Line 16) contains all vertices $u \in \sigma$ with $b(u) = \{j, r\}$ or $b(u) = \{k, r\}$ for $r \in [n] \setminus \{j, k\}$.

Finally we individualize both v and w to obtain the coloring $\pi_{v,w}$. The fragments are given by Lemma 3.9. In particular, we obtain $b(y) = \{i, k\}$. Apart from y, the intersection $\mathrm{ad}[v] \cap \mathrm{ad}[w]$ contains all vertices $u \in \sigma$ with $b(u) = \{j, r\}$ for $r \in [n] \setminus \{i, j, k\}$, and we add them to E_j (Line 25). Similarly, we construct the sets E_i and E_k (Lines 24 and 26).

From this explicit description, it is clear that $u \in \sigma$ is added to E_i precisely if $i \in b(u)$ (similarly for E_j and E_k). In particular, for distinct vertices $u_1, u_2 \in E_i$, we have $b(u) \cap b(v) = \{i\}$. Thus if the lists $\mathrm{label}[u]$ for $u \in E_i$ have a common entry, the label i has been considered before (recall that $|E_i| > 1$ holds). Otherwise, we add the current vertex number *vnr* to $\mathrm{label}[u]$ for all $u \in E_i$ (Line 27) and set $\tau(i) = \mathrm{vnr}$. This way, $\mathrm{label}[u]$ remains duplicate-free and only ever contains labels $\tau(l)$ for $l \in b(u)$. In particular, we always maintain the property $|\mathrm{label}[u]| \leq 2$. In each iteration of the while loop, one of the labels i and j was not considered before (due to $|\mathrm{label}[v]| \leq 1$). In particular, the loop is executed at most n times. When it stops, we have $|\mathrm{label}[v]| = 2$ for all vertices v. □

(Runtime of Algorithm 3.) Let n and m refer to the number of vertices and edges of $G(F)$. In each iteration of the main loop starting in Line 5, we assign at least one label to one of the edges. Hence, the main loop runs for $O(|\sigma|)$ many iterations, performing $O(|\sigma|)$ individualizations. All other operations of an iteration can be implemented in $O(|\sigma|)$. Using the fact that $|\sigma| \leq n$, we get an upper bound estimate of $O(|\sigma|(n+m)\log n)$

Example 3.11. We consider the formula F encoding the Ramsey problem from Example 2.11. Individualizing a vertex corresponding to a variable $x_{i,j}$ and applying the IR procedure distinguishes the vertices corresponding to variables $x_{k,l}$ with $\{k, l\} \cap \{i, j\} \neq \emptyset$ from the ones corresponding to variables $x_{k,l}$ with $\{k, l\} \cap \{i, j\} = \emptyset$, as the action of $\mathrm{Aut}(F)$ on the recolored graph will preserve these two sets. Translated to the underlying graph-theoretic Ramsey problem, this means that we distinguish the edges that share an endvertex with the fixed edge $\{i, j\}$ from those which do not share such an endvertex.

Johnson Action on Row Symmetry. Quite commonly, SAT instances which search for a graph, will search for a graph with a certain *vertex property*. For example, when asking for a k-colorable graph, there are (interchangeable) colors attached to each vertex of the graph. In order to detect a corresponding symmetry structure, we would like to detect blocks which correspond to the labels in the Johnson domain. The detection works by stabilizing vertices in other orbits, and checking whether they split the Johnson orbit into the vertices marked with a particular label and a remainder. If so, these blocks are collected and considered in the Johnson action. Finally, we run row symmetry detection on the collected blocks.

4 Implementation

We now give an overview of our new symmetry breaking tool SATSUMA. The input of our algorithm is a CNF formula F. The output is an equi-satisfiable CNF formula F'.

4.1 High-level Algorithm

We begin with a description of the high-level procedure implemented in SATSUMA. The high-level algorithm proceeds as follows:

(Step 1, Preprocessing.) Preprocess the CNF formula F using the unit and pure literal rule.

(Step 2, Remove Duplicates.) Remove duplicate clauses from the formula, by making a hash set with all clauses. To improve efficiency, there are additional separate hash sets for binary and ternary clauses.

(Step 3, Model Graph.) Construct a model graph from the given CNF formula using the optimized encoding described in Section 2.2.

(Step 4, Color Refinement.) Run color refinement on the obtained model graph. Non-trivial colors are used as candidates for the special detection algorithms.

(Step 5, Special Detection.) Run the algorithms described in the previous section on all non-trivial colors obtained in the last step. The algorithms are used in the following order:

(1) Johnson groups (Algorithm 3),
(2) row-column symmetry (Algorithm 2),
(3) row interchangeability (Algorithm 1).

The implementation uses further simple heuristics to quickly discard candidates. (For example, the algorithm probes the number of possible consecutive individualizations, which can be used to immediately discard certain candidates.)

Whenever a structure is found, all orbits covered by the structure are marked. Subsequent special detection algorithms only consider *unmarked* orbits. For each structure, we collect a specific set of generators. Indeed, we collect precisely the generators constructed in Algorithm 1, Algorithm 2, and Algorithm 3. Lastly, we maintain a vertex coloring of the model graph, which we call the *remainder coloring*: this coloring restricts the symmetries of the model graph to symmetries not yet covered by the detected groups.

(Step 6, General Detection.) Run symmetry detection (DEJAVU) on the graph colored with the remainder coloring.

(Step 7, Produce Breaking Constraints.) Lastly, produce lex-leader constraints for each collected generator. Before we can produce lex-leader constraints, we must however fix an ordering on the variables. The ordering used for matrix models simply orders the matrix row-by-row. For Johnson groups, we begin with the vertices of the first label (see Algorithm 3), then the remaining vertices of the second label, and so forth.

For parts not involved in particular structures, we use an ordering heuristic as introduced by BREAKID. Variables are ordered according to how often they occur in generators in ascending order. That is, the least-used variables appear first.

While duplicate literals and duplicate clauses are removed, SATSUMA will otherwise keep the original order of clauses, as well as the original order of literals in clauses.

4.2 Implementation Details

Technical Details. The tool is written in C++, and is freely available as open source software[1]. The tool DEJAVU (Anders 2024; Anders and Schweitzer 2021; Anders, Schweitzer, and Stieß 2023) is used for providing general-purpose symmetry detection, the individualization-refinement framework, and data structures for symmetries. We compiled SATSUMA with DEJAVU version 2.1.

[1]https://github.com/markusa4/satsuma, archived under swh:1:rev:98ff66db835140ee51fa5ef01b9aa274f9accc4c

Table 1. Benchmarks comparing solving times with no symmetry breaking and BreakID to satsuma. The SAT solver used is Kissat (kis). The timeout used is 5000 seconds. The columns "prep" refer to the time in seconds used by the respective symmetry breaking tool. Columns "solved" refer to the number of solved instances within the time limit, and "avg" is the average total time (*including* the time used for symmetry breaking).

family		Kissat		BreakID+kis			satsuma+kis		
name	size	solved	avg	prep	solved	avg	prep	solved	avg
sat2024	400	318	1476	58.3	322	1439	**13.1**	**330**	**1302**
php	16	2	4380	43.3	12	1259	**8.7**	**16**	**10.3**
coloring	55	22	3015	38.4	29	**2494**	**0.15**	**30**	2534
channel	10	2	4006	6.8	**10**	6.9	**0.15**	**10**	**1.31**
fpga	19	**19**	113	0.07	**19**	**0.14**	**0.01**	**19**	0.59
urquhart	6	2	3384	0.1	**3**	**2500**	**0.04**	**3**	**2500**
ramsey	14	6	3001	3.24	6	2862	**0.73**	**10**	**1452**
cliquecolor	21	9	3620	0.27	12	1945	**0.1**	**21**	**0.4**

Some parts of the implementation, in particular the generation of lex-leader constraints, are based on BreakID. We mention two technical differences between BreakID and satsuma. First, BreakID uses the symmetry detection tool saucy (Darga et al. 2004) instead of dejavu. Second, we use a more careful implementation of data structures and algorithms for the handling of symmetries. This in particular involves the way generators are stored, the orbit algorithm, and the way lex-leader constraints are computed.

Limits. We want to make sure that symmetry breaking does not take up too much time, and thus does not interfere with the SAT solver. In order to achieve this, we apply reasonable limits to our algorithms. This is standard for SAT preprocessing techniques. Even somewhat basic techniques, such as subsumption, are usually applied under certain resource limits.

By default, we use the following limits.

(1) We limit all special detection algorithms to only test 64 orbits. We found that the impact of the special detection algorithms is greatest on instances with few, large orbits. On instances with many orbits, the special detection algorithms sometimes incur significant overhead.
(2) Color refinement in special detection algorithms is limited to 16 million split operations. This prevents rare, expensive edge cases for very large graphs.
(3) We impose the following limits on dejavu. We skip components that exceed 20 million vertices, and stop if the size of the so-called Schreier table, which stores symmetry, exceeds 2GB. The algorithm furthermore contains a "budget" to track a particular kind of backtracking, not caused by symmetry. We allow a maximum "budget" of 64. In essence, limiting this budget mitigates any chance for exponential-time backtracking on difficult graphs. We refer to (Anders 2024) for more details.

We want to stress that in our testing, the limits in (3) only trigger on extremely large, highly symmetrical, or exceptionally difficult graphs. Furthermore, the limit for the "budget" did not improve running time in our testing. However, in a setting where we are trying to speed up a SAT solver with as little overhead as possible, we believe that it is a reasonable design choice to impose such a limit. All imposed limits can of course be overridden manually.

5 Benchmarks

We compare the state-of-the-art static symmetry breaking tool BreakID version 2.6 to satsuma.

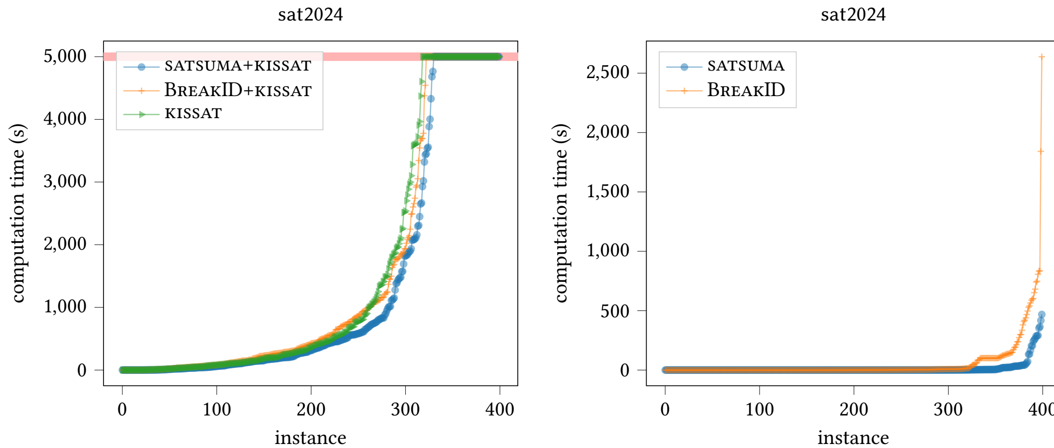


Fig. 10. Detailed plots for the SAT 2024 competition instances. The left plot shows overall performance, whereas the right plot shows the overhead incurred by the respective symmetry breaking tool.

As SAT solver, we use Kissat version 4.0.1 (Biere et al. 2024). The timeout for all benchmarks is 5000 seconds, which is the typical timeout used in the main track of the SAT competition. We separately measure the time spent on symmetry breaking and SAT solving. All benchmarks ran on a server using two Intel Xeon Silver 4214 with 12 cores each (24 cores total), 128GB of memory, running on Ubuntu 20.04. We ran up to 16 benchmarks in parallel.

For Kissat we used the default configuration. For BreakID, we mostly matched the configuration used in the SAT competition 2024. This means we deactivated the binary clause heuristic, used non-relaxed symmetry breaking constraints, and a limit of 100 for the depth of each individual breaking constraint. In the competition, BreakID used an internal time limit for symmetry detection of 300 seconds. In preliminary testing, we did however achieve significantly better results using a time limit of only 100 seconds, which also matches the limit used in (Devriendt, Bogaerts, Bruynooghe, and Denecker 2016). Note that this time limit is purely for *detection* and the tool may indeed run significantly longer.

We remark that both satsuma and BreakID implement similar heuristics for adding binary clauses (also called Schreier-Sims table cuts). Since we used BreakID competition settings, we disabled these binary clauses in *both* tools to ensure a fair comparison. It should, however, be mentioned that this heuristic is usually very effective on Tseitin instances.

Benchmark Instances. We run benchmarks on a variety of well-established instance families exhibiting symmetry (see Table 1) and, additionally, on the instances used in the main track of the most recent SAT competition.

The set sat2024 contains the instances used in the main track of the SAT competition 2024. The sets coloring, urquhart, fpga, and channel are part of the distribution of BreakID (Devriendt, Bogaerts, Bruynooghe, and Denecker 2016). We generate pigeonhole principle (php) instances, Ramsey instances, and clique coloring instances using the tool cnfgen (Lauria et al. 2017). The set of parameters for clique coloring is similar to (Junttila, Karppa, et al. 2020), but we added larger instances.

The set sat2024 contains a mix of satisfiable and unsatisfiable instances. The set fpga contains 10 unsatisfiable and 9 satisfiable instances. All instances of the remaining sets are unsatisfiable.

Regarding the detected symmetry structures of these instances, we find Johnson symmetry on the ramsey and cliquecolor families. On php, channel, and fpga, satsuma detects row-column symmetry, and BreakID the corresponding row interchangeability (see also (Devriendt, Bogaerts, Bruynooghe, and Denecker 2016; Sabharwal

2009)). The coloring instances exhibit a variety of different symmetries, but in particular also row symmetry (Devriendt, Bogaerts, Bruynooghe, and Denecker 2016). On urquhart, no structure is detected by either tool. The SAT competition instances sat2024 contain a large variety of symmetry structures.

SAT Benchmarks. An overview of the results can be found in Table 1. For the sat2024 set, Figure 10 contains a detailed plot of the results.

For the SAT competition set, we observe more solved instances and better average solving time of SATSUMA compared to the other configurations. Average solving time of the SATSUMA configuration is about 12% lower than for KISSAT, and 10% lower than the BREAKID configuration. The average PAR2 scores are 2501 for KISSAT, 2414 for the BREAKID configuration, and 2177 for the SATSUMA configuration. Hence, the PAR2 score of the SATSUMA configuration is about 13% lower compared to KISSAT.

Notably, the SAT 2024 set contains 6 clique coloring instances with Johnson symmetry, as well as 3 instances of variations of the pigeonhole principle (relativized and binary), which contain row symmetry. Furthermore, there are 8 instances of different varieties of Tseitin formulas, which mostly contain negation symmetry. On these families, SATSUMA, as expected, outperforms KISSAT. In total, SATSUMA solves 18 instances which KISSAT does not, while KISSAT solves 6 instances which SATSUMA does not. The losses are predominantly on *satisfiable* instances.

Looking at the preprocessing times, we can observe that around half of the speedup compared to BREAKID is due to the reduced computational overhead, and the other half due to improved SAT solver performance on the resulting instances.

Considering the results on the symmetry benchmarks, we observe that SATSUMA solves more instances, and average solving times are considerably lower on the cliquecolor and ramsey instances. We recall that these instances exhibit Johnson symmetry. On most sets with row and row-column symmetry, that is channel, coloring, and fpga, we observe that the number of solved instances and the average solving times are comparable. On the larger instances of php, BREAKID is not able to detect all symmetries within the 100s detection time limit. For the coloring family, we observe that SATSUMA solves one more instance than BREAKID.

In particular, we point out that SATSUMA compares favorably on instance families which exhibit Johnson symmetry. We believe this to be due to our detection of Johnson symmetry and the subsequent generation of more favorable constraints. Crucially, on all successfully solved instances of cliquecolor and ramsey, the *remainder contains no symmetry*: all symmetries are detected and in turn broken solely using the algorithms of this paper, and no general-purpose symmetry detection and breaking is applied.

We observe that the average time spent computing the symmetry breaking constraints is lower on all families for SATSUMA. A detailed analysis follows below.

Computational Overhead. We conduct further benchmarks to more precisely gauge the efficiency of BREAKID and SATSUMA. In order to gain a better understanding of the efficiency of the underlying algorithms, we deactivate all limits imposed by the tools. This means, we remove the 100s symmetry detection limit in BREAKID. In SATSUMA, we do not apply any of the limits discussed in Section 4.

In particular, we investigate how the algorithms scale. We test three different benchmark families: php, cliquecolor, and urquhart (generated using CNFGEN). For php, we increase the number of pigeons from 10 to 500 (with $n-1$ holes, respectively). For cliquecolor, we increase the number of vertices of the prospective graph from 10 to 1000 (the size of the clique is 3 and number of colors is 2). In urquhart, we use random 5-regular graphs, increasing the number of vertices from 10 to 5000. The chosen instance families cover different symmetry detection routines in SATSUMA: the family php essentially measures the runtime of our row-column routine, cliquecolor that of the Johnson routine, and urquhart uses general purpose symmetry detection and breaking.

Table 11 summarizes the results. In all instance families, SATSUMA substantially outperforms BREAKID. In particular, on the php and urquhart instances, the data suggest better asymptotic scaling of SATSUMA. These results agree with our observations regarding overhead from the first part of the benchmarks (see Table 1).

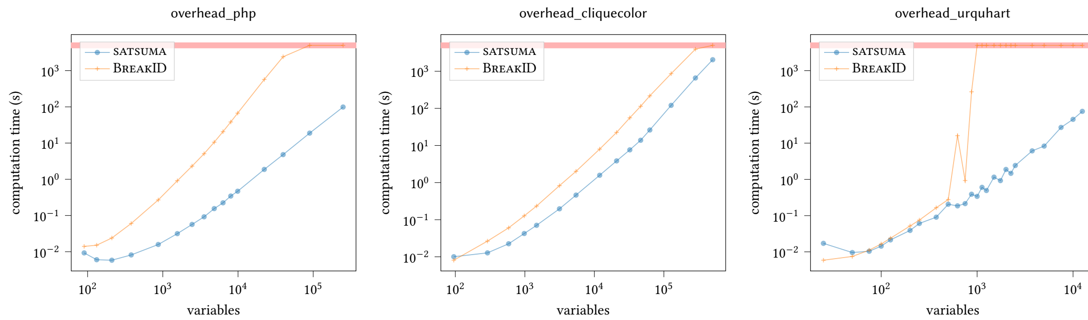


Fig. 11. Benchmarks comparing the computational overhead of BreakID to satsuma. The shown computation time is the time spent on computing symmetry breaking constraints for an instance using the respective tool. The red bar indicates the timeout of 5000 seconds.

We believe there are multiple reasons why satsuma runs faster than BreakID. First, we use a more compact graph encoding. Second, our new algorithms of Section 3 verify symmetries on the CNF formula instead of the model graph. This is advantageous because symmetries of the CNF only explicitly map literals, whereas symmetries of the model graph also explicitly map clauses. Third, the runtime of most routines in our implementation run proportional in the size of the *support* of symmetries, as opposed to the number of literals of F. Fourth, for general-purpose symmetry detection, dejavu seems to be more efficient in computing automorphism groups of SAT instances than saucy (Anders 2024). We mention that in particular in the urquhart instances, the high running time of BreakID is due to saucy taking a long time to compute symmetries.

Ablation Study. Lastly, we study the impact of individual components of the implementation. In particular, we test the effects of the *special detection algorithms*, as well as the *SAT preprocessing techniques* applied before symmetry breaking. The results are summarized in Table 2. The configuration no-special uses none of the special detection algorithms described in this paper, skipping Step (5) of the high-level algorithm described in Section 4.1. Instead, all symmetries become part of the remainder and are treated using the generic strategies in Step (6) and Step (7). The configuration no-preprocess does not use CNF preprocessing before symmetry breaking, skipping Step (1) of the high-level algorithm.

We observe that the special detection algorithms dramatically improve performance across most of the benchmark families. While overhead is sometimes slightly lower, this is clearly outweighed by improved SAT solver performance. While the impact of the SAT preprocessing techniques is not as dramatic, the results still seem to suggest that it is beneficial to use them. In particular on the SAT competition instances, more instances are solved while computational overhead is reduced.

6 Conclusions and Future Work

We described a new structure-based approach to symmetry breaking, and demonstrated both the effectiveness and efficiency of our new implementation satsuma. We believe there are many promising directions in which the present work could be expanded:

- Detect more group structures: in particular, a more generic approach to detect aggregates of groups would be of great interest. Another interesting case might be the symmetries of the family urquhart, which are isomorphic to C_2^k and have been studied previously (Luks and Roy 2004).
- Consider other breaking approaches for certain group structures. So far, we used the knowledge of group structures to pick out automorphisms, for which off-the-shelf lex-leader constraints are generated. Since

Table 2. Benchmarks comparing different configurations of SATSUMA. The first column NO-SPECIAL does not perform the special detection algorithms for group structures. The second column NO-PREPROCESS does not perform SAT preprocessing before symmetry breaking. The last column is the default configuration, as tested in previous benchmarks. The SAT solver used is KISSAT (KIS). The timeout used is 5000 seconds. The columns "prep" refer to the time in seconds used by the symmetry breaking tool. Columns "solved" refer to the number of solved instances within the time limit, and "avg" is the average total time (*including* the time used for symmetry breaking).

family		NO-SPECIAL+KIS			NO-PREPROCESS+KIS			SATSUMA+KIS		
name	size	prep	solved	avg	prep	solved	avg	prep	solved	avg
sat2024	400	12.5	321	1406	13.7	327	1327	**13.1**	**330**	**1302**
php	16	12.2	2	4375	**8**	**16**	9.8	**8.7**	**16**	**10.3**
coloring	55	**0.14**	22	3045	0.17	29	**2526**	0.15	**30**	2534
channel	10	**0.15**	2	4000	0.18	**10**	1.5	**0.15**	**10**	**1.31**
fpga	19	0.04	**19**	1.45	0.04	**19**	**0.67**	**0.01**	**19**	**0.59**
urquhart	6	**0.06**	**3**	**2500**	**0.06**	**3**	**2500**	**0.04**	**3**	**2500**
ramsey	14	**0.6**	6	2866	0.7	**10**	**1456**	0.73	**10**	1452
cliquecolor	21	0.11	8	3476	0.12	**21**	0.48	**0.1**	**21**	**0.4**

optimal handling of row-column symmetry and Johnson symmetry is known to be difficult problem (Anders, Brenner, et al. 2024b; Luks and Roy 2004), other breaking constraints could lead to better results. It could even be feasible to switch to tailored dynamic techniques depending which group structure is detected. For example, Johnson symmetry allows the use of symmetry reduction developed specifically for graph generation (Codish, Gange, et al. 2016; Codish, Miller, et al. 2019; Kirchweger and Szeider 2021).

- Another interesting direction could specifically concern improved heuristics for handling of the "remainder". As already pointed out in (Devriendt, Bogaerts, Bruynooghe, and Denecker 2016), one potential improvement could be to apply the random Schreier-Sims algorithm (Seress 2003) to produce more small symmetry breaking clauses.
- An enticing feature is proof-logging, as was recently introduced to BREAKID (Bogaerts, Gocht, et al. 2023) through the use of the VERIPB proof system.
- The new detection algorithms could be applied in other domains as well: for example, seeing as row interchangeability is also successfully used in MIP, it seems only natural that MIP instances may also contain richer structures.
- Sometimes symmetries are not present in a compiled CNF of a given problem, as, e.g., pointed out in (Junttila, Karppa, et al. 2020; Knuth 2022). A possible remedy is to allow the user to provide an auxiliary graph that models the original symmetry as is described in (Junttila, Karppa, et al. 2020). The methods proposed in this paper should readily generalize to this setting.

Acknowledgements

Supported by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (EngageS: grant No. 820148). Markus Anders acknowledges funding by the European Union (ERC, CertiFOX, 101122653). Views and opinions expressed are however those of the authors only and do not necessarily reflect those of the European Union or the European Research Council. Neither the European Union nor the granting authority can be held responsible for them. Sofia Brenner acknowledges funding from the German Research Foundation DFG (SFB-TRR 195 "Symbolic Tools in Mathematics and their Application" as well as grant 522790373) as well as a postdoc fellowship by the German Academic Exchange Service (DAAD).

References

F. A. Aloul, I. L. Markov, and K. A. Sakallah. 2003. “Shatter: efficient symmetry-breaking for boolean satisfiability.” In: *Proceedings of the 40th Design Automation Conference, DAC 2003, Anaheim, CA, USA, June 2-6, 2003*. ACM, 836–839. doi:10.1145/775832.776042.

F. A. Aloul, A. Ramani, I. L. Markov, and K. A. Sakallah. 2003. “Solving difficult instances of Boolean satisfiability in the presence of symmetry.” *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, 22, 9, 1117–1137. doi:10.1109/TCAD.2003.816218.

M. Anders. 2024. “Efficient Algorithms for Symmetry Detection.” en. Ph.D. Dissertation. Technische Universität Darmstadt, Darmstadt, xi, 205 Seiten. doi:https://doi.org/10.26083/tuprints-00028257.

M. Anders. 2022. “SAT Preprocessors and Symmetry.” In: *25th International Conference on Theory and Applications of Satisfiability Testing, SAT 2022, August 2-5, 2022, Haifa, Israel* (LIPIcs). Vol. 236. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 1:1–1:20. doi:10.4230/LIPICS.SAT.2022.1.

M. Anders, S. Brenner, and G. Rattan. 2024a. “Satsuma: Structure-Based Symmetry Breaking in SAT.” In: *27th International Conference on Theory and Applications of Satisfiability Testing (SAT 2024)* (Leibniz International Proceedings in Informatics (LIPIcs)). Ed. by S. Chakraborty and J.-H. R. Jiang. Vol. 305. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 4:1–4:23. ISBN: 978-3-95977-334-8. doi:10.4230/LIPIcs.SAT.2024.4.

M. Anders, S. Brenner, and G. Rattan. 2024b. “The Complexity of Symmetry Breaking Beyond Lex-Leader.” In: *30th International Conference on Principles and Practice of Constraint Programming, CP 2024, September 2-6, 2024, Girona, Spain* (LIPIcs). Ed. by P. Shaw. Vol. 307. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 3:1–3:24. doi:10.4230/LIPICS.CP.2024.3.

M. Anders and P. Schweitzer. 2021. “Parallel Computation of Combinatorial Symmetries.” In: *29th Annual European Symposium on Algorithms, ESA 2021, September 6-8, 2021, Lisbon, Portugal (Virtual Conference)* (LIPIcs). Vol. 204. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 6:1–6:18. doi:10.4230/LIPIcs.ESA.2021.6.

M. Anders, P. Schweitzer, and M. Soos. 2023. “Algorithms Transcending the SAT-Symmetry Interface.” In: *26th International Conference on Theory and Applications of Satisfiability Testing, SAT 2023, July 4-8, 2023, Alghero, Italy* (LIPIcs). Vol. 271. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 1:1–1:21. doi:10.4230/LIPICS.SAT.2023.1.

M. Anders, P. Schweitzer, and J. Stieß. 2023. “Engineering a Preprocessor for Symmetry Detection.” In: *21st International Symposium on Experimental Algorithms, SEA 2023, July 24-26, 2023, Barcelona, Spain* (LIPIcs). Vol. 265. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 1:1–1:21. doi:10.4230/LIPICS.SEA.2023.1.

V. Arvind, J. Köbler, G. Rattan, and O. Verbitsky. 2017. “Graph Isomorphism, Color Refinement, and Compactness.” *Comput. Complex.*, 26, 3, 627–685. doi:10.1007/S00037-016-0147-6.

G. Audemard, S. Jabbour, and L. Sais. 2007. “Symmetry Breaking in Quantified Boolean Formulae.” In: *IJCAI 2007, Proceedings of the 20th International Joint Conference on Artificial Intelligence, Hyderabad, India, January 6-12, 2007*, 2262–2267. http://ijcai.org/Proceedings/07/Papers/364.pdf.

L. Babai. 2016. “Graph isomorphism in quasipolynomial time [extended abstract].” In: *Proceedings of the 48th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2016, Cambridge, MA, USA, June 18-21, 2016*. ACM, 684–697. doi:10.1145/2897518.2897542.

L. Babai, E. M. Luks, and Á. Seress. 1987. “Permutation Groups in NC.” In: *Proceedings of the 19th Annual ACM Symposium on Theory of Computing, 1987, New York, New York, USA*. ACM, 409–420. doi:10.1145/28395.28439.

A. Biere, T. Faller, K. Fazekas, M. Fleury, N. Froleyks, and F. Pollitt. 2024. “CaDiCaL, Gimsatul, IsaSAT and Kissat Entering the SAT Competition 2024.” In: *Proc. of SAT Competition 2024 – Solver, Benchmark and Proof Checker Descriptions* (Department of Computer Science Report Series B). Ed. by M. Heule, M. Iser, M. Järvisalo, and M. Suda. Vol. B-2024-1. University of Helsinki, 8–10.

B. Bogaerts, S. Gocht, C. McCreesh, and J. Nordström. 2023. “Certified Dominance and Symmetry Breaking for Combinatorial Optimisation.” *J. Artif. Intell. Res.*, 77, 1539–1589. doi:10.1613/JAIR.1.14296.

B. Bogaerts, J. Nordström, A. Oertel, and Ç. U. Yıldırımoğlu. 2023. “BreakID-kissat in SAT Competition 2023 (System Description).” English. In: *Proceedings of SAT Competition 2023: Solver, Benchmark and Proof Checker Descriptions* (Department of Computer Science Series of Publications B). Department of Computer Science, University of Helsinki, Finland.

M. Codish, G. Gange, A. Itzhakov, and P. J. Stuckey. 2016. “Breaking Symmetries in Graphs: The Nauty Way.” In: *Principles and Practice of Constraint Programming - 22nd International Conference, CP 2016, Toulouse, France, September 5-9, 2016, Proceedings* (Lecture Notes in Computer Science). Vol. 9892. Springer, 157–172. doi:10.1007/978-3-319-44953-1_11.

M. Codish, A. Miller, P. Prosser, and P. J. Stuckey. 2019. “Constraints for symmetry breaking in graph representation.” *Constraints An Int. J.*, 24, 1, 1–24. doi:10.1007/S10601-018-9294-5.

J. M. Crawford, M. L. Ginsberg, E. M. Luks, and A. Roy. 1996. “Symmetry-Breaking Predicates for Search Problems.” In: *Proceedings of the Fifth International Conference on Principles of Knowledge Representation and Reasoning (KR'96), Cambridge, Massachusetts, USA, November 5-8, 1996*. Morgan Kaufmann, 148–159.

P. T. Darga, M. H. Liffiton, K. A. Sakallah, and I. L. Markov. 2004. “Exploiting structure in symmetry detection for CNF.” In: *Proceedings of the 41th Design Automation Conference, DAC 2004, San Diego, CA, USA, June 7-11, 2004*. ACM, 530–534. doi:10.1145/996566.996712.

J. Devriendt and B. Bogaerts. 2016. "BreakID: Static Symmetry Breaking for ASP (System Description)." *CoRR*, abs/1608.08447. http://arxiv.org/abs/1608.08447 arXiv: 1608.08447.

J. Devriendt, B. Bogaerts, and M. Bruynooghe. 2017. "Symmetric Explanation Learning: Effective Dynamic Symmetry Handling for SAT." In: *Theory and Applications of Satisfiability Testing - SAT 2017 - 20th International Conference, Melbourne, VIC, Australia, August 28 - September 1, 2017, Proceedings* (Lecture Notes in Computer Science). Vol. 10491. Springer, 83–100. doi:10.1007/978-3-319-66263-3_6.

J. Devriendt, B. Bogaerts, M. Bruynooghe, and M. Denecker. 2016. "Improved Static Symmetry Breaking for SAT." In: *Theory and Applications of Satisfiability Testing - SAT 2016 - 19th International Conference, Bordeaux, France, July 5-8, 2016, Proceedings* (Lecture Notes in Computer Science). Vol. 9710. Springer, 104–122. doi:10.1007/978-3-319-40970-2_8.

J. Devriendt, B. Bogaerts, B. D. Cat, M. Denecker, and C. Mears. 2012. "Symmetry Propagation: Improved Dynamic Symmetry Breaking in SAT." In: *IEEE 24th International Conference on Tools with Artificial Intelligence, ICTAI 2012, Athens, Greece, November 7-9, 2012*. IEEE Computer Society, 49–56. doi:10.1109/ICTAI.2012.16.

P. Flener, A. M. Frisch, B. Hnich, Z. Kiziltan, I. Miguel, J. Pearson, and T. Walsh. 2002. "Breaking Row and Column Symmetries in Matrix Models." In: *Principles and Practice of Constraint Programming - CP 2002, 8th International Conference, CP 2002, Ithaca, NY, USA, September 9-13, 2002, Proceedings* (Lecture Notes in Computer Science). Vol. 2470. Springer, 462–476. doi:10.1007/3-540-46135-3_31.

P. Flener, A. M. Frisch, B. Hnich, Z. Kızıltan, I. Miguel, and T. Walsh. 2001. *Matrix Modelling*. Technical Report APES-36-2001. APES group (2001).

I. P. Gent, K. E. Petrie, and J. Puget. 2006. "Symmetry in Constraint Programming." In: *Handbook of Constraint Programming*. Foundations of Artificial Intelligence. Vol. 2. Elsevier, 329–376. doi:10.1016/S1574-6526(06)80014-3.

T. A. Junttila, M. Karppa, P. Kaski, and J. Kohonen. 2020. "An adaptive prefix-assignment technique for symmetry reduction." *J. Symb. Comput.*, 99, 21–49. doi:10.1016/J.JSC.2019.03.002.

T. A. Junttila and P. Kaski. 2011. "Conflict Propagation and Component Recursion for Canonical Labeling." In: *Theory and Practice of Algorithms in (Computer) Systems - First International ICST Conference, TAPAS 2011, Rome, Italy, April 18-20, 2011. Proceedings* (Lecture Notes in Computer Science). Vol. 6595. Springer, 151–162. doi:10.1007/978-3-642-19754-3_16.

G. Katsirelos, N. Narodytska, and T. Walsh. 2010. "On the Complexity and Completeness of Static Constraints for Breaking Row and Column Symmetry." In: *Principles and Practice of Constraint Programming - CP 2010 - 16th International Conference, CP 2010, St. Andrews, Scotland, UK, September 6-10, 2010. Proceedings* (Lecture Notes in Computer Science). Vol. 6308. Springer, 305–320. doi:10.1007/978-3-642-15396-9_26.

M. Kirchweger, M. Scheucher, and S. Szeider. 2022. "A SAT Attack on Rota's Basis Conjecture." In: *25th International Conference on Theory and Applications of Satisfiability Testing, SAT 2022, August 2-5, 2022, Haifa, Israel* (LIPIcs). Vol. 236. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 4:1–4:18. doi:10.4230/LIPICS.SAT.2022.4.

M. Kirchweger and S. Szeider. 2021. "SAT Modulo Symmetries for Graph Generation." In: *27th International Conference on Principles and Practice of Constraint Programming, CP* (LIPIcs). Vol. 210. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 34:1–34:16. doi:10.4230/LIPIcs.CP.2021.34.

D. E. Knuth. 2022. *The Art of Computer Programming, Volume 4B: Combinatorial Algorithms*. Pearson Education. ISBN: 9780137926817.

M. Lauria, J. Elffers, J. Nordström, and M. Vinyals. 2017. "CNFgen: A Generator of Crafted Benchmarks." In: *Theory and Applications of Satisfiability Testing - SAT 2017 - 20th International Conference, Melbourne, VIC, Australia, August 28 - September 1, 2017, Proceedings* (Lecture Notes in Computer Science). Vol. 10491. Springer, 464–473. doi:10.1007/978-3-319-66263-3_30.

E. M. Luks and A. Roy. 2004. "The Complexity of Symmetry-Breaking Formulas." *Ann. Math. Artif. Intell.*, 41, 1, 19–45. doi:10.1023/B:AMAI.0000018578.92398.10.

B. D. McKay and A. Piperno. 2014. "Practical graph isomorphism, II." *J. Symb. Comput.*, 60, 94–112. doi:10.1016/j.jsc.2013.09.003.

H. Metin, S. Baarir, M. Colange, and F. Kordon. 2018. "CDCLSym: Introducing Effective Symmetry Breaking in SAT Solving." In: *Tools and Algorithms for the Construction and Analysis of Systems - 24th International Conference, TACAS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings, Part I* (Lecture Notes in Computer Science). Vol. 10805. Springer, 99–114. doi:10.1007/978-3-319-89960-2_6.

M. E. Pfetsch and T. Rehn. 2019. "A computational comparison of symmetry handling methods for mixed integer programs." *Math. Program. Comput.*, 11, 1, 37–93. doi:10.1007/s12532-018-0140-y.

A. Sabharwal. 2009. "SymChaff: exploiting symmetry in a structure-aware satisfiability solver." *Constraints An Int. J.*, 14, 4, 478–505. doi:10.1007/S10601-008-9060-1.

K. A. Sakallah. 2021. "Symmetry and Satisfiability." In: *Handbook of Satisfiability - Second Edition*. Frontiers in Artificial Intelligence and Applications. Vol. 336. IOS Press, 509–570. doi:10.3233/FAIA200996.

Á. Seress. 2003. *Permutation Group Algorithms*. Cambridge Tracts in Mathematics. Cambridge University Press. doi:10.1017/CBO9780511546549.

G. Tinhofer. 1991. "A note on compact graphs." *Discret. Appl. Math.*, 30, 2-3, 253–264. doi:10.1016/0166-218X(91)90049-3.

Received 31 March 2025; accepted 31 October 2025