Efficient Propagation Techniques for Handling Cyclic Symmetries in Binary Programs

Citation for published version (APA):
van Doornmalen, M. J., & Hojny, C. (2024). Efficient Propagation Techniques for Handling Cyclic Symmetries in Binary Programs. INFORMS Journal on Computing, 36(3), 868-883. https://doi.org/10.1287/ijoc.2022.0060

Document license:
TAVERNE

DOI:
10.1287/ijoc.2022.0060

Document status and date:
Published: 01/05/2024

Document Version:
Publisher’s PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:

- A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher’s website.
- The final author version and the galley proof are versions of the publication after peer review.
- The final published version features the final layout of the paper including the volume, issue and page numbers.

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal.

If the publication is distributed under the terms of Article 25fa of the Dutch Copyright Act, indicated by the “Taverne” license above, please follow below link for the End User Agreement:
www.tue.nl/taverne

Take down policy
If you believe that this document breaches copyright please contact us at:
openaccess@tue.nl
providing details and we will investigate your claim.
Efficient Propagation Techniques for Handling Cyclic Symmetries in Binary Programs

Jasper van Doornmalen, Christopher Hojny

Abstract. The presence of symmetries in binary programs typically degrades the performance of branch-and-bound solvers. In this article, we derive efficient variable fixing algorithms to discard symmetric solutions from the search space based on propagation techniques for cyclic groups. Our algorithms come with the guarantee to find all possible variable fixings that can be derived from symmetry arguments; that is, one cannot find more variable fixings than those found by our algorithms. Because every permutation symmetry group of a binary program has cyclic subgroups, the derived algorithms can be used to handle symmetries in any symmetric binary program. In experiments, we also provide numerical evidence that our algorithms handle symmetries more efficiently than other variable fixing algorithms for cyclic symmetries.

Keywords: symmetry handling • cyclic group • propagation • branch-and-bound

1. Introduction

We consider binary programs \(\max\{c^T x : Ax \leq b, x \in \{0,1\}^n\}\), with \(A \in \mathbb{R}^{m \times n}\), \(b \in \mathbb{R}^m\), and \(c \in \mathbb{R}^n\) for some positive integers \(m\) and \(n\). A standard method to solve binary programs is branch-and-bound (B&B), which iteratively explores the search space by splitting the initial binary program into subproblems; see Land and Doig (1960). Although B&B can solve binary programs with thousands of variables and constraints rather efficiently, the performance of B&B usually degrades drastically if symmetries are present, as it unnecessarily explores symmetric subproblems. Such a symmetry is a permutation \(\gamma\) of \([n] := \{1, \ldots, n\}\) that acts on a vector \(x \in \mathbb{R}^n\) by permuting its coordinates, that is, \(\gamma(x) := (x_{\gamma^{-1}(1)}, \ldots, x_{\gamma^{-1}(n)})\), and that adheres to the following two properties: (S1) it preserves feasibility, that is, \(Ax \leq b\) if and only if \(A\gamma(x) \leq b\), and (S2) it preserves the objective value, that is, \(c^T x = c^T \gamma(x)\). Two solutions \(x\) and \(y\) are symmetric if there exists a symmetry \(\gamma\) such that \(y = \gamma(x)\).

Various methods to remove symmetric parts from the search space have been proposed, for example, cutting planes, variable fixing or branching rules, and propagation methods; see below for references. The idea of many propagation algorithms is to enforce a lexicographic order on the solution space by excluding solutions that cannot be lexicographically maximal in their symmetry class. This is achieved as follows. In a subproblem, some variables might have been fixed, for example, because of branching decisions. Propagation algorithms look for further variables that need to be fixed to ensure that a solution \(x\) is not lexicographically smaller than its permutation \(\gamma(x)\). Usually propagation algorithms are either applied to individual permutations or to an entire group. The latter becomes important as groups may contain exponentially many permutations. Thus, propagating each individual permutation is, in general, computationally intractable. But even if the groups are small, propagating individual permutations rather than groups might find fewer fixings, as we will see in Section 3.

For permutation groups, the focus has been mostly on handling symmetries of \(S_n\), the symmetric group on \(n\) elements; see Bendotti et al. (2021) and Kaibel et al. (2011) for polynomial time propagation algorithms. One argument to focus on symmetric groups is that they arise in many applications. For instance, Pfetsch and Rehn (2019) investigated a test set of 194 instances from the Mixed-Integer Programming Library (MIPLIB) 2010 (see Koch et al. 2011); the majority of instances contain actions of \(S_n\). However, there exist also 101 instances whose symmetry groups contain actions not associated with a symmetric group. In only a few cases, Pfetsch and Rehn (2019) were able to classify these group actions; but in most cases, the group action is unknown. We believe that this is an important gap, as these unknown groups cannot be handled efficiently at the moment. For unknown
groups, usually only a set of generators $\Pi$ is known. To shed light on the structure of unknown groups, we propose to consider the cyclic groups being generated by the individual permutations in $\Pi$. This way, we can derive reductions based on subgroups rather than individual permutations, which has the potential to find way more symmetry-based variable fixings. In this article, we therefore develop efficient propagation algorithms for cyclic groups that come with the guarantee to find all possible symmetry-based variable fixings. To the best of our knowledge, this has not been done before. In particular, we are not aware of how the analysis of our guarantee can be carried out for state-of-the-art methods such as isomorphism pruning.

At first glance, finding algorithms for cyclic groups seems trivial, as cyclic shifts are very simple. But despite their simplicity, we have no understanding of the structure of binary points being lexicographically maximal with respect to (w.r.t.) cyclic group actions. In fact, the structure of these points is rather complicated and does not seem to follow an obvious pattern (Loos 2010, chapter 3.2.2). It has been open for at least 10 years to gain further insights into the structure of lexicographically maximal points for cyclic groups. We emphasize that although cyclic groups are generated by a single permutation $\gamma$, they might have superpolynomial size. That is, efficient algorithms for cyclic groups are not immediate.

Before we conclude this section with a literature review, we provide an outline of this article. In Section 2, we discuss our contribution by describing the framework of our propagation algorithms and comparing it with existing algorithms. Afterward, we present our propagation algorithms in Section 3. These algorithms are based on a novel characterization of lexicographically maximal elements w.r.t. certain cyclic groups. For these groups, we show that our algorithms find all possible symmetry-based variable fixings; that is, they are as strong as possible. Moreover, we provide a running time analysis of our algorithms. This analysis requires an algorithm to propagate all permutations in a set $\Pi \subseteq S_n$ in $O(n|\Pi|)$ time. This algorithm is outlined in Section 4 and detailed in the online appendix. We report on numerical experiments in Section 5. These experiments are based on instances from graph coloring problems in graphs with dominant cyclic symmetries, as well as general benchmark test sets. Compared with a comparable state-of-the-art method, our best results report an improvement of $12.0\%$ for the difficult-to-solve well-structured instances, and $6.5\%$ for the solvable general benchmark instances that our methods can be applied to. Finally, we conclude this paper in Section 6.

1.1. Literature Review

Handling symmetries in binary programs via propagation originates from constraint programming, where it is discussed, for example, by Crawford et al. (1996), Katsirelos et al. (2009, 2010), and Narodytska and Walsh (2013). For binary programs, Bendotti et al. (2021) describe a variable fixing algorithm for certain actions of symmetric groups. Further fixings can be found if the variables affected by the symmetric group are contained in set packing or partitioning constraints; see Kaibel et al. (2011). If, instead of an entire group, only a single permutation is considered, propagation algorithms that enforce that the solution vector is lexicographically not smaller than its permuted counterpart exist; see Bestuzheva et al. (2021) and Hojny and Pfetsch (2019). These algorithms are complete in the sense that they find all possible symmetry-based variable fixings derivable from a set of fixed variables. In contrast, orbital fixing by Margot (2003) and Ostrowski et al. (2011) can be used for arbitrary groups, however, without any guarantee on completeness. Moreover, Margot (2002, 2003) presents propagation pruning, a propagation technique to prune nodes of a B&B tree that do not contain lexicographically maximal solutions.

Besides propagation, other methods to handle symmetries in binary programs are cutting planes (Friedman 2007; Kaibel and Pfetsch 2008; Liberti 2008, 2012a, b; Liberti and Ostrowski 2014; Salvagnin 2018; Hojny and Pfetsch 2019; Hojny 2020), branching rules (Ostrowski et al. 2015, 2011), and model reformulations (Fischetti and Liberti 2012).

2. Framework and Basic Definitions

In this section, we describe our symmetry handling framework and discuss its relation to already existing symmetry handling methods. Moreover, we provide basic definitions that we use throughout this paper.

Let $\Gamma \subseteq S_n$ be a set of symmetries of a binary program. For any $x \in \{0,1\}^n$, all vectors in orbit($\Gamma, x$) := \{\gamma(x) : \gamma \in \Gamma\} have the same objective value (property (S2)), and either all or none of them is feasible for the binary program (property (S1)). We can thus handle symmetries by imposing an order on orbit($\Gamma, x$) and computing only the maximal elements w.r.t. this order. Our symmetry handling framework uses the standard lexicographic order, which is defined as follows. Let $k$ be an integer, and denote $[k] := \{i \in \mathbb{Z} : 1 \leq i \leq k\}$; that is, $k$ is the empty set if $k \leq 0$, and $\{1, \ldots, k\}$ otherwise. The standard lexicographic order regards $x \in \mathbb{R}^n$ as lexicographically larger than $y \in \mathbb{R}^n$, in formulae $x \succ y$, if there is $i \in [n]$ with $x_i > y_i$ and, for all $j \in [i-1]$, we have $x_j = y_j$. By $x \succeq y$, we denote
that either $x > y$ or $x = y$ holds. Then, $x$ is lexicographically maximal in its symmetry class if $x \geq \gamma(x)$ for all $\gamma \in \Gamma$. We denote the set of all $x \in \{0,1\}^n$ being not lexicographically smaller than their images $\gamma(x)$ by $\mathcal{X}_\gamma := \{x \in \{0,1\}^n : x \geq \gamma(x)\}$. In addition, for $\Pi \subseteq S_n$, denote $\mathcal{X}_\Pi := \bigcap_{\gamma \in \Pi} \mathcal{X}_\gamma$. If $\Pi$ is a permutation group, the intersection is taken over all group elements.

Embedding this symmetry handling framework into B&B works as follows. For a node of the B&B tree, let $I_0, I_1 \subseteq [n]$ be the indices of variables being fixed to zero and one, respectively, at this node. We denote by $F(I_0, I_1) := \{x \in \{0,1\}^n : x_i = 0 \text{ for } i \in F_0, x_i = 1 \text{ for } i \in I_1\}$ the binary points satisfying the fixings in $I_0$ and $I_1$. Then, our aim is to identify inclusion-wise maximal sets $I_0, I_1 \subseteq [n]$ such that $\mathcal{X}_{I_0} \cap F(I_0, I_1) = \mathcal{X}_{I_1} \cap F(I_0, I_1)$.

Every variable with index in $I_0 \setminus I_1$ and $I_1 \setminus I_0$ can be fixed to zero and one, respectively, at the current node of the branch-and-bound tree, because every solution that is lexicographically maximal in its symmetry class and adheres to the variable fixings $I_0$ and $I_1$ also adheres to the fixings in $I_0$ and $I_1$. Moreover, no further variables can be fixed because $I_0$ and $I_1$ are inclusion-wise maximal. An algorithm identifying $I_0$ and $I_1$ is called a complete propagation algorithm; if only a subset of $I_0$ or $I_1$ is identified, it is just called a propagation algorithm.

**Algorithm 1** (Find Complete Set of Fixings of $\mathcal{X}_{I_0} \cap F(I_0, I_1)$ with $\Gamma \subseteq S_n$)

- **Input:** $\Gamma \subseteq S_n$, sets $I_0, I_1 \subseteq [n]$  
- **Output:** message INFEASIBLE, or FEASIBLE and two subsets of $[n]$

1. **If** $\mathcal{X}_{I_0} \cap F(I_0, I_1) = \emptyset$ **then** return INFEASIBLE;
2. $t \leftarrow 0$, $(I_0', I_1') \leftarrow (I_0, I_1)$ (or supersets if Line 1 detected valid fixings);
3. **foreach** $i \in [n] \setminus (I_0 \cup I_1)$ **do**
   4. **If** $\mathcal{X}_{I_0} \cap F(I_0' \cup \{i\}, I_1') = \emptyset$ **then** $(I_0'^{i+1}, I_1'^{i+1}) \leftarrow (I_0', I_1' \cup \{i\})$;
   5. **If** $\mathcal{X}_{I_0} \cap F(I_0', I_1' \cup \{i\}) = \emptyset$ **then** $(I_0'^{i+1}, I_1'^{i+1}) \leftarrow (I_0' \cup \{i\}, I_1')$;
   6. $t \leftarrow t + 1$
4. return FEASIBLE, $(I_0', I_1')$;

To find $I_0'$ and $I_1'$, we propose to use the natural meta-algorithm Algorithm 1. The algorithm’s main idea is to check, for each index $i \in [n] \setminus (I_0 \cup I_1)$, whether $\mathcal{X}_{I_0} \cap F(I_0 \cup \{i\}, I_1)$ or $\mathcal{X}_{I_0} \cap F(I_0', I_1 \cup \{i\})$ is empty. If neither is empty, $x_i$ cannot be fixed to zero or one. Otherwise, if emptiness can be shown for adding $i$ to $I_0$, $x_i$ can be fixed to one and vice versa. Moreover, in Line 1, the algorithm checks whether infeasibility can be returned immediately. Then, the entire node of the B&B tree is infeasible and can be pruned. In later iterations, no infeasibility can be found because only valid fixings are added to $I_0$ and $I_1$.

We call Algorithm 1 a meta-algorithm as it does not specify how to decide $\mathcal{X}_{I_0} \cap F(I_0, I_1) = \emptyset$. In fact, for arbitrary groups, Algorithm 1 cannot run in polynomial time, unless $P = \text{coNP}$, as Babai and Luks (1983) show that deciding whether $x \in \mathbb{R}^n$ is lexicographically maximal in orbit($\Gamma, x$) is $\text{coNP}$-complete, which corresponds to the setting $I_0 \cup I_1 = [n]$. Our contribution is to show that Algorithm 1 can run in cubic time for certain cyclic groups.

### 2.1. Relation to Existing Methods and Contribution

To achieve this running time, we need an oracle that can answer whether, given $I_0, I_1 \subseteq [n]$, the set $\mathcal{X}_{I_0} \cap F(I_0, I_1)$ is empty in quadratic time. That is, the oracle needs to decide whether there is a lexicographically maximal point adhering to the fixings of $I_0$ and $I_1$. At first glance, this is exactly the problem isomorphism pruning is solving (see the literature overview in Section 1). At second glance, however, isomorphism pruning cannot be used in our framework: Isomorphism pruning requires either a specific branching rule (Margot 2002, 2003) or a lexicographic order adapted to the branching decisions (Ostrowski 2009). In particular, there is no unique lexicographic order in the latter approach, but potentially different orders for different nodes of the B&B tree. In contrast to this, our framework allows for an arbitrary branching rule, but relies on a fixed lexicographic order. Thus, isomorphism pruning is not applicable in our framework. Analogously, orbital fixing (Margot 2003, Ostrowski et al. 2011) and smallest-image fixing (Ostrowski 2009) cannot be used in our framework, as both require the same assumptions as isomorphism pruning.

Therefore, we need to develop new methods for testing emptiness in our framework, which is the contribution of this article. The reason why we suggest to apply our symmetry handling framework is that it allows us to prove that our algorithms are complete. That is, we find all possible variable fixings that can be derived from symmetry information and local variable fixings. As our analysis strongly depends on the interaction of the cyclic group with the standard lexicographic order, it is not immediate how this analysis can be carried out for isomorphism pruning.
2.2. Further Basic Definitions and Notations

In the algorithms that we present next, we make use of lexicographic order comparisons by considering only the first few entries of a vector. This defines a partial ordering. Given \( k \in [n + 1] \) and \( x, y \in \mathbb{R}^k \), we say \( x \equiv y \) if and only if \( x_i = y_i \) for all \( i \in [k - 1] \). To decide whether \( x \) and \( y \) can be distinguished \textit{up to position} \( k \), we write \( x \triangleright_k y \) if and only if there exists \( i \in [k - 1] \) such that \( x_i = y_i \) and \( x_i > y_i \). The relation \( x \equiv y \) holds if \( x \triangleright_k y \) or \( x \triangleright_k y \). These relations define the partial lexicographic order \( \equiv \). When \( k = n + 1 \), we write \( =, >, \) and \( \geq \) instead of \( \equiv \), \( >_k \), and \( \geq_k \), respectively. In this case, we say that \( x \) is \textit{equal} to, \textit{lexicographically greater than}, and \textit{not lexicographically smaller than} \( y \), respectively.

Let \( y \in S_n \) and \( \Pi \subseteq S_n \). Recall that \( \mathcal{X}_y = \{ x \in [0,1]^n : x \geq \gamma(x) \} \) and \( \mathcal{X}_{\Pi} := \cap_{y \in \Pi} \mathcal{X}_y \). We define \( \mathcal{X}_{\Pi}^{(k)} \) and \( \mathcal{X}_{\Pi}^{(k)} \) if we use \( \geq_k \) instead of \( \geq \), that is, \( \mathcal{X}_{\Pi}^{(k)} = \{ x \in [0,1]^n : x \geq_k \gamma(x) \} \) and \( \mathcal{X}_{\Pi}^{(k)} := \cap_{y \in \Pi} \mathcal{X}_y^{(k)} \).

We introduce basic notation on permutations. A \textit{cycle} of length \( k \) is an ordered tuple \( \zeta = (i_1, \ldots, i_k) \) without repeated entries. The \textit{support} of \( \zeta \), \( \text{supp}(\zeta) = \{ i_1, \ldots, i_k \} \), \( \zeta(i) := i_{i+1} \), where subscripts are modulo \( k \). Two cycles are \textit{disjoint} if their supports are disjoint. Each permutation \( \gamma \in S_n \) admits a \textit{disjoint cycle representation} \( \zeta_1, \ldots, \zeta_\ell \), where \( \bigcup_{\ell = 1}^\ell \text{supp}(\zeta_\ell) = [n] \) and \( \zeta_\ell(i) = \zeta_\ell(i) \) for the unique cycle \( \zeta_\ell \) with \( i \in \text{supp}(\zeta_\ell) \). The support of \( \gamma \) is \( \text{supp}(\gamma) = \{ i \in [n] : \gamma(i) \neq i \} \); we say \( \gamma \) affects \( i \in [n] \) if \( i \in \text{supp}(\gamma) \). For any \( i \in [\ell] \), \( \zeta_\ell \) can be a \textit{subcycle} of \( \gamma \).

A \textit{permutation group} is a set \( \Gamma \subseteq S_n \) endowed with the composition operation \( \circ \) that contains the identity permutation \( \text{id} \) and is closed under taking inverses and compositions. The inverse of permutation \( \gamma \) is denoted by \( \gamma^{-1} \), and the \( j \)-fold composition of \( \gamma \) with itself is denoted by \( \gamma^j \). To stress that a set of permutations \( \Gamma \) forms a permutation (sub)group of \( S_n \), we write \( \Gamma \leq S_n \). A set of permutations \( \Pi \subseteq \Gamma \leq S_n \) \textit{generates} \( \Gamma \) if every \( \gamma \in \Gamma \) can be written as a finite composition of elements from \( \Pi \). In this case, \( \Pi \) is called a set of \textit{generators} of \( \Gamma \), and we denote by \( \langle \Pi \rangle \) the group generated by \( \Pi \). If \( \Pi = \{ \gamma \} \), we write \( \langle \gamma \rangle \) instead of \( \langle \langle \gamma \rangle \rangle \). We refer to \( \langle \gamma \rangle \) as a \textit{cyclic group}.

3. Propagating Lexicographic Orders for Cyclic Groups

To effectively use Algorithm 1, we need an oracle to identify whether \( \mathcal{X}_\Pi \cap F(I_0, I_1) \) is empty for sets \( I_0, I_1 \subseteq [n] \). This section forms the main part of our paper, as it presents such an oracle for two types of cyclic groups. In particular, although these groups might be superpolynomially large, we will show that the oracle can run in quadratic time.

Before presenting our oracles, we stress that it is not sufficient to iteratively propagate individual permutations in \( \Gamma \). This means, given \( I_0 \) and \( I_1 \), we iterate over all \( \gamma \in \Gamma \) and check whether \( x \geq \gamma(x) \) allows us to derive further variable fixings. If no permutation allows us to derive a new fixing, the method stops. Otherwise, we update \( I_0 \) and \( I_1 \) and iterate over all permutations again as newly found fixings might trigger further fixings for other permutations. At termination, no further fixings can be derived from \( I_0 \) and \( I_1 \), and we say that \( I_0 \) and \( I_1 \) form a \textit{complete set of fixings for individual permutations}.

The following example shows a case where, although the given fixings are complete for individual permutations, further symmetry-based reductions can be derived by taking an entire cyclic group into account.

**Example 1.** Let \( \tilde{\gamma} = (1,2,3,4,5) \) and \( \Gamma = \langle \tilde{\gamma} \rangle \). Let \( I_0 = \{ 2,5 \} \) and \( I_1 = \emptyset \). One can show that \( X_\Pi \cap F(I_0, I_1) = \{ (0,0,0,0,0),(1,0,0,0,0),(1,0,1,0,0) \} \). Hence, \( x_4 = 0 \) for all \( x \in X_\Pi \cap F(I_0, I_1) \), index 4 can be added to \( I_0 \). However, this fixing cannot be derived from a single constraint \( x \geq \gamma(x) \) for \( \gamma \in \Gamma \), because individual constraints allow solutions with \( x_4 = 1 \); for \( \delta \in \{ \tilde{\gamma}^{-1}, \tilde{\gamma}^{-2}, \tilde{\gamma}^{-4} \} \), we find \( (1,0,1,1,0) \in X_\delta \) and \( (1,0,0,1,0) \in X_\gamma \).

In contrast to Algorithm 1, considering individual permutations thus might not be sufficient to find all possible fixings. Therefore, we discuss algorithms to decide \( \mathcal{X}_\Pi \cap F(I_0, I_1) = \emptyset \) for an entire cyclic group in this section. Our algorithms require structural properties of the cyclic groups that we detail next.

A cycle \( \zeta = (i_1, \ldots, i_k) \) is \textit{monotone} if there is exactly one \( j \in [k] \) with \( \zeta(i_j) < i_j \). For instance, \( (1,2,3,4) \) is monotone, whereas \( (1,3,2,4) \) is not. We call a permutation \( \gamma = \zeta_1 \ldots \zeta_\ell \) monotone if every subcycle is monotone. Moreover, \( \gamma \) is \textit{ordered} if, for any distinct \( i, j \in [\ell] \), the closed intervals \( [\min \text{supp}(\zeta_i), \max \text{supp}(\zeta_i)] \) and \( [\min \text{supp}(\zeta_j), \max \text{supp}(\zeta_j)] \) are disjoint, for example, \( (1,2)(3,4) \) is ordered and \( (1,4)(2,3) \) is not. Section 3.1 discusses the oracle for \( \Gamma \leq \langle \gamma \rangle \) with \( \gamma \) being a monotone cycle; the oracle for \( \gamma \) being a monotone and ordered permutation is discussed in Section 3.2. Although these requirements do not allow us to handle all cyclic groups, we will discuss in Section 5 how our algorithms can still be used to handle symmetries in \textit{arbitrary} binary programs.

3.1. Oracle for Monotone Cycles

In this section, we consider the case that \( \Gamma \leq \langle (1,2,\ldots,n) \rangle \), that is, \( \Gamma \) is a subgroup of a cyclic group that is generated by a monotone cycle. The task of the oracles is to decide whether \( \mathcal{X}_\Pi \cap F(I_0, I_1) \neq \emptyset \). Because \( \mathcal{X}_\Pi \subseteq \mathcal{X}_\gamma \), for
every \( \gamma \in \Gamma \), a necessary condition for \( \mathcal{X}_\Gamma \cap F(I_0, I_1) \) being nonempty is that \( \mathcal{X}_\gamma \cap F(I_0, I_1) \neq \emptyset \) for every \( \gamma \in \Gamma \). The core ingredient of our oracle is the, at first glance surprising, fact that this condition is also sufficient.

**Proposition 1.** Let \( I_0, I_1 \subseteq [n] \) be disjoint sets defining a complete set of fixings for the constraint \( x \geq \gamma(x) \) for each permutation \( \gamma \in \Gamma \leq \langle (1, \ldots, n) \rangle \). Then, \( \mathcal{X}_\Gamma \cap F(I_0, I_1) \neq \emptyset \) if and only if \( \mathcal{X}_\gamma \cap F(I_0, I_1) \neq \emptyset \) for every \( \gamma \in \Gamma \).

Before we present the proof of this proposition, we discuss its implications for Algorithm 1. In Lines 1, 4, and 5, Algorithm 1 needs to check whether there exists a lexicographically maximal point (w.r.t. \( \Gamma \)) adhering to a set of fixings. By Proposition 1, it is sufficient to check whether there exists, for each individual \( \gamma \in \Gamma \), a (potentially different) point \( x \in F(I_0, I_1) \) satisfying \( x \geq \gamma(x) \). To turn the meta-algorithm Algorithm 1 into an actual algorithm, we thus need to be able to decide whether \( \mathcal{X}_\gamma \cap F(I_0, I_1) \neq \emptyset \) for all \( \gamma \in \Gamma \). This can be done using a complete propagation algorithm for \( \mathcal{X}_\gamma \cap F(I_0, I_1) \) for all individual \( \gamma \in \Gamma \), as it can determine whether there exists \( \gamma \in \Gamma \) with \( \mathcal{X}_\gamma \cap F(I_0, I_1) = \emptyset \).

**Theorem 1.** Let \( I_0, I_1 \subseteq [n] \) be disjoint, let \( \Gamma \leq \langle (1, \ldots, n) \rangle \), and let \( f(\Gamma, n) \) be the time of determining the complete set of fixings for individual permutations in \( \Gamma \). Then, Algorithm 1 can be implemented to run in \( O(nf(\Gamma, n)) \) time.

**Proof.** By the previous discussion, Lines 1, 4, and 5 of Algorithm 1 can be carried out by an algorithm determining the complete set of fixings for individual permutations \( \gamma \in \Gamma \). In each iteration of the “foreach” loop and outside the loop, such an algorithm is called constantly often. The loop has size \( O(n) \), so the total running time is \( O(nf(\Gamma, n)) \).

**Theorem 2.** Let \( \Pi \subseteq S_n \), and let \( I_0, I_1 \subseteq [n] \) be disjoint. The complete set of fixings for individual permutations in \( \Pi \) that are derived from \( I_0 \) and \( I_1 \) can be found in \( O(n|\Pi|) \) time.

Although Theorem 2 is the core for our running time analysis of Algorithm 1, we postpone its details until Section 4 to not lose our focus on structural properties of cyclic groups. Together, Theorems 1 and 2 with \( \Pi = \Gamma \leq \langle (1, \ldots, n) \rangle \), implying \( |\Pi| \leq n \), yield the following.

**Corollary 1.** Let \( I_0, I_1 \subseteq [n] \) be disjoint index sets, and let \( \Gamma \leq \langle (1, \ldots, n) \rangle \). Then, Algorithm 1 can be implemented to run in \( O(n^2) \) time.

In preparation for the proof of Proposition 1, we present an argument that we will use frequently throughout the proofs of this and the next section.

**Remark 1.** Let \( I_0, I_1 \subseteq [n] \) define a complete set of fixings for constraint \( x \geq \gamma(x) \) for each \( \gamma \in \Gamma \). Suppose \( \gamma \in \Gamma \) with \( \mathcal{X}_\gamma \cap F(I_0, I_1) \neq \emptyset \). In the proof of Proposition 1, we frequently exploit implications between different elements of \( I_0 \) and \( I_1 \), which are based on completeness of fixings, as follows.

Let \( k \in [n] \) be such that \( x \gamma = \gamma(x) \) for each \( x \in \mathcal{X}_\gamma \cap F(I_0, I_1) \), which means \( x_i = \gamma(x) \in \{0, 1\} \) for all \( i < k \) and \( x \in \mathcal{X}_\gamma \cap F(I_0, I_1) \). Note that this can only be the case if, for every \( i \in [k-1] \), we have \( i, \gamma^{-1}(i) \in I_0 \) or \( i, \gamma^{-1}(i) \in I_1 \) by completeness of \( I_0 \) and \( I_1 \). The first implication that we exploit is the following: if \( k \in I_0 \), we also need \( x \gamma \in I_0 \) to ensure \( x \geq x \gamma \). Because the set \( I_0 \) is complete, we find \( \gamma^{-1}(k) \in I_0 \). Analogously, if \( \gamma^{-1}(k) \in I_1 \), then \( k \in I_1 \).

Equipped with these arguments, we present the missing proof.

**Proof of Proposition 1.** Let \( I_0, I_1 \subseteq [n] \) be disjoint sets defining, for all \( \gamma \in \Gamma \), a complete set of fixings for \( \mathcal{X}_\gamma \cap F(I_0, I_1) \). Above, we argued that \( \mathcal{X}_\gamma \cap F \neq \emptyset \) for each \( \gamma \in \Gamma \) is necessary for \( \mathcal{X}_\Gamma \cap F \neq \emptyset \). It remains to show that this condition is sufficient. To this end, we construct an explicit \( \bar{x} \in \mathcal{X}_\Gamma \cap F \) while assuming \( \mathcal{X}_\gamma \cap F \neq \emptyset \) for all \( \gamma \in \Gamma \).

First, assume \( I_0 \cup I_1 = \{n\} \). Then, \( F = \{\bar{x}\} \) for some \( \bar{x} \in \{0, 1\}^n \). Because, for all \( \gamma \in \Gamma \), \( \mathcal{X}_\gamma \cap F \neq \emptyset \), we conclude \( \bar{x} \geq \gamma(\bar{x}) \) for all \( \gamma \in \Gamma \). Consequently, \( \bar{x} \in \bigcap_{\gamma \in \Gamma} \mathcal{X}_\gamma \cap F = \mathcal{X}_\Gamma \cap F \).

Second, suppose \( I_0 \cup I_1 \subsetneq \{n\} \). Let \( i := \min([n] \setminus (I_0 \cup I_1)) \) and, for \( \gamma \in \Gamma \), let \( i_{\gamma} := \min\{\gamma^{-1}(i) : i \in [n] \setminus (I_0 \cup I_1)\} \). Then, \( i \) is the first unfixed entry of a vector \( x \in F \), and \( i_{\gamma} \) is the first unfixed entry of a vector \( \gamma(x) \) with \( x \in F \). If the permutation is clear from the context, we drop the subscript of \( i_{\gamma} \). Let \( \bar{x} \in F \) with \( \bar{x}_i = 1 \) if \( i \in I_1 \) or \( i_{\gamma} \), and \( \bar{x}_i = 0 \) otherwise. We claim that \( \bar{x} \in \mathcal{X}_\Gamma \cap F \), which completes the proof.

For the sake of contradiction, assume there exists \( \gamma \in \Gamma \) with \( \bar{x} \prec \gamma(\bar{x}) \). Let \( m = \min([\bar{i}], [i]) \). Then, \( x_m = \gamma(x) \) for all \( x \in \mathcal{X}_\gamma \cap F \) as the latter set is nonempty. Consequently, \( (\hat{x}, \gamma(\hat{x})) \neq (1, 0) \), as otherwise, \( \bar{x} \prec \gamma(\bar{x}) \). We will use this observation, which we refer to as (*), to find a contradiction in the following. We proceed by a case distinction.

**Case 1.** We have \( \hat{i} \leq i \). Then, \( \gamma^{-1}(i) \not\in I_1 \), as otherwise, the complete propagation algorithm for \( \mathcal{X}_\gamma \cap F \) had fixed \( \gamma \) to one by Remark 1. Because \( \bar{x} \prec \gamma(\bar{x}) \), \( \gamma \) is not the identity. Thus, \( \gamma \) is a nontrivial cyclic shift and as such has no
fixed points, which implies \( \gamma^{-1}(i) \neq i \). As \( \gamma^{-1}(i) \notin I_1 \) and the only 1-entry of \( x \) not being implied by \( I_1 \) is \( i \neq \gamma^{-1}(i) \), we conclude \( x(i) = x_{\prec \gamma^{-1}(i)} = 0 \). By (\( \ast \)), however, this is a contradiction, as \( \hat{x}_i = 1 \).

**Case 2.** We have \( i > i \) and \( i \notin I_0 \). If \( \gamma^{-1}(i) \notin I_0 \), there is \( x \in F \) with \( \gamma(x) = x_{\gamma^{-1}(i)} = 1 \). Because \( i < i \), this shows \( x = \gamma(x) = 0 \), \( x = \gamma(x) = 1 \), that is, \( x < \gamma(x) \). Consequently, \( \gamma^{-1}(i) \) can be added to \( I_0 \), contradicting completeness of \( I_0 \). Therefore, Case 2 implies \( \gamma^{-1}(i) \notin I_0 \), but this contradicts the definition of \( i \).

**Case 3.** We have \( i > i \) and \( i \notin I_0 \). Because \( i < i \), we have \( i \in I_0 \cup I_1 \). Combined with the second property of Case 3, \( i \notin I_0 \). Then, \( \gamma(x) = 1 \) needs to hold, as otherwise \( x > \gamma(x) \). This means that \( \gamma^{-1}(i) \notin I_1 \cup \{i\} \), as these are the only 1-entries of \( x \), and from the definition of \( i \), we conclude that \( \gamma^{-1}(i) = i \). To derive the contradiction in Case 3, we consider \( j := \min\{i \in [n] : x_i \neq \gamma(x)\} \). Then, \( \hat{x}_j = 0 \) by \( x < \gamma(x) \). It is not possible that \( j = i \), as \( \hat{x}_j = 0 \neq 1 = \hat{x}_i \).

**Claim 1.** If \( i > i \), any \( x \in X_\gamma \cap F \) with \( x_i = 0 \) has \( x_i = \hat{x}_i \) for all \( i = \min\{n, i^f - i - 1\} \).

**Proof of Claim 1.** Let \( x \in X_\gamma \cap F \) with \( x_i = 1 \). To prove the claim, we show \( x_i = \hat{x}_i \) for all \( i \leq b \), where \( b = \min\{n, i + i - i - 1\} \). We prove this by induction on \( b \). As \( x, \hat{x} \in F \), by \( x_i = \hat{x}_i \) and the definition of \( i \), \( x_i = \hat{x}_i \) for all \( i \leq b \), so the statement holds for all \( b \leq i \).

Consider \( b = \min\{n, i^f - i - 1\} + 1 \) and \( b > i \). Suppose \( x_i = \hat{x}_i \) holds for all \( i \leq b - 1 \) (\( \text{IH} \)). By the previous paragraph, this holds for \( b = i + 1 \). We show that \( x_i = \hat{x}_i \), so that the claim follows by induction. If \( b \in I_0 \cup I_1 \), then \( x_i = \hat{x}_i \) holds trivially as entry \( b \) is fixed, so we assume \( b \notin I_0 \cup I_1 \). By previous arguments, \( x_{b < i} = \hat{x}_{b < i} \). Fixing \( i \) and \( j \) and \( k \) and \( l \) to the appropriate entries of \( i \), we get \( x_{b < i} = \hat{x}_{b < i} \). Thus, \( x_{b < i} = \hat{x}_{b < i} \) for all \( b \leq i \).

\[
(1) \quad x_{b - i} = x_{\gamma(b - i)} = \gamma(x)_{b - i} = \gamma(x)_{b - i} = \gamma(x)_{b - i}. 
\]

In this, (\( \ast \)) uses that \( \gamma \) is a cyclic shift with \( \gamma^{-1}(i) = i \), such that \( \gamma^{-1}(i) = i + i - i - i \) \( (\text{IH}) \). By the previous paragraph, this holds for \( b = i + 1 \). Thus, \( x_{b - i} = \hat{x}_{b - i} \).

Otherwise, if \( x_{b - i} = 0 \), \( \text{IH} \) yields \( x_{b - i} = \gamma(x)_{b - i} \), \( \gamma(x)_{b - i} = \gamma(x)_{b - i} \). Combining these results yields \( x_{b - i} = \gamma(x)_{b - i} \).

Thus, \( (1) \) yields \( x_{b - i} = \gamma(x)_{b - i} = \gamma(x)_{b - i} \). As always, we have \( x_{b} = x_{b - i} \).

Recall that \( j \neq i \) and that \( \gamma^{-1}(i) = i \). As completeness of the fixing for permutation \( \gamma \) is assumed and \( i \notin I_0 \cup I_1 \), there always exists a vector \( x \in X_\gamma \cap F \) with \( x_i = 1 \). If the minimum in the claim evaluates to \( n \), then \( x = \hat{x} \), so \( x \in X_\gamma \cap F \). Otherwise, if the minimum evaluates to \( j = j < i \), then \( x_{\gamma(j)} = \hat{x}_{\gamma(j)} \), so especially \( x_j = \hat{x}_j = 0 \) as \( j > i \). Because \( \gamma \) is a cyclic shift with \( \gamma^{-1}(i) = i \), \( \gamma(x) = \gamma(x) \).

Thus, \( (1) \) yields \( x_{b - i} = \hat{x}_{b - i} \), and \( i > i \) that \( x = \hat{x} = \gamma(x) = \gamma(x) = x = 0, \gamma(x) = 1 \), such that \( x < \gamma(x) \), contradicting \( x \in X_\gamma \). Each case yields a contradiction, proving \( \hat{x} = x \).

### 3.2. Oracle for Monotone and Ordered Permutations

The setting of subgroups of \( \langle 1, \ldots, n \rangle \), which we discussed in the last section, is rather strict and might not appear in practice even if cyclic symmetries are present.

**Example 2.** The minimum edge cover of an undirected graph \( G = (V, E) \) selects the minimum subset of edges such that every vertex is incident to at least one selected edge. As a binary program, it is \( \min\{\sum_{e \in E} x_e : \sum_{e \in E} E x_e \geq 1 \} \) for all \( v \in V \), and \( x_e \in \{0, 1\} \). The decision variable \( x_e \) indicates whether edge \( e \in E \) is included in the edge cover solution. Let \( V = [6] \) and \( E = \{u, v\} \in V^2 : u - v \mod 6 \in \{1, 2\} \). Permuting the vertex indices by a cyclic shift \( \langle 1, \ldots, 6 \rangle \) yields isomorphic graphs. As such, a symmetry in the binary problem permutes the variable indices (that are edges) by applying the permutation to both end points of an edge, that is, \( \gamma = \gamma_1 \circ \gamma_2 \) with \( \gamma_1 = ((1, 2), (2, 3), (3, 4), (4, 5), (5, 6), (6, 1)) \) and \( \gamma_2 = ((1, 3), (2, 4), (3, 5), (4, 6), (1, 5), (2, 6)) \). By appropriately ordering the edges, \( \gamma \) can be made monotone and ordered.

Therefore, we consider the more general case of groups generated by a monotone and ordered permutation in this section. Although this requirement seems restrictive, we can achieve it by relabeling the variables as indicated in the previous example. The main reason why we focus on such groups is that we can derive a characterization of lexicographically maximal points that can be checked easily by an iterative scheme.
Lemma 1. Let $x \in \mathbb{R}^n$, and let $\gamma = \gamma_1 \ldots \gamma_m \in S_n$ be an ordered permutation such that $\max \text{supp}(\gamma_1) < \min \text{supp}(\gamma_2) < \max \text{supp}(\gamma_2) < \ldots < \max \text{supp}(\gamma_m)$. Let $M_0 := 1$, and for $c \in [m]$, let $M_c := \max \text{supp}(\gamma_c) + 1$. Then, there is $k \in \mathbb{Z}$ with $\gamma := \gamma_k \in \gamma$ with $\gamma(x) > x$ if and only if there is $c \in [m]$ such that $(P1) x = M_{c-1} \gamma(x)$ and $(P2) \hat{c}(x) > x$.

Proof. For fixed $\gamma := \gamma_k \in \gamma$, let $c = \min\{c' \in [m] : x \neq M_{c'} \gamma(x)\} \cup \{\infty\}$. If $c = \infty$, this means $x = \gamma(x)$. Thus, $c \in [m]$ is necessary for $x \neq \gamma(x)$. For this reason, assume $c \in [m]$. As $\gamma$ does not affect elements outside $\text{supp}(\gamma_c)$ and the cycles are ordered, $(\gamma_1 \ldots \gamma_m)(x) = M_{c-1} \gamma_c(x) = M_c x$. If $(P2)$ holds, this means $(\gamma_1 \ldots \gamma_m)(x) = M_{c-1} \gamma_c(x) = M_c x$, because the different cycles commute as they are disjoint. Analogously, if $(P2)$ does not hold, $x \geq M_c \gamma(x)$, and the definition of $c$ yields $x > M_c \gamma(x)$. Consequently, $(P1)$ and $(P2)$ are necessary and sufficient. □

Replacing $(P2)$ by $x > \hat{c}(x)$ shows that the corresponding permutation $\gamma$ yields $x = \gamma(x)$. We will exploit this observation in an efficient implementation of the meta-algorithm Algorithm 1. Another core ingredient of our analysis is a refined version of Proposition 1, which still considers the case of $(1, \ldots, n)$. Proposition 2. Let $I_0, I_1 \subseteq [n]$ with $I_0 \cup I_1 \subseteq [n]$ be disjoint index sets defining a complete set of fixings for every $\gamma \in \Gamma \subseteq (\{1, \ldots, n\})$. If $X_1 \cap F(I_0, I_1) \neq \emptyset$, then there exists a $x \in F(I_0, I_1)$ such that for every $\gamma \in \Gamma \setminus \{\emptyset\}$, $x > \gamma(x)$ holds.

Proof. Let $t := \min\{\min([n] \setminus (I_0 \cup I_1)) \}$ be the first unfixed entry. Based on the value of $t$, we construct $\tilde{x} \in X_1 \cap F(I_0, I_1)$ with $\tilde{x} > \gamma(x)$ for each $\gamma \in \Gamma \setminus \{\emptyset\}$.

Construction 1 $(t > \frac{1}{2}n)$. We claim that $\tilde{x} \in [0, 1]^n$ with $\tilde{x}_i = 1$ for $i \in I_1$ and $\tilde{x}_i = 0$ for $i \in [n] \setminus I_1$ has the desired properties. For the sake of contradiction, suppose $\tilde{x}$ does not satisfy $\tilde{x} > \gamma(\tilde{x})$ for some $\gamma \in \Gamma \setminus \{\emptyset\}$.

Case 1. There exists $\gamma \in \Gamma \setminus \{\emptyset\}$ such that $\tilde{x} = \gamma(\tilde{x})$. Then, $\tilde{x} = \delta(\gamma)$ for all $\delta \in (\gamma)$. As $\gamma \neq \emptyset$ is a cyclic shift, there is $\delta \in (\gamma)$ with $\delta(t) \leq \frac{1}{2}n < i$. Therefore, we may assume without loss of generality that $\gamma(i) \leq \frac{1}{2}n$. Combining this and the observation $\tilde{x}_{\gamma(i)} = \gamma(\tilde{x})_{\gamma(i)} = \tilde{x}_i = 0$, we conclude $\gamma(i) \in I_0$. As $\gamma$ is a cyclic shift, $\gamma^{-1}(i) = i + 1 - \gamma(i)$ (mod n), wherein the modular residual classes identify with $\{1, \ldots, n\}$. From $1 \leq \gamma(i) \leq \frac{1}{2}n < i \leq n$, we get that $1 \leq \gamma^{-1}(i) < i$ for all $1 \leq i < \gamma(i)$, showing that $i, \gamma^{-1}(i) \in I_0 \cup I_1$ for all $i < \gamma(i)$. By the assertion $\tilde{x} = \gamma(\tilde{x})$, all vectors $x \in F(I_0, I_1)$ thus have $x_{\gamma^{-1}(i)} = \gamma(x)_{\gamma^{-1}(i)}$ and satisfy $x_{\gamma^{-1}(i)} = 0$ by $\gamma(i) \in I_0$. Because $i \not\in I_0 \cup I_1$, any $x \in F(I_0, I_1)$ with $x > \gamma(x)$ must have $0 = x_{\gamma^{-1}(i)} > \gamma(x)_{\gamma^{-1}(i)} = \tilde{x}_i$, such that $x_i = 0$, which violates completeness for permutation $\gamma$. This is the desired contradiction.

Case 2. There is $\gamma \in \Gamma \setminus \{\emptyset\}$ with $\tilde{x} > \gamma(\tilde{x})$. Let $j := \min\{\min([n] : \exists i : \tilde{x}_i \neq \gamma(\tilde{x})_i)\}$, and let $i := \min\{\min([n] : \gamma^{-1}(i) \notin I_0 \cup I_1)\}$ be the first nonfixed entry in a vector permuted by $\gamma$. We derive some properties of such vectors $x \in F(I_0, I_1)$. Every vector $x \in F(I_0, I_1)$ satisfies $x_i = \tilde{x}_i$ for $i < \gamma(i)$ and $x_{\gamma(i)} = \gamma(\tilde{x})_{\gamma(i)}$, for $i < \gamma(i)$. Also, $0 = \tilde{x}_i < \gamma(\tilde{x})_{\gamma(i)} = 1$, showing $j \in I_1$ and $\gamma^{-1}(j) \in I_1$. As $X_1 \cap F(I_0, I_1)$ is nonempty and $\tilde{x} > \gamma(\tilde{x})$, $j \geq \min\{\min\{\emptyset\}, \tilde{x}\}$. Thus, $x = \gamma(x)$ for every vector $x \in F(I_0, I_1)$. We distinguish three cases..

Case 2a. We have $i < \min\{\emptyset\}$. By definition of $i$, there is $x \in X_1 \cap F(I_0, I_1)$ such that $1 \neq \gamma^{-1}(i) = \gamma(x)$. Because $i < \tilde{x}$, we find $x_i = \tilde{x}_i = \gamma(\tilde{x})_i = 0$ and thus $i \in I_0$. So all $x \in F(I_0, I_1)$ have $x_i = \gamma(x)$, and $x_i = 0$. But then $\gamma(x)_i = 1$ is not possible, as this implies $x > \gamma(x)$. To ensure $x > \gamma(x)$, entry $\gamma^{-1}(i)$ must be fixed to zero as well. This contradicts completeness of the fixings. Therefore, $i \geq \min\{\min\{\emptyset\}, \tilde{x}\}$, and we conclude from $j \geq \min\{\min\{\emptyset\}, \tilde{x}\}$ that $i > \tilde{x}$ and $i > \gamma(i)$.

Case 2b. We have $i = \gamma(i)$. Then $\gamma(\tilde{x})_{\gamma(i)} = 1$, that is, $\gamma^{-1}(i) \in I_1$, contradicting completeness.

Case 2c. We have $j > i$. Because $i > \frac{1}{2}n$, there is exactly one sequence of $i - 1$ consecutive numbers modulo $n$ in $I_0 \cup I_1$ followed by a noncontained number. This sequence is $1, \ldots, i$. In the vector permuted by $\gamma$, this sequence is mapped to $1 + (i - 1), \ldots, i + (i - 1)$, as it is a cyclic shift, $i$ is the first nonfixed entry in the vector permuted by $\gamma$, and $i > i > \frac{1}{2}n$. Hence, $\gamma^{-1}(i) = i + 1 - j$ (mod n). Then $i > \tilde{x}$ follows from $\gamma \neq \emptyset$, so $i > \tilde{x}$ and $j > i$. As such, any vector $x \in F(I_0, I_1)$ has $x_{\gamma^{-1}(i)} = \gamma(x)_{\gamma^{-1}(i)}$. From this we conclude $\gamma^{-1}(i) \notin I_0$, because otherwise, completeness of $I_1$ would imply $i \in I_1$ to ensure $x \geq \gamma^{-1}(i) \gamma(x)$, contradicting the definition of $i$. Thus, $\gamma^{-1}(i) \in I_0$ because $i > \tilde{x}$. However, this leads to a contradiction with completeness of permutation $\gamma^{-1} \in \Gamma$, as we show next.

The explicit formula above for $\gamma^{-1}(i)$ implies $\gamma(i) = i + 1 - i$ (mod n). As $\frac{1}{2}n < i < n$, the first unfixed entry in the vector permuted by $\gamma^{-1}$ is $\gamma^{-1}(i) = 2i - 1$. Applying permutation $\gamma^{-1}$ to any vector permutes the entries $1 + (i - 1), \ldots, i + (i - 1)$ to $1, \ldots, i$. Because for any vector $x \in F(I_0, I_1)$ we have $x_{\gamma^{-1}(i)} = \gamma(x)$, we can apply $\gamma^{-1}$ on either side to find $\gamma^{-1}(x) = \gamma(x) \gamma^{-1}(\gamma(x))$, so $x = \gamma(x) \gamma^{-1}(x)$. By the previous paragraph, $\gamma^{-1}(i) \in I_0$, so any $x \in F(I_0, I_1)$ has $x_{\gamma^{-1}(i)} = \gamma(x)$ and $x_{\gamma^{-1}(i)} = 0$. As the set of fixings is complete for $x \geq \gamma(x)$, all $x \in F(I_0, I_1)$ have $\gamma^{-1}(x) = x_i = 0$, so $i \in I_0$. This contradicts that $i \notin I_0 \cup I_1$ and completes the proof of Case 2.
\[ \hat{x} < \frac{1}{n}. \] For the sake of contradiction, suppose that \( \check{x} = \gamma(\hat{x}) \) for some \( \gamma \in \Gamma \setminus \{ \text{id} \} \). Then especially \( \check{x} = \delta(\hat{x}) \) for all \( \delta \in \langle \gamma \rangle \). Because \( \hat{x}_1 = 1 \), for all \( \delta \in \langle \gamma \rangle \), we have \( \delta(\hat{x}_1) = 1 \), that is, orbit(\( \langle \gamma \rangle, \hat{x} \)) \subseteq I_1 \cup \{ \text{id} \}.

Let \( \delta \in \langle \gamma \rangle \setminus \{ \text{id} \} \) and \( i_\delta = \min \{ i \in [n] : \delta^{-1}(i) \not\in I_0 \cup I_1 \} \). If \( \hat{i} \leq i_\delta \), then \( x = 0(\hat{x}) \) for all \( x \in F(I_0, I_1) \). Also, \( \delta(x)_1 = 1 \) follows from orbit(\( \langle \gamma \rangle, \hat{x} \)) \subseteq I_1 \cup \{ \text{id} \} and that \( \delta \neq \text{id} \) is cyclic, such that \( \delta^{-1}(\hat{i}) \neq \hat{i} \). But then completeness of the fixings for permutation \( \delta \) implies that \( \hat{i} \in I_1 \), which violates the definition of \( \hat{i} \). Hence, \( \hat{i} > i_\delta \) for all \( \delta \in \langle \gamma \rangle \setminus \{ \text{id} \} \).

As \( i_\delta < \hat{i} \), we have \( i_\delta \in I_0 \cup I_1 \), and \( x = 0(\hat{x}) \) for all \( x \in F(I_0, I_1) \). If \( i_\delta \in I_0 \), completeness of the fixings for permutation \( \delta \) yields \( \delta^{-1}(i_\delta) \in I_0 \), violating the definition of \( i_\delta \). Thus, \( i_\delta \in I_1 \). and as such, from \( 1 = \hat{x}_1 = \delta(\check{x}_1) \) follows \( \delta^{-1}(i_\delta) \in I_1 \cup \{ \text{id} \} \). Again, by definition of \( i_\delta \), \( \delta^{-1}(i_\delta) \not\in I_0 \cup I_1 \), so \( \delta^{-1}(i_\delta) = \hat{i} \). Thus, \( \hat{i} = \delta(\hat{i}) \). As \( \gamma \neq \text{id} \) is a cyclic shift and \( \hat{i} \leq \frac{1}{2}n \), there is \( \delta \in \langle \gamma \rangle \setminus \{ \text{id} \} \) with \( i_\delta = \delta(\hat{i}) > \frac{1}{2}n \geq \hat{i} \), contradicting \( \hat{i} > i_\delta \) for all \( \delta \in \langle \gamma \rangle \setminus \{ \text{id} \} \).

To present our oracle, we require some concepts from group theory that we explain next. Given a set \( I \subseteq [n] \) and a group \( \Gamma \leq S_n \), the **pointwise stabilizer** is \( \text{STAB}(I, \Gamma) := \{ \gamma \in \Gamma : \gamma(i) = i \text{ for all } i \in I \} \). The **setwise stabilizer** is \( \text{stab}(I, \Gamma) := \{ \gamma \in \Gamma : \gamma(i) \in I \text{ for all } i \in I \} \). The orbit of a solution \( x \) with respect to a group \( \Gamma \) is \( \langle \gamma(x) : \gamma \in \Gamma \rangle \). Last, the **restriction** of \( \gamma \in S_n \) to \( I \) is \( \text{restr}(\gamma, I) \), defined as \( \text{restr}(\gamma, I) = i \gamma(i) \text{ for } i \in I \) and \( i \gamma(i) = i \text{ for } i \notin I \). For groups \( \Gamma \leq S_n \), \( \text{restr}(\Gamma, I) := \{ \text{restr}(\gamma, I) : \gamma \in \Gamma \} \). Note that \( \gamma \in \text{STAB}_I \) if and only if \( \gamma(I) = I \) and that \( \text{restr}(\Gamma, I) \subseteq \text{STAB}_I \) if and only if \( \gamma \) is a union of orbits of \( \Gamma \).

Using this notation and Proposition 2, we can provide an efficient oracle for cyclic groups generated by monotone and ordered \( \gamma = \zeta_1 \ldots \zeta_m \). We assume that the cycle indices respect the order of the cycles, that is, \( \text{max supp}(\zeta_1) < \text{max supp}(\zeta_2) < \ldots < \text{max supp}(\zeta_m) \). Our oracle is summarized in Algorithm 2. The intuition behind the oracle is based on Lemma 1: \( \gamma \in \Gamma \cap F(I_0, I_1) \) if \( \text{already the restriction of } \gamma \text{ to } \Gamma \text{ to the first cycle } \zeta_1 \text{ has no lexicographically maximal element adhering to the fixings, which follows from orderliness. Otherwise, we want to use Lemma 1 to decide emptiness. As we deal with sets of fixings rather than a single vector, condition } x \in M \gamma \text{ needs to be handled carefully. Therefore, we consider certain stabilizers of } 0- \text{ and } 1- \text{ fixings. Property (P2)} \text{ is then checked via Line 4; that is, instead of checking the feasibility of the permutation's entire support, we check only the support of a single cycle.}

**Algorithm 2** (Oracle for Cyclic Groups Generated by Monotone and Ordered Permutations)

**Input:** Group \( \Gamma \leq \langle \gamma \rangle \) for monotone and ordered \( \gamma = \zeta_1 \ldots \zeta_m \in S_n \) with \( N_c = \zeta_c \) for \( c \in [m] \), sets \( I_0, I_1 \subseteq [n] \)

**Output:** The message \text{INFEASIBLE} or \text{FEASIBLE}

1. \( \Delta_1 \leftarrow \Gamma \)
2. for \( c \leftarrow 1, \ldots, m \) do
3. \( \{ f_0, f_1 \} \leftarrow \) complete fixings of \( X_{\gamma} \cap F(I_0, I_1) \) for \( \gamma \in \text{restr}(\Delta_c, N_c) \);
4. if \( X_{\text{restr}(\Delta_c, N_c)} \cap F(I_0 \cup N_c, f_1) \cap N_c = \emptyset \) then
5. \( \text{return } \text{INFEASIBLE} \)
6. if \( N_c \setminus f_0 \cup f_1 \neq \emptyset \) then
7. \( \Delta_{c+1} \leftarrow \text{stab}(f_0 \cap N_c, \Delta_c) \cap \text{stab}(f_1 \cap N_c, \Delta_c) \)
8. else
9. \( \Delta_{c+1} \leftarrow \text{STAB}(N_c, \Delta_c) \)
10. \( \text{return } \text{FEASIBLE} \)

**Theorem 3.** Let \( \gamma = \zeta_1 \ldots \zeta_m \in S_n \) be monotone and ordered, let \( I_0, I_1 \subseteq [n] \) be disjoint, and let \( \Gamma \leq \langle \gamma \rangle \). Then, Algorithm 1 can be implemented to run in \( O(n^3) \) time.

We stress the qualitative difference between Theorems 1 and 3. The former could achieve a running time of \( O(n^3) \) for groups of size \( n \), and the latter achieves the same running time even if the group is superpolynomially large in \( n \).

**Proof.** We show that Algorithm 2 correctly determines in \( O(n^2) \) time whether \( X_{\gamma} \cap F(I_0, I_1) = \emptyset \). We call the oracle \( O(n) \) times, yielding a cubic running time for Algorithm 1.

To prove that Algorithm 2 is an oracle, we show that it returns the correct statement. Suppose the algorithm returns infeasible in the first iteration. This means that \( X_{\text{restr}(\Gamma, N_1)} \cap F(I_0 \cap N_1, f_1) \cap N_1 = \emptyset \). Consequently, \( X_{\gamma} \cap F(I_0, I_1) \) is empty by Lemma 1; that is, the algorithm correctly reports infeasibility. Otherwise, if it does not report infeasibility, we might detect infeasibility in later iterations. Based on Lemma 1, we need to stabilize solutions on the first cycle. On the one hand, if \( N_1 \subseteq f_0 \cup f_1 \), all entries of lexicographically maximal vectors in \( F(I_0, I_1) \) are fixed on \( N_1 \). This means solutions are stabilized by replacing \( \Gamma \) by \( \text{stab}(f_n \cap N_1, \Gamma) \cap \text{stab}(f_1 \cap N_1, \Gamma) \). On the other hand, if \( N_1 \setminus f_0 \cup f_1 \neq \emptyset \), Proposition 2 guarantees that there is a solution on \( N_1 \) that is lexicographically larger than all of its nontrivial permutations. Hence, by Lemma 1, we can find infeasibility only in later iterations, by
considering permutations that fix all entries on $N_1$; that is, we replace $\Gamma$ by $\text{STAB}(N_1, \Gamma)$. This update is done at the end of the iteration, where $\Delta_2$ is defined.

In the following iterations, we can use exactly the same arguments to find that $\Delta_2$ stabilizes all solutions on the previous cycles. This means we can implicitly remove the previous cycles and hypothetically assume that cycle $c$ is the first cycle, where we use $\Delta_1$ instead of $\Gamma$. Consequently, if infeasibility is yielded, a correct statement is found.

It remains to show $X_\gamma \cap F(I_0, I_1) \neq \emptyset$ if the algorithm returns feasibility. We construct $x \in X_\gamma \cap F(I_0, I_1)$ via sub-vectors $x^i \in \{0, 1\}^N$ speciﬁcing each cycle $c \in [m]$. If $N_c \subseteq I_0^c \cup I_1^c$, all entries on $N_c$ are ﬁxed, and we deﬁne $x^c$ accordingly; otherwise, let $x^c \in \{0, 1\}^N$ respect the 0-ﬁxings $I_0^c \cap N_c$ and 1-ﬁxings $I_1^c \cap N_c$, and satisfy $x^c > \gamma(x^c)$ for all nontrivial $\gamma \in \text{restr}(\Delta_1, N_c)$, which exists by Proposition 2. We claim $x \geq \gamma(x)$ for all $\gamma \in \Gamma$. If the claim were wrong, there would exist $c \in [m]$ and $\gamma \in \Gamma$ such that $x$ and $\gamma(x)$ coincide on the first $c - 1$ cycles and $x^c < \gamma(x^c)$. Because $\gamma$ stabilizes the first $c - 1$ cycles, $\gamma \in \Delta_c$. Consequently, $N_c \subseteq I_0^c \cup I_1^c$, because otherwise $x^c > \gamma(x^c)$ for all $\gamma \in \Delta_c$ that do not stabilize cycle $c$ pointwise (i.e., $\text{restr}(\gamma, N_c) \neq \emptyset$), by construction. This, however, means that $X_{\text{restr}(\Delta_1, N_c)} \cap F(I_0^c, I_1^c) = \emptyset$ and Algorithm 2 steps into Line 5. A contradiction to our assumption.

It remains to prove the quadratic running time. Because deciding the emptiness of $X_{\text{restr}(\Delta_1, N_c)} \cap F(I_0, I_1)$ can be easily done once the complete sets of ﬁxings are known by Proposition 1, the critical steps are computing the complete sets of ﬁxings and ﬁnding the stabilizer subgroups. The complete sets of ﬁxings for cycle $c$ can be found in $O(|N_c|^2) \cdot \text{time}$ by Proposition 2, because $|\text{restr}(\Delta_1, N_c)| = |N_c|$. Moreover, stabilizers can also be found in $O(|N_c|^2 + n)$: The groups $\Delta_c$ can be encoded by a power $k_c \in \mathbb{Z}$ such that $\Delta_c = \left( \tau^{k_c} \right)$, for the generator $\gamma$ of $\Gamma$, such that $k_1 = 1$. To compute the subgroup that stabilizes the ﬁxings $I_0^c \cap N_c$ and $I_1^c \cap N_c$, it suﬃces to ﬁnd $\ell = \min\{p \in \mathbb{N} | k_c \equiv 0 \mod p\}$ and replace $k_c$ by $k_c / \ell$. This is possible in $O(|N_c|^2)$ time. For stabilizing $N_c$ pointwise, $k_{c+1} = \text{lcm}(k_c, |N_c|)$. Because $k_c \leq |\Gamma| < 2^n$, the asymptotic running time of computing the least common multiples is $O(2^n \cdot |N_c|) = O(n)$. Because $\sum_{c=1}^{n} (|N_c|^2 + n) \leq n\sum_{c=1}^{n} (|N_c| + 1) = n^2 + nm = O(n^2)$, the running time of Algorithm 2 is quadratic.

4. Propagation of Individual Permutations in a Set

This section provides the intuition of our proof of Theorem 2. For details, we refer the reader to the online appendix. Throughout this section, we assume $\Pi \subseteq S_m$ and that $I_0, I_1 \subseteq \{n\}$ are disjoint. To find the complete set of ﬁxings for individual permutations in $\Pi$, we can, in principle, use the iterative algorithm sketched in the beginning of Section 3; that is, we propagate each constraint $x \geq \gamma(x)$ for $\gamma \in \Pi$. If no permutation allows us to ﬁnd a new ﬁxing, we stop, and otherwise we repeat this mechanism. Even if propagating a single constraint can be done in $O(n)$ time, this algorithm has a running time of $O(n^2 |\Pi|)$, as each newly found running time might trigger another round of propagating all $\gamma \in \Pi$. To derive the $O(n |\Pi|)$ running time of Theorem 2, we will introduce suitable data structures and implications between different permutations. We make use of the following terminology.

A ﬁxing is a tuple $(i, b) \in \{0, 1\} \times \{0\} \text{ encoding that entry } i \text{ is ﬁxed to } b \text{. The converse ﬁxing of } f = (i, b) \text{ is } f = (i, 1 - b)$. A ﬁxing is valid for $\gamma \in \Pi$ and $(I_0, I_1)$ if all $x \in X_\gamma \cap F(I_0, I_1)$ have $x_i = b$. Fixing $(i, b)$ is applied if $i$ is added to $I_0$. A conﬂict is a set $C \subseteq \{0\} \times \{0\}$, if $X_\gamma \cap F(I_0^c, I_1^c) = \emptyset$ for $I_0^c = I_0 \cup \{(i, b) \in C : b = 0\}$ and $I_1^c = I_1 \cup \{(i, b) \in C : b = 1\}$. That is, applying all ﬁxings in $C$ turns $X_\gamma \cap F(I_0, I_1)$ infeasible. Also the following type of conﬂicts will be important. Let $k \in \{n + 1\}$, $x \in \{0, 1\}^n$, and $\gamma \in \Pi$. Note that $x \prec_k \gamma(x)$ implies $x \prec \gamma(x)$, and both statements coincide if $k = n + 1$. A k-conﬂict for $\gamma$ is a conﬂict $C$ such that all $x \in F(I_0, I_1)$ with $x_i = b$ for $(i, b) \in C$ have $x \prec_k \gamma(x)$.

The core of our algorithm to prove Theorem 2 is a careful analysis of conﬂicts for $\gamma \in \Pi$. Before we present our algorithm, we provide some intuition.

Example 3. Let $\zeta = (1, 6, 8, 4, 7, 2, 5)$, $\varphi = (1, 5)(2, 6)(3, 7)(4, 8)$, and $\Pi = \{\zeta, \varphi\}$, and let $I_0 = \{4, 6\}$ and $I_1 = \{5\}$ be encoded by $x = \langle \omega, \omega, \omega, 0, 1, 0, \omega, \omega \rangle$, where $\omega$ represents an unﬁxed entry. To ﬁnd further entries that can be ﬁxed, we construct k-conﬂicts for individual $\gamma \in \Pi$ and increasing values of $k$. If we ﬁnd a conﬂict $\{(i, b)\}$, then $i = 1 - b$ is valid. Figure 1 illustrates our discussion.

Select $\zeta$ and set $k = 2$. Then, $\{(1, 0)\}$ is a 2-conﬂict for $\zeta$ because $x_1 \leftarrow 0$ yields $x \prec_2 \zeta(x)$. Hence, we can safely apply ﬁxing $(1, 1)$. No 3-conﬂicts (respectively, 4-conﬂicts) of size one exist for $\zeta$ as $x_2, x_7$ are both are unﬁxed (respectively, 3 is a ﬁxed point). Checking for 5-conﬂicts, we have $(x_4, x_9) = (0, \_)$ if the values of $x_2$ and $x_7$ are the same, $x_8$ must become zero; otherwise, $x \prec_5 \zeta(x)$. For $k = 6$, we encounter $(x_4, x_9) = (1, \_)$ in this case, if $x$ and $\zeta(x)$ are equal up to entry 5 and $x_9 = 0$, then no 6-conﬂict for $\zeta$ with cardinality one exists. Otherwise, if $x_2 = 1$, we can continue. For $k = 7$, we ﬁnd $(x_6, x_1) = (0, 1)$, which means that $x \prec_7 \zeta(x)$ if for all entries $i < 6$, we have that the value of $x_i$ is the same as $\zeta(x)$. If $x_7 = 1$, to ensure $x \prec_7 \zeta(x)$, we must have $x_2 = 1$ and $x_8 = 0$, but in that case, $x \prec_7 \zeta(x)$, so $\{(7, 1)\}$ is a 7-conﬂict for $\zeta$. Hence, apply ﬁxing $(7, 0)$.  

van Doornmalen and Højny: Efficient Propagation Techniques for Cyclic Symmetries
INFORMS Journal on Computing, 2024, vol. 36, no. 3, pp. 868–883, © 2024 INFORMS
Applying similar steps to \( \varphi \) does not lead to further fixings. Indeed, assigning \( x_2 \leftarrow 0 \) and \( x_3 \leftarrow 1 \) yields \( x >_3 \varphi(x) \), and if \( x_2 \leftarrow 1 \), then \( x >_3 \varphi(x) \).

The example carefully considers \( k \)-conflicts for a permutation where \( k \) is increased by one in each iteration. If a conflict consisting of one fixing is found, an appropriate fixing is applied. This idea is formalized in Algorithm 3.

**Algorithm 3** (Determine the Complete Set of Fixings for Each Individual Constraint \( x \geq \gamma(x) \) for all \( \gamma \in \Pi \))

**Input:** Set of permutations \( \Pi \subseteq S_n \), and initial set of fixings \( (I_0, I_1) \)

**Output:** INFEASIBLE, or FEASIBLE and the set of fixings that is complete for each individual permutation in \( \Pi \).

1. if \( F(I_0, I_1) = \emptyset \) then return INFEASIBLE
2. \( t \leftarrow 0; (I_0', I_1') \leftarrow (I_0, I_1) \);
3. foreach \( \gamma \in \Pi \) do \( i_\gamma \leftarrow 1 \);
4. while there is a \( \gamma \in \Pi \) not satisfying sufficient conditions for completeness do
5. \( i_\gamma \leftarrow i_\gamma + 1; t \leftarrow t + 1; \)
6. repeat
7. if there is a \( \delta \in \Pi \) with \( i_\delta \)-conflict \( \emptyset \) then return INFEASIBLE
8. else if there is a \( \delta \in \Pi \) with \( i_\delta \)-conflict \((i, b)\) and \( i \not\in I_{1-b} \) then
9. \[ \text{Apply } (i, 1-b): (I_{b+1}'', I_{1-b}'') \leftarrow (I_{b}', I_{1-b}' \cup \{i\}); t \leftarrow t + 1; \]
10. else break “repeat”-loop
11. return FEASIBLE, \((I_0', I_1')\);

For the ease of analysis, we maintain a time stamp \( t \) for the sets \( I_0 \) and \( I_1 \) at iteration \( t \). We iterate over all permutations and construct \( k \)-conflicts for increasing values of \( k \). For each \( \gamma \in \Pi \), the algorithm keeps track of a counter \( i_\gamma \) to remember that \( i_\gamma \)-conflicts have been computed. If a conflict of size one is found, the algorithm applies the converse fixing, and conflicts of size zero immediately show that \( X_{\Pi} \cap F(I_0, I_1) \) is empty. The algorithm terminates if it knows that the complete set of fixings for individual permutations \( \gamma \in \Pi \) has been found. This is achieved via checking the sufficient conditions for completeness of a permutation \( \gamma \in \Pi \). This is a set of conditions that, if satisfied, guarantee that no further fixings can be derived from \( \gamma \in \Pi \). In the online appendix, we specify what these sufficient conditions are, show how to test for the existence of \( i_\gamma \)-conflicts, and prove correctness. Moreover, if appropriate data structures are used, we show that the algorithm terminates, and that executing each line of the algorithm is asymptotically a constant-time operation. Because at most \( n \) different fixings can be applied before infeasibility is yielded, and because one sufficient condition for completeness is \( i_\gamma > n \), the number of iterations of the inner loop is worst-case \( O(n|\Pi|) \), which yields the claimed running time.

### 5. Computational Results

This section investigates the practical performance of Algorithm 1. We are particularly interested in the following questions for cyclic groups \( \Gamma \) generated by monotone and ordered permutations:

Q1. How quickly can binary programs be solved when handling symmetries by Algorithm 1 compared with propagating the individual constraints \( x \geq \delta(x) \) for all \( \delta \in \Gamma \)?

Q2. Removing the “foreach” loop in Algorithm 1 reduces the running time to \( O(n^2) \) for groups generated by a monotone and ordered permutation. But then Algorithm 1 prunes only nodes not admitting lexicographically
maximal solutions. To fix at least some variables (without guarantee of completeness), we can impose the fixings of sets \( J^c_1 \) found by Algorithm 2 for individual cycles \( c \). Are binary programs solved more efficiently using the complete Algorithm 1 or the modified version?

Q3. How do the dedicated propagation methods for cyclic groups compare with the state-of-the-art methods isomorphism pruning and orbital fixing?

Questions Q1 and Q2 are meaningful only if \(|\Gamma| > 2\), as otherwise there is no difference between Algorithm 1 and propagation of an individual constraint. To answer these questions, we have implemented our methods in the Solving Constraint Integer Programming (SCIP) (Bestuzheva et al. 2021). SCIP already provides methods to compute symmetries of a mixed-integer program (MIP), and it enforces \( x \geq \gamma(x) \) for a symmetry \( \gamma \) of an MIP via propagation and separation methods for so-called symresack and orbisack constraints; see Hojny and Pfetsch (2019). If the symmetry group of an MIP is a product group and one of its factors defines an action associated with a certain symmetric group, SCIP applies orbital fixing (see Kaibel et al. 2011, Bendotti et al. 2021) to handle the action of the entire factor. We have extended this code by a plug-in that implements the enforcement of \( x \geq \delta(x) \) for all \( \delta \in \langle \gamma \rangle \). This plug-in uses Algorithm 1 if the representation of \( \gamma \) is monotone and ordered. Otherwise, we use Algorithm 3, where \( \Pi \) consists of all nonidentity group elements of \( \langle \gamma \rangle \).

To answer Question Q3, we compare the dedicated methods for cyclic groups to isomorphism pruning enhanced with orbital fixing. In our comparison, we use two variants of isomorphism pruning, one in which we used a branching rule based on a fixed variable order (Margot 2002) and one in which the lexicographic order for comparisons is adapted to the branching decisions (Ostrowski 2009), as discussed in Section 2.1.

5.1. Computational Setup

We extended the MIP framework SCIP 8.0.2 (release version, git hash 5f0473c) with our symmetry code. Our code is publicly available at GitHub\(^a\) and van Doornmalen and Hojny (2023). As the linear programming solver we use CPLEX 20.1.0.0. SCIP detects symmetries of an MIP by building an auxiliary graph (Salvagnin 2005, Margot 2010, Pfetsch and Rehn 2019) and computing its automorphism group \( \Gamma \) using bliss 0.77 (Junttila and Kaski 2015). This way, we find permutations \( \gamma_1, \ldots, \gamma_k \) that generate \( \Gamma \). SCIP checks whether \( \Gamma \) is a product group \( \Gamma_1 \otimes \cdots \otimes \Gamma_r \), because the symmetries of each factor \( \Gamma_i \) can be handled independently (cf. Hojny and Pfetsch 2019).

Based on the generators, SCIP heuristically decides whether a factor \( \Gamma_i \) can be completely handled by orbital fixing (cf. Hojny and Pfetsch 2019). The remaining factors \( \Gamma_i \) are handled by selecting a subset of permutations \( \Pi \subseteq \Gamma_i \) depending on one of the following settings, where \( \gamma \in \Pi \):

- nosym: No symmetry handling is applied (also no orbital fixing).
- gen: Only propagate constraint \( x \geq \gamma(x) \) individually.
- group: Propagate the individual constraints \( x \geq \delta(x) \) for all \( \delta \in \langle \gamma \rangle \) \{id\}.
- weak: Propagate \( x \geq \delta(x) \) for all \( \delta \in \langle \gamma \rangle \) \{id\} using the modified Algorithm 1.
- strong: Propagate \( x \geq \delta(x) \) for all \( \delta \in \langle \gamma \rangle \) \{id\} using Algorithm 1.

In the last two settings, the oracle uses Algorithm 2 if \( \gamma \) is monotone and ordered, and Algorithm 3 otherwise. Completeness of Algorithm 2 is guaranteed in the strong setting only when \( \gamma \) is monotone and ordered. Because \( \gamma \in \Pi \) does not satisfy this requirement in general, we implemented a simple heuristic to relabel the variables. The heuristic iterates over all \( \gamma \in \Pi \), and if none of its affected variables have been relabeled yet, it relabels these to make the generator monotone and ordered. For instance, for \( \Pi = \{\gamma_1, \gamma_2, \gamma_3\} \) with \( \gamma_1 = (1, 8, 7, 3) \), \( \gamma_2 = (3, 4, 5, 8) \), and \( \gamma_3 = (2, 5, 6, 9, 4) \), the heuristic relabels \( \gamma_1 \) does not relabel \( \gamma_2 \) because it also affects variable 3, and relabels \( \gamma_3 \). This results in permutations \( \gamma'_1 = (1', 2', 3', 4') \), \( \gamma'_2 = (5', 6', 7', 8') \), and \( \gamma'_2 = (4', 9', 6', 2') \), where a prime indicates relabeled variable space. We implemented three variants of the heuristic that sorts each permutation’s cycles nonincreasingly by cycle length (max), nondecreasingly by cycle length (min), increasingly in the minimum element of each cycle based on original variable order (respect). Moreover, we sort the generators nonincreasingly w.r.t. the largest subcycle size and then nonincreasingly w.r.t. \( |\langle \gamma \rangle| \). We also compare it to the original variable labeling (original).

Recall that cyclic groups can have superpolynomial size. As Algorithm 3 with setting group allocates memory for each permutation, we limit the number of considered permutations from \( \langle \gamma \rangle \) in that case. Let \( s = |\text{supp}(\gamma)| \). If \( |\langle \gamma \rangle| \geq 10^6 \) or \( s|\langle \gamma \rangle| > 5 \cdot 10^6 \), we only propagate symmetries for \( S := \{\gamma^j : j \in [k]\} \), where \( k \in \mathbb{Z}_+ \) is maximal such that \( |S| \leq 10^4 \) and \( s|S| \leq 5 \cdot 10^4 \).

For isomorphism pruning and orbital fixing, we use the implementation by Pfetsch and Rehn (2019) that builds upon SCIP as a branch-and-bound framework. We thank Marc Pfetsch for providing us the latest version of this code, which we further adapted to be compatible with SCIP 8.0.2. The code for isomorphism pruning and orbital fixing uses PermLib 0.2.9 (Rehn and Schürmann 2010) for group-theoretical computations. Because symmetry reductions depend on the branch-and-bound tree structure, this method is not compatible with (tree)
restarted in SCIP, so we have disabled those. Moreover, the incompatible presolver reductions reported by Pfetsch and Rehn (2019) are also disabled in our experiments. Because isomorphism pruning can be costly, Pfetsch and Rehn (2019) consider different variants. The most competitive variant according to their results disables isomorphism pruning and orbital fixing in a node of the B&B tree and all its descendants if the symmetry group used for deriving symmetry reductions by any of these methods becomes trivial. They refer to this setting as the “no subtree” variant, and we refer the reader to Pfetsch and Rehn (2019) for more details. Preliminary runs on our test sets show that for the aggregated results, the “no subtree” setting are superior over the variant that handles symmetries in every node. This is consistent with the findings of Pfetsch and Rehn (2019). As such, in the following, all results are for the “no subtree” setting. We report results for two variants of isomorphism pruning and orbital fixing:

- isopr mib: This always branches on the unfixed variable with minimal index (Margot 2002);
- isopr: The standard branching rule of SCIP is used.

The running time $t_i$ per instance $i$ is reported in shifted geometric mean $\prod_{i=1}^{n} (t_i + s)^{1/2} - s$ with shift $s = 10$ s. The number of instances solved within the time limit of two hours per instance is denoted by $(S)$. If the time limit is hit, we report a running time of two hours for that run. We will compare the average results of all test instances for the various relabeling heuristics and the aggregated results. Instances ran in parallel on a Linux cluster with Intel Xeon Platinum 8260 processors, reserving one thread and 10.6 GB of memory per instance. Performance variability was tackled by running all instances with five different random seeds.

### 5.2. Flower Snarks

Flower snark graphs (Isaacs 1975, Fiorini and Wilson 1977), described in Figure 2, are undirected graphs with chromatic index 4; that is, there is no edge coloring with three colors such that incident edges are colored differently. Deciding whether an edge coloring with three colors exists for an undirected graph $\mathcal{G} = (V, E)$ is equivalent to deciding whether

$$S_\mathcal{G} = \left\{ x \in \{0,1\}^{E[3]} : x_{e,k} + x_{e',k} \leq 1 \text{ for } k \in [3], \ e, e' \in E \text{ with } |e \cap e'| = 1, \ \sum_{k \in [3]} x_{e,k} = 1 \text{ for all } e \in E \right\}$$

is empty, which can be done by binary programming. Margot (2007) studied this binary program for the flower snarks $J_5$, $J_{15}$, and $J_{21}$, and we also use these snarks in our experiments as they admit cyclic symmetries. Flower snarks are defined for odd $n \geq 3$ and have an automorphism group of order $4n$. The group is generated by a cycle $\gamma$ of order $2n$ and a reflection. Symmetries in the problem are therefore given by these graph automorphisms, and by interchanging the edge colors. These symmetries are automatically identified by SCIP and are the permutations included in the set $\Pi_1$ (defining a single factor $\Gamma_1 = \Gamma$).

Our test set consists of all instances with odd parameters $n \in \{3, \ldots, 49\}$ for the four different labeling strategies described above. We regard each relabeled instance as a distinct instance, as our aim is not primarily to solve the coloring problem for a specific graph, but to compare the effects of different symmetry handling methods. In Table 1, we present the aggregated results for these instances. In comparison with setting nosym, applying any

---

**Figure 2.** Flower Snark Graph $J_5$ and Construction of $J_n$

Notes. Graph $J_n$ is defined for odd $n \geq 3$ and has $4n$ vertices labeled by $a_i, b_i, c_i, d_i$ for $i \in [n]$. For $i \in [n]$, connect $a_i$ to $b_{i+1}$, $c_i$, $d_i$; make cycle $(b_1, b_2, \ldots, b_n)$; and, for $i < n$, connect $c_i$ to $d_{i+1}$, and $d_i$ to $c_{i+1}$. Last, connect $c_n$ to $c_1$ and $d_n$ to $d_1$. The automorphism group generators are $(a_1, \ldots, a_n)(b_1, \ldots, b_n)(c_1, \ldots, c_n, d_1, \ldots, d_n)$, and $(c_1, d_1) \cdot \prod_{i=1}^{n-1}(a_i, a_{i+2}, \ldots, d_i, d_{i+2}, \ldots, d_n, d_{i+2}, \ldots, d_n)$. 

Table 1. 3-Edge Coloring on Flower Snark Graphs $J_n$ for Odd $n \in \{3, \ldots, 43\}$

| Relabeling   | Nosym | gen | group | weak | strong | isopr mib | isopr |
|--------------|-------|-----|-------|------|--------|----------|-------|
| Original     | 912.12| 53  | 260.10| 89   | 301.88 | 79       | 261.46| 86   |
| Max          | —     | —   | —     | —    | —      | —        | —     | —    |
| Min          | —     | —   | 148.89| 103  | 217.85 | 87       | 188.83| 97   |
| Respect      | —     | —   | 257.54| 82   | 253.48 | 82       | 246.39| 84   |
| Aggregated   | 912.12| 44% | 275.89| 70%  | 289.59 | 66%      | 263.93| 71%  |
| Compared to group | +215% | -33% | -5%  | +6%   | —      | —        | -9%   | +7%  |

form of symmetry handling substantially decreases the running time and allows us to solve many more instances. We can also see that the labeling of the cycles of the generator $\gamma$ has a big impact on the running time. For the original labeling, gen and our settings weak and strong are comparable; for the min labeling, gen is faster; and for the remaining max and respect labeling, weak and strong beat setting gen. Averaging all labeling strategies, our algorithms in settings weak and strong are roughly 4.6% faster than gen.

At first glance, an improvement of 4.6% seems rather small as the flower snark instances have a large cyclic subgroup. Besides cyclic symmetries, however, all settings (except nosym) also handle reflection symmetries. A possible explanation for the relatively small improvement of strong and weak in comparison with gen is that handling reflections yields further variable fixings that can be exploited by propagating the single constraint $x \geq \gamma(x)$. To support this hypothesis, we conducted a small experiment. We disable handling of reflections in all symmetry handling settings and rerun the experiments for all relabeling strategies and a single seed. The different settings then achieve mean running times of 860.56s (nosym), 288.72s (gen), 299.02s (group), 251.3s (weak), and 241.38s (strong). Settings weak and strong are thus by 13.4% and 16.4%, respectively, faster than gen. We conclude that weak and strong are substantially faster than gen if the dominating symmetry is a cyclic symmetry that does not interact with other symmetries.

Turning to Question Q1, we see that individually handling the constraints $x \geq \gamma(x), \gamma \in \langle \gamma \rangle$ is considerably slower for all tested relabeling strategies. This effect is even more pronounced if we consider only the more difficult instances in our test set (each setting needed at least 10s to solve such an instance). Table 2 reports on these experiments. On average, weak and strong are faster than group by 11.8% and 12.0%, respectively. Thus, when the size of the cyclic group gets larger, exploiting the cyclic group structure via our novel algorithms in settings weak and strong is important. Regarding Q2, neither weak nor strong dominates the other strategy; on average, they perform comparably. This might indicate that for flower snark instances, weak already finds many variable fixings and a complete algorithm is not needed in this case. Reflecting on Q3, isopr mib is inferior to every other variant that handles symmetries, which might be due to an unfortunate variable indexing of these well-structured instances. The isopr setting does not rely on the variable indexing in this way, and performs significantly better. Although isomorphism pruning leads to a great improvement in comparison with no symmetry handling, it is substantially slower than our dedicated methods for handling symmetries of cyclic groups. We thus conclude that developing dedicated tools for specific group structures can lead to substantial improvements of state-of-the-art methods.

5.3. MIPLIB Test Set

In the introduction, we mentioned that many instances from MIPLIB 2010 contain unknown symmetry groups. In the following, we investigate whether handling symmetries of cyclic groups solves these instances more
efficiently than just handling symmetries of generators. Therefore, we extracted all instances from MIPLIB 2010 and MIPLIB 2017 (Gleixner et al. 2021) for which SCIP finds symmetries. Only for 38 of these instances, SCIP reports a generator \( \gamma \) for which \( |\langle \gamma \rangle| > 2 \), that is, nontrivial cyclic groups. Because only 10 of these instances are solvable by any of our settings within two hours, we identified more cyclic groups as follows. Let \( \Pi' \) be the generators found by SCIP that are not handled by orbitopal fixing. We compute \( \Pi'' = \Pi \cup \{ \gamma \circ \delta : \gamma, \delta \in \Pi \} \) and handle all \( \gamma' \in \Pi'' \) with \( |\langle \gamma' \rangle| > 2 \) as cyclic groups. This results in 169 instances. We left out the (solvable) instance supportcase29 because of memory overflow in the group setting. Regarding Q1, this shows that for large cyclic groups, our algorithms are essential.

Table 3 shows the results of the experiments on all 168 instances, run with five seeds each, totaling to 840 runs per setting and relabeling heuristic. Many instances are not solvable by any setting, and nonsolved instances are counted as running for 7,200 s, meaning that the effect of symmetry handling is thus demagnified. Moreover, in the isopr mib setting, there were nine runs that reached the memory limit. These instances are reported as not solved and using the time limit. Regarding Q3, this illustrates the potential expensive and memory-intensive group-theoretical computations required by these methods.

Table 4 presents the results restricted to the 89 solvable instances. Regarding Q1, our weak and strong variants achieve better mean running times than group for all relabeling strategies, and, on average, achieve improvements of 6.5\% and 5.5\%, respectively. Regarding Q2, it seems that strong performs slightly worse than weak. A possible explanation is that, in contrast to flower snark instances, there are many cyclic groups in MIPLIB instances. Thus, not necessarily all generators of cyclic groups can be labeled in an ordered and monotone way; in particular, the strong setting is not guaranteed to be complete for all cyclic groups. The additional time spent in the strong setting does thus not pay off for the diverse MIPLIB instances.

Settings weak and strong improve upon gen by 4.3\%–5.3\%. Although this improvement does not seem overwhelming at first glance, we believe that it is a substantial improvement of gen for the following reason: the generators of all newly found cyclic groups have cycles of length at most six; that is, the cyclic groups are rather small. The impact of handling only the generator in setting gen is thus relatively high, whereas we hypothesize that for larger cyclic groups, the improvements of weak and strong should be even more pronounced.

Regarding Q3, we observe that both isopr mib and isopr perform, on average, significantly worse than the remaining settings, even worse than nosym. The latter might have different explanations. On the one hand, this might be due to the necessary disabling of presolvers and restarts. To verify that this is not due to restarts, we disabled restarts in the nosym setting and found an average of 1,586.75 s and solves 322 instances, which is still significantly faster than isopr mib and isopr. On the other hand, a possible explanation for isomorphism pruning being slower is that symmetry handling is not necessarily the key component for solving MIPLIB instances, which is in contrast to the flower snark instances. In particular, for MIPLIB, it might be possible that

| Table 3. Relevant MIPLIB 2010 and MIPLIB 2017 Instances |
|----------------------------------------------------------|
| Relabeling     | nosym Time (s) | gen Time (s) | group Time (s) | weak Time (s) | strong Time (s) | isopr mib Time (s) | Isopr Time (s) |
|----------------|----------------|--------------|---------------|--------------|----------------|------------------|----------------|
| Original       | 1,468.97       | 333          | 1,119.75      | 404          | 1,217.35       | 400              | 1,170.45        |
| Max            | —              | —            | 1,298.85      | 380          | 1,285.68       | 393              | 1,231.29        |
| Min            | —              | —            | 1,343.39      | 379          | 1,324.89       | 383              | 1,297.61        |
| Respect        | —              | —            | 1,246.48      | 397          | 1,238.44       | 390              | 1,196.39        |
| Aggregated     | 1,468.97       | 40%          | 1,257.53      | 46%          | 1,265.91       | 47%              | 1,223.02        |
| Compared to group | +16%        | -15%         | -1%           | 0%           | -3%            | +1%              | +60%           |

| Table 4. Relevant MIPLIB 2010 and MIPLIB 2017 Solvable Instances |
|----------------------------------------------------------|
| Relabeling     | nosym Time (s) | gen Time (s) | group Time (s) | weak Time (s) | strong Time (s) | isopr mib Time (s) | Isopr Time (s) |
|----------------|----------------|--------------|---------------|--------------|----------------|------------------|----------------|
| Original       | 364.69         | 333          | 228.01        | 404          | 254.55         | 400              | 235.98          |
| Max            | —              | —            | 288.29        | 380          | 282.70         | 393              | 260.18          |
| Min            | —              | —            | 307.51        | 379          | 299.44         | 383              | 287.74          |
| Respect        | —              | —            | 266.40        | 397          | 263.11         | 390              | 246.17          |
| Aggregated     | 364.69         | 74%          | 270.95        | 87%          | 274.42         | 87%              | 256.83          |
| Compared to group | +33%        | -15%         | -1%           | -0%          | -6%            | +1%              | -5%            | +144%           | +34%           | +114%           | -27%           |
many nodes are needed in a B&B tree to find an optimal solution or the right cutting plane. In this case, isomorphism pruning might be computationally too expensive, whereas our dedicated methods for cyclic groups run in polynomial time. That is, the symmetry reductions that can be detected by isomorphism pruning for MIPLIB instances do not compensate for its long running time.

6. Conclusion
The algorithms developed in this article allow us to efficiently handle symmetries of cyclic groups. If the generator of the cyclic group is monotone and ordered, we provide a cubic-time algorithm (strong) that guarantees to find all symmetry-based variable fixings that can be derived from a set of given fixings. We also provide a weaker but faster (quadratic-time) variant (weak) that does not come with a guarantee of completeness.

As illustrated by the flower snark and MIPLIB test sets, our algorithms clearly dominate the iterated propagation of state-of-the-art symmetry handling methods, both constraint-based (gen and group) and isomorphism pruning and orbital fixing (isopr mib and isopr). Although strong has a higher computational cost than weak, we observe an improvement of the required running time for the well-structured flower snark instances. Whereas flower snark instances only have one cyclic generator, the MIPLIB instances might admit several cyclic generating permutations that cannot be made monotone and ordered simultaneously. On the more diverse MIPLIB test set, we observe that our weaker weak implementation outperforms strong. Apparently, the higher computational cost does not always outweigh the potential of finding more fixings for such instances. We thus conclude that the dedicated methods that exploit the structure of cyclic groups developed in this article clearly help to improve the performance of symmetry handling in SCIP.

The numerical results also show interesting possibilities for future research. On the one hand, it is important to identify which instances benefit from (aggressive) symmetry handling. On the other hand, because completeness of our methods is guaranteed only for (sub)groups generated by monotone and ordered permutations γ, it would be helpful to derive methods that achieve completeness even if one of the assumptions on γ is dropped. In the experiments, we ensured these conditions by relabeling the solution vectors on which the symmetry handling constraints acted. It turns out, for instance, for the flower snark instances, that this chosen relabeling highly impacts the required running time of SCIP. Alternatively, determining what a good relabeling could be and finding sophisticated relabeling methods could be beneficial. This, however, is out of scope of this paper.

Acknowledgments
The authors thank Rudi Pendavingh, Marc E. Pfetsch, and Frits Spieksma for carefully reading a preliminary version of the paper and their suggestions to improve its presentation. The authors also thank Rob Eggermont for fruitful discussions about permutation groups and anonymous referees for their valuable feedback.

Endnotes
1 To ensure that γ(x) defines a group action, that is, (γ′γ′′)(x) = γ′(γ′′(x)), permutation γ−1 is required in the index set.
2 This might be the case if, for example, γ has multiple disjoint cycles of pairwise distinct prime lengths.
3 See the GitHub repository at https://github.com/JasperNL/scip-symmetope.

References
Babai L, Luks EM (1983) Canonical labeling of graphs. Proc. 15th Annual ACM Sympos. Theory Comput. (Association for Computing Machinery, New York), 171–183.

Bendotti P, Fouilhoux P, Rotten C (2021) Orbitopal fixing for the full (sub-)orbitope and application to the unit commitment problem. Math. Programming 186:337–372.

Bestuzheva K, Besançon M, Chen WK, Chiola A, Donkiewicz T, van Doornmalen J, Eifler L, et al. (2021) The SCIP optimization suite 8.0. Preprint, submitted December 16, http://www.optimization-online.org/DB_HTML/2021/12/8728.html.

Crawford J, Ginsberg M, Luks E, Roy A (1996) Symmetry-breaking predicates for search problems. Proc. Fifth Internat. Conf. Knowledge Representation Reasoning (Morgan Kaufman, Burlington, MA), 148–159.

Fiorini S, Wilson RJ (1977) Edge-Colourings of Graphs. Research Notes in Mathematics No. 16 (Pitman Publishing Limited, Lanham, MD).

Fischetti M, Liberti L (2012) Orbital shrinking. Mahjoub AR, Markakis V, Milis I, Paschos VT, eds. Combinatorial Optimization. Lecture Notes in Computer Science, vol. 7422 (Springer, Berlin), 48–58.

Friedman EJ (2007) Fundamental domains for integer programs with symmetries. Dress A, Xu Y, Zhu B, eds. Combinatorial Optimization and Applications. Lecture Notes in Computer Science, vol. 4616 (Springer, Berlin), 146–153.

Gleixner A, Hendel G, Gamrath G, Achterberg T, Bastubbe M, Berthold T, Christophel PM, et al. (2021) MIPLIB 2017: Data-driven compilation of the 6th Mixed-Integer Programming Library. Math. Programming Comput. 13:443–490.

Hojny C (2020) Packing, partitioning, and covering symmetries. Discrete Appl. Math. 283:689–717.

Hojny C, Pfetsch ME (2019) Polytopes associated with symmetry handling. Math. Programming 175:197–240.

Isaacs R (1975) Infinite families of nontrivial trivalent graphs which are not Tait colorable. Amer. Math. Monthly 82(3):221–239.
Junttila T, Kaski P (2015) bliss: A tool for computing automorphism groups and canonical labelings of graphs. http://www.tcs.hut.fi/Software/bliss/.

Kabel V, Pfetsch ME (2008) Packing and partitioning orbitopes. Math. Programming 114(1):1–36.

Kabel V, Peinhardt M, Pfetsch ME (2011) Orbital fixing. Discrete Optim. 8(4):595–610.

Katsirelos G, Narodytska N, Walsh T (2009) Breaking generator symmetry. Preprint, submitted September 28, https://arxiv.org/abs/0909.5099.

Katsirelos G, Narodytska N, Walsh T (2010) On the complexity and completeness of static constraints for breaking row and column symmetry. Cohen D, ed. Principles and Practice of Constraint Programming (Springer, Berlin), 305–320.

Koch T, Achterberg T, Andersen E, Bastert O, Berthold T, Bixby RE, Danna E, et al. (2011) MIPLIB 2010. Math. Programming Comput. 3(2):103–163.

Land AH, Doig AG (1960) An automatic method of solving discrete programming problems. Econometrica 28(3):497–520.

Liberti L (2008) Automatic generation of symmetry-breaking constraints. Yang B, Du DZ, Wang CA, eds. Combinatorial Optimization and Applications. Lecture Notes in Computer Science, vol. 5165 (Springer, Berlin), 328–338.

Liberti L (2012a) Reformulations in mathematical programming: Automatic symmetry detection and exploitation. Math. Programming 131(1–2):273–304.

Liberti L (2012b) Symmetry in mathematical programming. Lee J, Leyffer S, eds. Mixed Integer Nonlinear Programming. IMA Volumes in Mathematics and its Applications, vol. 154 (Springer, New York), 236–286.

Liberti L, Ostrowski J (2014) Stabilizer-based symmetry breaking constraints for mathematical programs. J. Global Optim. 60:183–194.

Loos A (2010) Describing orbitopes by linear inequalities and projection based tools. Unpublished PhD thesis, Otto-von-Guericke-Universität Magdeburg, Magdeburg, Germany.

Margot F (2002) Pruning by isomorphism in branch-and-cut. Math. Programming 94(1):71–90.

Margot F (2003) Exploiting orbits in symmetric ILP. Math. Programming 98(1–3):3–21.

Margot F (2007) Symmetric ILP: Coloring and small integers. Discrete Optim. 4(1):40–62.

Margot F (2010) Symmetry in integer linear programming. Jünger M, Liebling TM, Naddef D, Nemhauser GL, Pulleyblank WR, Reinelt G, Rinaldi G, Wolsey LA, eds. 50 Years of Integer Programming 1958–2008 (Springer, Berlin), 647–686.

Narodytska N, Walsh T (2013) Breaking symmetry with different orderings. Schulte C, ed. Principles and Practice of Constraint Programming. Lecture Notes in Computer Science, vol. 8124 (Springer, Berlin), 545–561.

Ostrowski J (2009) Symmetry in integer programming. Unpublished doctoral dissertation, Lehigh University, Bethlehem, PA.

Ostrowski J, Anjos MF, Vannelli A (2015) Modified orbital branching for structured symmetry with an application to unit commitment. Math. Programming 150(1):99–129.

Ostrowski J, Linderoth J, Rossi F, Smriglio S (2011) Orbital branching. Math. Programming 126(1):147–178.

Pfetsch ME, Rehn T (2019) A computational comparison of symmetry handling methods for mixed integer programs. Math. Programming Comput. 11(1):37–93.

Rehn T, Schürmann A (2010) C++ tools for exploiting polyhedral symmetries. Fukuda K, Hoev Jvd, Joswig M, Takayama N, eds. Mathematical Software. Lecture Notes in Computer Science, vol. 6327 (Springer, Berlin), 295–298.

Salvagnin D (2005) A dominance procedure for integer programming. Unpublished Master’s thesis, University of Padova, Padova, Italy.

Salvagnin D (2018) Symmetry breaking inequalities from the Schreier-Sims table. van Hoeve WJ, ed. Integration of Constraint Programming, Artificial Intelligence, and Operations Research (Springer International Publishing, Cham, Switzerland), 521–529.

van Doornmalen J, Hojny C (2023) Code repository for “Efficient propagation techniques for handling cyclic symmetries in binary programs.” https://dx.doi.org/10.5281/zenodo.10182549.