Journal of Symbolic Computation ••• (••••) •••–•••

Contents lists available at ScienceDirect

Journal of Symbolic Computation

www.elsevier.com/locate/jsc

An adaptive prefix-assignment technique for symmetry reduction

Tommi Junttila, Matti Karppa, Petteri Kaski, Jukka Kohonen

Aalto University, Department of Computer Science, Finland

ARTICLE INFO

Article history:
Received 11 June 2018
Accepted 28 January 2019
Available online xxxx

Keywords:
Symmetry breaking
Symmetry reduction
Isomorph rejection
Constraint programming
SAT
Canonical extension

ABSTRACT

This paper presents a technique for symmetry reduction that adaptively assigns a prefix of variables in a system of constraints so that the generated prefix-assignments are pairwise nonisomorphic under the action of the symmetry group of the system. The technique is based on McKay's canonical extension framework (McKay, 1998). Among key features of the technique are (i) adaptability—the prefix sequence can be user-prescribed and truncated for compatibility with the group of symmetries; (ii) parallelizability—prefix-assignments can be processed in parallel independently of each other; (iii) versatility—the method is applicable whenever the group of symmetries can be concisely represented as the automorphism group of a vertex-colored graph; and (iv) implementability—the method can be implemented relying on a canonical labeling map for vertex-colored graphs as the only nontrivial subroutine. To demonstrate the practical applicability of our technique, we have prepared an experimental open-source implementation of the technique and carry out a set of experiments that demonstrate ability to reduce symmetry on hard instances. Furthermore, we demonstrate that the implementation effectively parallelizes to compute clusters with multiple nodes via a message-passing interface.

© 2019 Published by Elsevier Ltd.

E-mail addresses: tommi.junttila@aalto.fi (T. Junttila), matti.karppa@aalto.fi (M. Karppa), petteri.kaski@aalto.fi (P. Kaski), kohonen@cs.helsinki.fi (J. Kohonen).

https://doi.org/10.1016/j.jsc.2019.03.002
0747-7171/© 2019 Published by Elsevier Ltd.

1. Introduction

1.1. Symmetry reduction

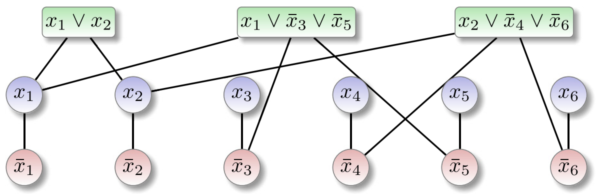
Systems of constraints often have substantial symmetry. For example, consider the following system of Boolean clauses:

$$(x_1 \vee x_2) \wedge (x_1 \vee \bar{x}_3 \vee \bar{x}_5) \wedge (x_2 \vee \bar{x}_4 \vee \bar{x}_6)\,. \tag{1}$$

The associative and commutative symmetries of disjunction and conjunction induce symmetries between the variables of (1), a fact that can be captured by stating that the group Γ generated by the two permutations $(x_1\ x_2)(x_3\ x_4)(x_5\ x_6)$ and $(x_4\ x_6)$ consists of all permutations of the variables that map (1) to itself. That is, Γ is the *automorphism group* of the system (1), cf. Section 2.

Known symmetry in a constraint system is a great asset from the perspective of solving the system, in particular since symmetry enables one to disregard partial assignments that are *isomorphic* to each other under the action of Γ on the space of partial assignments. Techniques for such *isomorph rejection*[1] (Swift, 1960) (alternatively, *symmetry reduction* or *symmetry breaking*) are essentially mandatory if one desires an exhaustive traversal of the (pairwise nonisomorphic) assignments of a highly symmetric system of constraints, or if the system is otherwise difficult to solve, for example, with many "dead-end" partial assignments compared with the actual number of solutions.

A prerequisite to symmetry reduction is that the symmetries are known. In many cases it is possible to automatically discover and compute these symmetries to enable practical and automatic symmetry reduction. In this context the dominant computational approach for combinatorial systems of constraints is to represent Γ via the automorphism group of a vertex-colored graph that captures the symmetries in the system. Carefully engineered tools for working with symmetries of vertex-colored graphs (Darga et al., 2004; Junttila and Kaski, 2007; McKay, 1981; McKay and Piperno, 2014) and permutation group algorithms (Butler, 1991; Seress, 2003) then enable one to perform symmetry reduction. For example, for purposes of symmetry computations we may represent (1) as the following vertex-colored graph (for interpretation of the colors in the figure, the reader is referred to the web version of this article):



(2)

In particular, the graph representation (2) enables us to discover and reduce symmetry to avoid redundant work when solving the underlying system (1).

1.2. Our contribution

The objective of this paper is to present a novel technique for symmetry reduction on systems of constraints. The technique is based on adaptively assigning values to a prefix of the variables so that the obtained prefix-assignments are pairwise nonisomorphic under the action of Γ. The technique can be seen as an instantiation of McKay's (1998) influential canonical extension framework for isomorph-free exhaustive generation.

To give a brief outline of the technique, suppose we are working with a system of constraints over a finite set U of variables that take values in a finite set R. Suppose furthermore that $\Gamma \leq \mathrm{Sym}(U)$

[1] A term introduced by Swift (1960); cf. Hall and Knuth (1965) for a survey on early work on exhaustive computer search and combinatorial analysis.

is the automorphism group of the system. Select k distinct variables $u_1, u_2, \ldots, u_k$ in U. These k variables form the *prefix sequence* considered by the method. The technique works by assigning values in R to the variables of the prefix, in prefix-sequence order, with u_1 assigned first, then u_2, then u_3, and so forth, so that at each step the partial assignments so obtained are pairwise nonisomorphic under the action of Γ. For example, in (1) the partial assignments $x_1 \mapsto 0$, $x_2 \mapsto 1$ and $x_1 \mapsto 1$, $x_2 \mapsto 0$ are isomorphic since $(x_1\ x_2)(x_3\ x_4)(x_5\ x_6) \in \Gamma$ maps one assignment onto the other; in total there are three nonisomorphic assignments to the prefix x_1, x_2 in (1), namely (i) $x_1 \mapsto 0$, $x_2 \mapsto 0$, (ii) $x_1 \mapsto 0$, $x_2 \mapsto 1$, and (iii) $x_1 \mapsto 1$, $x_2 \mapsto 1$. Each partial assignment that represents an isomorphism class can then be used to reduce redundant work when solving the underlying system by standard techniques—in the nonincremental case, the system is augmented with a symmetry-breaking predicate requiring that one of the nonisomorphic partial assignments holds, while in the incremental setting (Heule et al., 2011; Wieringa, 2011) the partial assignments can be solved independently or even in parallel.

Our contribution in this paper lies in how the isomorph rejection is implemented at the level of isomorphism classes of partial assignments by careful reduction to McKay's (1998) isomorph-free exhaustive generation framework. The key technical contribution is that we observe how to generate the partial assignments in a normalized form that enables both *adaptability* (that is, the prefix $u_1, u_2, \ldots, u_k$ can be arbitrarily selected to match the structure of Γ) and precomputation of the extending variable-value orbits along a prefix.

Among further key features of the technique are:

1. *Implementability.* The technique can be implemented by relying on a canonical labeling map for vertex-colored graphs (cf. Junttila and Kaski, 2007; McKay, 1981; McKay and Piperno, 2014) as the only nontrivial subroutine that is invoked once for each partial assignment considered.
2. *Versatility.* The method is applicable whenever the group of symmetries can be concisely represented as a vertex-colored graph; cf. (1) and (2). This is useful in particular when the underlying system has symmetries that are not easily discoverable from the final constraint encoding, for example, due to the fact that the constraints have been compiled or optimized[2] from a higher-level representation in a symmetry-obfuscating manner. A graphical representation can represent such symmetry directly and independently of the compiled/optimized form of the system.
3. *Parallelizability.* As a corollary of implementing McKay's (1998) framework, the technique does not need to store representatives of isomorphism classes in memory to perform isomorph rejection, which enables easy parallelization since the partial assignments can be processed independently of each other.

The required mathematical preliminaries on symmetry and McKay's framework are reviewed in Sections 2 and 3, respectively. The main technical contribution of this paper is developed in Sections 4 and 5. We start in Section 4 by developing the prefix-assignment technique for variable symmetry, and extend this in Section 5 to account for symmetry both in the variables and in the values of the variables. Our development in Sections 4 and 5 relies on an abstract group Γ, with the understanding that a concrete implementation can be designed e.g. in terms of a vertex-colored graph representation, as will be explored in Section 6.

To demonstrate the practical applicability of our technique, we have prepared an open-source parallel implementation (Karppa, 2018). The implementation is structured as a preprocessor that works with an explicitly given graph representation and utilizes the *nauty* (McKay, 1981; McKay and Piperno, 2014) canonical labeling software for vertex-colored graphs as a subroutine to prepare an exhaustive collection of nonisomorphic prefix assignments relative to a user-supplied prefix of variables, and the Message Passing Interface (MPI) for parallelization. Further details of the implementation are presented in Section 7. In Section 8, we report on a set of experiments that (i) demonstrate the ability

[2] For a beautiful illustration, we refer to Knuth's (2011, §7.1.2, Fig. 10) example of optimum Boolean chains for 5-variable symmetric Boolean functions—from each optimum chain it is far from obvious that the chain represents a symmetric Boolean function. (See also Example 17.)

to reduce symmetry on hard instances, (ii) study the serendipity of an auxiliary graph for encoding the symmetries in an instance, (iii) give examples of instances with hard combinatorial symmetry where our technique performs favorably in comparison with earlier techniques, and (iv) study the parallel speedup obtainable when we distribute the symmetry reduction task to a compute cluster with multiple compute nodes.

1.3. Earlier work

A classical way to exploit symmetry in a system of constraints is to augment the system with so-called symmetry-breaking predicates (SBP) that eliminate either some or all symmetric solutions, see e.g. (Aloul et al., 2003; Crawford et al., 1996; Gent et al., 2006; Grayland et al., 2009; Sakallah, 2009; Jefferson and Petrie, 2011). Such constraints are typically lexicographic leader (lex-leader) constraints that are derived from a generating set for the group of symmetries Γ. Among recent work in this area, Devriendt et al. (2016) extend the approach by presenting a more compact way for expressing SBPs and a method for detecting "row interchangeabilities". Itzhakov and Codish (2016) present a method for finding a set of symmetries whose corresponding lex-leader constraints are enough to completely break symmetries in search problems on small (10-vertex) graphs; this approach is extended by Codish et al. (2016) by adding pruning predicates that simulate the first iterations of the equitable partition refinement algorithm of *nauty* (McKay, 1981; McKay and Piperno, 2014). Heule (2016) shows that small complete symmetry-breaking predicates can be computed by considering arbitrary Boolean formulas instead of lex-leader formulas.

Our present technique can be seen as a method for producing symmetry-breaking predicates by augmenting the system of constraints with the disjunction of the nonisomorphic partial assignments. The main difference to the related work above is that our technique does not produce the symmetry-breaking predicate from a set of generators for Γ but rather the predicate is produced recursively, and with the possibility for parallelization, by classifying orbit representatives up to isomorphism using McKay's (1998) framework. As such our technique breaks all symmetry with respect to the prescribed prefix, but comes at the cost of additional invocations of graph-automorphism and canonical-labeling tools. This overhead and increased symmetry reduction in particular means that our technique is best suited for constraint systems with hard combinatorial symmetry that is not easily capturable from a set of generators, such as symmetry in combinatorial classification problems (Kaski and Östergård, 2006). In addition to McKay's (1998) canonical extension framework, other standard frameworks for isomorph-free exhaustive generation in this context include *orderly algorithms* due to Faradžev (1978) and Read (1978), as well as the homomorphism principle for group actions due to Kerber and Laue (1998).

It is also possible to break symmetry within a constraint solver during the search by dynamically adding constraints that rule out symmetric parts of the search space (cf. Chu et al., 2014; Gent et al., 2006). If we use the nonisomorphic partial assignments produced by our technique as assumption sequences (cubes) in the incremental cube-and-conquer approach (Heule et al., 2011; Wieringa, 2011), our technique can be seen as a restricted way of breaking the symmetries in the beginning of the search, with the benefit—as with cube-and-conquer—that the portions of the search space induced by the partial assignments can be solved in parallel, either with complete independence or with appropriate sharing of information (such as conflict clauses) between the parallel nodes executing the search. For further work in dynamic symmetry breaking, cf. Devriendt et al. (2017, 2012), Benhamou and Sais (1994), Benhamou et al. (2010a,b), Sabharwal (2009), Schaafsma et al. (2009).

For work on isomorphism and canonical labeling techniques, cf. Babai (2016), Grohe et al. (2018), Lokshtanov et al. (2017), McKay and Piperno (2014).

2. Preliminaries on group actions and symmetry

This section reviews relevant mathematical preliminaries and notational conventions for groups, group actions, symmetry, and isomorphism for our subsequent development. (Cf. Butler, 1991; Dixon and Mortimer, 1996; Humphreys, 1996; Kaski and Östergård, 2006; Kerber, 1999; Seress, 2003 for further reference.)

2.1. Groups and group actions

Let Γ be a finite group and let Ω be a finite set (the domain) on which Γ acts. For two groups Λ and Γ, let us write $\Lambda \le \Gamma$ to indicate that Λ is a subgroup of Γ. We use exponential notation for group actions, and accordingly our groups act from the right. That is, for an object $X \in \Omega$ and $\gamma \in \Gamma$, let us write X^γ for the object in Ω obtained by acting on X with γ. Accordingly, we have $X^{(\beta\gamma)} = (X^\beta)^\gamma$ for all $\beta, \gamma \in \Gamma$ and $X \in \Omega$. For a finite set V, let us write $\mathrm{Sym}(V)$ for the group of all permutations of V with composition of mappings as the group operation.

2.2. Orbit and stabilizer, the automorphism group

For an object $X \in \Omega$ let us write $X^\Gamma = \{X^\gamma : \gamma \in \Gamma\}$ for the *orbit* of X under the action of Γ and $\Gamma_X = \{\gamma \in \Gamma : X^\gamma = X\} \le \Gamma$ for the *stabilizer* subgroup of X in Γ. Equivalently we say that Γ_X is the *automorphism group* of X and write $\mathrm{Aut}(X) = \Gamma_X$ whenever Γ is clear from the context; if we want to stress the acting group we write $\mathrm{Aut}_\Gamma(X)$.

We write $\Omega/\Gamma = \{X^\Gamma : X \in \Omega\}$ for the set of all orbits of Γ on Ω. For $\Lambda \le \Gamma$ and $\gamma \in \Gamma$, let us write $\Lambda^\gamma = \gamma^{-1}\Lambda\gamma = \{\gamma^{-1}\lambda\gamma : \lambda \in \Lambda\} \le \Gamma$ for the *γ-conjugate* of Λ. For all $X \in \Omega$ and $\gamma \in \Gamma$ we have $\mathrm{Aut}(X^\gamma) = \mathrm{Aut}(X)^\gamma$. That is, the automorphism groups of objects in an orbit are conjugates of each other.

2.3. Isomorphism

We say that two objects are *isomorphic* if they are in the same orbit of Γ in Ω. In particular, $X, Y \in \Omega$ are isomorphic if and only if there exists an *isomorphism* $\gamma \in \Gamma$ from X to Y that satisfies $Y = X^\gamma$. An isomorphism from an object to itself is an *automorphism*. Let us write $\mathrm{Iso}(X, Y)$ for the set of all isomorphisms from X to Y. When X and Y are isomorphic, we have $\mathrm{Iso}(X, Y) = \mathrm{Aut}(X)\gamma = \gamma\mathrm{Aut}(Y)$ where $\gamma \in \mathrm{Iso}(X, Y)$ is arbitrary. Let us write $X \cong Y$ to indicate that X and Y are isomorphic. If we want to stress the group Γ under whose action isomorphism holds, we write $X \cong_\Gamma Y$.

2.4. Elementwise action on tuples and sets

Suppose that Γ acts on two sets, Ω and Σ. We extend the action to the Cartesian product $\Omega \times \Sigma$ elementwise by defining $(X, S)^\gamma = (X^\gamma, S^\gamma)$ for all $(X, S) \in \Omega \times \Sigma$ and $\gamma \in \Gamma$. Isomorphism extends accordingly; for example, we say that (X, S) and (Y, T) are isomorphic and write $(X, S) \cong (Y, T)$ if there exists a $\gamma \in \Gamma$ with $Y = X^\gamma$ and $T = S^\gamma$. Suppose that Γ acts on a set U. We extend the action of Γ on U to an elementwise action of Γ on subsets $W \subseteq U$ by setting $W^\gamma = \{w^\gamma : w \in W\}$ for all $\gamma \in \Gamma$ and $W \subseteq U$. In what follows we will tacitly work with these elementwise actions on tuples and sets unless explicitly otherwise indicated.

2.5. Canonical labeling and canonical form

A function $\kappa : \Omega \to \Gamma$ is a *canonical labeling map* for the action of Γ on Ω if

(K) for all $X, Y \in \Omega$ it holds that $X \cong Y$ implies $X^{\kappa(X)} = Y^{\kappa(Y)}$ (canonical labeling).

For $X \in \Omega$ we say that $X^{\kappa(X)}$ is the *canonical form* of X in Ω. From (K) it follows that isomorphic objects have identical canonical forms, and the canonical labeling map gives an isomorphism that takes an object to its canonical form.

We assume that the act of computing $\kappa(X)$ for a given X produces as a side-effect a set of generators for the automorphism group $\mathrm{Aut}(X)$.

3. McKay's canonical extension method

This section reviews McKay's (1998) canonical extension method for isomorph-free exhaustive generation. Mathematically it will be convenient to present the method so that the isomorphism classes are captured as orbits of a group action of a group Γ, and extension takes place in one step from "seeds" to "objects" being generated, with the understanding that the method can be applied inductively in multiple steps so that the "objects" of the current step become the "seeds" for the next step. For completeness and ease of exposition, we also give a correctness proof for the method. We stress that all material in this section is well known. (Cf. Kaski and Östergård, 2006.)

3.1. *Objects and seeds*

Let Ω be a finite set of *objects* and let Σ be a finite set of *seeds*. Let Γ be a finite group that acts on Ω and Σ. Let κ be a canonical labeling map for the action of Γ on Ω.

3.2. *Extending seeds to objects*

Let us connect the objects and the seeds by means of a relation $e \subseteq \Omega \times \Sigma$ that indicates which objects can be built from which seeds by extension. For $X \in \Omega$ and $S \in \Sigma$ we say that X *extends* S and write XeS if $(X, S) \in e$. We assume the relation e satisfies

(E1) e is a union of orbits of Γ, that is, $e^{\Gamma} = e$ (invariance), and
(E2) for every object $X \in \Omega$ there exists a seed $S \in \Sigma$ such that XeS (completeness).

For a seed $S \in \Sigma$, let us write $e(S) = \{X \in \Omega : XeS\}$ for the set of all objects that extend S.

3.3. *Canonical extension*

We associate with each object a particular isomorphism-invariant extension by which we want to extend the object from a seed. A function $M : \Omega \to \Sigma$ is a *canonical extension map* if

(M1) for all $X \in \Omega$ it holds that $(X, M(X)) \in e$ (extension), and
(M2) for all $X, Y \in \Omega$ we have that $X \cong Y$ implies $(X, M(X)) \cong (Y, M(Y))$ (canonicity).

That is, (M1) requires that X is in fact an extension of $M(X)$ and (M2) requires that isomorphic objects have isomorphic canonical extensions. In particular, $X \mapsto (X, M(X))$ is a well-defined map from Ω/Γ to e/Γ.

3.4. *Generating objects from seeds*

Let us study the following procedure, which is invoked for exactly one representative $S \in \Sigma$ from each orbit in Σ/Γ:

(P) Let $S \in \Sigma$ be given as input. Iterate over all $X \in e(S)$. Perform zero or more isomorph rejection tests on X and S. If the tests indicate we should accept X, visit X.

We will equip procedure (P) with *isomorph rejection tests* that will ensure that the procedure visits exactly one object from each isomorphism class of objects. Let us first consider the case when there are no isomorph rejection tests.

Lemma 1. *The procedure (P) visits every isomorphism class of objects in Ω at least once.*

Proof. To see that every isomorphism class is visited, let $Y \in \Omega$ be arbitrary. By (E2), there exists a $T \in \Sigma$ with YeT. By our assumption on how procedure (P) is invoked, T is isomorphic to a unique S such that procedure (P) is invoked with input S. Let $\gamma \in \Gamma$ be an associated isomorphism with $S^\gamma = T$. By (E1) and YeT, we have XeS for $X = Y^{\gamma^{-1}}$. By the structure of procedure (P) we observe that X is visited and $X \cong Y$. Since Y was arbitrary, all isomorphism classes are visited at least once. □

Let us next modify procedure (P) so that any two visits to the same isomorphism class of objects originate from the same procedure invocation. Let $M : \Omega \to \Sigma$ be a canonical extension map. Whenever we construct X by extending S in procedure (P), let us visit X if and only if

$$\text{(T1)} \quad (X, S) \cong (X, M(X)).$$

Lemma 2. *The procedure (P) equipped with the test (T1) visits every isomorphism class of objects in Ω at least once. Furthermore, any two visits to the same isomorphism class must (i) originate by extension from the same procedure invocation on input S, and (ii) belong to the same* Aut(S)*-orbit of this seed S.*

Proof. Suppose that X is visited by extending S and Y is visited by extending T, with $X \cong Y$. By (T1) we must thus have $(X, S) \cong (X, M(X))$ and $(Y, T) \cong (Y, M(Y))$. Furthermore, from (M2) we have $(X, M(X)) \cong (Y, M(Y))$. Thus, $(X, S) \cong (Y, T)$ and hence $S \cong T$. Since $S \cong T$, we must in fact have $S = T$ by our assumption on how procedure (P) is invoked. Since X and Y were arbitrary, any two visits to the same isomorphism class must originate by extension from the same seed. Furthermore, we have $(X, S) \cong (Y, S)$ and thus $X \cong_{\mathrm{Aut}(S)} Y$. Let us next observe that every isomorphism class of objects is visited at least once. Indeed, let $Y \in \Omega$ be arbitrary. By (M1), we have $YeM(Y)$. In particular, there is a unique $S \in \Sigma$ with $S \cong M(Y)$ such that procedure (P) is invoked with input S. Let $\gamma \in \Gamma$ be an associated isomorphism with $S^\gamma = M(Y)$. By (E1), we have XeS for $X = Y^{\gamma^{-1}}$. Furthermore, $X \cong Y$ implies by (M2) that $(X, M(X)) \cong (Y, M(Y)) = (X^\gamma, S^\gamma) \cong (X, S)$, so (T1) holds and X is visited. Since $X \cong Y$ and Y was arbitrary, every isomorphism class is visited at least once. □

Let us next observe that the outcome of test (T1) is invariant on each Aut(S)-orbit of extensions of S.

Lemma 3. *For all $\alpha \in$* Aut(S) *we have that (T1) holds for (X, S) if and only if (T1) holds for (X^α, S).*

Proof. From $X \cong X^\alpha$ and (M2) we have $(X, M(X)) \cong (X^\alpha, M(X^\alpha))$. Thus, $(X, S) \cong (X, M(X))$ if and only if $(X^\alpha, S) = (X^\alpha, S^\alpha) \cong (X, S) \cong (X, M(X)) \cong (X^\alpha, M(X^\alpha))$. □

Lemma 3 in particular implies that we obtain complete isomorph rejection by combining the test (T1) with a further test that ensures complete isomorph rejection on Aut(S)-orbits. Towards this end, let us associate an arbitrary order relation on every Aut(S)-orbit on $e(S)$. Let us perform the following further test:

$$\text{(T2)} \quad X = \min X^{\mathrm{Aut}(S)}.$$

The following lemma is immediate from Lemma 2 and Lemma 3.

Lemma 4. *The procedure (P) equipped with the tests (T1) and (T2) visits every isomorphism class of objects in Ω exactly once.*

3.5. A template for canonical extension maps

We conclude this section by describing a template of how to use an arbitrary canonical labeling map $\kappa : \Omega \to \Gamma$ to construct a canonical extension map $M : \Omega \to \Sigma$.

For $X \in \Omega$ construct the canonical form $Z = X^{\kappa(X)}$. Using the canonical form Z only, identify a seed T with ZeT. In particular, such a seed must exist by (E2). (Typically this identification can be carried out by studying Z and finding an appropriate substructure in Z that qualifies as T. For example, T may be the minimum seed in Σ that satisfies ZeT. Cf. Lemma 10.) Once T has been identified, set $M(X) = T^{\kappa(X)^{-1}}$.

Lemma 5. *The map $X \mapsto M(X)$ above is a canonical extension map.*

Proof. By (E1) we have $XeM(X)$ because $Z^{\kappa(X)^{-1}} = X$, $T^{\kappa(X)^{-1}} = M(X)$, and ZeT. Thus, (M1) holds for M. To verify (M2), let $X, Y \in \Omega$ with $X \cong Y$ be arbitrary. Since $X \cong Y$, by (K) we have $X^{\kappa(X)} = Z = Y^{\kappa(Y)}$. It follows that $M(X) = T^{\kappa(X)^{-1}}$ and $M(Y) = T^{\kappa(Y)^{-1}}$, implying that $\gamma = \kappa(X)\kappa(Y)^{-1}$ is an isomorphism witnessing $(X, M(X)) \cong (Y, M(Y))$. Thus, (M2) holds for M. □

4. Generation of partial assignments via a prefix sequence

This section describes an instantiation of McKay's method that generates partial assignments of values to a set of variables U one variable at a time following a *prefix sequence* at the level of isomorphism classes given by the action of a group Γ on U. Let R be a finite set where the variables in U take values.

4.1. Partial assignments, isomorphism, restriction

For a subset $W \subseteq U$ of variables, let us say that a *partial assignment* of values to W is a mapping $X : W \to R$. Isomorphism for partial assignments is induced by the following group action. Let $\gamma \in \Gamma$ act on $X : W \to R$ by setting $X^\gamma : W^\gamma \to R$ where X^γ is defined for all $u \in W^\gamma$ by

$$X^\gamma(u) = X(u^{\gamma^{-1}}) . \tag{3}$$

Lemma 6. *The action (3) is well-defined.*

Proof. We observe that for the identity $\epsilon \in \Gamma$ of Γ we have $X^\epsilon = X$. Furthermore, for all $\gamma, \beta \in \Gamma$ and $u \in W^{\gamma\beta} = (W^\gamma)^\beta$ we have

$$X^{\gamma\beta}(u) = X\big(u^{(\gamma\beta)^{-1}}\big) = X\big((u^{\beta^{-1}})^{\gamma^{-1}}\big) = X^\gamma\big(u^{\beta^{-1}}\big) = (X^\gamma)^\beta(u) . \quad \square$$

For an assignment $X : W \to R$, let us write $\underline{X} = W$ for the underlying set of variables assigned by X. Observe that the underline map is a homomorphism of group actions in the sense that

$$\underline{X^\gamma} = \underline{X}^\gamma \tag{4}$$

holds for all $\gamma \in \Gamma$ and $X : W \to R$. For $Q \subseteq \underline{X}$, let us write $X|_Q$ for the restriction of X to Q.

4.2. The prefix sequence and generation of normalized assignments

We are now ready to describe the generation procedure. We recommend referring to Example 7 below as a running example that illustrates the following definitions and the generation procedure for the system of clauses (1).

Let us begin by prescribing the prefix sequence. Let $u_1, u_2, \ldots, u_k$ be k distinct elements of U and let $U_j = \{u_1, u_2, \ldots, u_j\}$ for $j = 0, 1, \ldots, k$. In particular we observe that

$$U_0 \subseteq U_1 \subseteq \cdots \subseteq U_k$$

with $U_j \setminus U_{j-1} = \{u_j\}$ for all $j = 1, 2, \ldots, k$.

For $j = 0, 1, \ldots, k$ let Ω_j consist of all partial assignments $X : W \to R$ with $W \cong U_j$. Or what is the same, using the underline notation, Ω_j consists of all partial assignments X with $\underline{X} \cong U_j$.

We rely on canonical extension to construct exactly one object from each orbit of Γ on Ω_j, using as seeds exactly one object from each orbit of Γ on Ω_{j-1}, for each $j = 1, 2, \ldots, k$. We assume the availability of canonical labeling maps $\kappa : \Omega_j \to \Gamma$ for each $j = 1, 2, \ldots, k$.

Our construction procedure will work with objects that are in a normal form to enable precomputation for efficient execution of the subsequent tests for isomorph rejection. Towards this end, let us say that $X \in \Omega_j$ is *normalized* if $\underline{X} = U_j$. It is immediate from our definition of Ω_j and (3) that each orbit in Ω_j/Γ contains at least one normalized object.

Let us begin with a high-level description of the construction procedure, to be followed by the details of the isomorph rejection tests and a proof of correctness. Fix $j = 1, 2, \ldots, k$ and study the following procedure, which we assume is invoked for exactly one normalized representative $S \in \Omega_{j-1}$ from each orbit in Ω_{j-1}/Γ:

(P') Let a normalized $S \in \Omega_{j-1}$ be given as input. For each $p \in u_j^{\mathrm{Aut}(U_{j-1})}$ and each $r \in R$, construct the assignment

$$X : U_{j-1} \cup \{p\} \to R$$

defined by $X(p) = r$ and $X(u) = S(u)$ for all $u \in U_{j-1}$. Perform the isomorph rejection tests (T1') and (T2') on X and S. If both tests accept, visit $X^{\nu(p)}$ where $\nu(p) \in \mathrm{Aut}(U_{j-1})$ normalizes X.

From an implementation point of view, it is convenient to precompute the orbit $u_j^{\mathrm{Aut}(U_{j-1})}$ together with group elements $\nu(p) \in \mathrm{Aut}(U_{j-1})$ for each $p \in u_j^{\mathrm{Aut}(U_{j-1})}$ that satisfy $p^{\nu(p)} = u_j$. Indeed, a constructed X with $\underline{X} = U_{j-1} \cup \{p\}$ can now be normalized by acting with $\nu(p)$ on X to obtain a normalized $X^{\nu(p)}$ isomorphic to X.

4.3. The isomorph rejection tests

Let us now complete the description of procedure (P') by describing the two isomorph rejection tests (T1') and (T2'). This subsection only describes the tests with an implementation in mind, the correctness analysis is postponed to the following subsection.

Let us assume that the elements of U have been arbitrarily ordered and that $\kappa : \Omega_j \to \Gamma$ is a canonical labeling map. Suppose that X has been constructed by extending a normalized S with $\underline{X} = \underline{S} \cup \{p\} = U_{j-1} \cup \{p\}$. The first test is:

(T1') Subject to the ordering of U, select the minimum $q \in U$ such that $q^{\kappa(X)^{-1}\nu(p)} \in u_j^{\mathrm{Aut}(U_j)}$. Accept if and only if $p \cong_{\mathrm{Aut}(X)} q^{\kappa(X)^{-1}}$.

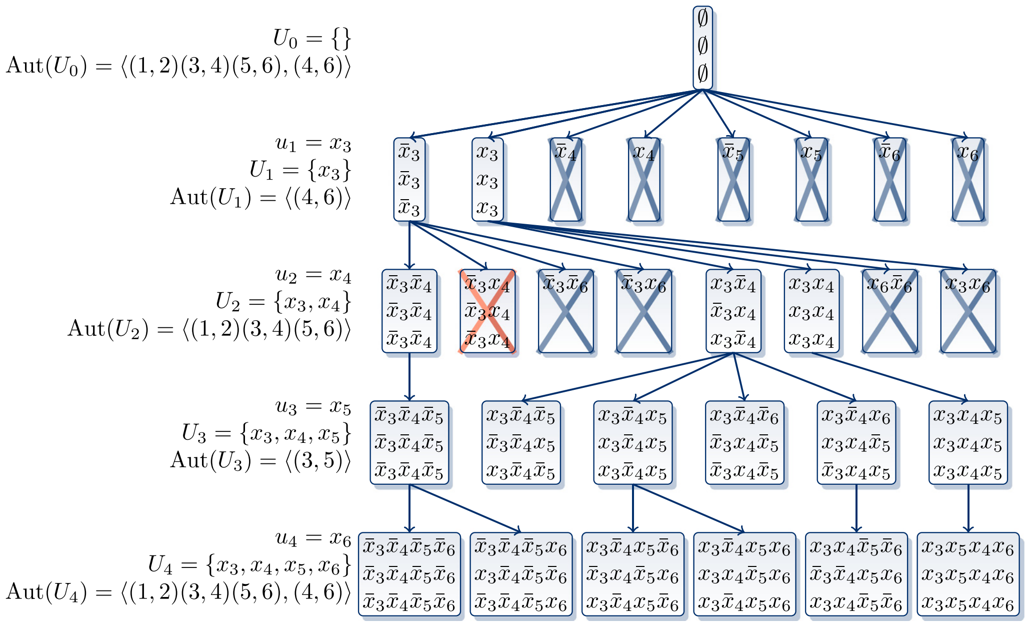
From an implementation perspective we observe that we can precompute the orbit $u_j^{\mathrm{Aut}(U_j)}$. Furthermore, the only computationally nontrivial part of the test is the computation of $\kappa(X)$ since we assume that we obtain generators for $\mathrm{Aut}(X)$ as a side-effect of this computation. Indeed, with generators for $\mathrm{Aut}(X)$ available, it is easy to compute the orbits $U/\mathrm{Aut}(X)$ and hence to test whether $p \cong_{\mathrm{Aut}(X)} q^{\kappa(X)^{-1}}$.

Let us now describe the second test:

(T2') Accept if and only if $p = \min p^{\mathrm{Aut}(S)}$ subject to the ordering of U.

From an implementation perspective we observe that since S is normalized we have $\mathrm{Aut}(S) \le \mathrm{Aut}(\underline{S}) = \mathrm{Aut}(U_{j-1})$ and thus the orbit $u_j^{\mathrm{Aut}(U_{j-1})}$ considered by procedure (P') partitions into one or more $\mathrm{Aut}(S)$-orbits. Furthermore, generators for $\mathrm{Aut}(S)$ are readily available (due to S itself getting accepted in the test (T1') at an earlier level of recursion), and thus the orbits $u_j^{\mathrm{Aut}(U_{j-1})}/\mathrm{Aut}(S)$ and their minimum elements are cheap to compute. Thus, a fast implementation of procedure (P') will in most cases execute the test (T2') before the more expensive test (T1').

Example 7. We display below a possible search tree for the system of clauses (1) and the prefix sequence x_3, x_4, x_5, x_6. Each node in the search tree displays the prefix assignment X (top), its canonical version $X^{\kappa(X)}$ (middle) and its normalized version $X^{\nu(p)}$ (bottom). The variables have the Boolean domain $\{\mathbf{f}, \mathbf{t}\}$ and the assignments are given in the literal form; for example, we write $\bar{x}_3 x_4$ for the assignment $\{x_3 \mapsto \mathbf{f}, x_4 \mapsto \mathbf{t}\}$. (For interpretation of the colors in the figure(s), the reader is referred to the web version of this article.)


$U_0 = \{\}$
$\mathrm{Aut}(U_0) = \langle(1,2)(3,4)(5,6),(4,6)\rangle$
$u_1 = x_3$
$U_1 = \{x_3\}$
$\mathrm{Aut}(U_1) = \langle(4,6)\rangle$
$u_2 = x_4$
$U_2 = \{x_3, x_4\}$
$\mathrm{Aut}(U_2) = \langle(1,2)(3,4)(5,6)\rangle$
$u_3 = x_5$
$U_3 = \{x_3, x_4, x_5\}$
$\mathrm{Aut}(U_3) = \langle(3,5)\rangle$
$u_4 = x_6$
$U_4 = \{x_3, x_4, x_5, x_6\}$
$\mathrm{Aut}(U_4) = \langle(1,2)(3,4)(5,6),(4,6)\rangle$
$\emptyset$ $\emptyset$ $\emptyset$
$\bar{x}_3$ $\bar{x}_3$ $\bar{x}_3$
x_3 x_3 x_3
$\bar{x}_4$
x_4
$\bar{x}_5$
x_5
$\bar{x}_6$
x_6
$\bar{x}_3\bar{x}_4$ $\bar{x}_3\bar{x}_4$ $\bar{x}_3\bar{x}_4$
$\bar{x}_3x_4$ $\bar{x}_3x_4$ $\bar{x}_3x_4$
$\bar{x}_3\bar{x}_6$
$\bar{x}_3x_6$
$x_3\bar{x}_4$ $\bar{x}_3x_4$ $x_3\bar{x}_4$
x_3x_4 x_3x_4 x_3x_4
$x_6\bar{x}_6$
x_3x_6
$\bar{x}_3\bar{x}_4\bar{x}_5$ $\bar{x}_3\bar{x}_4\bar{x}_5$ $\bar{x}_3\bar{x}_4\bar{x}_5$
$x_3\bar{x}_4\bar{x}_5$ $\bar{x}_3\bar{x}_4x_5$ $x_3\bar{x}_4\bar{x}_5$
$x_3\bar{x}_4x_5$ $x_3\bar{x}_4x_5$ $x_3\bar{x}_4x_5$
$x_3\bar{x}_4\bar{x}_6$ $\bar{x}_3x_4\bar{x}_5$ $\bar{x}_3x_4\bar{x}_5$
$x_3\bar{x}_4x_6$ $x_3x_4\bar{x}_5$ $\bar{x}_3x_4x_5$
$x_3x_4x_5$ $x_3x_4x_5$ $x_3x_4x_5$
$\bar{x}_3\bar{x}_4\bar{x}_5\bar{x}_6$ $\bar{x}_3\bar{x}_4\bar{x}_5\bar{x}_6$ $\bar{x}_3\bar{x}_4\bar{x}_5\bar{x}_6$
$\bar{x}_3\bar{x}_4\bar{x}_5x_6$ $x_3\bar{x}_4\bar{x}_5\bar{x}_6$ $\bar{x}_3\bar{x}_4\bar{x}_5x_6$
$x_3\bar{x}_4x_5\bar{x}_6$ $\bar{x}_3x_4\bar{x}_5x_6$ $x_3\bar{x}_4x_5\bar{x}_6$
$x_3\bar{x}_4x_5x_6$ $x_3x_4\bar{x}_5x_6$ $x_3\bar{x}_4x_5x_6$
$x_3x_4\bar{x}_5\bar{x}_6$ $\bar{x}_3\bar{x}_4x_5x_6$ $x_3x_4\bar{x}_5\bar{x}_6$
$x_3x_5x_4x_6$ $x_3x_5x_4x_6$ $x_3x_5x_4x_6$

The nodes with a red cross are nodes eliminated by the test (T1') and the ones with a blue cross are eliminated by the test (T2'). (For convenience of display, these eliminated nodes are only drawn in the first three levels above.) For instance, the node with $X = \bar{x}_3 x_4$ is eliminated by the test (T1') because the minimum q such that $q^{\kappa(X)^{-1}\nu(x_4)} \in x_4^{\mathrm{Aut}(U_2)} = \{x_3, x_4\}$ when $\kappa(X) = \{x_3 \mapsto x_3, x_4 \mapsto x_4\}$ and $\nu(x_4) = \{x_3 \mapsto x_3, x_4 \mapsto x_4\}$ is x_3 and $x_4 \not\cong_{\mathrm{Aut}(X)} q^{\kappa(X)^{-1}} = x_3$ as $\mathrm{Aut}(X) = \{\epsilon\}$. On the other hand, the node with $X = \bar{x}_3\bar{x}_6$ is eliminated by the test (T2') as $x_6 \neq \min x_6^{\mathrm{Aut}(\bar{x}_3)}$ and $x_6^{\mathrm{Aut}(\bar{x}_3)} = \{x_4, x_6\}$. We observe that the search tree has dead-end nodes that do not extend to any full prefix assignment.

4.4. Correctness

We now establish the correctness of procedure (P') together with the tests (T1') and (T2') by reduction to McKay's framework and Lemma 4. Fix $j = 1, 2, \dots, k$. Let us start by defining the extension relation $e \subseteq \Omega_j \times \Omega_{j-1}$ for all $X \in \Omega_j$ and $S \in \Omega_{j-1}$ by setting $X e S$ if and only if

$$\text{there exists a } \gamma \in \Gamma \text{ such that } \underline{X}^{\gamma} = U_j,\ \underline{S}^{\gamma} = U_{j-1}, \text{ and } X^{\gamma}|_{U_{j-1}} = S^{\gamma}. \tag{5}$$

This relation is well-defined in the context of McKay's framework:

Lemma 8. *The relation (5) satisfies (E1) and (E2).*

Proof. To establish (E1), let $X \in \Omega_j$ and $S \in \Omega_{j-1}$ be arbitrary. It suffices to show that for all $\beta \in \Gamma$ we have XeS if and only if $X^\beta e S^\beta$. Let $\beta \in \Gamma$ be arbitrary. By (4), for all $\gamma \in \Gamma$ we have $\underline{X}^\gamma = U_j$ if and only if $\underline{X}^{\beta\beta^{-1}\gamma} = \underline{X}^{\beta\beta^{-1}\gamma} = \underline{X}^\gamma = U_j$. Similarly, for any $\gamma \in \Gamma$ we have $\underline{S}^\gamma = U_{j-1}$ if and only if $\underline{S}^{\beta\beta^{-1}\gamma} = \underline{S}^{\beta\beta^{-1}\gamma} = \underline{S}^\gamma = U_{j-1}$. Finally, for any $\gamma \in \Gamma$ that satisfies $\underline{X}^\gamma = U_j$ and $\underline{S}^\gamma = U_{j-1}$ (equivalently, $\beta^{-1}\gamma$ satisfies $\underline{X^\beta}^{\beta^{-1}\gamma} = U_j$ and $\underline{S^\beta}^{\beta^{-1}\gamma} = U_{j-1}$), we have $X^\gamma|_{U_{j-1}} = S^\gamma$ if and only if $(X^\beta)^{\beta^{-1}\gamma}|_{U_{j-1}} = X^\gamma|_{U_{j-1}} = S^\gamma = (S^\beta)^{\beta^{-1}\gamma}$. To establish (E2), observe that for an arbitrary $X \in \Omega_j$ there exists a $\gamma \in \Gamma$ with $\underline{X}^\gamma = U_j$, and thus XeS holds for $S = T^{\gamma^{-1}}$, where T is obtained from $Y = X^\gamma$ by deleting the assignment to the variable u_j. □

The following lemma establishes that the iteration in procedure (P') constructs exactly the objects $X \in e(S)$; cf. procedure (P).

Lemma 9. *Let $S \in \Omega_{j-1}$ be normalized. For all $X \in \Omega_j$ we have XeS if and only if there exists a $p \in u_j^{\mathrm{Aut}(U_{j-1})}$ with $\underline{X} = U_{j-1} \cup \{p\}$ and $X|_{U_{j-1}} = S$.*

Proof. From (5) we have that XeS if and only if there exists a $\gamma \in \Gamma$ with $\underline{X}^\gamma = U_j$, $\underline{S}^\gamma = U_{j-1}$, and $X^\gamma|_{U_{j-1}} = S^\gamma$. Since S is normalized, we have $\underline{S} = U_{j-1}$ and hence $U_{j-1}^\gamma = \underline{S}^\gamma = U_{j-1}$. Thus, $\gamma \in \mathrm{Aut}(U_{j-1})$ and

$$X|_{U_{j-1}} = X^{\gamma\gamma^{-1}}|_{U_{j-1}} = (X^\gamma|_{U_{j-1}})^{\gamma^{-1}} = (S^\gamma)^{\gamma^{-1}} = S\,. \tag{6}$$

Thus, to establish the "only if" direction of the lemma, take $p = u_j^{\gamma^{-1}}$, and for the "if" direction, take $\gamma \in \mathrm{Aut}(U_{j-1})$ with $p^\gamma = u_j$. □

Next we show the correctness of the test (T1') by establishing that it is equivalent with the test (T1) for a specific canonical extension function M. Towards this end, let us use the assumed canonical labeling map $\kappa : \Omega_j \to \Gamma$ to build a canonical extension function M using the template of Lemma 5. In particular, given an $X \in \Omega_j$ as input with $\underline{X} = U_{j-1} \cup \{p\}$, first construct the canonical form $Z = X^{\kappa(X)}$. In accordance with (T1'), select the minimum $q \in U$ such that $q^{\kappa(X)^{-1}\nu(p)} \in u_j^{\mathrm{Aut}(U_j)}$. Now construct $M(X)$ from X by deleting the value of $q^{\kappa(X)^{-1}}$.

Lemma 10. *The mapping $X \mapsto M(X)$ is well-defined and satisfies both (M1) and (M2).*

Proof. From (4) we have both $\mathrm{Aut}(Z) \le \mathrm{Aut}(\underline{Z})$ and $\underline{Z}^{\kappa(X)^{-1}\nu(p)} = U_j$. Thus,

$$\mathrm{Aut}(Z)^{\kappa(X)^{-1}\nu(p)} \le \mathrm{Aut}(\underline{Z})^{\kappa(X)^{-1}\nu(p)} = \mathrm{Aut}(U_j)\,.$$

It follows that the choice of q depends on Z and u_j but not on the choices of $\kappa(X)$ or $\nu(p)$. Furthermore, we observe that $q \in \underline{Z}$ and $q^{\kappa(X)^{-1}} \in \underline{X}$. Thus, the construction of $M(X)$ is well-defined and (M2) holds by Lemma 5.

To verify (M1), observe that since $q^{\kappa(X)^{-1}\nu(p)} \in u_j^{\mathrm{Aut}(U_j)}$, there exists an $\alpha \in \mathrm{Aut}(U_j)$ with $q^{\kappa(X)^{-1}\nu(p)\alpha} = u_j$. Thus, for $\gamma = \nu(p)\alpha$ we have $\underline{X}^\gamma = (U_{j-1} \cup \{p\})^{\nu(p)\alpha} = U_j^\alpha = U_j$, $\underline{M(X)}^\gamma = (U_j \setminus \{q^{\kappa(X)^{-1}\nu(p)}\})^\alpha = U_{j-1}$, and $X^\gamma|_{U_{j-1}} = M(X)^\gamma$. Thus, from (5) we have $XeM(X)$ and thus (M1) holds. □

To complete the equivalence between (T1') and (T1), observe that since X and p determine S by $X|_{\underline{X}\setminus\{p\}} = S$, and similarly X and $q^{\kappa(X)^{-1}}$ determine $M(X)$ by $X|_{\underline{X}\setminus\{q^{\kappa(X)^{-1}}\}} = M(X)$, the test (T1) is equivalent to testing whether $(X, p) \cong (X, q^{\kappa(X)^{-1}})$ holds, that is, whether $p \cong_{\mathrm{Aut}(X)} q^{\kappa(X)^{-1}}$ holds. Observe that this is exactly the test (T1').

It remains to establish the equivalence of (T2') and (T2). We start with a lemma that captures the $\mathrm{Aut}(S)$-orbits considered by (T2).

Lemma 11. *For a normalized $S \in \Omega_{j-1}$ the orbits in $e(S)/\mathrm{Aut}(S)$ are in a one-to-one correspondence with the elements of $(u_j^{\mathrm{Aut}(U_{j-1})}/\mathrm{Aut}(S)) \times R$.*

Proof. From (3) we have $\mathrm{Aut}(S) \leq \mathrm{Aut}(\underline{S}) = \mathrm{Aut}(U_{j-1})$ since S is normalized. Furthermore, Lemma 9 implies that every extension $X \in e(S)$ is uniquely determined by the variable $p \in u_j^{\mathrm{Aut}(U_{j-1})} \cap \underline{X}$ and the value $X(p) \in R$. Since the action (3) fixes the values in R elementwise, for any $X, X' \in e(S)$ we have $X \cong_{\mathrm{Aut}(S)} X'$ if and only if both $p \cong_{\mathrm{Aut}(S)} p'$ and $X(p) = X'(p')$. The lemma follows. $\square$

Now order the elements $X \in e(S)$ based on the lexicographic ordering of the pairs $(p, X(p)) \in u_j^{\mathrm{Aut}(U_{j-1})} \times R$. Since the action (3) fixes the values in R elementwise, we have that (T2') holds if and only if (T2) holds for this ordering of $e(S)$. The correctness of procedure (P') equipped with the tests (T1') and (T2') now follows from Lemma 4.

4.5. Selecting a prefix

This section gives a brief discussion on how to select the prefix. Let $U_k = \{u_1, u_2, \ldots, u_k\}$ be the set of variables in the prefix sequence. It is immediate that there exist $|R|^k$ distinct partial assignments from U_k to R. Let us write R^{U_k} for the set of these assignments. The group Γ now partitions R^{U_k} into orbits via the action (3), and it suffices to consider at most one representative from each orbit to obtain an exhaustive traversal of the search space, up to isomorphism. Our goal is thus to select the prefix U_k so that the setwise stabilizer Γ_{U_k} has comparatively few orbits on R^{U_k} compared with the total number of such assignments. In particular, the ratio of the number of orbits $|R^{U_k}/\Gamma_{U_k}|$ to the total number of mappings $|R|^k$ can be viewed as a proxy for the achieved symmetry reduction and as a rough[3] proxy for the speedup factor obtained compared with no symmetry reduction at all.

A yet further consideration in selecting a prefix is the extent of symmetry remaining under the action (3) once a partial assignment $X : U_k \to R$ has been fixed. This is precisely captured by the group $\mathrm{Aut}(X)$, whose generators are available for each generated X as a side-effect of executing the canonical labeling map κ for X to perform the test (T1'). Thus, as a side-effect of procedure (P') we obtain a precise characterization of the symmetry remaining at each generated partial assignment X. This remaining symmetry together with the ratio $|R^{U_k}/\Gamma_{U_k}|$ to $|R|^k$ can be used to assess whether a prefix is appropriate to achieve a desired extent of symmetry reduction.

4.6. Subroutines

By our assumption, the canonical labeling map κ produces as a side-effect a set of generators for the automorphism group $\mathrm{Aut}(X)$ for a given input X. We also assume that generators for the groups $\mathrm{Aut}(U_j)$ for $j = 0, 1, \ldots, k$ can be precomputed by similar means. This makes the canonical labeling map essentially the only nontrivial subroutine needed to implement procedure (P'). Indeed, the orbit computations required by tests (T1') and (T2') are implementable by elementary permutation group algorithms (Butler, 1991; Seress, 2003). Section 6 describes how to implement κ by reduction to vertex-colored graphs.[4]

[3] Here it should be noted that executing the symmetry reduction carries in itself a nontrivial computational cost. That is, there is a tradeoff between the potential savings in solving the system gained by symmetry reduction versus the cost of performing symmetry reduction. For example, if the instance has no symmetry and Γ is a trivial group, then executing symmetry reduction merely makes it more costly to solve the system.

[4] Reduction to vertex-colored graphs is by no means the only possibility to obtain the canonical labeling map to enable (P'), (T1'), and (T2'). Another possibility would be to represent Γ directly as a permutation group and use dedicated permutation-group algorithms (Leon, 1991, 1997). Our present choice of vertex-colored graphs is motivated by easy availability of carefully engineered implementations for working with vertex-colored graphs.

5. Value symmetries

The previous section considered prefix-assignment generation subject to an action of a group Γ on the set of variables U. In this section, we extend the framework so that it captures symmetries in values assigned to variables, or *value symmetries*. Towards this end, we extend the domain that records the symmetries from U to $U \times R$, where R is the set of values that can be assigned to the variables in U. Accordingly, in what follows we assume that the group Γ acts on $U \times R$ as well as on U, the latter by restriction.

The action of the group Γ on $U \times R$ may not be completely arbitrary, however, because we want partial assignments $X : W \to R$ with $W \subseteq U$ to remain well-defined functions under the action of Γ. This property is naturally captured by the *wreath product* group $\mathrm{Sym}(R) \wr \mathrm{Sym}(U)$ and its natural action on $U \times R$.

5.1. The wreath product and its actions

We will follow the convention that $\mathrm{Sym}(R) \wr \mathrm{Sym}(U)$ acts on $U \times R$ by first acting on U and then on R. For accessibility and convenience, we review our conventions in detail. The group $\mathrm{Sym}(R) \wr \mathrm{Sym}(U)$ consists of all pairs (π, σ), where $\pi \in \mathrm{Sym}(U)$ is a permutation of U and $\sigma : U \to \mathrm{Sym}(R)$ associates a permutation $\sigma(u) \in \mathrm{Sym}(R)$ with each element $u \in U$. In particular, $\mathrm{Sym}(R) \wr \mathrm{Sym}(U)$ has order $|U|! \cdot (|R|!)^{|U|}$.

The product of two elements $(\pi_1, \sigma_1), (\pi_2, \sigma_2) \in \mathrm{Sym}(R) \wr \mathrm{Sym}(U)$ is defined by $(\pi, \sigma) = (\pi_1, \sigma_1)(\pi_2, \sigma_2)$, where

$$\pi = \pi_1 \pi_2 \tag{7}$$

and for all $u \in U$ we set

$$\sigma(u) = \sigma_1(u^{\pi_2^{-1}})\sigma_2(u)\,. \tag{8}$$

The inverse of an element $(\pi, \sigma) \in \mathrm{Sym}(R) \wr \mathrm{Sym}(U)$ is thus given by $(\pi, \sigma)^{-1} = (\rho, \tau)$, where

$$\rho = \pi^{-1} \tag{9}$$

and for all $u \in U$ we have

$$\tau(u) = \sigma(u^{\pi})^{-1}\,. \tag{10}$$

An element $(\pi, \sigma) \in \mathrm{Sym}(R) \wr \mathrm{Sym}(U)$ acts on an element $u \in U$ by

$$u^{(\pi,\sigma)} = u^{\pi} \tag{11}$$

and on a pair $(u, r) \in U \times R$ by

$$(u, r)^{(\pi,\sigma)} = (u^{\pi}, r^{\sigma(u^{\pi})})\,. \tag{12}$$

Here in particular the intuition is that we first act on (u, r) with π to obtain (u^{π}, r), and then act with $\sigma(u^{\pi})$ to obtain $(u^{\pi}, r^{\sigma(u^{\pi})})$. Extend the action (11) elementwise to subsets $W \subseteq U$.

5.2. Partial assignments and isomorphism

Let Γ be a subgroup of $\mathrm{Sym}(R) \wr \mathrm{Sym}(U)$ and let Γ act on U and $U \times R$ by (11) and (12), respectively. Furthermore, we let an element $\gamma = (\pi, \sigma) \in \Gamma$ act on a partial assignment $X : W \to R$ with $W \subseteq U$ to produce the partial assignment $X^{\gamma} : W^{\pi} \to R$ defined for all $u \in W^{\pi}$ by

$$X^{\gamma}(u) = X(u^{\pi^{-1}})^{\sigma(u)}\,. \tag{13}$$

In analogy with Lemma 6, let us verify that the value-permuting action (13) is well-defined.

Lemma 12. *The action* (13) *is well-defined.*

Proof. We observe that for the identity $\epsilon \in \Gamma$ of $\Gamma \leq \mathrm{Sym}(R) \wr \mathrm{Sym}(U)$, we have $X^{\epsilon} = X$. Furthermore, for all $\gamma_1 = (\pi_1, \sigma_1) \in \Gamma$, $\gamma_2 = (\pi_2, \sigma_2) \in \Gamma$, and $u \in W^{\gamma_1\gamma_2} = (W^{\gamma_1})^{\gamma_2}$, by (13), (7), and (8), we have

$$X^{\gamma_1\gamma_2}(u) = X(u^{(\pi_1\pi_2)^{-1}})^{\sigma_1(u^{\pi_2^{-1}})\sigma_2(u)} = X(u^{\pi_2^{-1}\pi_1^{-1}})^{\sigma_1(u^{\pi_2^{-1}})\sigma_2(u)}$$
$$= X^{\gamma_1}(u^{\pi_2^{-1}})^{\sigma_2(u)} = (X^{\gamma_1})^{\gamma_2}(u) . \quad \square$$

Let us recall that for $X : W \to R$ we write $\underline{X} = W$ for the underlying set of variables assigned by X. In analogy with Section 4.1, the underline map is a homomorphism of group actions that satisfies (4) for the action (13) and the action (11) extended elementwise to subsets of U. Isomorphism for partial assignments is now induced by the action (13).

5.3. Generating normalized assignments

Working with the group action (13), let $u_1, u_2, \ldots, u_k$ be k distinct elements of U, and let $U_j = \{u_1, u_2, \ldots, u_j\}$ for $j = 0, 1, \ldots, k$. Let Ω_j consist of all partial assignments $X : W \to R$ with $W \cong U_j$. We construct exactly one object from each orbit of Γ on Ω_j, using as seeds exactly one object from each orbit of Γ on Ω_{j-1}, for each $j = 1, 2, \ldots, k$, assuming the availability of canonical labeling maps $\kappa : \Omega_j \to \Gamma$. We say the assignment $X \in \Omega_j$ is *normalized* if $\underline{X} = U_j$.

We now present a version of the procedure (P') modified for the group action (13). Let us fix $j = 1, 2, \ldots, k$. We assume that the procedure is invoked for exactly one normalized representative $S \in \Omega_{j-1}$ from each orbit in Ω_{j-1}/Γ.

(P") Let a normalized $S \in \Omega_{j-1}$ be given as input. For each $p \in u_j^{\mathrm{Aut}(U_{j-1})}$ and each $r \in R$, construct the assignment $X : U_{j-1} \cup \{p\} \to R$ defined by $X(p) = r$ and $X(u) = S(u)$ for all $u \in U_{j-1}$. Perform the isomorph rejection tests (T1') and (T2") on X and S. If both tests accept, visit $X^{\nu(p)}$ where $\nu(p) \in \mathrm{Aut}(U_{j-1})$ normalizes X.

In particular, procedure (P") has two differences compared with procedure (P'). First, the underlying group action is (13). Second, the test (T2') has been replaced with a new test (T2") to account for more extensive orbits of pairs (p, r) under the action of $\mathrm{Aut}(S)$.

5.4. The isomorph rejection tests

Assume that the elements of U, R, and $U \times R$ have been arbitrarily ordered and that $\kappa : \Omega_j \to \Gamma$ is a canonical labeling map. Suppose that X has been constructed by extending a normalized S with $\underline{X} = \underline{S} \cup \{p\} = U_{j-1} \cup \{p\}$ and $X(p) = r$. Let us first recall the test (T1') for convenience:

(T1') Subject to the ordering of U, select the minimum $q \in U$ such that $q^{\kappa(X)^{-1}\nu(p)} \in u_j^{\mathrm{Aut}(U_j)}$. Accept if and only if $p \cong_{\mathrm{Aut}(X)} q^{\kappa^{-1}(X)}$.

The new isomorph rejection test is as follows:

(T2") Accept if and only if $(p, r) = \min\,(p, r)^{\mathrm{Aut}(S)}$ subject to the ordering of $U \times R$.

5.5. Correctness

We now establish the correctness of the modified procedure (P"). Fix $j = 1, 2, \ldots, k$. Define the extension relation $e \subseteq \Omega_j \times \Omega_{j-1}$ as in (5). This relation is well-defined in the context of McKay's framework under the modified group action.

Lemma 13. *The relation* (5) *satisfies (E1) and (E2) when the group action is as defined in* (13).

Proof. Identical to Lemma 8 since (4) holds for the action (13) and the action (11) extended elementwise to subsets of U. □

The correctness analysis of the test (T1') proceeds identically as in Section 4.1, relying on (4) in the proof of Lemma 10. To establish the correctness of the new test (T2"), we first observe that the counterpart of Lemma 9 holds for the modified group action.

Lemma 14. *Let $S \in \Omega_{j-1}$ be normalized. For all $X \in \Omega_j$, we have XeS if and only if there exists a $p \in u_j^{\mathrm{Aut}(U_{j-1})}$ with $\underline{X} = U_{j-1} \cup \{p\}$ and $X|_{U_{j-1}} = S$.*

Proof. First observe that (6) holds for the action (13). Then proceed as in the proof of Lemma 9. □

Let us now proceed to the counterpart of Lemma 11.

Lemma 15. *For a normalized $S \in \Omega_{j-1}$, the orbits in $e(S)/\mathrm{Aut}(S)$ are in a one-to-one correspondence with the orbits in $(u_j^{\mathrm{Aut}(U_{j-1})} \times R)/\mathrm{Aut}(S)$.*

Proof. Lemma 14 implies that every extension $X \in e(S)$ is uniquely determined by the variable $p \in u_j^{\mathrm{Aut}(U_{j-1})} \cap \underline{X}$ and the value $X(p) \in R$. That is, the elements in $e(S)$ are in one-to-one correspondence with elements in $u_j^{\mathrm{Aut}(U_{j-1})} \times R$.

Let $\mathrm{Aut}(S)$ act on $e(S)$ via (13); this action is well-defined by Lemma 13 and (E1) since for all $\alpha \in \mathrm{Aut}(S)$ we have XeS if and only if $X^\alpha eS$. Let $\mathrm{Aut}(S)$ act on $u_j^{\mathrm{Aut}(U_{j-1})} \times R$ via (12); this action is well-defined because S is normalized and hence $\mathrm{Aut}(S) \le \mathrm{Aut}(\underline{S}) = \mathrm{Aut}(U_{j-1})$ holds by (4).

Let $X, Y \in e(S)$ be arbitrary with $\underline{X} = U_{j-1} \cup \{p\}$ and $\underline{Y} = U_{j-1} \cup \{q\}$. We now claim that $X \cong_{\mathrm{Aut}(S)} Y$ holds under the action (13) if and only if $(p, X(p)) \cong_{\mathrm{Aut}(S)} (q, Y(q))$ holds under the action (12). To see this, first observe that for all $\alpha \in \mathrm{Aut}(S)$ we have $U_{j-1}^\alpha = \underline{S}^\alpha = \underline{S} = U_{j-1}$ by (4) since S is normalized. Furthermore, $X|_{U_{j-1}} = Y|_{U_{j-1}} = S$. Thus, by (13) it holds that for all $\alpha = (\pi, \sigma) \in \mathrm{Aut}(S)$ with $\pi \in \mathrm{Sym}(U)$ and $\sigma : U \to \mathrm{Sym}(R)$ we have $Y = X^\alpha$ if and only if $q = p^\pi$ and $Y(q) = X^\alpha(q) = X(q^{\pi^{-1}})^{\sigma(q)} = X(p)^{\sigma(p^\pi)}$. Or what is the same by (12), if and only if $(q, Y(q)) = (p^\pi, X(p)^{\sigma(p^\pi)}) = (p, X(p))^\alpha$. □

Order the elements $X \in e(S)$ based on the lexicographic ordering of the pairs $(p, X(p)) \in u_j^{\mathrm{Aut}(U_{j-1})} \times R$. We now have that (T2") holds if and only if (T2) holds for this ordering of $e(S)$. The correctness of procedure (P") equipped with the tests (T1') and (T2") now follows from Lemma 4.

6. Representation using vertex-colored graphs

This section describes one possible approach to represent the group of symmetries $\Gamma \le \mathrm{Sym}(U)$ of a system of constraints over a finite set of variables U taking values in a finite set R. Our representation of choice will be vertex-colored graphs over a fixed finite set of vertices V. In particular, isomorphisms between such graphs are permutations $\gamma \in \mathrm{Sym}(V)$ that map edges onto edges and respect the colors of the vertices; that is, every vertex in V maps to a vertex of the same color under γ. It will be convenient to develop the relevant graph representations in steps, starting with the representation of the constraint system and then proceeding to the representation of setwise stabilizers and partial assignments. These representations are folklore (see e.g. Kaski and Östergård, 2006) and are presented here for completeness of exposition only.

6.1. *Representing the constraint system*

To capture $\Gamma \cong \mathrm{Aut}(G)$ via a vertex-colored graph G with vertex set V, it is convenient to represent the variables U directly as a subset of vertices $U \subseteq V$ such that no vertex in $V \setminus U$ has a color that agrees with a color of a vertex in U. We then seek a graph G such that $\mathrm{Aut}(G) \le \mathrm{Sym}(U) \times \mathrm{Sym}(V \setminus U)$ projected to U is exactly Γ. In most cases such a graph G is concisely obtainable by encoding the system of constraints with additional vertices and edges joined to the vertices representing the variables in U. We discuss two examples.

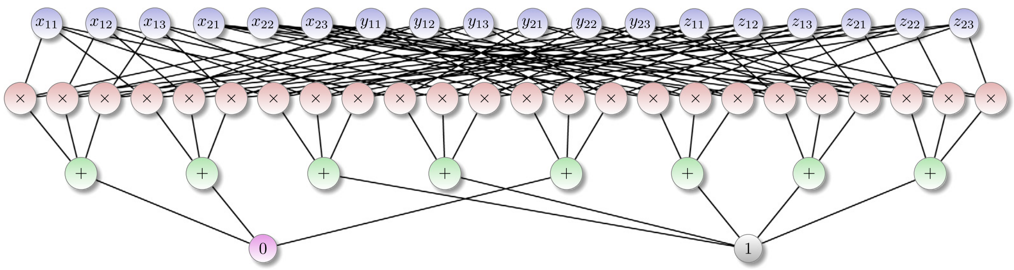
Example 16. Consider the system of clauses (1) and its graph representation (2). The latter can be obtained as follows. First, introduce a blue vertex for each of the six variables of (1). These blue vertices constitute the subset U. Then, to accommodate negative literals, introduce a red vertex joined by an edge to the corresponding blue vertex representing the positive literal. These edges between red and blue vertices ensure that positive and negative literals remain consistent under isomorphism. Finally, introduce a green vertex for each clause of (1) with edges joining the clause with each of its literals. It is immediate that we can reconstruct (1) from (2) up to labeling of the variables even after arbitrary color-preserving permutation of the vertices of (2). Thus, (2) represents the symmetries of (1).

Let us next discuss an example where it is convenient to represent the symmetry at the level of original constraints rather than at the level of clauses.

Example 17. Consider the following system of eight cubic equations over 24 variables taking values modulo 2:

$$\begin{aligned}
&x_{11}y_{11}z_{11} + x_{12}y_{12}z_{12} + x_{13}y_{13}z_{13} = 0 \qquad && x_{21}y_{11}z_{11} + x_{22}y_{12}z_{12} + x_{23}y_{13}z_{13} = 0\\
&x_{11}y_{11}z_{21} + x_{12}y_{12}z_{22} + x_{13}y_{13}z_{23} = 0 && x_{21}y_{11}z_{21} + x_{22}y_{12}z_{22} + x_{23}y_{13}z_{23} = 1\\
&x_{11}y_{21}z_{11} + x_{12}y_{22}z_{12} + x_{13}y_{23}z_{13} = 1 && x_{21}y_{21}z_{11} + x_{22}y_{22}z_{12} + x_{23}y_{23}z_{13} = 1\\
&x_{11}y_{21}z_{21} + x_{12}y_{22}z_{22} + x_{13}y_{23}z_{23} = 1 && x_{21}y_{21}z_{21} + x_{22}y_{22}z_{22} + x_{23}y_{23}z_{23} = 1
\end{aligned}$$

This system seeks to decompose a $2 \times 2 \times 2$ tensor (whose elements appear on the right hand sides of the equations) into a sum of three rank-one tensors. The symmetries of addition and multiplication modulo 2 imply that the symmetries of the system can be represented by the following vertex-colored graph:


x_{11} x_{12} x_{13} x_{21} x_{22} x_{23} y_{11} y_{12} y_{13} y_{21} y_{22} y_{23} z_{11} z_{12} z_{13} z_{21} z_{22} z_{23}
×
+
0
1

Indeed, we encode each monomial in the system with a product-vertex, and group these product-vertices together by adjacency to a sum-vertex to represent each equation, taking care to introduce two uniquely colored constant-vertices to represent the right-hand side of each equation.

Remark. The representation built directly from the system of polynomial equations in Example 17 concisely captures the symmetries in the system independently of the final encoding of the system (e.g. as CNF) for solving purposes. In particular, building the graph representation from such a final CNF encoding (cf. Example 16) results in a less compact graph representation and obfuscates the symmetries of the original system, implying less efficient symmetry reduction.

6.2. Representing the values

In what follows it will be convenient to assume that the graph G contains a uniquely colored vertex for each value in R. (Cf. the graph in Example 17.) That is, we assume that $R \subseteq V \setminus U$ and that $\mathrm{Aut}(G)$ projected to R is the trivial group.

6.3. Representing setwise stabilizers in the prefix chain

To enable procedure (P') and the tests (T1') and (T2'), we require generators for $\mathrm{Aut}(U_j) \le \Gamma$ for each $j = 0, 1, \ldots, k$. More generally, given a subset $W \subseteq U$, we seek to compute a set of generators for the setwise stabilizer $\mathrm{Aut}_\Gamma(W) = \Gamma_W = \{\gamma \in \Gamma : W^\gamma = W\}$, with $W^\gamma = \{w^\gamma : w \in W\}$. Assuming we have available a vertex-colored graph G that represents Γ by projection of $\mathrm{Aut}_{\mathrm{Sym}(V)}(G)$ to U, let us define the graph $G \uparrow W$ by selecting one vertex $r \in R$ and joining each vertex $w \in W$ with an edge to the vertex r. It is immediate that $\mathrm{Aut}_{\mathrm{Sym}(V)}(G \uparrow W)$ projected to U is precisely $\mathrm{Aut}_\Gamma(W)$.

6.4. Representing partial assignments

Let $X : W \to R$ be an assignment of values in R to variables in $W \subseteq U$. Again to enable procedure (P') together with the tests (T1') and (T2'), we require a canonical labeling $\kappa(X)$ and generators for the automorphism group $\mathrm{Aut}(X)$. Again assuming we have a vertex-colored graph G that represents Γ, let us define the graph $G \uparrow X$ by joining each vertex $w \in W$ with an edge to the vertex $X(w) \in R$. It is immediate that $\mathrm{Aut}_{\mathrm{Sym}(V)}(G \uparrow X)$ projected to U is precisely $\mathrm{Aut}_\Gamma(X)$. Furthermore, a canonical labeling $\kappa(X)$ can be recovered from $\kappa(G \uparrow X)$ and the canonical form $(G \uparrow X)^{\kappa(G \uparrow X)}$.

6.5. Using tools for vertex-colored graphs

Given a vertex-colored graph G as input, practical tools exist for computing a canonical labeling $\kappa(G) \in \mathrm{Sym}(V)$ and a set of generators for $\mathrm{Aut}(G) \le \mathrm{Sym}(V)$. Such tools include *bliss* (Junttila and Kaski, 2007), *nauty* (McKay, 1981; McKay and Piperno, 2014), and *traces* (McKay and Piperno, 2014). Once the canonical labeling and generators are available in $\mathrm{Sym}(V)$ it is easy to map back to Γ by projection to U so that corresponding elements of Γ are obtained.

7. Parallel implementation

This section outlines the parallel implementation of our technique into a tool called `reduce`. The implementation is written in C++ and structured as a preprocessor that works with an explicitly given graph representation. In the absence of such an input graph, the graph is constructed automatically from CNF as described in Section 6. The *nauty* (McKay, 1981; McKay and Piperno, 2014) canonical labeling software for vertex-colored graphs is utilized as a subroutine.

7.1. Backtracking search for partial assignments

The backtracking search for partial assignments is implemented using a stack that stores nodes of the search tree. (Recall Example 7 for an illustration of a search tree.) Each node X in the stack represents the complete subtree of the search tree rooted at X. Initially, the stack consists of the empty assignment, which represents the entire search tree.

Throughout the search, we maintain the invariant that the nodes stored in the stack represent pairwise node-disjoint subtrees of the search tree, which enables us to work through the contents of the stack in arbitrary order and to distribute the contents of the stack to multiple compute nodes as necessary; we postpone a detailed discussion of the distribution of the stack and parallelization of the search to Section 7.2.

Viewed as a sequential process, the search proceeds by iterating the following work procedure until the stack is empty:

(W) Pop an assignment X_ℓ from the stack. Unless X_ℓ is the empty assignment (that is, unless $\ell = 0$), it will have the form $X_\ell : U_{\ell-1} \cup \{p_\ell\} \to R$ for some $p_\ell \in u_\ell^{\mathrm{Aut}(U_{\ell-1})}$. Furthermore, X_ℓ extends the normalized assignment $S_{\ell-1} = X|_{U_{\ell-1}}$. Execute the test (T1') on X_ℓ and $S_{\ell-1}$. If the test (T1') fails, reject the subtree of X_ℓ from further consideration. If either $\ell = 0$ or the test (T1') passes, then normalize X_ℓ to obtain $S_\ell = X_\ell^{\nu(p_\ell)}$ with $S_\ell : U_\ell \to R$. At this point S_ℓ has been accepted as the unique representative of its isomorphism class. If $\ell = k$, then output the full prefix assignment S_ℓ. If $\ell \le k-1$, proceed to consider extensions of S_ℓ at level $\ell+1$ as follows. Iterate over each variable-value pair $(p_{\ell+1}, r)$ with $p_{\ell+1} \in u_{\ell+1}^{\mathrm{Aut}(U_\ell)}$ and $r \in R$. Construct the assignment $X_{\ell+1} : U_\ell \cup \{p_{\ell+1}\} \to R$ by setting $X_{\ell+1}(p_{\ell+1}) = r$ and $X_{\ell+1}(u) = S_\ell(u)$ for all $u \in U_\ell$. For each constructed $X_{\ell+1}$, perform the test (T2'). If the test (T2') passes, push $X_{\ell+1}$ to the stack.

We observe that the procedure (W) above implements procedure (P') using the stack to maintain the state of the search. In particular, when a single worker process executes the search, we obtain a standard depth-first traversal of the search tree. However, we also observe that procedure (W) pushes *all* the child nodes of S_ℓ to the stack before consulting the stack for further work. This enables multiple worker processes, all executing procedure (W), to work in parallel, if we take care to ensure that (i) push and pop operations to the stack are atomic, and (ii) the termination condition is changed from the stack being empty to the stack being empty and all worker procedures being idle. Furthermore, as presented in more detail in what follows, we can distribute the stack across multiple compute nodes by appropriately communicating push and pop requests between nodes.

7.2. *Parallelization and distributing the stack*

We parallelize the search using the OpenMPI implementation (Gabriel et al., 2004) of the Message Passing Interface (MPI) (Message Passing Interface Forum, 2015; Pacheco, 1997; Gropp et al., 2014). We provide two different communication modes, both of which rely on a *master–slave* paradigm with N processes. The *master* process with rank 0 distributes the work to $N-1$ *worker* processes that, in turn, communicate their results back to the master process. We now proceed with a more detailed description of the two communication modes.

Master stack mode. In the simpler of the two modes, the master process stores the entire stack. The worker processes interact with the master directly, making push and pop requests to the master process via MPI messages. While inefficient in terms of communication and in terms of potentially overwhelming the master node, this mode provides load balancing that is empirically adequate for a small number of compute nodes and instances whose search tree is not too wide.

Hierarchical stack mode. The hierarchical stack mode divides the $N-1$ worker nodes into M classes, each of which is associated with a subset of levels of the search tree. Each worker process maintains a *local stack* for nodes at their respective levels. Whenever a worker process pushes an assignment, the assignment is stored in the local stack if the level of the assignment belongs to the levels associated with the node; otherwise, the assignment is communicated to the master process which then pushes the assignment to the *global stack* maintained in the master process. Whenever a worker process pops an assignment, the worker process first consults its local stack and pops the assignment from the local stack if an assignment is available; otherwise, the worker process makes a pop request to the master process, which supplies an assignment from the global stack as soon as an assignment of one of the levels associated with the worker becomes available. This strategy helps in cases where the search tree becomes very wide; in our experiments, we found that a simple thresholding into one low-level process that processes levels $1, 2, \ldots, t$, and $N-2$ high level processes that process levels $t+1, t+2, \ldots, k$ was sufficient.

For both modes of communication, the master process keeps track of the worker processes that are idle, that is, workers that have sent pop requests that have not been serviced. If all workers are idle and the global stack is empty, the master process instructs all worker processes to exit and then exits itself.

These communication modes serve as a proof-of-concept of the practical parallelizability of our present technique for symmetry reduction. For parallelization to very large compute clusters, we expect that more advanced communication strategies will be required (see, for example, Dinan et al., 2009; Pezzi et al., 2007 or Pacheco, 1997); however, the implementation of such strategies is beyond the scope of the present work.

8. Experiments

This section documents an experimental evaluation of our parallel implementation of the adaptive prefix-assignment technique. Our main objective is to demonstrate the effective parallelizability of the approach, but we will also report on experiments comparing the performance of our tool (without parallelization) with existing tools that do not parallelize.

8.1. Instances

Let us start by defining the families of input instances used in our experiments. First, we study the usefulness of an auxiliary symmetry graph with systems of polynomial equations aimed at discovering the tensor rank of a small $m \times m \times m$ tensor $T = (t_{ijk})$ modulo 2, with $t_{ijk} \in \{0, 1\}$ and $i, j, k = 1, 2, \ldots m$. Computing the rank of a given tensor is NP-hard (Håstad, 1990).[5] In precise terms, we seek to find the minimum r such that there exist three $m \times r$ matrices $A, B, C \in \{0, 1\}^{m\times r}$ such that for all $i, j, k = 1, 2, \ldots, m$ we have

$$\sum_{\ell=1}^{r} a_{i\ell} b_{j\ell} c_{k\ell} = t_{ijk} \pmod 2 . \tag{14}$$

Such instances are easily compilable into CNF with A, B, C constituting three matrices of Boolean variables so that the task becomes to find the minimum r such that the compiled CNF instance is satisfiable. Independently of the target tensor T, such instances have a symmetry group of order at least $r!$ due to the fact that the columns of the matrices A, B, C can be arbitrarily permuted so that (14) maps to itself. In our experiments, we select the entries of T uniformly at random so that the number of 1s in T is exactly n. We use the first three rows of the matrix A as the prefix sequence.

As a further family of instances with considerable symmetry, we study the *Clique Coloring Problem* (CCP) that yields empirically difficult-to-solve instances for contemporary SAT solvers (Manthey, 2014). For positive integer parameters n, s, and t, the CCP asks whether there exists an undirected t-colorable graph on n nodes such that the graph contains a complete graph K_s as a subgraph. Such instances are unsatisfiable if $s > t$. The particular encoding that we use (see Manthey, 2014) is as follows. Introduce variables $x_{i,j}$ for $1 \le i, j \le n$ with $i \ne j$ to indicate the presence of an edge joining vertex i and j, variables $y_{p,j}$ for $1 \le p \le s$ with $1 \le j \le n$ to indicate that vertex j occupies slot p in a clique, and variables $z_{i,k}$ for $1 \le i \le n$ and $1 \le k \le t$ to indicate that vertex i has color k. The clauses are

1. $\bigwedge_{1\le p\le s} \bigvee_{1\le j\le n} y_{p,j}$,
2. $\bigwedge_{1\le p\le s} \bigwedge_{1\le q\le s: p\ne q} \bigwedge_{1\le j\le n} \overline{y_{p,j}} \vee \overline{y_{q,j}}$,
3. $\bigwedge_{1\le p\le s} \bigwedge_{1\le q\le s: p\ne q} \bigwedge_{1\le i\le n} \bigwedge_{1\le j\le n: i\ne j} \overline{y_{p,i}} \vee \overline{y_{q,j}} \vee x_{i,j}$,

[5] Yet considerable interest exists to determine tensor ranks of small tensors, in particular tensors that encode and enable fast matrix multiplication algorithms; cf. Alekseev (2014, 2015), Alekseev and Smirnov (2013), Alekseyev (1985), Bläser (1999, 2003), Courtois et al. (2012), Hopcroft and Kerr (1971), Laderman (1976), Strassen (1969), Winograd (1971). For numerical work on discovering small low-rank tensor decompositions, cf. Benson and Ballard (2015), Huang et al. (2017), Smirnov (2013).

4. $\bigwedge_{1\le k\le t}\bigwedge_{1\le i\le n}\bigwedge_{1\le j\le n: i\ne j}\overline{z_{i,k}}\vee\overline{z_{j,k}}\vee\overline{x_{i,j}}$, and
5. $\bigwedge_{1\le i\le n}\bigvee_{1\le k\le t} z_{i,k}$.

We consider unsatisfiable instances with parameters $s\in\{5,6\}$, $t=s-1$, and let n vary from 15 to 20 in the case of $s=5$ and from 12 to 24 when $s=6$. We use the variables $y_{1,1}, y_{1,2}, \ldots, y_{1,n}$ as the prefix sequence. The auxiliary graph for encoding the symmetries is constructed as follows. Introduce a vertex for each variable $x_{i,j}$, for each variable $y_{p,j}$, and for each variable $z_{i,k}$. These vertices are colored with three distinct colors, one color for each type of variable. Next, introduce three types of auxiliary vertices, with each type colored with its own distinct color. Introduce vertices $1, 2, \ldots, n$ for the n nodes, vertices $1', 2', \ldots, s'$ for the s clique slots, and vertices $1'', 2'', \ldots, t''$ for the t node colors. Thus, in total the graph consists of $n(n-1)+sn+tn+n+s+t$ vertices colored with six distinct colors. To complete the construction of the auxiliary graph, introduce edges to the graph so that each variable $x_{i,j}$ is joined to the nodes i and j, each variable $y_{p,j}$ is joined to clique slot p' and to the node j, and each variable $z_{i,k}$ is joined to the node i and to the node color k''.

We study the parallelizability of our algorithm using two input instances with hard combinatorial symmetry. The first instance, which we call $R(4,4;18)$ in what follows, is an unsatisfiable CNF instance that asks whether there exists an 18-node graph with the property that neither the graph nor its complement contains the complete graph K_4 as a subgraph. That is, we ask whether the Ramsey number $R(4,4)$ satisfies $R(4,4)>18$. (In fact, $R(4,4)=18$ (Graham et al., 1990).) No auxiliary graph is provided to accompany this instance. The second instance consists of an empty CNF over 36 variables together with an auxiliary graph that encodes the isomorphism classes of 9-node graphs by inserting a variable vertex in the middle of each of the $\binom{9}{2}=36$ edges of the complete graph K_9. Applying `reduce` with a length-36 prefix sequence (listing the 36 variable vertices in any order) yields a complete listing of all the 274668 isomorphism classes of 9-node graphs. The number of isomorphism classes of graphs of order n is the sequence A000088 in the *Online Encyclopedia of Integer Sequences*.

Finally, we study the residual symmetry and the effect of symmetry reduction on the sizes of the automorphism group with the $R(4,4;18)$ instance described above, as well as instances arising from the Pigeon-Hole Principle (PHP), and the construction of Steiner Triple Systems (STS). A PHP instance asks whether we can assign n pigeons into $n-1$ holes such that each hole is occupied by at most one pigeon, with the variables encoding whether pigeon i occupies hole j. We use 11 and 12 pigeons in our experiments. An STS instance asks whether, given a universe of n elements, we can choose a set of triples (3-subsets) of the elements such that each pair of elements occurs in exactly one chosen triple; the $\binom{n}{3}$ variables encode whether a triple is included in the system. It is well-known that such sets of triples exist if and only if $n\equiv 1$ or $n\equiv 3 \pmod 6$. In our experiments, we choose $n=16, 18, 20$ so the instances are unsatisfiable. We use the following prefix sequences: for $R(4,4;18)$, we simply fix either one or two vertices, yielding a sequence of length 17 or 33, respectively. For PHP instances, the prefix sequence simply fixes one or two pigeons, that is, has length $n-1$ or $2(n-1)$. For STS instances, the prefix sequence is chosen by fixing one triple (say, $\{1,2,3\}$), and one pair present in the triple (say, $\{1,2\}$), and then fixing all triples that contain the pair and thus cannot be included in the set if the initial triple is included, yielding a prefix of length $n-2$.

8.2. Hardware and software configuration

The experiments were performed on a cluster of Dell PowerEdge C4130 compute nodes, each equipped with two Intel Xeon E5-2680v3 CPUs (12 cores per CPU, 24 cores per node) and 128 GiB (8×16 GiB) of DDR4-2133 main memory, running the CentOS 7 distribution of GNU/Linux. Comparative experiments were executed by allocating a single core on a single CPU of a compute node. All experiments were conducted as batch jobs using the `slurm` batch scheduler, and running between one to four physical nodes, with one to 24 cores allocated in each node, using one MPI process per core. OpenMPI version 2.1.1 was used as the MPI implementation.

8.3. Symmetry reduction tools and SAT solvers

We report on three methods for symmetry reduction: (1) no reduction ("raw"), (2) breakid version 2.1-152-gb937230-dirty[6] (Devriendt et al., 2016), (3) our technique ("reduce") with a user-selected prefix. Three different SAT solvers were used in the experiments: lingeling and ilingeling version bbc-9230380 (Biere, 2016), and glucose version 4.1 (Audemard and Simon, 2016). We use the incremental solver ilingeling together with the incremental CNF output of reduce, which is available from the command line of reduce via the "-i" modifier and is piped, as is, to ilingeling.

8.4. Experiments on parallel speedup

This section documents experiments that study the wall-clock running time of symmetry reduction using our tool reduce as we increase the number of CPU cores and compute nodes participating in parallel symmetry reduction. The range of the experiments was between one to four compute nodes, with one to 24 cores allocated in each node. One MPI process was launched per core. Each node was exclusively reserved for the experiment. In addition to the wall-clock running time, we measure the *total reserved time* that is obtained by recording, for each core, the length of the time interval the core is reserved for an experiment, and taking the sum of these time intervals. The total reserved time conservatively tracks the total resources consumed by an experiment in a batch job environment regardless of whether each allocated core is running or idle.

The results of our parallel speedup experiments are displayed in Fig. 1. The top-left plot in the figure displays the parallel speedup (ratio of parallel wall-clock running time to sequential running time) of running our tool reduce on the instance $R(4,4;18)$ with a length-33 prefix sequence as a function of the number of cores used for one, two, and four allocated compute nodes. We also display the line $y = x$ for reference to compare against perfect linear speedup. As the number of cores grows, in the top-left plot we observe linear scaling of the speedup as a function of the number of cores. The slope of the speedup yet remains somewhat short of the perfect $y = x$ scaling. This is most likely due to the use of the master stack mode and associated communication overhead. The top-right plot displays the total reserved time to demonstrate the total resource usage in addition to the parallel speedup. Table 1 displays the number of canonical partial assignments at different levels of the search tree explored by reduce.

The two plots in the middle row of Fig. 1 display the parallel speedup and the total reserved time of executing our tool reduce on the instance A000088 (with $n = 9$ and a length-36 prefix sequence) in the master stack mode. This instance requires extensive stack access with many easy instances of canonical labeling (cf. Table 2 and compare with Table 1); accordingly we observe poor speedup from parallelization in the master stack mode. The two plots in the bottom row of Fig. 1 show an otherwise identical experiment but now executed in hierarchical stack mode with the threshold parameter set to $t = 21$, in which case both the parallel speedup obtained and the total resource usage become substantially better.

When the number of processes is small, Fig. 1 reveals inefficiency in terms of the total reserved time compared with a larger number of processes. This inefficiency is explained by two factors. First, when the number of processes is small, a significant fraction of the total reserved time is used by the master process which does not contribute work to the exploration of the search tree but does consume reserved time from the start to the end of the computation. As soon as more worker processes start exploring the search tree, the total reserved time decreases because the time consumed by the master process decreases. Second, in hierarchical stack mode, a small number of processes means that some of the worker nodes processing lower levels of the search tree can run out of work—but will still consume total reserved time—as assignments in these levels are exhausted, while the small number of processes assigned to work on the higher levels of the tree still remain at work. This bottleneck can be alleviated by increasing the number of workers associated with the higher levels.

[6] We thank Bart Bogaerts for implementing custom graph input in breakid.

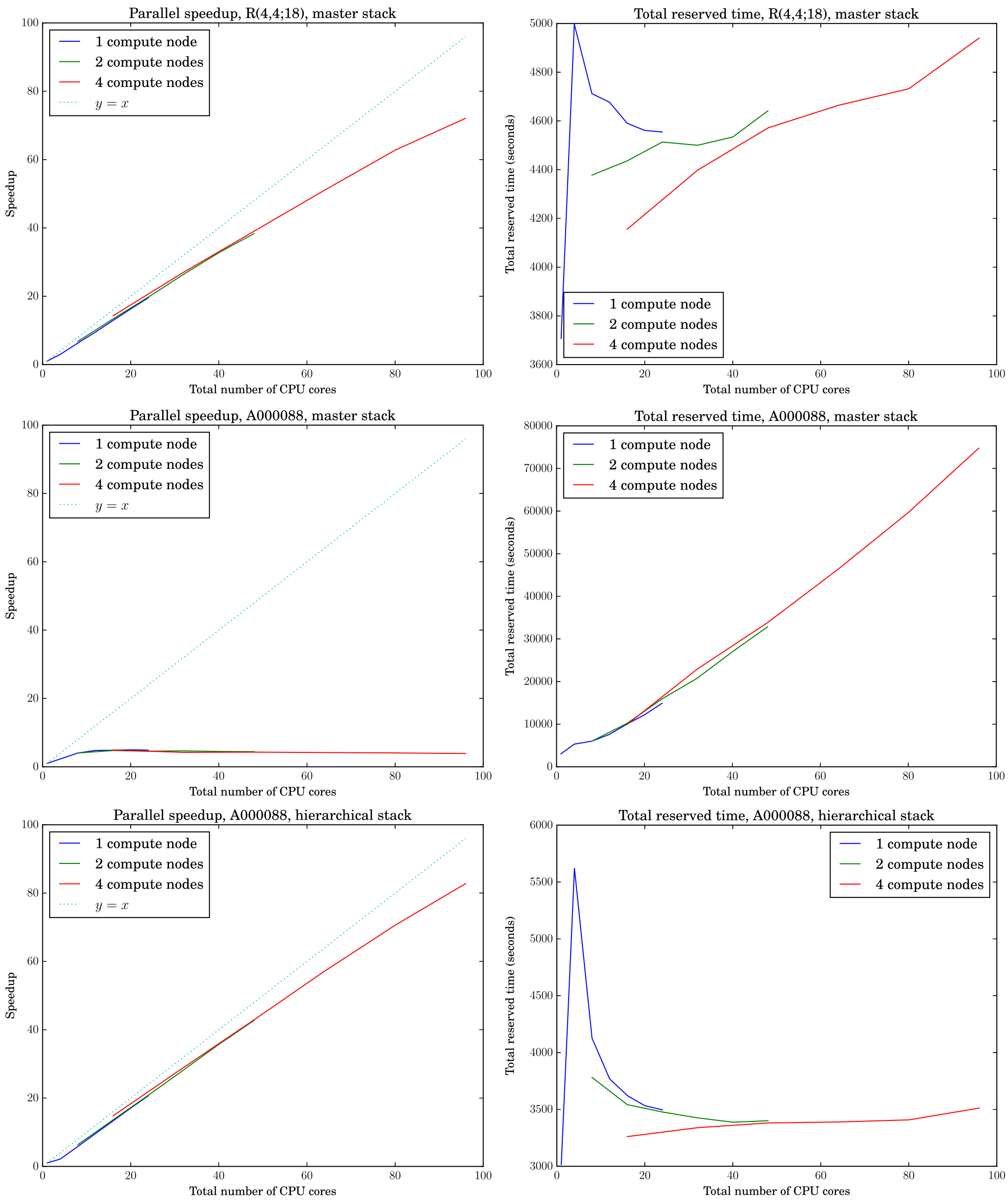


Fig. 1. The plots on the left display the parallel speedup factor obtained with increasing number of cores. The plots on the right display the total reserved time in seconds of all the cores with increasing number of cores. The first row shows the instance $R(4, 4; 18)$, with parallelization executed using the master stack mode of communication. We observe solid parallel speedup with increasing number of cores. The second row shows the instance A000088 using the master stack mode of communication. Here the speedup is unsatisfactory due to extensive accesses to the master stack caused by a wide search tree with many easy instances of canonical labeling (cf. Table 2). This bottleneck can be alleviated by using the hierarchical stack mode. The third row shows the instance A000088 executed in the hierarchical stack mode. Now we observe solid parallel speedup with increasing number of cores. The peaks in the total reserved time for a small number of cores are caused by an unbalanced work allocation between the cores that eases with increasing number of cores; see Section 8.4.

Table 1
The number of canonical partial assignments in the instance $R(4, 4; 18)$ at different levels of the search tree explored by `reduce`.

level	assignments	level	assignments	level	assignments
1	2	12	13	23	1848
2	3	13	14	24	2400
3	4	14	15	25	2970
4	5	15	16	26	3520
5	6	16	17	27	4004
6	7	17	18	28	4368
7	8	18	96	29	4550
8	9	19	300	30	4480
9	10	20	560	31	4080
10	11	21	910	32	3264
11	12	22	1344	33	1050

Table 2
The number of canonical partial assignments in the instance A000088 at different levels of the search tree explored by `reduce`. We observe that the search tree is considerably wider at the intermediate levels compared with the last level.

level	assignments	level	assignments	level	assignments
1	2	13	336	25	346376
2	3	14	336	26	47418
3	4	15	140	27	644016
4	5	16	1216	28	3256288
5	6	17	5256	29	4336496
6	7	18	9936	30	508140
7	8	19	13664	31	5245032
8	9	20	13104	32	19768096
9	42	21	2676	33	2409488
10	120	22	34500	34	13814848
11	200	23	183120	35	4147832
12	280	24	328032	36	274668

8.5. Experiments comparing with other tools

We compared our present tool `reduce` against the tool `breakid` (Devriendt et al., 2016). Since `breakid` does not parallelize, no parallelization was used in these experiments and all experiments were executed using a single compute core. All running times displayed in the tables that follow are in seconds, with "t/o" indicating a time-out of 25 hours of wall-clock time. Other compute load was in general present on the compute nodes where these experiments were run.

Table 3 shows the results of a tensor rank computation modulo 2 for two random tensors T with $m = 5$, $n = 9$ and $m = 5$, $n = 20$ with (top table) and without (bottom table) an auxiliary graph. When $m = 5$ and $n = 9$, the tensor has rank 8 and decompositions for rank 7 and 8 are sought. When $m = 5$ and $n = 20$, the tensor has rank 9 and decompositions of rank 8 and 9 are sought. For both tensors we observe decreased running time due to symmetry reduction. Comparing the top and bottom tables, we observe the relevance of the graph representation of the symmetries in (14), which are not easily discoverable from the compiled CNF. As the auxiliary graph, we used the graph representation of the system (14), constructed as in Example 17.

Table 4 shows the results of applying `breakid` and our tool `reduce` as preprocessors for solving instances of the Clique Coloring Problem. We observe that for sufficiently large instances, our tool is faster than `breakid` in the combined runtime of preprocessor and solver.

Table 5 compares running times of `reduce` on instances of the Clique Coloring Problem (i) using the graph automatically constructed from CNF, and (ii) using a tailored auxiliary graph constructed as described in Section 8.1. For these instances, the available symmetry can be easily discovered directly

Table 3
Comparing different tools for preprocessing and then solving instances with hard symmetry not easily discoverable from a compiled CNF encoding. Here the instances ask for modulo-2 tensor decompositions for two 5 × 5 × 5 tensors with (top) and without (bottom) an auxiliary graph. We observe that the auxiliary graph gives a marked improvement in the running times of preprocessing, making all the instances tractable. Without the auxiliary graph to highlight the symmetry, the two unsatisfiable instances are intractable within the timeout threshold of 90,000 seconds. All running times are in seconds.

with auxiliary graph

m	r	n	raw		prep. breakid	breakid		prep. reduce	reduce			Sat?
			glucose	lingeling		glucose	lingeling		glucose	lingeling	ilingeling	
5	7	9	t/o	t/o	0.28	30.07	30.73	87.35	77.87	66.40	47.67	No
5	8	9	0.36	3.91	0.63	0.33	5.61	290.70	1.69	13.93	0.50	Yes
5	8	20	t/o	t/o	0.61	1078.54	1273.49	290.78	2641.97	7699.27	885.01	No
5	9	20	1.44	1.28	1.68	72.09	26.33	881.85	228.06	371.09	40.57	Yes

without auxiliary graph

m	r	n	raw		prep. breakid	breakid		prep. reduce	reduce			Sat?
			glucose	lingeling		glucose	lingeling		glucose	lingeling	ilingeling	
5	7	9	t/o	t/o	0.64	t/o	t/o	t/o	n/a	n/a	n/a	No
5	8	9	0.36	3.91	0.60	0.40	1.22	t/o	n/a	n/a	n/a	Yes
5	8	20	t/o	t/o	0.32	t/o	t/o	t/o	n/a	n/a	n/a	No
5	9	20	1.44	1.28	1.59	3.30	15.10	t/o	n/a	n/a	n/a	Yes

Table 4
Comparing different tools for preprocessing and then solving instances with hard symmetry. Here the instances ask for solutions to the Clique Coloring Problem with parameters indicated on the left. On this family of instances, our tool `reduce` is faster than the tool `breakid` for sufficiently large parameters. All instances are unsatisfiable. All running times are in seconds. A timeout threshold of 90,000 seconds was applied.

n	s	t	raw		prep. breakid	breakid		prep. reduce	reduce		
			glucose	lingeling		glucose	lingeling		glucose	lingeling	ilingeling
15	5	4	722.32	811.59	2.82	126.35	154.27	19.34	19.82	31.61	32.92
16	5	4	1038.88	1839.05	10.68	238.52	570.19	8.07	25.62	39.55	38.31
17	5	4	4481.54	8865.19	1.94	597.35	498.20	12.81	105.35	57.96	54.66
18	5	4	2709.96	4762.23	9.96	559.86	460.70	14.11	40.68	74.71	66.19
19	5	4	6701.65	6819.77	10.66	586.81	651.10	19.13	107.73	106.62	85.38
20	5	4	8901.20	7777.35	1.15	1294.31	1579.77	25.37	157.91	134.55	248.56
12	6	5	38835.75	15517.28	9.87	1602.96	745.83	2.42	1190.58	677.38	751.27
13	6	5	26017.87	50312.82	9.91	7032.11	3506.61	9.68	1439.84	1440.30	1420.10
14	6	5	t/o	t/o	2.28	8417.69	5384.79	17.18	2360.88	5559.26	2543.99
15	6	5	t/o	t/o	9.05	10537.53	7316.61	6.49	3504.82	4104.10	4140.57
16	6	5	t/o	t/o	9.46	41355.16	27699.48	30.52	6858.69	5612.36	5708.18
17	6	5	t/o	t/o	0.94	t/o	t/o	11.48	11329.16	20597.16	10460.73
18	6	5	t/o	t/o	5.34	t/o	t/o	23.81	17347.43	52703.19	16873.36
19	6	5	t/o	t/o	7.18	t/o	t/o	82.93	29689.84	19969.09	21195.04
20	6	5	t/o	t/o	3.38	t/o	t/o	29.36	76600.29	35850.94	27035.80
21	6	5	t/o	t/o	1.50	t/o	t/o	148.64	t/o	45963.02	48542.48
22	6	5	t/o	t/o	1.74	t/o	t/o	198.15	t/o	61414.88	66279.68
23	6	5	t/o	t/o	1.91	t/o	t/o	267.67	t/o	t/o	78463.34

from the CNF encoding, but we observe that the use of the tailored auxiliary graph does result in faster preprocessing times for `reduce`.

8.6. Sizes of automorphism groups and residual symmetry

To study the sizes of automorphism groups and residual symmetry after symmetry reduction, Table 6 lists the runtimes for some well-known instances, as well as the size $|\mathrm{Aut}(U_0)|$ of the initial automorphism group and the size $|\mathrm{Aut}(U_k)|$ of the setwise stabilizer of the prefix variables U_k. The

Table 5
The effect of a tailored auxiliary graph on the running time of `reduce` when applied on instances of the Clique Coloring Problem with parameters indicated on the left. We see that there is a marked decrease in the running times when the auxiliary graph is available. All running times are in seconds.

n	s	t	reduce	
			with graph	without graph
15	5	4	19.34	399.25
16	5	4	8.07	158.47
17	5	4	12.81	223.24
18	5	4	14.11	1151.05
19	5	4	19.13	1629.34
20	5	4	25.37	586.78
12	6	5	2.42	139.14
13	6	5	9.68	61.46
14	6	5	17.18	381.85
15	6	5	6.49	587.87
16	6	5	30.52	881.09
17	6	5	11.48	322.53
18	6	5	23.81	450.28
19	6	5	82.93	2388.18
20	6	5	29.36	851.68
21	6	5	148.64	1195.23
22	6	5	198.15	1543.51
23	6	5	267.67	5545.20

Table 6
Instances with large automorphism groups. The column `raw` shows the solver runtime without preprocessing. The columns `breakid` and `reduce` display the runtime of the preprocessor and the solver after augmenting the CNF with a symmetry-breaking predicate produced by the preprocessor. All instances are unsatisfiable. The column k shows the length of the prefix used by `reduce`, and the column "cubes" lists the number of partial assignments generated by `reduce`—in each case, we observe effective symmetry reduction, with the number of generated partial assignments substantially less than $|R|^{|U_k|} = 2^k$. The column $|\mathrm{Aut}(U_0)|$ shows the size of the initial automorphism group of the instance and the column $|\mathrm{Aut}(U_k)|$ displays the size of the setwise stabilizer of the prefix U_k. All running times are in seconds. A timeout threshold of 4 hours was applied.

instance	raw glucose	breakid		reduce		k	cubes	$\|\mathrm{Aut}(U_0)\|$	$\|\mathrm{Aut}(U_k)\|$
		prep.	glucose	prep.	glucose				
STS(16)	25.83	0.77	17.17	17.89	1.67	14	15	20922789888000	174356582400
STS(18)	1506.65	0.86	724.57	41.48	25.43	16	17	6402373705728000	41845579776000
STS(20)	t/o	0.78	t/o	90.64	1745.17	18	19	2432902008176640000	12804747411456000
PHP(11)	116.84	0.30	3.18	2.00	5.46	10	11	144850083840000	13168189440000
PHP(12)	t/o	0.42	27.51	111.91	7.56	22	203	19120211066880000	289700167680000
$R(4, 4; 18)$	t/o	0.73	1330.23	16.23	1.47	17	18	6402373705728000	355687428096000

table also lists the length k of the prefix sequence and the number of isomorphism classes of partial assignments (cubes) generated. For all instances considered, we observe that the number cubes generated is substantially less than the total number of cubes $|R|^{|U_k|} = 2^k$ without symmetry reduction; that is, the action of the large setwise stabilizer $\mathrm{Aut}(U_k)$ gives rise to only a few large orbits in R^{U_k}, as is desirable from the perspective of obtaining effective symmetry reduction. In general, there is a tradeoff in efficiency between symmetry reduction and the subsequent solver running time on the symmetry-reduced instances. For example, in Table 6, had a prefix of length only 11 been chosen for PHP(12), the time taken by the symmetry reduction and the solver in total would have remained almost the same, but considerably more time would have been spent by the solver. Although our tool `reduce` compares favorably with `breakid` in the STS and Ramsey instances, we observe that

Table 7
Residual symmetry after symmetry reduction. Each of the four sub-tables below illustrates the residual symmetry in the generated partial assignments arising from a particular problem instance. For each size $|\mathrm{Aut}(X)|$ of the automorphism group, we display the number of generated partial assignments X with an automorphism group of the indicated size. The length of the prefix sequence is indicated by k. For the PHP(12) instance, two prefixes of lengths $k = 11$ and $k = 22$ are shown to illustrate the tradeoff between prefix length and residual symmetry. A similar tradeoff can be witnessed for the $R(4, 4; 18)$ instance with prefixes of lengths $k = 17$ and $k = 33$, with the latter prefix postponed to Table 8 due to lack of space. The running times of `reduce` were as follows: 12.24 seconds for PHP(12) with $k = 11$, 111.91 seconds for PHP(12) with $k = 22$, 90.64 seconds for $STS(20)$ with $k = 18$, and 16.23 seconds for $R(4, 4; 18)$ with $k = 17$. In particular, we observe decreased residual symmetry with longer prefixes, at the cost of increased running time for symmetry reduction.

$R(4, 4; 18)$, $k = 17$	
$\lvert\mathrm{Aut}(X)\rvert$	#
14631321600	2
18289152000	2
28740096000	2
57480192000	2
149448499200	2
523069747200	2
2615348736000	2
20922789888000	2
355687428096000	2
Total	18

STS(20), $k = 18$	
$\lvert\mathrm{Aut}(X)\rvert$	#
263363788800	1
292626432000	2
402361344000	2
689762304000	2
1494484992000	2
4184557977600	2
15692092416000	2
83691159552000	2
711374856192000	2
12804747411456000	2
Total	19

PHP(12), $k = 11$	
$\lvert\mathrm{Aut}(X)\rvert$	#
3448811520000	2
4828336128000	2
9656672256000	2
28970016768000	2
144850083840000	2
1593350922240000	2
Total	12

PHP(12), $k = 22$	
$\lvert\mathrm{Aut}(X)\rvert$	#
1567641600	1
2090188800	5
3135283200	7
3483648000	1
4180377600	7
5225472000	12
6270566400	2
6967296000	2
8360755200	2
10450944000	10
12541132800	5
15676416000	10
20901888000	16
25082265600	2
31352832000	16
36578304000	5
52254720000	5
62705664000	12
73156608000	7
104509440000	2
109734912000	12
146313216000	3
292626432000	14
313528320000	5
438939648000	5
627056640000	2
877879296000	7
1316818944000	5
1755758592000	2
2633637888000	7
5267275776000	2
13168189440000	5
26336378880000	2
144850083840000	1
289700167680000	2
Total	203

`breakid` is quite efficient in the case of PHP instances; nevertheless, our tool can still handle these instances as well.

Tables 7 and 8 give a more fine-grained view into the residual symmetry still present in the instances after preprocessing with `reduce`. In more precise terms, for each generated partial assignment X, the tables list the size of the automorphism group $|\mathrm{Aut}(X)|$. We observe that the residual symmetry in general depends on the values X assigns to the variables in U_k. For example, when the partial assignment assigns all variables in U_k to the same value, we have $\mathrm{Aut}(X) = \mathrm{Aut}(U_k)$; while such partial assignments have substantial residual symmetry, one can expect that the resulting instances are easy to solve. We again observe a tradeoff in the choice of prefix, as illustrated by the two chosen prefixes for the PHP(12) and $R(4, 4; 18)$ instances: using a longer prefix yields less residual

Table 8
This table complements Table 8 by tabulating residual symmetry for $R(4,4;18)$ at prefix length $k = 33$. Since there are in total 1050 generated partial assignments, the results have been divided into four columns. The running time for `reduce` was 4153.15 seconds.

$R(4,4;18)$, $k=33$							
$\lvert\mathrm{Aut}(X)\rvert$	#	$\lvert\mathrm{Aut}(X)\rvert$	#	$\lvert\mathrm{Aut}(X)\rvert$	#	$\lvert\mathrm{Aut}(X)\rvert$	#
414720	10	4147200	4	46448640	4	3251404800	4
518400	4	4838400	4	50803200	14	3657830400	4
622080	10	5806080	32	52254720	24	3832012800	4
663552	2	6220800	4	58060800	32	4790016000	10
691200	10	7257600	24	87091200	38	5225472000	4
829440	14	7741440	4	101606400	4	6227020800	2
1036800	28	8709120	10	130636800	10	9580032000	4
1088640	2	9676800	24	159667200	10	11496038400	10
1244160	4	10368000	10	174182400	28	12454041600	28
1382400	4	12441600	10	203212800	24	22992076800	4
1451520	24	13063680	10	239500800	10	37362124800	10
1658880	4	14515200	24	261273600	28	74724249600	4
1728000	2	17418240	28	319334400	4	87178291200	10
2073600	28	20736000	4	435456000	24	174356582400	14
2177280	4	21772800	24	479001600	28	348713164800	4
2419200	10	23224320	10	958003200	34	1307674368000	10
2903040	20	24883200	4	1625702400	4	2615348736000	4
3110400	10	25401600	4	1828915200	10	20922789888000	2
3456000	4	26127360	4	1916006400	14	41845579776000	4
3628800	24	29030400	36	2612736000	10		
3870720	10	43545600	34	2874009600	24		

symmetry, but it may be unnecessary to go all the way, since the time spent for symmetry reduction could be better spent by the solver.

Acknowledgements

The research leading to these results has received funding from the European Research Council under the European Union's Seventh Framework Programme (FP/2007-2013) / ERC Grant Agreement 338077 "Theory and Practice of Advanced Search and Enumeration" (M.K., P.K., and J.K.). We gratefully acknowledge the use of computational resources provided by the Aalto Science-IT project at Aalto University. We thank Tomi Janhunen and Bart Bogaerts for useful discussions, and the anonymous reviewers for their comments that helped to improve the exposition in this paper.

A preliminary conference abstract of this paper appeared in T. Junttila, M. Karppa, P. Kaski, and J. Kohonen, "An Adaptive Prefix-Assignment Technique for Symmetry Reduction", in *Theory and Applications of Satisfiability Testing — SAT 2017*, Lecture Notes in Computer Science, Vol. 10491, Springer, 2017, pp. 101–118.

References

Alekseev, V.B., 2014. On bilinear complexity of multiplication of 5×2 matrix by 2×2 matrix. In: Physics and Mathematics. In: Uchenye Zapiski Kazanskogo Universiteta. Seriya Fiziko-Matematicheskie Nauki, vol. 156. Kazan University, Kazan, pp. 19–29.
Alekseev, V.B., 2015. On bilinear complexity of multiplication of $m \times 2$ and 2×2 matrices. Chebyshevskiĭ Sb. 16 (4), 11–27.
Alekseev, V.B., Smirnov, A.V., 2013. On the exact and approximate bilinear complexities of multiplication of 4×2 and 2×2 matrices. Proc. Steklov Inst. Math. 282 (1), 123–139.
Alekseyev, V.B., 1985. On the complexity of some algorithms of matrix multiplication. J. Algorithms 6 (1), 71–85.
Aloul, F.A., Sakallah, K.A., Markov, I.L., 2003. Efficient symmetry breaking for Boolean satisfiability. In: Proc. IJCAI 2003. Morgan Kaufmann, pp. 271–276.
Audemard, G., Simon, L., 2016. Extreme cases in SAT problems. In: Proc. SAT 2016. In: Lecture Notes in Computer Science, vol. 9710. Springer, pp. 87–103.
Babai, L., 2016. Graph isomorphism in quasipolynomial time. In: Wichs, D., Mansour, Y. (Eds.), Proceedings of the 48th Annual ACM SIGACT Symposium on Theory of Computing. STOC 2016. ACM, pp. 684–697. http://dl.acm.org/citation.cfm?id=2897518.
Benhamou, B., Nabhani, T., Ostrowski, R., Saıdi, M.R., 2010a. Dynamic symmetry breaking in the satisfiability problem. In: Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning. LPAR-16.

Benhamou, B., Nabhani, T., Ostrowski, R., Saidi, M.R., 2010b. Enhancing clause learning by symmetry in SAT solvers. In: Proceedings of the 22nd IEEE International Conference on Tools with Artificial Intelligence, vol. 1. ICTAI 2010, pp. 329–335.

Benhamou, B., Sais, L., 1994. Tractability through symmetries in propositional calculus. J. Autom. Reason. 12 (1), 89–102.

Benson, A.R., Ballard, G., 2015. A framework for practical parallel fast matrix multiplication. In: Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. PPoPP 2015, pp. 42–53.

Biere, A., 2016. Splatz, lingeling, plingeling, treengeling, YalSAT entering the SAT competition 2016. In: Proceedings of SAT Competition 2016: Solver and Benchmark Descriptions, vol. B-2016-1. Department of Computer Science Series of Publications, University of Helsinki, pp. 44–45.

Bläser, M., 1999. Lower bounds for the multiplicative complexity of matrix multiplication. Comput. Complex. 8 (3), 203–226.

Bläser, M., 2003. On the complexity of the multiplication of matrices of small formats. J. Complex. 19 (1), 43–60.

Butler, G., 1991. Fundamental Algorithms for Permutation Groups. Lecture Notes in Computer Science, vol. 559. Springer.

Chu, G., de la Banda, M.G., Mears, C., Stuckey, P.J., 2014. Symmetries, almost symmetries, and lazy clause generation. Constraints 19 (4), 434–462.

Codish, M., Gange, G., Itzhakov, A., Stuckey, P.J., 2016. Breaking symmetries in graphs: the nauty way. In: Proc. CP 2016. In: Lecture Notes in Computer Science, vol. 9892. Springer, pp. 157–172.

Courtois, N.T., Hulme, D., Mourouzis, T., 2012. Multiplicative complexity and solving generalized Brent equations with SAT solvers. In: Proceedings of the Third International Conference on Computational Logics, Algebras, Programming, Tools, and Benchmarking. COMPUTATION TOOLS 2012, pp. 22–27.

Crawford, J.M., Ginsberg, M.L., Luks, E.M., Roy, A., 1996. Symmetry-breaking predicates for search problems. In: Proc. KR. 1996. Morgan Kaufmann, pp. 148–159.

Darga, P.T., Liffiton, M.H., Sakallah, K.A., Markov, I.L., 2004. Exploiting structure in symmetry detection for CNF. In: Proc. DAC 2004. ACM, pp. 530–534.

Devriendt, J., Bogaerts, B., Bruynooghe, M., 2017. Symmetric explanation learning: effective dynamic symmetry handling for SAT. In: Theory and Applications of Satisfiability Testing. SAT 2017. Springer International Publishing, pp. 83–100.

Devriendt, J., Bogaerts, B., Bruynooghe, M., Denecker, M., 2016. Improved static symmetry breaking for SAT. In: Proc. SAT 2016. In: Lecture Notes in Computer Science, vol. 9710. Springer, pp. 104–122.

Devriendt, J., Bogaerts, B., Cat, B.D., Denecker, M., Mears, C., 2012. Symmetry propagation: improved dynamic symmetry breaking in SAT, vol. 1. In: IEEE 24th International Conference on Tools with Artificial Intelligence. ICTAI 2012, pp. 49–56.

Dinan, J., Larkins, D.B., Sadayappan, P., Krishnamoorthy, S., Nieplocha, J., 2009. Scalable work stealing. In: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, pp. 1–11.

Dixon, J.D., Mortimer, B., 1996. Permutation Groups. Graduate Texts in Mathematics, vol. 163. Springer.

Faradžev, I.A., 1978. Constructive enumeration of combinatorial objects. In: Problèmes Combinatoires et Théorie des Graphes. In: Colloq. Internat. CNRS, vol. 260. CNRS, pp. 131–135.

Gabriel, E., Fagg, G.E., Bosilca, G., Angskun, T., Dongarra, J.J., Squyres, J.M., Sahay, V., Kambadur, P., Barrett, B., Lumsdaine, A., Castain, R.H., Daniel, D.J., Graham, R.L., Woodall, T.S., 2004. Open MPI: goals, concept, and design of a next generation MPI implementation. In: Proceedings, 11th European PVM/MPI Users' Group Meeting, pp. 97–104.

Gent, I.P., Petrie, K.E., Puget, J., 2006. Symmetry in constraint programming. In: Handbook of Constraint Programming. Vol. 2 of Foundations of Artificial Intelligence. Elsevier, pp. 329–376.

Graham, R.L., Rothschild, B.L., Spencer, J.H., 1990. Ramsey Theory. John Wiley & Sons.

Grayland, A., Jefferson, C., Miguel, I., Roney-Dougal, C.M., 2009. Minimal ordering constraints for some families of variable symmetries. Ann. Math. Artif. Intell. 57 (1), 75–102.

Grohe, M., Neuen, D., Schweitzer, P., Wiebking, D., 2018. An improved isomorphism test for bounded-tree-width graphs. CoRR arXiv:1803.06858.

Gropp, W., Lusk, E., Skjellum, A., 2014. Using MPI: Portable Parallel Programming with the Message-Passing Interface. The MIT Press.

Hall Jr., M., Knuth, D.E., 1965. Combinatorial analysis and computers. Am. Math. Mon. 72 (2, part 2), 21–28.

Håstad, J., 1990. Tensor rank is NP-complete. J. Algorithms 11 (4), 644–654.

Heule, M., Kullmann, O., Wieringa, S., Biere, A., 2011. Cube and conquer: guiding CDCL SAT solvers by lookaheads. In: Proc. HVC 2011. In: Lecture Notes in Computer Science, vol. 7261. Springer, pp. 50–65.

Heule, M.J.H., 2016. The quest for perfect and compact symmetry breaking for graph problems. In: Proc. SYNASC 2016. IEEE Computer Society, pp. 149–156.

Hopcroft, J.E., Kerr, L.R., 1971. On minimizing the number of multiplications necessary for matrix multiplication. SIAM J. Appl. Math. 20 (1), 30–36.

Huang, J., Rice, L., Matthews, D.A., van de Geijn, R.A., 2017. Generating families of practical fast matrix multiplication algorithms. In: 2017 IEEE International Parallel and Distributed Processing Symposium. IPDPS 2017, pp. 656–667.

Humphreys, J.F., 1996. A Course in Group Theory. Oxford University Press, Oxford.

Itzhakov, A., Codish, M., 2016. Breaking symmetries in graph search with canonizing sets. Constraints 21 (3), 357–374.

Jefferson, C., Petrie, K.E., 2011. Automatic generation of constraints for partial symmetry breaking. In: 17th International Conference Principles and Practice of Constraint Programming. CP 2011. In: Lecture Notes in Computer Science, vol. 6876. Springer, pp. 729–743.

Junttila, T., Kaski, P., 2007. Engineering an efficient canonical labeling tool for large and sparse graphs. In: Proc. ALENEX 2007. SIAM.

Karppa, M., 2018. Reduce. GitHub repository. https://github.com/mkarppa/reduce.

Kaski, P., Östergård, P.R.J., 2006. Classification Algorithms for Codes and Designs. Algorithms and Computation in Mathematics, vol. 15. Springer-Verlag.

Kerber, A., 1999. Applied Finite Group Actions, 2nd edition. Algorithms and Combinatorics, vol. 19. Springer.

Kerber, A., Laue, R., 1998. Group actions, double cosets, and homomorphisms: unifying concepts for the constructive theory of discrete structures. Acta Appl. Math. 52 (1–3), 63–90.
Knuth, D.E., 2011. The Art of Computer Programming. Vol. 4A. Combinatorial Algorithms. Part 1. Addison-Wesley.
Laderman, J.D., 1976. A noncommutative algorithm for multiplying 3×3 matrices using 23 multiplications. Bull. Am. Math. Soc. 82, 126–128.
Leon, J.S., 1991. Permutation group algorithms based on partitions, I: theory and algorithms. J. Symb. Comput. 12 (4–5), 533–583.
Leon, J.S., 1997. Partitions, refinements, and permutation group computation, II. In: Groups and Computation. In: DIMACS Series in Discrete Mathematics and Theoretical Computer Science, vol. 28. American Mathematical Society, pp. 123–158.
Lokshtanov, D., Pilipczuk, M., Pilipczuk, M., Saurabh, S., 2017. Fixed-parameter tractable canonization and isomorphism test for graphs of bounded treewidth. SIAM J. Comput. 46 (1), 161–189.
Manthey, N., 2014. Generating clique coloring problem formulas. In: Proceedings of SAT Competition 2014: Solver and Benchmark Descriptions, vol. B-2014-2. Department of Computer Science Series of Publications B, University of Helsinki, p. 89.
McKay, B.D., 1981. Practical graph isomorphism. Congr. Numer. 30, 45–87.
McKay, B.D., 1998. Isomorph-free exhaustive generation. J. Algorithms 26 (2), 306–324.
McKay, B.D., Piperno, A., 2014. Practical graph isomorphism, II. J. Symb. Comput. 60, 94–112.
Message Passing Interface Forum, 2015. MPI: a message-passing interface standard. Version 3.1. Standard specification.
Pacheco, P., 1997. Parallel Programming with MPI. Morgan Kaufmann.
Pezzi, G.P., Cera, M.C., Mathias, E., Maillard, N., Navaux, P.O.A., 2007. On-line scheduling of MPI-2 programs with hierarchical work stealing. In: 19th International Symposium on Computer Architecture and High Performance Computing. SBAC-PAD 2007, pp. 247–254.
Read, R.C., 1978. Every one a winner; or, how to avoid isomorphism search when cataloguing combinatorial configurations. Ann. Discrete Math. 2, 107–120.
Sabharwal, A., 2009. Symchaff: exploiting symmetry in a structure-aware satisfiability solver. Constraints 14 (4), 478–505.
Sakallah, K.A., 2009. Symmetry and satisfiability. In: Handbook of Satisfiability. In: Frontiers in Artificial Intelligence and Applications, vol. 185. IOS Press, pp. 289–338.
Schaafsma, B., Heule, M.J.H., van Maaren, H., 2009. Dynamic symmetry breaking by simulating Zykov contraction. In: Theory and Applications of Satisfiability Testing. SAT 2009. Springer, pp. 223–236.
Seress, Á., 2003. Permutation Group Algorithms. Cambridge University Press, Cambridge.
Smirnov, A.V., 2013. The bilinear complexity and practical algorithms for matrix multiplication. Comput. Math. Math. Phys. 53 (12), 1781–1795.
Strassen, V., 1969. Gaussian elimination is not optimal. Numer. Math. 13 (4), 354–356.
Swift, J.D., 1960. Isomorph rejection in exhaustive search techniques. In: Combinatorial Analysis. American Mathematical Society, pp. 195–200.
Wieringa, S., 2011. The iCNF file format. http://www.siert.nl/icnf/. (Accessed April 2017).
Winograd, S., 1971. On multiplication of 2×2 matrices. Linear Algebra Appl. 4 (4), 381–388.