Symmetric Core Learning for Pseudo-Boolean Optimization by Implicit Hitting Sets

Hannes Ihalainen
University of Helsinki, Finland

Jeremias Berg
University of Helsinki, Finland

Matti Järvisalo
University of Helsinki, Finland

Bart Bogaerts
KU Leuven, Leuven, Belgium
Vrije Universiteit Brussel, Brussels, Belgium

Abstract

We propose symmetric core learning (SCL) as a novel approach to making the implicit hitting set approach (IHS) to constraint optimization more symmetry-aware. SCL has the potential of significantly reducing the number of iterations and, in particular, the number of calls to an NP decision solver for extracting individual unsatisfiable cores. As the technique is focused on generating symmetric cores to the hitting set component of IHS, SCL is generally applicable in IHS-style search for essentially any constraint optimization paradigm. In this work, we focus in particular on integrating SCL to IHS for pseudo-Boolean optimization (PBO), as earlier proposed static symmetry breaking through lex-leader constraints generated before search turns out to often degrade the performance of the IHS approach to PBO. In contrast, we show that SCL can improve the runtime performance of a state-of-the-art IHS approach to PBO and generally does not impose significant overhead in terms of runtime performance.

2012 ACM Subject Classification Mathematics of computing → Combinatorial optimization; Theory of computation → Constraint and logic programming

Keywords and phrases Implicit hitting sets, symmetries, unsatisfiable cores, pseudo-Boolean optimization

Digital Object Identifier 10.4230/LIPIcs.CP.2025.33

Supplementary Material *Software*: `https://doi.org/10.5281/zenodo.15630156`

Funding *Hannes Ihalainen*: Partially funded by Research Council of Finland under grant 356046
Jeremias Berg: Research Council of Finland under grant 362987
Matti Järvisalo: Partially funded by Research Council of Finland under grant 356046
Bart Bogaerts: Partially funded by Fonds Wetenschappelijk Onderzoek – Vlaanderen (project G064925N) and by the European Union (ERC, CertiFOX, 101122653). Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Research Council. Neither the European Union nor the granting authority can be held responsible for them.

1 Introduction

Symmetries are intrinsically present in various types of computationally hard decision and optimization problems, ranging from, e.g., scheduling and timetabling through the resolution of mathematical conjectures by automated reasoning to the analysis of systems broadly construed. As automated reasoning and constraint optimization solvers constitute the de facto approach in such settings, different ways of making solvers more symmetry-aware

have been developed: techniques that break symmetries in a preprocessing phase [2, 21, 3], techniques that learn such breaking constraints on the fly [29, 27, 45], techniques that guarantee asymmetric branching inside a solver [35], and techniques that extend the conflict learning mechanism applied during search within a solver [7, 20]. All these techniques aim to avoid unnecessary exploration of symmetric parts of search spaces underlying declarative models of real-world problems.

In this work, we propose symmetric core learning (SCL) as a novel approach to making the implicit hitting set approach (IHS) [33, 34, 30, 38] to constraint optimization more symmetry-aware. Our motivation for focusing on IHS is two-fold. Firstly, IHS offers a generic framework for developing constraint optimization solvers; IHS has been shown to yield efficient solvers for various constraint optimization languages, including maximum satisfiability (MaxSAT) [14, 15, 16, 36], MaxSAT modulo theories [12, 23], quantified MaxSAT [25, 31] answer set programming [37], finite-domain constraint optimization [18, 17, 28], and, among the most recent ones, pseudo-Boolean optimization (PBO) [41, 42]. Secondly, what sets IHS apart from other typical solving approaches is that IHS iteratively combines two types of constraint solvers. The first solver is a decision solver for the constraint language at hand, used for identifying so-called unsatisfiable cores expressing inconsistencies in the input formula. The second solver is a hitting set optimizer, typically implemented through an integer programming (IP) solver, for identifying ways of ruling out the so-far identified cores from consideration towards an optimal solution of the original problem. As we will explain, the particular way in which IHS search is structured offers a novel way of integrating symmetry-awareness to IHS-based approaches to constraint optimization, referred to as symmetric core learning, or SCL for short.

Symmetric core learning employs knowledge of instance-level symmetries extracted before search, and can be considered a new type of a dynamic symmetry exploitation technique in the style of symmetric explanation learning [20], with the key insight that learning symmetric images of cores extracted within IHS iterations can benefit the search process. A key difference of SCL compared to lex-leader symmetry breaking (which is in many domains the current state-of-the-art symmetry handling technique) is that, whereas lex-leader symmetry breaking adds symmetry breaking constraints to an input instance before invoking a solver in hope that some of the constraints might prove useful during search, SCL does not bloat the input instance with *potentially* useful symmetry breaking constraints. Instead, SCL employs symmetry information on the fly during IHS search to generate likely beneficial symmetric cores. As a high number of (symmetric) cores may need to be extracted in the worst case for termination, SCL has the potential of speeding up IHS by avoiding a high number of potentially time-consuming calls to the core-extracting decision solver, and instead can produce symmetric cores with polynomial delay from the cores extracted. Specifically, as the subsequently-computed hitting sets over the cores are required to also relax away the sources of inconsistencies expressed by the symmetric cores generated with SCL, SCL has potential for significantly lowering the number of IHS iterations needed for termination.

We focus on pseudo-Boolean optimization, also known as 0–1 IP solving, for which IHS is a competitive solving approach [42]. As we will empirically show, earlier-proposed static lex-leader symmetry breaking applied before search generally degrades the performance of IHS for PBO. This also holds for the recently-proposed extension of symmetry breaking to so-called weak symmetries broken by enforcing dominance constraints that ensure the preservation of optimal solutions [44]. This motivates the study of alternative ways of integrating symmetries into IHS search for PBO. Symmetric core learning provides one such alternative approach to making use of symmetries in IHS. At the same time, SCL makes use of weak symmetries

which discount the impact of the objective function at hand without needing dominance constraints. Beyond explicitly generating symmetric cores, we also investigate a compact representation of symmetric cores that has the potential of representing an exponential set of symmetric cores as a single abstract core [8] over a set of extension variables by making use of linear constraints native to the IP solver. In contrast to lex-leader symmetry breaking, we show that SCL can improve the runtime performance of a state-of-the-art IHS approach to PBO, and generally does not impose significant runtime overheads. Complementing the empirical results, we also point out fundamental differences in the effects of lex-leader symmetry breaking and SCL on IHS.

2 Preliminaries

2.1 Pseudo-Boolean Optimization

A *literal* ℓ is a Boolean variable x or its negation $\overline{x} = 1 - x$, where variables take values 0 (false) or 1 (true). A *pseudo-Boolean (PB) constraint* C is a 0–1 linear inequality $\sum_i a_i \ell_i \geq A$, where a_i and A are integers. Without loss of generality, we often assume PB constraints to be in *normal form*, meaning that the literals ℓ_i are over distinct variables, each *coefficient* a_i is *positive* and the *degree (of falsity)* A is non-negative. A *pseudo-Boolean formula* F is a conjunction $\bigwedge_j C_j$ of PB constraints. When convenient, we view F as a *set* of constraints. For a constraint C, $\text{LIT}(C)$ is the set of literals that appear in C. For a pseudo-Boolean formula F, $\text{LIT}(F) = \bigcup_{C \in F} \text{LIT}(C)$. An *objective* O is an expression $\sum_i w_i \ell_i + lb$ where the coefficients w_i and the constant lb are integers (lb stands for "lower bound": when the w_i are all possible, this constant term is indeed a lower bound on the cost).

A *substitution* (sometimes also called a *witness*) ω is a mapping of variables to 0, 1 and literals. An *assignment* α is a substitution that maps each variable in its domain into $\{0, 1\}$. An assignment is *complete* for F if it assigns a value to each variable of interest (where the set of variables should be clear from the context). We write $\omega(C)$ for the constraint obtained from C by replacing each variable x in the domain of ω by $\omega(x)$ (and implicitly normalizing); $\omega(O)$, $\omega(\ell)$, $\omega(F)$ are defined analogously.

A (normalized) PB constraint $C = \sum_i a_i \ell_i \geq A$ is *trivial* if $A = 0$ and *contradictory* if $\sum_i a_i < A$. The constraint C is *satisfied* by α (denoted $\alpha \models C$) if $\alpha(C)$ is trivial. A PB formula is satisfied if all of its constraints are satisfied. An instance of the *pseudo-Boolean optimization problem* is a tuple (F, O) with F a PB formula and O an objective to minimize. A complete assignment α is a *solution* of (F, O) if $\alpha \models F$ and is optimal if also $\alpha(O) \leq \beta(O)$ for each solution β of (F, O). The optimal cost of (F, O) is $\alpha(O)$ for an optimal solution of (F, O).

Following [41], a constraint C is an *unsatisfiable core* of (F, O) if the following conditions hold: (i) the literals in C are objective literals or their negations and (ii) all assignments α that satisfy F also satisfy C (denoted by $F \models C$). In other words, an unsatisfiable core is an implied constraint that only mentions objective literals.

▶ Remark 1. This definition of unsatisfiable core might sound somewhat unconventional; its naming has grown historically as follows. Originally an "unsatisfiable core" was defined a subset of the original formula that is unsatisfiable. In the context of assumption-based SAT solving (also employed for SAT-based optimization) fresh variables are used to represent constraints in the input formula and such a core became "a subset of the assumptions that cannot jointly be set to a specific value" (or alternatively, a clause over assumption variables learned by the solver). In the PBO setting, this was then further generalized to be any PB constraint over assumptions (objective variables) learned by the PB solver.

The following proposition forms the basis for how the IHS approach to PBO uses cores to compute optimal solutions.

▶ **Proposition 2** (e.g. [41]). *Let (F, O) be a PBO instance, α_{best} an optimal solution to (F, O), $\mathcal{K}$ a set of cores of (F, O), and γ an optimal solution to $(\mathcal{K}, O)$. Then $\gamma(O) \leq \alpha_{best}(O)$.*

In words, Proposition 2 states that the cost of the optimal (minimum-cost) solutions to any set of cores is a lower bound on the optimal cost of the instance.

2.2 Symmetries in Pseudo-Boolean Optimization

A substitution σ is called a (syntactic) *weak symmetry of* (F, O) if $\sigma(F) = F$. If $\sigma(O) = O$ further holds, σ is a (strong) symmetry. For strong symmetries, α is an optimal solution of (F, O) if and only if $\alpha \circ \sigma$ is an optimal solution of (F, O). We also consider a subset of weak symmetries that we call *core-preserving symmetries*. A weak symmetry σ is core-preserving if $\sigma(\ell)$ is an objective literal if and only if ℓ is. Similarly to weak symmetries, and in contrast to strong ones, a core-preserving symmetry σ is not guaranteed to preserve the cost of solutions, i.e., $\alpha(O)$ need not equal $\alpha \circ \sigma(O)$. In contrast to weak symmetries, however, core-preserving symmetries are guaranteed to map cores to cores, i.e., $\sigma(C)$ is a core of (F, O) if and only if C is. If Σ is a set of symmetries, we write $\langle\Sigma\rangle$ for the group generated by Σ. If Σ is a set of symmetries acting on a set S (e.g., acting on the set of cores as just defined), the *orbit* of $s \in S$ under Σ, denoted $\mathcal{O}(s, \Sigma)$, is the set $\{\sigma(s) \mid \sigma \in \langle\Sigma\rangle\}$.

When symmetries are present, they can slow down search of a PBO solver, causing the solver to explore many symmetric subareas of the search space of solutions. One way to deal with symmetric search spaces is to add symmetry-breaking constraints. Specifically, given a (strong) symmetry σ, a common approach is to add a (set of) constraint(s) that is satisfied by an assignment α if and only if α is lexicographically smaller than (or equal to) $\alpha \circ \sigma$ [1]. This can be achieved with a single pseudo-Boolean constraint LL_σ (called a *lex-leader* constraint) of the form

$$\sum_i 2^i \cdot x_i \leq \sum_i 2^i \cdot \sigma(x_i)$$

or by expressing the constraint as a set of constraints with smaller coefficients using auxiliary variables (see for instance [21]). Since weak symmetries do not preserve the costs of solutions, lex-leader constraints over a weak symmetry might change the optimal cost of a PBO instance. Van Caudenberg and Bogaerts [44] generalized the lex-leader constraint to weak symmetries, which then becomes a *dominance constraint*

$$2^{n+1} \cdot O + \sum_{i=1}^{n} 2^i \cdot x_i \leq 2^{n+1} \cdot \sigma(O) + \sum_{i=1}^{n} 2^i \cdot \sigma(x_i),$$

which states that the objective value must be smaller than or equal to the objective of the symmetric assignment, and, furthermore, *if* the objective values are equal, the assignment must be lexicographically smaller than its symmetric counterpart.

An alternative to defining symmetries as *substitutions* is to define symmetries as *permutations of literals* respecting negation, i.e., via a permutation π of the literals such that $\pi(\overline{x}) = \overline{\pi}(x)$ for each x. We will use *disjoint cycle* notation and write, for instance, $(x\overline{y}z)$ (or $(x\overline{y}z)(\overline{x}y\overline{z})$ if we also want to spell out the redundant information) for the symmetry that maps x to $\overline{y}$, $\overline{y}$ to z and z to x and all other variables to themselves.

1 PBO-IHS(F, O)
  Input: A PBO instance (F, O)
  Output: An optimal solution α_{best}
2 $(\alpha_{best}, sat?) \leftarrow$ PB-Solve(F);
3 **if** *not sat?* **then**
4 **return** "no feasible solutions";
5 $ub \leftarrow \alpha_{best}(O)$; $lb \leftarrow -\infty$; $\mathcal{K} \leftarrow \emptyset$;
6 $\boldsymbol{\Sigma} \leftarrow$ Compute-Symmetries($\mathbf{F}$) ;
7 **while** *TRUE* **do**
8 $\gamma \leftarrow$ Min-Sol($\mathcal{K}, O$);
9 $lb \leftarrow \gamma(O)$;
10 **if** $ub = lb$ **then break**;
11 $(C, \alpha', ub') \leftarrow$ Extract-Core(γ, F, O);
12 **if** $ub' < ub$ **then** $ub \leftarrow ub'$; $\alpha_{best} \leftarrow \alpha$;
13 **if** $ub = lb$ **then break**;
14 $\mathbf{K} \leftarrow$ SCL($\mathbf{C}, \mathcal{K}, \boldsymbol{\Sigma}$) ;
15 $\mathcal{K} \leftarrow \mathcal{K} \cup \{C\} \cup \mathbf{K}$;
16 **return** α_{best};

Algorithm 1 PBO-IHS. Text highlighted in yellow marks the symmetric core learning extension.

Min-Sol($\mathcal{K}, O$):
minimize: O
subject to:

$$C \qquad \forall C \in \mathcal{K}$$
$$\ell \in \{0, 1\} \qquad \forall \ell \in \text{LIT}(\mathcal{K})$$

return:
$\{\ell \mid \ell = 1 \text{ in opt. soln}\} \cup$
$\{\overline{\ell} \mid \ell = 0 \text{ in opt. soln}\}$

Figure 1 Hitting set IP.

▶ **Example 3.** Let $F = \{x + \overline{y} \geq 1, 2\overline{x} + y + z \geq 2, 2y + \overline{x} + z \geq 1\}$ and $O = x + y$. Then $\sigma = (x\overline{y})$ is a weak symmetry of (F, O) since applying σ to F swaps the last two constraints; σ is neither a core-preserving symmetry since σ maps the objective literal x to the non-objective literal $\overline{y}$, nor a strong symmetry since $\sigma(O) = \overline{y} + \overline{x}$ which is not $x + y$.

In practice, there can be exponentially many symmetries and hence breaking all of them, e.g., via lex-leader constraints, is in general infeasible. The most common way of dealing with this is to simply break a set of *generators* of the symmetry group, without formal guarantees on completeness, i.e., on that all symmetries would be broken. A set of generators is typically detected via a *graph* representing a simplification of the syntax tree of the formula and computing automorphisms of this graph [2]. Some tools go further then detecting an arbitrary set of generators, aiming to identify *structure* in the symmetry group [21, 4, 3]. As an example, these tools aim to detect *row interchangeability* [19], representing a symmetry as a matrix M of literals such that each two rows of the matrix are interchangeable in the sense that σ_{ij} maps every literal M_{ik} to M_{jk}, each M_{jk} to M_{ik}, and other literals to themselves. If one uses the "right" set of generators, and the "right" order of the variables at hand, the entire group generated by such a matrix can be broken with a polynomially-sized set of symmetry breaking constraints.

2.3 The Implicit Hitting Set Approach to PBO

Algorithm 1 details PBO-IHS, the implicit hitting set algorithm for pseudo-Boolean optimization [41, 42]. The highlighted lines correspond to the symmetric core learning extension we detail in Section 3 and should be ignored in this section.

Given a PBO instance (F, O), PBO-IHS begins by checking the existence of solutions of (F, O) by invoking PB decision procedure PB-Solve on F (Line 2). The call returns an

indicator *sat?* for satisfiability and a solution α_{best} in the positive case. Given α_{best}, an upper bound ub and a lower bound lb on the optimal cost of the instance are initialized to $\alpha_{best}(O)$ and $-\infty$, respectively, on line 5. During the search, α_{best} will always contain the currently best-known solution for which $\alpha_{best}(O) = ub$. A set $\mathcal{K}$ of cores of the instance is initialized to $\emptyset$ and the main search loop (lines 7-15) entered. The main search loop iterates until $lb = ub$, at which point α_{best} is known to be optimal. Each iteration begins with the computation of an optimal solution γ of the instance $(\mathcal{K}, O)$ consisting of the cores found so far and the objective (Line 9). In practice, γ is computed by solving the hitting set IP in Figure 1 with an integer programming (IP) solver. By Proposition 2, $\gamma(O)$ will be a lower bound on the optimal cost of the instance. Since no cores are ever removed from $\mathcal{K}$, the values of $\gamma(O)$ will be non-decreasing over the iterations, so lb can always be updated on Line 9. If the algorithm does not terminate on Line 10, the procedure `Extract-Core` next computes a new core C of (F, O) not satisfied by γ. This is achieved by invoking a decision procedure for PB constraints of F under the assumptions γ. The result of this call will either be a solution that extends γ, or a constraint C (learned using standard conflict analysis) falsified by γ but for which $F \models C$.

Each extracted core in IHS witnesses that the current γ cannot be extended to an optimal solution of the original instance. As a byproduct, `Extract-Core` also often obtains a solution α' of cost $\alpha'(O) = ub'$, which it also returns. The cost of α' can (but need not) be lower than the current upper bound; hence, termination is checked on Line 13. If the algorithm does not terminate, the new core is added to $\mathcal{K}$, and IHS reiterates.

3 Symmetric Core Learning for IHS

We detail *symmetric core learning* (SCL) for IHS-based PBO solving as our main contribution.

3.1 The Basic Idea

SCL is based on the observation that given a PBO instance (F, O), a core C, and a core-preserving symmetry σ, the constraint $\sigma(C)$ obtained by applying σ to C is also a core of (F, O). Symmetric core learning uses core-preserving symmetries of a PBO instance to generate new cores *dynamically during search*, with the key aims of decreasing the number of calls to the `Extract-Core` and `Min-Sol` procedures of `PBO-IHS`, while at the same time avoiding the generation of (a potentially large number) of lex-leader constraints before search.

▶ **Example 4.** Consider the PBO instance (F, O) with $F = \{x_i + y_j \geq 1 \mid i = 1, \ldots, n, j = 1, \ldots, m\}$, $O = \sum_{i=1}^{n} x_i + \sum_{j=1}^{m} y_j$ and $n \leq m$. Examples of core-preserving symmetries of (F, O) are $\sigma_1 = (x_1 x_2 \ldots x_n)$ and $\sigma_2 = (y_1 y_2 \ldots y_n)$. The core $C = x_1 + y_1 \geq 1$ and the symmetry σ_1 together imply the cores $\sigma_1(C) = x_2 + y_1 \geq 1$, $\sigma_1^2(C) = \sigma_1(\sigma_1(C)) = x_3 + y_1 \geq 1$, $\ldots$, $\sigma_1^{n-1}(C) = x_n + y_1 \geq 1$.

The IHS approach to PBO extended with symmetric core learning is obtained by including the highlighted lines of Algorithm 1. During the initialization phase, the algorithm computes a set of core-preserving symmetries of the PBO instance to be solved on Line 6. The symmetries are used during each iteration of the search loop by the `SCL` procedure on Line 14 to generate a set K of cores that are symmetric to the core C computed by `Extract-Core`, with the aim of increasing the number of cores added to $\mathcal{K}$ in each iteration of IHS search.

An intuition underlying symmetric core learning is that if we have a symmetry σ and a core C, then computing a symmetric core $\sigma(C)$ is very fast. In contrast, extracting cores with the subroutine `Extract-Core` invokes a decision procedure for an NP-hard constraint

```
1 Explicit-SCL(C, K, Σ)
    Input: A core C, a set of all so-far extracted cores K, and a set of symmetries Σ.
    Output: Set S of cores implied by C and the symmetries in Σ.
2   S ← {C};   coreQueue.init(C);
3   while coreQueue.size()>0 do
4     C ← coreQueue.pop();
5     for σ ∈ Σ do
6       if σ(C) ∉ K ∪ S then
7         S ← S ∪ {σ(C)};   coreQueue.push(σ(C));
8         if |S| ≥ constrLim or Σ_{C∈S} |LIT(C)| ≥ litLim then return S;
9   return S;
```

Algorithm 2 Explicit symmetric core learning

language, which could require a significant amount of time. Furthermore, in theory, a single symmetry can be used to generate many different symmetric cores. Thus, we expect symmetric core learning to be efficient for solving instances where the time required to compute core-preserving symmetries and invoking `SCL` is lower than the time needed to extract enough cores to terminate using only `Extract-Core`.

We detail two variants of symmetric core learning, i.e., instantiations of the `SCL` procedure of Algorithm 1. In *explicit symmetric core learning*, detailed in Section 3.2, the generated cores are explicitly added to the accumulated set of cores and thus as individual constraints to the hitting set IP. In *compact symmetric core learning*, detailed in Section 3.3, the generated cores are instead more compactly encoded with a small number of linear constraints that are then added to the hitting set IP instead of the cores themselves.

3.2 Explicit Symmetric Core Learning

In explicit symmetric core learning, the cores generated via symmetries are directly (or explicitly) added to the hitting set IP as individual constraints. Algorithm 2 details our breadth-first style implementation of explicit symmetric core learning. Given a core C computed by `Extract-Core` in the current iteration of IHS search, the set $\mathcal{K}$ of all cores found during search and a set Σ of core-preserving symmetries, `Explicit-SCL` first initialises a set S of cores that are symmetric to C, and a queue *coreQueue* of cores to be processed on Line 2. Then, the algorithm enters the while-loop between Lines 3 and 8, during which the cores in *coreQueue* are processed in breadth-first order as follows. For a core C, the algorithm iterates through each symmetry $\sigma \in \Sigma$. If $\sigma(C)$ is a previously unseen core, $\sigma(C)$ is added to the set S and pushed into *coreQueue* to be processed later. After adding a new core to S the algorithm checks for two termination criteria: whether $|S|$, the number of new symmetric cores that have been generated, exceeds a user-defined parameter *constrLim*, and whether $\sum_{C \in S} |\text{LIT}(C)|$, the sum of the number of literals in the new symmetric cores, exceeds the parameter *litLim*. As detailed in the following, these early termination criteria are employed to escape situations in which Algorithm 2 would otherwise produce an excessive number of symmetric cores to the point that overall search performance could suffer.

A central challenge in realizing explicit symmetric core learning is that naively applying symmetries to cores may result in a very large set of cores that do not help the IHS search terminate faster. More precisely, while a single core with n literals can, in theory, generate a number of symmetric cores exponential in n (e.g., in a degenerate case where any permutation of variables is a symmetry), the following observations demonstrate that the effect of explicit

symmetric core learning on IHS search depends on the PBO instance. The first observation demonstrates how explicit symmetric core learning can reduce the number of iterations of the IHS main loop.

▶ **Observation 5.** *Consider the PBO instance (F, O), the symmetries σ_1 and σ_2 from Example 4, and assume for the example that* `Extract-Core` *only returns at-least-one constraints (i.e., propositional clauses) as cores. An optimal solution α to (F, O) assigns exactly the n variables x_i to 1. As every at-least-one core extracted results in at most one new variable being assigned to 1 by solutions returned by* `Min-Sol`*,* `PBO-IHS` *without symmetric core learning (i.e. Algorithm 1 without the highlighted parts) invokes the* `Extract-Core` *and* `Min-Sol` *subroutine at least n times before termination. In contrast, given the symmetries σ_1, σ_2, and a single core of form $x_i + y_j \geq 1$ for some i and j, Algorithm 2 can generate $n \times m$ symmetric cores, allowing* `PBO-IHS` *with symmetric core learning (i.e. Algorithm 1 with the highlighted parts) to terminate after a single invocation of* `Extract-Core` *and two invocations of* `Min-Sol`*.*

The next observation demonstrates that IHS with unrestricted explicit symmetric core learning can lead to the extraction of unnecessarily many cores during IHS search.

▶ **Observation 6.** *Assume that, invoked on the PBO instance (F, O) from Example 4, the IHS algorithm has extracted the n cores $x_i + y_i \geq 1$ for $i = 1, \ldots, n$. In the next call to* `Min-Sol`*, one of the minimum-cost solutions that can be returned assigns all x_i variables to 1 and all y_j variables to 0. Given this solution, the next call to* `Extract-Core` *will not find a core, and hence IHS terminates after extracting n cores. In contrast, as detailed in Observation 5, IHS with unrestricted explicit SCL and the symmetries $\sigma_1 = (x_1 \ldots x_n)$ and $\sigma_2 = (y_1 \ldots y_m)$ is guaranteed to add $n \times m$ cores to $\mathcal{K}$ before termination.*

In contrast to the simple instance in Observation 6, in practical settings adding too many constraints to the hitting set IP can significantly increase the time needed to compute a minimum-cost solution. Identifying which cores allow IHS to make fast progress towards termination is an open research challenge. We limit explicit SCL in attempt to balance between harnessing the potential of symmetric cores to speed up search (Observation 5) and mitigating the risk of having to spent unnecessary effort to extract cores (Observation 6).

3.3 Compact Symmetric Core Learning

Compact symmetric core learning aims to express a large number of symmetric cores with a smaller number of constraints in the hitting set IP, making use of linear constraints native to IP. The following examples provide intuition for the approach.

▶ **Example 7.** Consider again the PBO instance (F, O), the core $C = x_1 + y_1 \geq 1$, and the symmetries σ_1 and σ_2 from Example 4. Using two fresh auxiliary Boolean variables s_x and s_y all of the $n \times m$ cores implied by C and σ_1 and σ_2 can be captured by the constraints $n \cdot s_x \leq x_1 + \cdots + x_n$, $m \cdot s_y \leq y_1 + \cdots + y_m$ (which enforces s_x (s_y) to be smaller than each of the x_i (y_i)), and $s_x + s_y \geq 1$.

As the minimum-cost solutions to the extracted cores are computed with an IP solver, we are not restricted to using binary variables for expressing a set of cores. Indeed, integer variables can be employed for representing sets of symmetric cores more compactly.

▶ **Example 8.** Consider a PBO instance with two symmetries $\sigma_1 = (x_1 x_2 x_3)(y_1 y_2 y_3)$ and $\sigma_2 = (z_1 z_2 z_3)(w_1 w_2 w_3)$. Given the core $C = x_1 + 2y_1 + z_1 + w_1 \geq 3$, the group generated by σ_1 and σ_2 yields the symmetric cores $x_i + 2y_i + z_j + w_j \geq 3$ for $i = 1, \ldots, 3$ and $j = 1, \ldots, 3$. All

these cores can be represented compactly with the constraints $s_{xy} \leq x_i + 2y_i$ (for $1 \leq i \leq 3$), $s_{zw} \leq w_i + z_i$ (for $1 \leq i \leq 3$), and $s_{xy} + s_{zw} \geq 3$, where s_{xy} and s_{zw} are (non-binary) *integer* variables.

We continue with a high-level description of compact SCL. Assume that we partition a core $C = \sum_\ell a^\ell \ell \geq B$ as $\sum_{\ell \in S} a^\ell \ell + \sum_{\ell \notin S} a^\ell \ell \geq B$ for a subset S of its literals. Now assume a set Σ of symmetries for which $\sigma(\sum_{\ell \notin S} a^\ell \ell) = \sum_{\ell \notin S} a^\ell \ell$ for each $\sigma \in \Sigma$. Consider the orbit $\mathcal{O}(\sum_{\ell \in S} a^\ell \ell, \Sigma)$ of the partition of the constraint containing the literals in S. Compact symmetric core learning generates a compact representation for a set of cores implied by C and Σ by replacing $\sum_{\ell \in S} a^\ell \ell$ with an integer variable s_S and the definition $s_S \leq \min(\mathcal{O}(\sum_{\ell \in S} a^\ell \ell, \Sigma))$. In the rest of this section, we fill in the details of this high-level description of compact SCL.

As illustrated in Examples 7 and 8, compact SCL can substitute several subsets of a core with new variables and in doing so represent many symmetric cores more compactly. What is happening in these examples is that (certain) subgroups of the group of all symmetries act independently on parts of the core rather than on the whole core. However, as the following observation shows, independent rewriting of parts of the cores is not sound in case symmetries interact too much. In the rest of this section, we will formalize conditions that guarantee the soundness of rewriting parts of the core.

▶ **Observation 9.** *Assume that a PBO instance has the symmetries $\sigma_1 = (x_1 x_2 x_3)(y_4 y_2 y_3)$, $\sigma_2 = (y_1 y_2 y_3)(x_4 x_2 x_3)$ and the core $C = x_1 + y_1 \geq 1$. The orbit of x_1 under $\{\sigma_1\}$ is $\{x_1, x_2, x_3\}$ and the orbit of y_1 under $\{\sigma_2\}$ is $\{y_1, y_2, y_3\}$. Compact SCL could introduce both $s_x + y_1 \geq 1$, $s_x \leq \min(\{x_1, x_2, x_3\})$ and $x_1 + s_y \geq 1$, $s_y \leq \min(\{y_1, y_2, y_3\})$. Introducing $s_x + s_y \geq 1$ with $s_x \leq \min(\{x_1, x_2, x_3\})$ and $s_y \leq \min(\{y_1, y_2, y_3\})$, however, is not correct since these would also represent the constraint $x_2 + y_2 \geq 1$ that is not a core implied by C, σ_1 and σ_2.*

Observation 9 demonstrates that the choice of symmetries under which to compute orbits can not be done independently for all subsets to be replaced. One way to interpret Observation 9 is that if we wish to introduce a new variable defined by the orbit $\mathcal{O}_2$ into a core that already has another new variable defined by the orbit $\mathcal{O}_1$, we need to ensure that the symmetries used to compute the orbit $\mathcal{O}_2$ "behave well" together with those used to compute $\mathcal{O}_1$. The following definition formalizes this notion of well-behavedness.

▶ **Definition 10.** *Let C be a core and assume that Σ_i is a set of symmetries for each $i \leq k$. We call $\langle \Sigma_i \mid 1 \leq i \leq k \rangle$ a symmetry decomposition of C if C can be written as $L_1 + \cdots + L_k \geq A$ and the following conditions hold:*

1. *For each $\sigma_i \in \Sigma_i$, $\sigma_i(\sum_{j>i} L_j) = \sum_{j>i} L_j$.*
2. *For each $\sigma_i \in \Sigma_i$, each $j < i$, and each $L'_j \in \mathcal{O}(L_j, \Sigma_j)$, it holds that $\sigma_i(L'_j) \in \mathcal{O}(L_j, \Sigma_j)$.*

For intuition on Definition 10, assume that we are computing a compact representation for a set of cores symmetric to C partitioned as $L_1 + \cdots + L_k \geq A$. If we process the partitions in the order of $L_1, L_2, \ldots, L_k$, then a symmetry decomposition of C defines which symmetries we apply to generate the orbit $\mathcal{O}_i$ for each L_i. To be a symmetry decomposition, we should be able to split our core so that each symmetry in the i^{th} set stabilizes the parts of the core beyond i as well as the orbits in all of the earlier parts of the core. These conditions together guarantee that every constraint represented by the compact representation is a core.

▶ **Proposition 11.** *Let $\langle \Sigma_i \mid 1 \leq i \leq k \rangle$ be a symmetry decomposition of $C = L_1 + \cdots + L_k \geq A$. If $L'_i \in \mathcal{O}(L_i, \Sigma_i)$ for all i, then $L'_1 + \cdots + L'_k \geq A$ is a core.*

Proof Sketch. This core can be constructed iteratively from k to 1 as follows. Since $L'_k \in \mathcal{O}(L_k, \Sigma_k)$, there is a symmetry σ_k such that $\sigma_k(L_k) = L'_k$ and hence $\sigma_k(L_1) + \cdots + \sigma_k(L_{k-1}) + L'_k$ is a core. The second item of Definition 10 guarantees that there exists a symmetry σ_{k-1} that stabilizes L_k and such that $\sigma_{k-1}(L_{k-1}) = \sigma_k^{-1}(L'_{k-1})$. As a consequence, $\sigma_k \circ \sigma_{k-1}(C) = \sigma_k \circ \sigma_{k-1}(L_1 + \cdots + L_{k-2}) + L'_{k-1} + L'_k$. Iteratively applying this construction yields the desired core. ◀

In order to use a symmetry decomposition, we will replace each L_i by new variables intuitively to represent all its symmetric images at once. Before giving the formal definition, we give one more example of how to get even more compact encodings.

▶ **Example 12.** Consider a PBO instance where $x_1, \ldots, x_n$ are interchangeable without touching the term $y_1 + 2y_2 + 3y_3$. In other words, assume that the group of stabilizers of $y_1 + 2y_2 + 3y_3$ acts symmetrically on $\{x_1, \ldots, x_n\}$ (meaning that for each i, j, it contains a permutation that swaps x_i and x_j and maps each other x_k to itself). Additionally, assume that $x_1 + 2x_2 + 3x_3 + y_1 + 2y_2 + 3y_3 \geq 5$ is a core. In that case, there are cubically many cores: for each triple of different values of i, j and k, $x_i + 2x_j + 3x_k + y_1 + 2y_2 + 3y_3 \geq 5$ is a core. We can introduce three counter variables: v_1, v_2, and v_3 such that v_i is false if at least i of the x variables are false using $(n - i + 1)v_i \leq x_1 + x_2 \cdots + x_n$. Given these three counting constraints, all the cores are captured by the single core $v_3 + 2v_2 + 3v_1 + y_1 + 2y_2 + 3y_3 \geq 5$.

We are now ready to formally define the compact encoding of cores that we use.

▶ **Definition 13.** *Let $\langle \Sigma_i \mid 1 \leq i \leq k \rangle$ be a symmetry decomposition of $C = L_1 + \cdots + L_k \geq A$. Assume that $L_i = \sum_j w_j \ell_j$ (where the coefficients w_j are wlog assumed to be decreasing). The* compact encoding *of L_i consists of:*

Case 1: if $\langle \Sigma_i \rangle$ acts symmetrically on a set L that includes all ℓ_j. *(1) constraints $(|L_i| - j + 1)c_{i,j} \leq \sum_j \ell_j$ defining fresh counting variables $c_{i,j}$ (for $1 \leq j \leq |L_i|$): and (2) replacing L_i by $\sum_j w_j c_{i,j}$ in C;*

Case 2: otherwise *(1) constraints $V_i \leq L'_i$ introducing a fresh integer variables V_i for each $L'_i \in \mathcal{O}(L_i, \Sigma_i)$, and (2) replacing L_i by V_i in C.*

Case 1 of Definition 13 was shown in Example 7, Case 2 in Example 8. In the worst case, i.e., when the **Case 1** optimization is not used, compact SCL uses $1 + \sum_i |\mathcal{O}(L_i, \Sigma_i)|$ constraints to represent $\prod_i |\mathcal{O}(L_i, \Sigma_i)|$ cores. (In the special case $|L_i| = 1$, we use 1 constraint instead of $|\mathcal{O}(L_i, \Sigma_i)|$ constraints to encode the semantics of V_i.)

Implementing compact SCL requires care and various heuristic choices. As a first way of realize compact SCL, we implemented a procedure which, given a core C and a set Σ of symmetries, heuristically computes different partitions of C and subsets of Σ as candidate symmetry decomposition of C. The procedure prioritizes finding symmetry groups that act symmetrically on a set of literals (to enable the Case 1 optimization of Definition 13). Our procedure for computing a symmetry decomposition of C and the orbits of a partition under it resemble explicit SCL as detailed in Algorithm 2. When orbits containing more than the proposed partition itself are found, C is modified by replacing the partition under consideration with a new variable defined based on the orbit. Technical details on our current implementation of this approach to implementing compact SCL are provided in Appendix A.

In contrast to explicit SCL, compact SCL can decrease the best-case number of constraints needed in the hitting set IP solved by `Min-Sol` during IHS search.

▶ **Observation 14.** *Consider the PBO instance (F, O) and the symmetries σ_1 and σ_2 from Example 4. Observation 6 illustrated that IHS search can terminate after having extracted*

n cores of form $x_i + y_i \geq 1$ for $i = 1, \ldots, n$. However, any assignment that sets exactly one from each pair x_i, y_i to 1 and all other variables to 0 is a minimum cost solution to the integer program formed over these n cores that `Min-Sol` *solves. Only one of these 2^n minimum-cost solutions will results in termination of IHS search, namely the one that assigns all x_i to 1 and all y_i to 0. To guarantee termination using only cores of form $x_i + y_j \geq 1$ (i.e., the at-least-one cores of (F, O) that have a minimum number of literals), IHS (with or without explicit SCL) needs to add at least $2n$ of them to $\mathcal{K}$ since with any fewer, there is at least one x_i that appears only in one core $x_i + y_j \geq 1$, meaning that there is a minimum-cost hitting set that assigns x_i to 0 and y_j to 1. On the other hand, the set $\{x_i + y_i \geq 1 \mid 1 \leq i \leq n\} \cup \{x_i + y_{i+1} \geq 1 \mid 1 \leq i \leq n\}$ of $2n$ cores only has one minimum-cost hitting set, assigning all x_i to 1.*

In contrast, as detailed in Example 7, assuming the first core extracted is $x_1 + y_1 \geq 1$, compact SCL will result in the constraints $s_x \leq \min(\{x_i \mid i = 1, \ldots, n\})$, $s_y \leq \min(\{y_i \mid i = 1, \ldots, m\})$ and $s_x + s_y \geq 1$, where s_x and s_y are fresh binary variables. In total, compact SCL will in this case add 3 constraints to $\mathcal{K}$: (1) $ns_x \leq \sum_{i=1}^{n} x_i$, (2) $ms_y \leq \sum_{i=1}^{m} y_i$ and (3) $s_x + s_y \geq 1$. In the next iteration, the only minimum-cost solution that `Min-Sol` *can return sets all x_i and s_x to 1, and the other variables to 0. With this solution, IHS with compact SCL terminates with 3 constraints (that represent $n \times m$ cores) in $\mathcal{K}$.*

Compact SCL and Abstract Cores. Finally, we note that compact SCL has a similar flavor to a technique called *abstract cores* proposed for IHS in the context of maximum satisfiability [8]. An IHS solver using abstract cores will (i) heuristically select a subset S of objective literals, (ii) introduce new count variables o_k with the definition $o_k \leftrightarrow \sum_{\ell \in S} \ell \geq k$, and (iii) use the `Extract-Core` subroutine to extract so-called abstract cores, i.e., core constraints that contain both original objective literals and the new count variables. A central difference between abstract cores and compact SCL is that rather than using `Extract-Core` to prove that the new variables appear in cores, the way in which compact SCL chooses which objective literals should be grouped together is guaranteed to cover a large number of symmetric cores of the PBO instance.

3.4 SCL and Heuristic Optimizations in IHS

Practical implementations of IHS employ various additional heuristic optimizations which are central for runtime efficiency. This is also the case for PBO-IHS. The two heuristics that affect our implementation of SCL are forms of *hardening*, including reduced cost fixing [5] and *weight-aware core extraction* (WCE) [16, 9].

Hardening refers to fixing the value of an objective literal ℓ during search. There are two ways in which that can happen: (i) if the coefficient of ℓ is strictly greater than the current upper-bound ub, then the literal is fixed to 0 since setting it to 1 would lead to a worse solution than the current best known; or (ii) via reduced cost fixing [5], i.e., if the reduced cost of ℓ in the LP relaxation of the hitting set IP detailed in Figure 1 imply that ℓ is set either to 0 or 1 in the optimal solutions. Hardening affects SCL in that symmetries that map non-fixed literals to fixed ones are removed from Σ when symmetric cores are generated.

Weight-aware core extraction aims to delay calls to `Min-Sol` by extracting multiple cores falsified by a solution γ in each iteration. Intuitively this decreases the number of times that the hitting set IP needs to be solved [42, 41]. With SCL, we interleave the calls to the `SCL` procedure between the core-extraction steps: after `Extract-Core` obtains each new core C, we compute symmetric cores to it and then invoke `Extract-Core` again, asking for a core that is falsified by γ and not any of the cores that have been computed in this iteration. In

practice, this is achieved by removing at least one literal from every obtained core from the assumptions posed in the next query to `Extract-Core`. The `Min-Sol` procedure is invoked only when `Extract-Core` does not find more cores.

3.5 SCL and Symmetry Breaking

We end this section with a short observation comparing SCL and symmetry breaking with lex-leader constraints in the context of IHS. In contrast to SCL, symmetry breaking with lex-leader constraints does not result in the IHS search extracting more of the original cores of the PBO instance, but rather introduces new cores that change the search.

▶ **Observation 15.** *Consider the PBO instance (F, O) and the symmetry σ_1 detailed in Example 4 as a strong symmetry of (F, O). Applying lex-leader symmetry breaking on (F, O) adds the constraint $LL_x = \sum_{i=1}^{n} 2^i x_i \leq \sum_{i=1}^{n-1} 2^i x_{i+1} + 2^n x_1$. The constraints $x_1 + \overline{x_i} \geq 1$ for $2 \leq i \leq n$ are examples of cores of $(F \cup \{LL_x\}, O)$ that are not cores of (F, O). In other words, symmetry breaking can (implicitly) introduce new cores to the original instance.*

4 Experiments

To evaluate the impact of SCL, we implemented the PBO-IHS algorithm from scratch in C++, using RoundingSat 2 [22] as the PB core extractor, Gurobi 9.5.2 as the IP solver for hitting set computation, and BreakID [21] to compute symmetries and lex-leader constraints. Our PBO-IHS re-implementation (without symmetry breaking) outperforms the original PBO-IHS implementation [41, 42] on the benchmark set used in [41, 42], solving 948 instances against 920 for the implementation by the original authors. (For a comparison of PBO-IHS with other approaches, see [41, 42].) Our PBO-IHS implementation includes the optimizations described in [41, 42], including assumption shuffling, weight-aware core extraction (WCE) [5], solution-improving search at stagnation [42], reduced cost fixing [5], and hardening. Most relevant in terms of SCL are assumption shuffling and WCE which both concern core extraction: SCL (when used) is applied after assumption shuffling, and within the WCE loop, i.e., after each core extraction step. We evaluate the following variants of PBO-IHS.

- **Baseline**: our PBO-IHS implementation without symmetry techniques.
- **LL**: Baseline extended with lex-leader symmetry breaking.
- **LLS**: LL restricted to only break *strong* symmetries (see [44]).
- **ExplicitSCL**: Baseline extended with explicit SCL.
- **CompactSCL**: ExplicitSCL extended with compact SCL.

We use 100-second time limit for BreakID in the LL and SCL variants for computing symmetries, and 1 h for the overall runtime; reported runtimes include symmetry computation. For LL, we run BreakID with the same parameters as in of the original work on weak symmetries for lex-leader symmetry breaking for PBO [44] (the detection of row interchangeability and weak symmetry breaking are enabled). For LLS, weak symmetries are disabled. Based on preliminary experimentation, with ExplicitSCL and CompactSCL detection of row interchangeability is disabled. For the early termination heuristics, we use $constrLim = 100$ and $litLim = 6000$. These value choices were not rigorously tuned, but the parameters are important to cap the symmetric core generation, since in general there may be exponentially many symmetric cores. (It should be noted that LL also has similar limiting parameters to avoid overloading the solver with symmetry breaking constraints.) Our implementation applies compact SCL only when the estimated number of cores covered is greater than the estimated number of auxiliary constraints needed. Explicit SCL is applied in each iteration

after first applying compact SCL; when explicit SCL is applied to a compacted core, we only use symmetries that do not change the definitions of the count variables in the core.

The experiments reported on were run on 2.40-GHz Intel Xeon Gold 6148 machines with 381-GB RAM in RHEL under a 32-GB memory limit. Implementation code, data and benchmark generation scripts are available at `https://doi.org/10.5281/zenodo.15630156`.

4.1 Experiments on Highly Symmetric Benchmarks

We start by considering two examples of problem domains known to have a high number of symmetries intrinsically. The first, which we will refer to as the CC problem, asks to maximize the size of the largest clique over all graphs with n nodes that admit a k-coloring where n and k are given parameters. Additionally, we consider a variant of the problem in which all nodes are given a distinct weight from $1, \ldots, n$, and the task is to maximize sum-of-weights of the nodes in the clique. We will refer to this variant as "weighted CC" and talk about "unweighted CC" when the task is to maximize the size of the clique. In our experiments, we use an extension of the encoding from [32] to optimization, the full details of the encoding are in Appendix B. Beyond being highly symmetric (as explained in short), the CC problem draws motivations from proof complexity: tautologies of the type "a graph either does not contain a clique of size m, or is not $(m-1)$-colorable", expressed similarly to the encoding we use, are exponentially hard for cutting planes [32].

Note that the size of the largest clique in the graphs that admit a k-coloring is k. There are many symmetric variants of graphs with n nodes that admit both a k-coloring and a clique of size k. The encoding of CC results in PBO instances in which the objective contains n variables that are indicators for a node **not** being included in the clique. With this intuition, any at-least-one core corresponds to a subset of nodes that cannot all be included in the clique. Since the maximum size of the clique is k, it follows that any subset of nodes that contains $k+1$ nodes corresponds to a (subset-minimal) core. Termination of IHS using only cores of this form requires in total $\binom{n}{k+1}$ cores that rule out all such subsets from consideration.

The CC instances contain two types of easy-to-grasp symmetries: "node-symmetries" over the node indices, and "color-symmetries" over color indices. Both types are present in weighted and unweighted CC, but very notably, the node-symmetries that, informally speaking, map nodes to other nodes are strong only in the unweighted variant. To see this, observe that mapping nodes to other nodes preserves the cost of solutions only in the unweighted case. When invoked on the PB encoding of the CC problem (unweighted or weighted) explicit SCL, using the node-symmetries and a single minimal at-least-one core of size $k+1$, can generate $\binom{n}{k+1} - 1$ other cores corresponding to all other $\binom{n}{k+1} - 1$ subsets of size $k+1$. As such, IHS with explicit SCL can terminate after extracting a single core from `Extract-Core`, but enumerates an exponential number of symmetric cores and adds them to the hitting set IP. In contrast, given one core of size $k+1$, and the node-symmetries, compact SCL can generate a single cardinality constraint that limits the maximum number of nodes in any clique to k, resulting in termination of IHS in the next iteration with only a single constraint added to the hitting set IP.

Regarding lex-leader-based symmetry breaking on unweighted CC instances, if the node-symmetries are broken completely (which is not guaranteed to be the case, but is a best-case scenario that LL heuristically aims to achieve), the additional constraints added to the PBO instance essentially enforce that the maximum clique consists of k specific nodes instead of any k nodes. Then the instance extended with lex-leader constraints has $n-k$ unit cores that directly enforce that the other nodes can not be in the maximum clique. When invoked

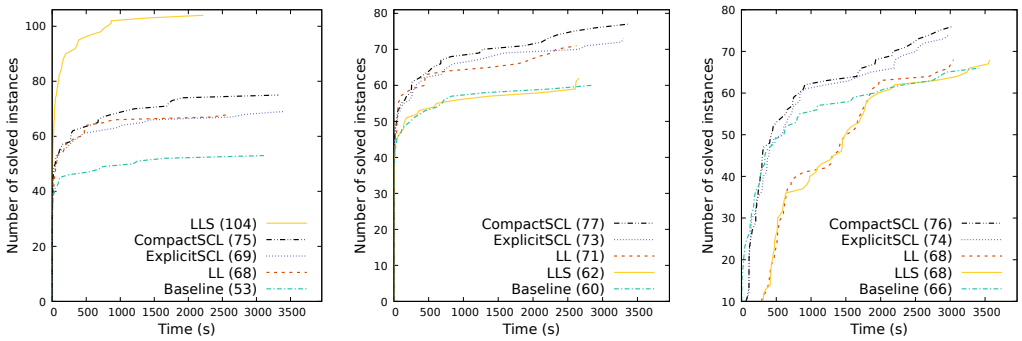

Figure 2 Impact of symmetry techniques on number of instances solved. Left: unweighted CC, Middle: weighted CC, Right: Haplotype inference.

on the extended instance, IHS can terminate after extracting $n - k$ of these (short) cores. (On weighted CC instances, the node-symmetries can be broken quite similarly when weak symmetry detection and dominance constraints are used: then the constraints enforce that the maximum clique consists of the k nodes with largest weights.) Further, in contrast to SCL, the fact that lex-leader symmetry breaking adds constraints to core-extractor can decrease core-extraction times. Thus we expect symmetry breaking to be very effective in these instances. The main question is how close IHS with (explicit/compact) SCL comes to the performance of IHS with lex-leader symmetry breaking.

To experiment with CC, we generated instances with all combinations of $n = 5..29$ and $k = 2..n$, yielding a total of 400 unweighted and 400 weighted benchmarks. On unweighted CC (see Figure 2 left), explicit SCL solves 16 more instances than the Baseline (IHS without symmetry techniques) with 69 vs 53 solved benchmarks. Compact SCL is even more effective, solving 6 more instances than explicit SCL (75 vs 69 instances). Using lex-leader with the full set of weak symmetries (LL) leads to 68 instances solved, which is less than either variant of SCL. Interestingly, restricting lex-leader-based symmetry breaking to the set of strong symmetries (LLS in the figure) leads to the overall best performance on unweighted CC and 104 instances solved. The results suggest that restricting the types of symmetries that BreakID can find (in this case to symmetries that map objective variables to objective variables), allows for better recognition of the relevant structure of the instance.

On the weighted CC instances (Figure 2 middle) the baseline solves 60 instances and lex-leader symmetry breaking restricted to strong symmetries only marginally improves with 62. When allowed to use all weak symmetries, IHS with lex-leader symmetry breaking improves to 71 solved instances. Our SCL approaches obtain the best performance and notable improvements over lex-leader-based breaking; explicit SCL solves 73 instances and compact SCL an impressive 77. A potential explanation for the big difference in performance of LLS between the unweighted and the weighted case is that node-symmetries are not strong in the weighted variant (whereas they are in the unweighted).

As a second, more practically oriented highly symmetric problem domain, we consider haplotype inference pedigrees problem [24], and in particular the task of finding a set of haplotypes that best explains given set of genotypes. For experimenting with this problem domain, we used the 100 instances submitted to Pseudo-Boolean competition 2011. It should be noted that, as explained in [24], these instances already include domain-specific symmetry breaking constraints added by hand for increased performance, enforcing a predefined order for the two haplotypes that produce a genotype. Hence an interesting question is whether

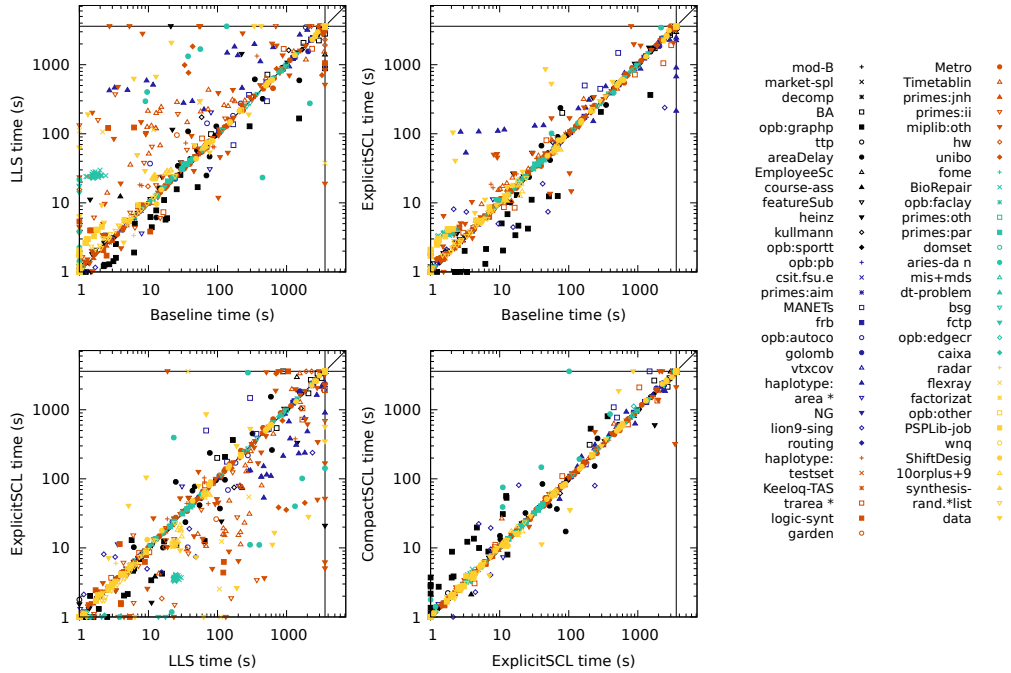

Figure 3 Runtime comparisons. Top left: Baseline vs. LLS, top right: Baseline vs. ExplicitSCL, bottom left: LLS vs. ExplicitSCL, bottom right: ExplicitSCL vs. CompactSCL

the different symmetry techniques can improve IHS performance further on these instances. Again, (compact) SCL provides the greatest performance improvements (see Figure 2 right): Baseline without symmetry breaking solves 66 instances, which is improved by lex-leader symmetry breaking to 68 instances by both variants, LL and LLS—however, with explicit SCL the number of solved instances goes up to 74, and with compact SCL further to 76.

4.2 Experiments on a Wider Set of Benchmarks

We also investigate the relative impact of lex-leader symmetry breaking and SCL more generally on a wide set of PBO benchmarks including instances from over 60 domains as used in the the original papers presenting PBO-IHS [41, 42]. Our main interest here is whether the different types of symmetry techniques are viable to use by default even when there are no guarantees that the input instances would be highly symmetric. The benchmark set consists of 1786 instances across tens of different benchmark domains; see [40] for more details. We disregard LL here since LLS turned out to perform better than LL (see Appendix C).

A pairwise per-instance runtime comparison of Baseline, LLS and ExplicitSCL is shown in Figure 3. The total number of instances solved is: Baseline 948, LLS 947, ExplicitSCL 954, and CompactSCL 947. LLS results in significant runtime degradation compared to Baseline (Figure 3 top left). Indeed, the impact lex-leader symmetry breaking has on PBO-IHS appears very different to results presented in [44] on the runtime impact of lex-leader symmetry breaking on other PBO solving approaches, where (mild) runtime improvements were reported. ExplicitSCL consistently and most often significantly outperforms LLS (Figure 3 bottom left). A reason for this is that the PB solver used for core extraction tends to use significantly more time when invoked on a satisfiable instance. This follows the intuition that, in contrast to the dynamically applied SCL, the lex-leader constraints all added in the beginning of IHS search

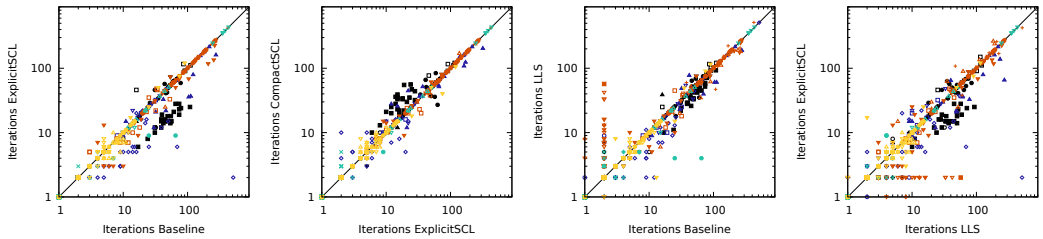

Figure 4 Number of IHS iterations. Left: Baseline vs. ExplicitSCL, Middle left: ExplicitSCL vs. CompactSCL, Middle right: Baseline vs. LLS, Right: LLS vs. ExplicitSCL.

rule out symmetric solutions and hence may make finding solutions more difficult; more details on the runtime differences are provided in Appendix D. Similarly, ExplicitSCL on most instances significantly outperforms LLS (Figure 3 bottom left). ExplicitSCL also tends to have a performance-improvement impact on PBO-IHS, as observed in Figure 3 (top right), despite the symmetry computation overhead (up to 100 s). In contrast, despite CompactSCL performing very well on selected highly symmetric domains (recall Section 4.1), on this wider heterogenous set of benchmarks, CompactSCL performs slightly worse compared to ExplicitSCL (Figure 3 bottom right).

ExplicitSCL tends to terminate with fewer IHS main loop iterations than Baseline (Figure 4 left), suggesting that symmetric cores added by SCL are indeed helpful in decreasing required number of core extraction steps. Between ExplicitSCL and CompactSCL, there appear to be only more minor differences both ways (Figure 4 middle left). The relative number of iterations between LLS and ExplicitSCL (Figure 4 middle right) and LLS and Baseline (Figure 4 top left) do not fully explain the runtime degradation caused by LLS (recall Figure 3 right), which suggests that lex-leader constraints indeed significantly slow down the IHS iterations on average.

5 Conclusions

We presented symmetric core learning, SCL, as a dynamic approach to symmetry-aware IHS search. The approach works by generating symmetric unsatisfiable cores on-the-fly, thereby strengthen the hitting set IP. SCL has the potential for quickly improving bounds and, especially, avoiding unnecessary costly calls to the core extractor. Beyond explicitly enumerating symmetric cores, we proposed a compact way of representing a set of symmetric cores, with potential for compacting the hitting set IP. Empirically, SCL can improve the runtime of IHS for PBO especially on highly symmetric problem domains and, in contrast to lex-leader symmetry breaking, is not generally detrimental to IHS performance. For future work, SCL mainly assumes that the hitting set problems in IHS are computed using an IP solver; hence SCL could be applied in IHS instantiations for other constraint languages as well. In fact, a technique similar to (explicit) SCL has been added recently to IHS-based generation of MUSes [10] based on generating *symmetric satisfiable subsets* instead of *symmetric cores*. Another direction for future work is to investigate *proof logging* for IHS with SCL. For *symmetry breaking*, proof logging techniques have recently been added to a proof system for reasoning with PB constraints [11]. A proof system for *symmetric learning* in a SAT context has also been proposed [43]. An open question is whether these two ideas could be combined to yield proofs for SCL for pseudo-Boolean optimization.

References

1. Fadi A. Aloul, Arathi Ramani, Igor L. Markov, and Karem A. Sakallah. ShatterPB: symmetry-breaking for pseudo-Boolean formulas. In Masaharu Imai, editor, *Proceedings of the 2004 Conference on Asia South Pacific Design Automation: Electronic Design and Solution Fair 2004, Yokohama, Japan, January 27-30, 2004*, pages 883–886. IEEE Computer Society, 2004. doi:10.1109/ASPDAC.2004.179.
2. Fadi A. Aloul, Karem A. Sakallah, and Igor L. Markov. Efficient symmetry breaking for Boolean satisfiability. *IEEE Trans. Computers*, 55(5):549–558, 2006. doi:10.1109/TC.2006.75.
3. Markus Anders, Sofia Brenner, and Gaurav Rattan. satsuma: Structure-based symmetry breaking in SAT. In Supratik Chakraborty and Jie-Hong Roland Jiang, editors, *27th International Conference on Theory and Applications of Satisfiability Testing, SAT 2024, August 21-24, 2024, Pune, India*, volume 305 of *LIPIcs*, pages 4:1–4:23. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2024. doi:10.4230/LIPICS.SAT.2024.4.
4. Markus Anders, Pascal Schweitzer, and Mate Soos. Algorithms transcending the SAT-symmetry interface. In Meena Mahajan and Friedrich Slivovsky, editors, *26th International Conference on Theory and Applications of Satisfiability Testing, SAT 2023, July 4-8, 2023, Alghero, Italy*, volume 271 of *LIPIcs*, pages 1:1–1:21. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023. doi:10.4230/LIPICS.SAT.2023.1.
5. Fahiem Bacchus, Antti Hyttinen, Matti Järvisalo, and Paul Saikko. Reduced cost fixing in MaxSAT. In Beck [6], pages 641–651. doi:10.1007/978-3-319-66158-2_41.
6. J. Christopher Beck, editor. *Principles and Practice of Constraint Programming - 23rd International Conference, CP 2017, Melbourne, VIC, Australia, August 28 - September 1, 2017, Proceedings*, volume 10416 of *Lecture Notes in Computer Science*. Springer, 2017. doi:10.1007/978-3-319-66158-2.
7. Belaid Benhamou, Tarek Nabhani, Richard Ostrowski, and Mohamed Réda Saïdi. Enhancing clause learning by symmetry in SAT solvers. In *22nd IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2010, Arras, France, 27-29 October 2010 - Volume 1*, pages 329–335. IEEE Computer Society, 2010. doi:10.1109/ICTAI.2010.55.
8. Jeremias Berg, Fahiem Bacchus, and Alex Poole. Abstract cores in implicit hitting set MaxSAT solving. In Luca Pulina and Martina Seidl, editors, *Theory and Applications of Satisfiability Testing - SAT 2020 - 23rd International Conference, Alghero, Italy, July 3-10, 2020, Proceedings*, volume 12178 of *Lecture Notes in Computer Science*, pages 277–294. Springer, 2020. doi:10.1007/978-3-030-51825-7_20.
9. Jeremias Berg and Matti Järvisalo. Weight-aware core extraction in SAT-based MaxSAT solving. In Beck [6], pages 652–670. doi:10.1007/978-3-319-66158-2_42.
10. Ignace Bleukx, Hélène Verhaeghe, Bart Bogaerts, and Tias Guns. Exploiting symmetries in MUS computation. In Toby Walsh, Julie Shah, and Zico Kolter, editors, *AAAI-25, Sponsored by the Association for the Advancement of Artificial Intelligence, February 25 - March 4, 2025, Philadelphia, PA, USA*, pages 11122–11130. AAAI Press, 2025. doi:10.1609/AAAI.V39I11.33209.
11. Bart Bogaerts, Stephan Gocht, Ciaran McCreesh, and Jakob Nordström. Certified dominance and symmetry breaking for combinatorial optimisation. *J. Artif. Intell. Res.*, 77:1539–1589, 2023. doi:10.1613/jair.1.14296.
12. Alessandro Cimatti, Alberto Griggio, Bastiaan Joost Schaafsma, and Roberto Sebastiani. A modular approach to MaxSAT modulo theories. In Järvisalo and Van Gelder [26], pages 150–165. doi:10.1007/978-3-642-39071-5_12.
13. Nadia Creignou and Daniel Le Berre, editors. *Theory and Applications of Satisfiability Testing - SAT 2016 - 19th International Conference, Bordeaux, France, July 5-8, 2016, Proceedings*, volume 9710 of *Lecture Notes in Computer Science*. Springer, 2016. doi:10.1007/978-3-319-40970-2.
14. Jessica Davies and Fahiem Bacchus. Solving MAXSAT by solving a sequence of simpler SAT instances. In Jimmy Ho-Man Lee, editor, *Principles and Practice of Constraint Programming*

- *CP 2011 - 17th International Conference, CP 2011, Perugia, Italy, September 12-16, 2011. Proceedings*, volume 6876 of *Lecture Notes in Computer Science*, pages 225–239. Springer, 2011. doi:10.1007/978-3-642-23786-7_19.

15 Jessica Davies and Fahiem Bacchus. Exploiting the power of mip solvers in MaxSAT. In Järvisalo and Van Gelder [26], pages 166–181. doi:10.1007/978-3-642-39071-5_13.

16 Jessica Davies and Fahiem Bacchus. Postponing optimization to speed up MAXSAT solving. In Schulte [39], pages 247–262. doi:10.1007/978-3-642-40627-0_21.

17 Toby O. Davies, Graeme Gange, and Peter J. Stuckey. Automatic logic-based benders decomposition with minizinc. In Satinder Singh and Shaul Markovitch, editors, *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, February 4-9, 2017, San Francisco, California, USA*, pages 787–793. AAAI Press, 2017. doi:10.1609/AAAI.V31I1.10654.

18 Erin Delisle and Fahiem Bacchus. Solving weighted CSPs by successive relaxations. In Schulte [39], pages 273–281. doi:10.1007/978-3-642-40627-0_23.

19 Jo Devriendt, Bart Bogaerts, and Maurice Bruynooghe. BreakIDGlucose: On the importance of row symmetry in SAT. In *CSPSAT, Vienna, 18 July 2014*, pages 1–17, July 2014. URL: https://lirias.kuleuven.be/handle/123456789/456639.

20 Jo Devriendt, Bart Bogaerts, and Maurice Bruynooghe. Symmetric explanation learning: Effective dynamic symmetry handling for SAT. In Serge Gaspers and Toby Walsh, editors, *Theory and Applications of Satisfiability Testing - SAT 2017 - 20th International Conference, Melbourne, VIC, Australia, August 28 - September 1, 2017, Proceedings*, volume 10491 of *Lecture Notes in Computer Science*, pages 83–100. Springer, 2017. doi:10.1007/978-3-319-66263-3_6.

21 Jo Devriendt, Bart Bogaerts, Maurice Bruynooghe, and Marc Denecker. Improved static symmetry breaking for SAT. In Creignou and Le Berre [13], pages 104–122. doi:10.1007/978-3-319-40970-2_8.

22 Jan Elffers and Jakob Nordström. Divide and conquer: Towards faster pseudo-Boolean solving. In Jérôme Lang, editor, *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, July 13-19, 2018, Stockholm, Sweden*, pages 1291–1299. ijcai.org, 2018. doi:10.24963/ijcai.2018/180.

23 Katalin Fazekas, Fahiem Bacchus, and Armin Biere. Implicit hitting set algorithms for maximum satisfiability modulo theories. In Didier Galmiche, Stephan Schulz, and Roberto Sebastiani, editors, *Automated Reasoning - 9th International Joint Conference, IJCAR 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings*, volume 10900 of *Lecture Notes in Computer Science*, pages 134–151. Springer, 2018. doi:10.1007/978-3-319-94205-6_10.

24 Ana Graça, Inês Lynce, João Marques-Silva, and Arlindo L. Oliveira. Efficient and accurate haplotype inference by combining parsimony and pedigree information. In Katsuhisa Horimoto, Masahiko Nakatsui, and Nikolaj Popov, editors, *Algebraic and Numeric Biology - 4th International Conference, ANB 2010, Hagenberg, Austria, July 31- August 2, 2010, Revised Selected Papers*, volume 6479 of *Lecture Notes in Computer Science*, pages 38–56. Springer, 2010. doi:10.1007/978-3-642-28067-2_3.

25 Alexey Ignatiev, Mikolás Janota, and João Marques-Silva. Quantified maximum satisfiability. *Constraints An Int. J.*, 21(2):277–302, 2016. doi:10.1007/S10601-015-9195-9.

26 Matti Järvisalo and Allen Van Gelder, editors. *Theory and Applications of Satisfiability Testing - SAT 2013 - 16th International Conference, Helsinki, Finland, July 8-12, 2013. Proceedings*, volume 7962 of *Lecture Notes in Computer Science*. Springer, 2013. doi:10.1007/978-3-642-39071-5.

27 Markus Kirchweger and Stefan Szeider. SAT modulo symmetries for graph generation and enumeration. *ACM Trans. Comput. Log.*, 25(3):1–30, 2024. doi:10.1145/3670405.

28 Javier Larrosa, Conrado Martínez, and Emma Rollon. Theoretical and empirical analysis of cost-function merging for implicit hitting set WCSP solving. In Michael J. Wooldridge, Jennifer G. Dy, and Sriraam Natarajan, editors, *Thirty-Eighth AAAI Conference on Artificial Intelligence,*

AAAI 2024, Thirty-Sixth Conference on Innovative Applications of Artificial Intelligence, IAAI 2024, Fourteenth Symposium on Educational Advances in Artificial Intelligence, EAAI 2014, February 20-27, 2024, Vancouver, Canada, pages 8057–8064. AAAI Press, 2024. doi:10.1609/AAAI.V38I8.28644.

29 Hakan Metin, Souheib Baarir, Maximilien Colange, and Fabrice Kordon. CDCLSym: Introducing effective symmetry breaking in SAT solving. In Dirk Beyer and Marieke Huisman, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 24th International Conference, TACAS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings, Part I*, volume 10805 of *Lecture Notes in Computer Science*, pages 99–114. Springer, 2018. doi:10.1007/978-3-319-89960-2_6.

30 Erick Moreno-Centeno and Richard M. Karp. The implicit hitting set approach to solve combinatorial optimization problems with an application to multigenome alignment. *Oper. Res.*, 61(2):453–468, 2013. doi:10.1287/OPRE.1120.1139.

31 Andreas Niskanen, Jere Mustonen, Jeremias Berg, and Matti Järvisalo. Computing smallest MUSes of quantified Boolean formulas. In Georg Gottlob, Daniela Inclezan, and Marco Maratea, editors, *Logic Programming and Nonmonotonic Reasoning - 16th International Conference, LPNMR 2022, Genova, Italy, September 5-9, 2022, Proceedings*, volume 13416 of *Lecture Notes in Computer Science*, pages 301–314. Springer, 2022. doi:10.1007/978-3-031-15707-3_23.

32 Pavel Pudlák. Lower bounds for resolution and cutting plane proofs and monotone computations. *J. Symb. Log.*, 62(3):981–998, 1997. doi:10.2307/2275583.

33 James A. Reggia, Dana S. Nau, and Pearl Y. Wang. Diagnostic expert systems based on a set covering model. *Int. J. Man Mach. Stud.*, 19(5):437–460, 1983. doi:10.1016/S0020-7373(83)80065-0.

34 Raymond Reiter. A theory of diagnosis from first principles. *Artif. Intell.*, 32(1):57–95, 1987. doi:10.1016/0004-3702(87)90062-2.

35 Ashish Sabharwal. Symchaff: exploiting symmetry in a structure-aware satisfiability solver. *Constraints An Int. J.*, 14(4):478–505, 2009. doi:10.1007/s10601-008-9060-1.

36 Paul Saikko, Jeremias Berg, and Matti Järvisalo. LMHS: A SAT-IP hybrid MaxSAT solver. In Creignou and Le Berre [13], pages 539–546. doi:10.1007/978-3-319-40970-2_34.

37 Paul Saikko, Carmine Dodaro, Mario Alviano, and Matti Järvisalo. A hybrid approach to optimization in answer set programming. In Michael Thielscher, Francesca Toni, and Frank Wolter, editors, *Principles of Knowledge Representation and Reasoning: Proceedings of the Sixteenth International Conference, KR 2018, Tempe, Arizona, 30 October - 2 November 2018*, pages 32–41. AAAI Press, 2018. URL: https://aaai.org/ocs/index.php/KR/KR18/paper/view/18021.

38 Paul Saikko, Johannes Peter Wallner, and Matti Järvisalo. Implicit hitting set algorithms for reasoning beyond NP. In Chitta Baral, James P. Delgrande, and Frank Wolter, editors, *Principles of Knowledge Representation and Reasoning: Proceedings of the Fifteenth International Conference, KR 2016, Cape Town, South Africa, April 25-29, 2016*, pages 104–113. AAAI Press, 2016. URL: http://www.aaai.org/ocs/index.php/KR/KR16/paper/view/12812.

39 Christian Schulte, editor. *Principles and Practice of Constraint Programming - 19th International Conference, CP 2013, Uppsala, Sweden, September 16-20, 2013. Proceedings*, volume 8124 of *Lecture Notes in Computer Science*. Springer, 2013. doi:10.1007/978-3-642-40627-0.

40 Pavel Smirnov. Pseudo-boolean optimization by implicit hitting sets. Master's thesis, University of Helsinki, 2021. URL: http://hdl.handle.net/10138/340857.

41 Pavel Smirnov, Jeremias Berg, and Matti Järvisalo. Pseudo-Boolean optimization by implicit hitting sets. In Laurent D. Michel, editor, *27th International Conference on Principles and Practice of Constraint Programming, CP 2021, Montpellier, France (Virtual Conference), October 25-29, 2021*, volume 210 of *LIPIcs*, pages 51:1–51:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.CP.2021.51.

794 **42** Pavel Smirnov, Jeremias Berg, and Matti Järvisalo. Improvements to the implicit hitting set
795 approach to pseudo-Boolean optimization. In Kuldeep S. Meel and Ofer Strichman, editors,
796 *25th International Conference on Theory and Applications of Satisfiability Testing, SAT 2022,*
797 *August 2-5, 2022, Haifa, Israel*, volume 236 of *LIPIcs*, pages 13:1–13:18. Schloss Dagstuhl -
798 Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPICS.SAT.2022.13.
799 **43** Rodrigue Konan Tchinda and Clémentin Tayou Djamégni. On certifying the UNSAT result
800 of dynamic symmetry-handling-based SAT solvers. *Constraints An Int. J.*, 25(3-4):251–279,
801 2020. doi:10.1007/s10601-020-09313-2.
802 **44** Daimy Van Caudenberg and Bart Bogaerts. Symmetry and dominance breaking for pseudo-
803 Boolean optimization. In Toon Calders, Celine Vens, Jefrey Lijffijt, and Bart Goethals,
804 editors, *Artificial Intelligence and Machine Learning - 34th Joint Benelux Conference, BNA-*
805 *IC/Benelearn 2022, Mechelen, Belgium, November 7-9, 2022, Revised Selected Papers*, volume
806 1805 of *Communications in Computer and Information Science*, pages 149–166. Springer, 2022.
807 doi:10.1007/978-3-031-39144-6_10.
808 **45** Daimy Van Caudenberg, Bart Bogaerts, and Leandro Vendramin. Incremental SAT-based
809 enumeration of solutions to the yang-baxter equation. In Arie Gurfinkel and Marijn Heule,
810 editors, *Tools and Algorithms for the Construction and Analysis of Systems - 31st International*
811 *Conference, TACAS 2025, Held as Part of the International Joint Conferences on Theory*
812 *and Practice of Software, ETAPS 2025, Hamilton, ON, Canada, May 3-8, 2025, Proceedings,*
813 *Part II*, volume 15697 of *Lecture Notes in Computer Science*, pages 3–22. Springer, 2025.
814 doi:10.1007/978-3-031-90653-4_1.

```
1  Compact-SCL(C, K, Σ)
       Input: A core C, a set of all so-far extracted cores K, and a set of symmetries Σ.
       Output: A set of constraints, S ∪ D, that compactly encode a set of symmetric
               cores of C.
2      S ← {C};   D ← ∅;   coreQueue.init({(C, ∅)});
3      constrBudgetCompact ← constrLim;   litBudgetCompact ← litLim;
        constrBudgetExplicit ← constrLim;   litBudgetExplicit ← litLim;
4      while coreQueue.size()>0 do
5          (C, D) ← coreQueue.pop();
6          if constrBudgetCompact > 0 and litBudgetCompact > 0 then
7              C ← Try-Compact-SCL(C, D, Σ);
8              if |C| > 0 then
9                  S ← S \ {C};
10                 for (C', D') ∈ C do
11                     D ← D ∪ D';
12                     S ← S ∪ {C'};
13                     coreQueue.push((C', D'));
14                     constrBudgetCompact ← constrBudgetCompact − 1;
                        litBudgetCompact ← litBudgetCompact − |LIT(C')|;
15                 continue;
16         if constrBudgetExplicit > 0 and litBudgetExplicit > 0 then
17             for σ ∈ Filter-Symmetries(Σ, D) do
18                 if σ(C) ∉ K ∪ S then
19                     S ← S ∪ {σ(C)};
20                     coreQueue.push((σ(C), D));
21                     constrBudgetExplicit ← constrBudgetExplicit − 1;
                        litBudgetExplicit ← litBudgetExplicit − |LIT(C)|;
22     return S ∪ D;
```

Algorithm 3 Compact SCL combined with explicit SCL.

A Implementing CompactSCL

Algorithm 3 details our implementation of compact SCL combined with explicit SCL (the instantiation of `SCL` to be called by Algorithm 1). Given a core C extracted by `Extract-Core`, the algorithm initializes the set S of cores to be returned, a set $\mathcal{D}$ of definitions of new count variables, and a queue *coreQueue* of cores to be processed. Each pair in *coreQueue* consists of a core constraint C, and the definitions D of any count variables in C. The cores are then processed in breadth-first order.

When processing a core C, compact SCL is tried first by the `Try-Compact-SCL` method attempting to compute a compact representation of a set of cores symmetric to C (Lines 6–15). The procedure returns a set $\mathcal{C}$ of pairs (C, D), where C is a core constraint, and a set D contains the definitions of count variables that appear in C. Each pair is a compacted core that represents a set of cores that are symmetric to C. When compact SCL on C is successful ($|\mathcal{C}| > 0$), all pairs in $\mathcal{C}$ are added to S, the new definitions to $\mathcal{D}$ and the original C removed (as the constraints in $\mathcal{C}$ represent it). Finally, each pair (C, D) in $\mathcal{C}$ is added to *coreQueue* and the next core popped on Line 5. If instead compact SCL on C is unsuccessful (i.e., if $|\mathcal{C}| = 0$ so the continue statement on Line 15 is not reached), or if the budget for compact SCL is exceeded (at which point the if statement on Line 6 returns false), explicit SCL is

```
1  Try-Compact-SCL(C, D, Σ)
       Input: A core C, definitions of count variables D, and a set of symmetries Σ.
       Output: A set of core constraints 𝒞 and definition constraints 𝒟, that compactly
               encode symmetric cores implied by C.
2      𝒞 ← ∅;
3      for k = 1, ..., |LIT(C)| − 1 do
4          𝒮 ← Select-Candidate-Partitions(C, k);
5          if Check-If-Good(C, S, k) = false then continue;
6          for L ∈ 𝒮 do
7              if |LIT(L)| = |{ℓ}| = 1 then
8                  ℒ_in, ℒ_out ← Find-Interchangeable-Lits(C, D, ℓ, Σ);
9                  if |ℒ_in| > 1 and |ℒ_out| > 0 then
10                     C′, D′ ← Add-Count-Variables(C, ℒ_in, ℒ_out);
11                     𝒞 ← 𝒞 ∪ {(C′, D ∪ D′)};
12                     continue;
13             𝒪 ← Compute-Orbit(C, D, L, Σ);
14             if |𝒪| < 2 then continue;
15             C′, D′ ← Add-Count-Variable(C, L, 𝒪);
16             𝒞 ← 𝒞 ∪ {(C′, D ∪ D′)};
17         if |𝒞| > 0 then return 𝒞;
18     return 𝒞;
```

Algorithm 4 Applying compact symmetric core learning on a single core.

instead applied on C (Lines 16–21). Explicit SCL operates as described in Section 3.2 with the exception that now—since we are applying explicit SCL to compacted cores—only a subset of Σ is used. In Line 17 method `Filter-Symmetries`(Σ, D) is invoked to select the symmetries in Σ that keep the definitions of count variables in C intact, i.e., symmetries that can be used when explicit SCL is applied in conjunction with compact SCL.

For intuition of how our approach produces compacted cores in terms of symmetry decompositions, Algorithm 3 can be seen as computing a compact representation of cores symmetric to C by building a partitioning $L_1 + \cdots + L_k \geq A$ of C step-by-step. The first time `Try-Compact-SCL` is invoked on C, different ways of selecting L_1 are tried. For each candidate L_1 resulting in an orbit of size at least 2, a new compacted core with the chosen L_1 replaced by a new count variable is introduced. The compacted cores are then reconsidered in subsequent iterations, in which `Try-Compact-SCL` tries ways of selecting L_2, and so on.

Algorithm 4 details `Try-Compact-SCL`. Given a (potentially compacted) core C, definitions D of the count variables in C, and a set Σ of symmetries, the algorithm first initializes an empty set $\mathcal{C}$ of new compacted cores on Line 2. The loop on Lines 3–17 then looks for a new subset of terms L_i of the symmetry decomposition that is being built for C, prioritizing small $|\text{LIT}(L_i)|$ (i.e., small values of k). For each k, the method `Select-Candidate-Partitions` returns a set $\mathcal{S}$ of "candidate L_is". As computing all subsets of C with size k is infeasible, for $k > 1$, we select the candidate L_i based on the symmetries in Σ. More precisely, $\mathcal{S}$ contains only sets L for which a symmetry in Σ (or row in a matrix-symmetry) shares with the core exactly the literals in L. After computing $\mathcal{S}$, Algorithm 4 first heuristically estimates if applying compact SCL with the sets in $\mathcal{S}$ would result in compacted cores where the number of auxiliary constraints is smaller than the number of cores covered. (The check is here rather than elsewhere for practical implementation reasons.) If not, the sets $L \in \mathcal{S}$ are not

1 `Find-Interchangeable-Lits`(C, D, ℓ, Σ)
  Input: A core C, the definition constraints D of the count variables in C, a literal ℓ that appears in C, and a set of symmetries Σ.
  Output: Sets $\mathcal{L}_{in}$ and $\mathcal{L}_{out}$, where $\{\ell\} \subseteq \mathcal{L}_{in} \subseteq \text{LIT}(C)$, $\mathcal{L}_{in} \cap \text{LIT}(C) = \emptyset$ and $\mathcal{L}_{in} \cup \mathcal{L}_{out}$ is a set of "interchangeable" literals.
2 $\mathcal{L} \leftarrow \{\ell\}$;
3 **while** `Extend-Outside-Core`$(\mathcal{L}, C, D, \Sigma)$ ***or*** `Extend-In-Core`$(\mathcal{L}, C, D, \Sigma)$ **do**
4 **continue**;
5 **while** $2 \leq |\mathcal{O} \cap C| \leq |\mathcal{O}| - 1$ **do**
6 $\mathcal{L}' \leftarrow$ `Check-Interch`$(\mathcal{L}, C, D, \Sigma)$;
7 **if** $\mathcal{L}' = \mathcal{L}$ **then return** $\mathcal{L} \cap \text{LIT}(C)$, $\mathcal{L} \setminus \text{LIT}(C)$;
8 **else** $\mathcal{L} \leftarrow \mathcal{L}'$;
9 **return** $\{\ell\}$, $\emptyset$;

Algorithm 5 Finding a set of interchangeable literals.

used for compact SCL. Otherwise the algorithm checks for each $L \in \mathcal{S}$ if the L can be used as the next L_i. The special treatment of candidate Ls of size 1 corresponds to Case 1 of Definition 13. When $\text{LIT}(L) = \{\ell\}$ the call to `Find-Interchangeable-Lits`(C, D, ℓ, Σ) first attempts to find a set of literals $\mathcal{L}$ and a set of symmetries that allow permuting the set $\mathcal{L}$ in any order, while keeping rest of C unchanged. The call returns the set $\mathcal{L}$ of interchangeable literals divided into two sets: $\mathcal{L}_{in}$ containing the literals of $\mathcal{L}$ that appear in core C, and $\mathcal{L}_{out}$ containing the rest. If $|\mathcal{L}_{in}| > 1$ and $|\mathcal{L}_{out}| > 0$ i.e. at least two of the interchangeable literals are in C, and at least one is not, the method `Add-Count-Variables` is invoked to generate a new compacted core C', in which the literals in $\mathcal{L}_{in}$ are replaced by count variables defined by constraints in D' (Line 10). Then $\mathcal{C}$ is extended to include C' and $\mathcal{D}$ to include D (Line 11). When $|L| > 1$ or Case 1 optimization does not result in a compacted core (i.e. the if statement on Line 9 returns false), `Compute-Orbit` instead finds an orbit $\mathcal{O}$ for L under a subset of symmetries in Σ that satisfies the requirements for L being part of a symmetry decomposition. If $\mathcal{O}$ includes something in addition to L, `Add-Count-Variable` next generates an compacted core C' in which L is replaced by a count variable defined by the constraints in D (Line 15). C' is then added to $\mathcal{C}$ and D to $\mathcal{D}$ (Line 16). Algorithm 4 continues iterating over increasing sizes of k until a compact core can be generated.

Algorithm 5 details `Find-Interchangeable-Lits`, for computing interchangeable literals for a literal ℓ in a core C. Initially, the candidate set $\mathcal{L}$ only contains ℓ (Line 2). The first stage of search (Lines 3–4) heuristically add literals one by one to $\mathcal{L}$. The second stage (Lines 5–8) checks if the resulting $\mathcal{L}$ is a set of interchangeable literals. If not, it tries to identify a subset of $\mathcal{L}$ that is. The first stage uses two methods to introduce literals into $\mathcal{L}$. `Extend-Outside-Core`$(\mathcal{L}, C, D, \Sigma)$ looks for symmetries that map a literal $l \in \mathcal{L}$ to some literal $l' \notin \mathcal{L}$, while keeping the rest of $\mathcal{L}$ and C stable. `Extend-In-Core`$(\mathcal{L}, C, D, \Sigma)$ checks for each pair $l \in \mathcal{L}$ and $l' \in (\text{LIT}(C) \setminus \mathcal{L})$, if there is a symmetry that maps l' and l to each other while keeping the rest of C, definitions in D, and the rest of $\mathcal{L}$ stable.

When the construction of the set $\mathcal{L}$ is finished, the algorithm proceeds to the second stage in which `Check-Interch`$(\mathcal{L}, C, D, \Sigma)$ checks if $\mathcal{L}$ contains a set of interchangeable literals. `Check-Interch` tries to rebuild $\mathcal{L}$ by adding new literals one by one, only using symmetries that do not map any $l \in \mathcal{L}$ to some $l' \notin \mathcal{L}$. The implementation ensures that there is a set of symmetries that allow permuting literals in $\mathcal{L}$ into any order. First, `Check-Interch`$(\mathcal{L}, C, D, \Sigma)$ initializes $\mathcal{L}' = \{l\}$ with some $l \in \mathcal{L}$. Then it looks for symmetries that swap some literals $l \in \mathcal{L}'$ and $l' \in (\mathcal{L} \setminus \mathcal{L}')$, while keeping $\mathcal{L}' \setminus \{l\}$, $(\mathcal{L} \setminus \mathcal{L}') \setminus \{l'\}$, and

```
1  Compute-Orbit(C, D, L, Σ)
       Input: A core C, the definition constraints D of the count variables in C, a
              subset L of the terms of C, and a set of symmetries Σ.
       Output: An orbit 𝒪 of L computed on a subset of Σ that satisfies the definition
               of symmetry decomposition.
2      𝒪 ← {L};
3      orbitQueue.init(L);
4      Σ' ← Filter-Symmetries(Σ, C, D, L);
5      while orbitQueue.size() > 0 do
6          L ← orbitQueue.pop();
7          for σ ∈ Σ' do
8              if σ(L) ∈ 𝒪 then continue;
9              𝒪 ← 𝒪 ∪ {σ(L)};
10             orbitQueue.push(σ(L));
11     return 𝒪;
```

Algorithm 6 Computing an orbit for L.

the rest of C stable. If such l and l' are found, $\mathcal{L}'$ is extended to include l'. If in the end $\mathcal{L} = \mathcal{L}'$, `Check-Interch`$(\mathcal{L}, C, D, \Sigma)$ finishes by returning the set $\mathcal{L}$. If $\mathcal{L} \neq \mathcal{L}'$, the process is reiterated by initializing $\mathcal{L}' = \{l\}$ with a different $l \in \mathcal{L}$. If there is no more new $l \in \mathcal{L}$ with which $\mathcal{L}'$ could be initialized, `Check-Interch` returns the $\mathcal{L}'$ computed on the last l, which will always be the first ℓ given to `Find-Interchangeable-Lits` to ensure that the literal ℓ used when calling `Find-Interchangeable-Lits` will be included in the set $\mathcal{L}$ returned. If `Check-Interch`$(\mathcal{L}, C, D, \Sigma)$ finds $\mathcal{L}'$ such that $\mathcal{L} = \mathcal{L}'$, the set $\mathcal{L}$ is divided to literals in the core and outside the core, and these two sets are then returned (Line 7). Otherwise the procedure is reiterated with the set $\mathcal{L}'$ returned by `Check-Interch` (Line 8).

Finally, Algorithm 6 details the computation of an orbit. The procedure resembles explicit SCL as detailed in Algorithm 2 in that, starting from L itself, it explicitly enumerates all members of the orbit in breadth-first order. To ensure, that the symmetries used satisfy the definition of a symmetry decomposition, the set of symmetries to be considered during breadth-first search is filtered in Line 4. In particular, in our implementation, a symmetry $\sigma \in \Sigma$ is removed from consideration if any of the following conditions hold for it. 1) σ maps some literal $\ell \in C$ to a literal not in C, or to a literal in C that has a different coefficient in C. 2) C contains a Boolean count variable s_x introduced in previous iterations of compact learning and σ maps a literal that appears in the definition of s_x to a literal that does not. 3) C contains an integer count variable s_x and σ maps any literal in the definition of σ to any literal other than itself.

B Details on the Encoding of the CC instances

Extending the encoding from [32], the PBO encoding of the CC instances has the following variables.

- $e_{i,j}$ for $1 \leq i, j \leq n$ indicates an edge between nodes i and j
- $r_{i,c}$ for each node $1 \leq i \leq n$ and color $1 \leq c \leq t$ indicates that the node i is colored with thecolor c.
- $q_{k,i}$ for $1 \leq i, k \leq n$ indicates that the node i is the kth node to be included in a largest clique.
- b_i for $1 \leq i \leq n$ are objective variables indicating that node i is not in a largest clique.

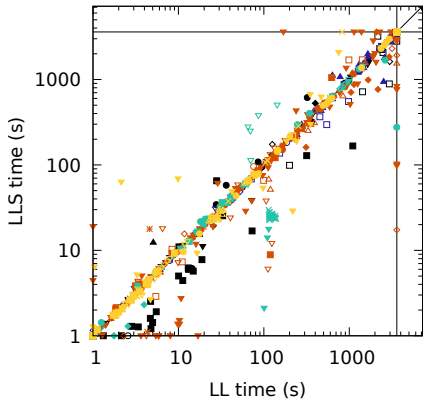

Figure 5 Runtime Comparison: LL vs. LLS.

The objective is to minimize $\sum_{i=1}^{n} b_i$ in the unweighted CC and $\sum_{i=1}^{n} ib_i$ in the weighted CC, subject to

- $\sum_k q_{k,i} \leq 1$ for $1 \leq i \leq n$ (Each node i is the kth node in the clique for at most one value of k.)
- $e_{i,j} - q_{k,i} - q_{k',j} \geq -1$ for $1 \leq i, j, k, k' \leq n$
  (If nodes i and j are the kth and k'th nodes in a largest clique, there must be an edge between i and j.)
- $e_{i,j} + r_{i,c} + r_{j,c} \leq 2$ for $1 \leq i, j \leq n$, $1 \leq c \leq t$
  (Adjacent nodes must have different colors.)
- $b_i + \sum_k q_{k,i} \geq 1$ (If node i is in the clique, it is the k:th node for some k..)

The scripts used to generate the CC instances are provided in the paper supplement.

C Comparing LL and LLS

Runtime comparison of LL and LLS is shown in Figure 5, LLS exhibiting better performance.

D Further Analysis of LLS vs ExplicitSCL

Fig. 6 shows a comparison of runtime LLS and ExplicitSCL spent in BreakID, core extraction, the IP solver, and solution-improving search (SIS). ExplicitSCL uses less time in each of the components. However, since ExplicitSCL generally solves instances with a smaller number of iterations (recall Figure 4), the difference in total runtime spent in core extraction, the IP solver and SIS is partially explained by the difference in the number of IHS iterations required for termination. For further details, Fig. 7 shows a comparison of the average time spent per iteration in each of the core extraction, IP solver and SIS component: the main difference in runtimes is due to time spent in core extraction and SIS. Digging deeper into the runtime spent in core extraction and SIS, we observed that for core extraction, the main runtime difference is due to calls which turn out to be satisfiable. This follows the intuition that the lex-leader constraints—all added at the beginning of IHS search—which rule out symmetric solutions may make it harder for a PB solver to find a solution. For SIS the main runtime difference is due to calls on which the PB solver is terminated without solution due exceeding the runtime limit enforced in PBO-IHS on the SIS solver calls.

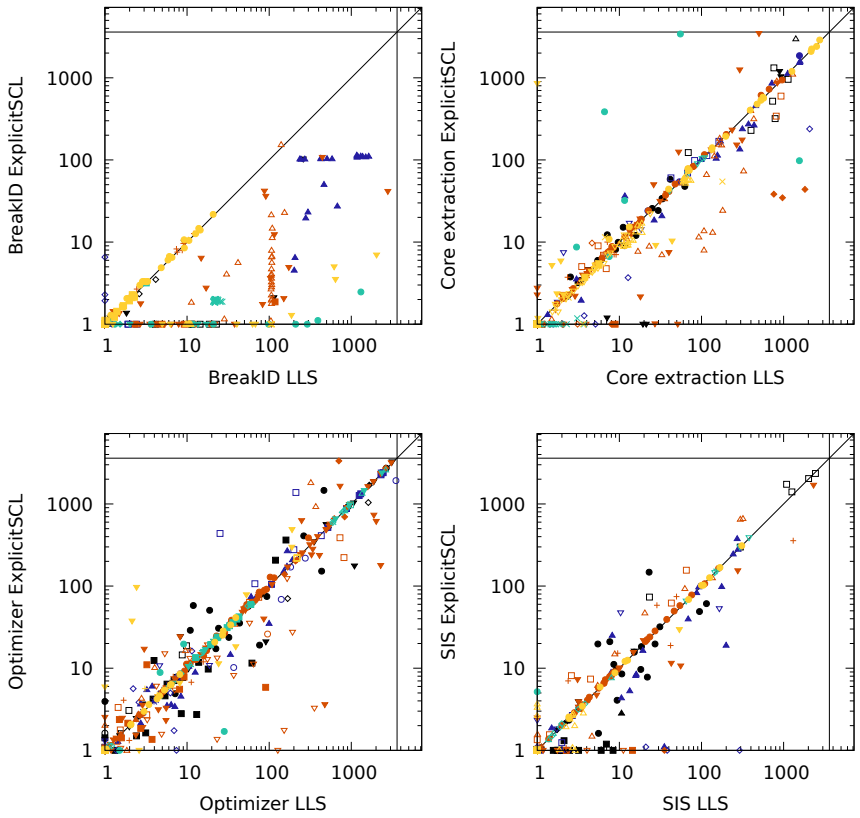

Figure 6 Time spent in BreakID (top left), core extraction (top right), IP solver (bottom left) and SIS (bottom right).

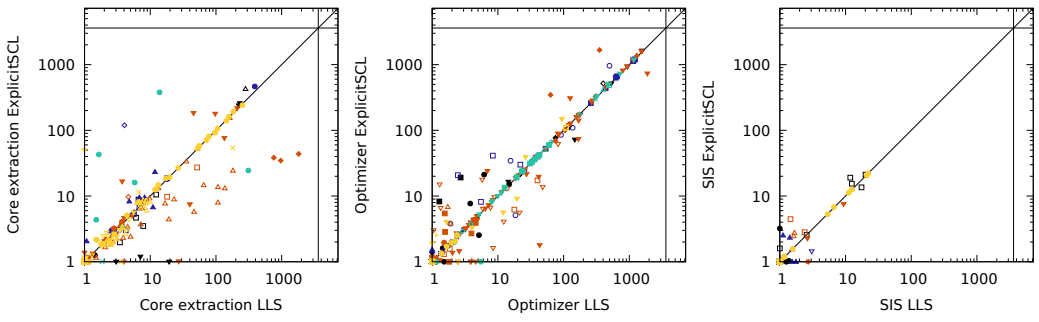

Figure 7 Average time spent per iteration in core extraction (left), IP solver (middle) and SIS (right).