Lifting Symmetry Breaking Constraints with Inductive Logic Programming

Alice Tarzariol[1] , **Martin Gebser**[1,2] , **Konstantin Schekotihin**[1]
[1]University of Klagenfurt, Austria
[2]Graz University of Technology, Austria
alice.tarzariol@aau.at , martin.gebser@aau.at , konstantin.schekotihin@aau.at

Abstract

Efficient omission of symmetric solution candidates is essential for combinatorial problem solving. Most of the existing approaches are instance-specific and focus on the automatic computation of Symmetry Breaking Constraints (SBCs) for each given problem instance. However, the application of such approaches to large-scale instances or advanced problem encodings might be problematic. Moreover, the computed SBCs are propositional and, therefore, can neither be meaningfully interpreted nor transferred to other instances. To overcome these limitations, we introduce a new model-oriented approach for Answer Set Programming that lifts the SBCs of small problem instances into a set of interpretable first-order constraints using the Inductive Logic Programming paradigm. Experiments demonstrate the ability of our framework to learn general constraints from instance-specific SBCs for a collection of combinatorial problems. The obtained results indicate that our approach significantly outperforms a state-of-the-art instance-specific method as well as the direct application of a solver.

1 Introduction

Many combinatorial problems are relatively easy to model with the current declarative programming paradigms. Nevertheless, when the size of input instances starts to grow, solving them might become infeasible because of a large number of possible solution candidates [Dodaro *et al.*, 2016]. In many cases, these candidates are symmetric, i.e., one candidate can simply be obtained from another by renaming constants. In order to deal with large problem instances, the ability to encode *Symmetry Breaking Constraints* (SBCs) in a problem representation becomes an essential skill for programmers. However, the identification of symmetric solutions and the formulation of constraints that remove them is a tedious and time-consuming task. As a result, various tools emerged that avoid the computation of symmetric solutions by, for instance, automatically finding a set of SBCs using properties of permutation groups, or applying specific search methods that detect and ignore symmetric states; see [Sakallah, 2009; Walsh, 2012; Margot, 2010] for an overview.

Respective approaches can be distinguished into *instance-specific* and *model-oriented* ones. The former methods identify symmetries for a particular instance at hand by computing and adding ground SBCs to the problem representation [Puget, 2005; Cohen *et al.*, 2006; Drescher *et al.*, 2011]. Unfortunately, computational advantages do not carry forward to large-scale instances or advanced encodings, where *instance-specific* symmetry breaking often requires as much time as it takes to solve the original problem. Moreover, ground SBCs generated by *instance-specific* approaches are *(i)* not transferable, since the knowledge obtained is limited to a single instance; *(ii)* usually hard to interpret and comprehend; *(iii)* derived from permutation group generators, whose computation is itself a combinatorial problem; and *(iv)* often redundant and might result in a degradation of the solving performance.

Elevated *model-oriented* approaches aim to find general SBCs that depend less on a particular instance. The method of Devriendt *et al.* [2016] uses local domain symmetries of a given first-order theory. SBCs are generated by identifying argument positions in atoms of a formula that comprise object variables defined over the same subset of a domain given in the input. As a result, the computation of lexicographical SBCs is very fast. However, the method considers each first-order formula separately and cannot reliably remove symmetric solutions, as it requires the analysis of several formulas at once. Mears *et al.* [2008] compute SBCs by generating small instances of parametrized constraints programs, and then find candidate symmetries using SAUCY [Darga *et al.*, 2004; Codenotti *et al.*, 2013] – a graph automorphism detection tool. Next, the algorithm removes all candidate symmetries that are valid only for some of the generated examples as well as those that cannot be proven to be parametrized symmetries using heuristic graph-based techniques. This approach can be seen as a simplified learning procedure that utilizes only negative examples represented by the generated SBCs.

In this work, we introduce a novel *model-oriented* method for Answer Set Programming (ASP) [Lifschitz, 2019] that aims to generalize the process of discarding redundant solution candidates for instances of a target domain using Inductive Logic Programming (ILP) [Cropper *et al.*, 2020]. The goal is to lift the SBCs of small problem instances and to obtain a set of interpretable first-order constraints. Such con-

straints cut the search space while preserving the satisfiability of a problem for the considered instance distribution, which improves the solving performance, especially in the case of unsatisfiability. The particular contributions of our work are:

- We suggest a method to generate a training set including positive and negative examples, allowing an ILP method to learn first-order SBCs for the problem at hand.
- We define the components of an ILP learning task enabling the generation of lexicographical SBCs for ASP.
- We show how to apply our method iteratively, to revise constraints when new permutation group generators are added or more training instances become available.
- We conduct experiments on variants of the pigeon-hole problem as well as the house-configuration problem [Friedrich *et al.*, 2011]. The obtained results show the benefits of our approach that significantly outperforms a state-of-the-art *instance-specific* method as well as an ASP solver without SBCs.

The structure of this paper is the following: a brief overview of the preliminaries is given in Section 2. Section 3 describes our approach, while Section 4 illustrates its implementation and specifies the components of an ILP learning task. In Section 5, we present and discuss experimental results for some combinatorial problems. Lastly, Section 6 concludes the paper and outlines directions for future work.

2 Background

2.1 Answer Set Programming

Answer Set Programming (ASP) [Lifschitz, 2019] is a declarative programming paradigm that applies non-monotonic reasoning and relies on the stable model semantics [Gelfond and Lifschitz, 1991]. Over the past decades, ASP has attracted considerable interest thanks to its elegant syntax, expressiveness, and efficient systems implementations, showing promising results in numerous domains that include, e.g., industrial, robotics, and biomedical applications [Erdem *et al.*, 2016; Falkner *et al.*, 2018]. We will briefly define the syntax and semantics of ASP, and refer the reader to [Gebser *et al.*, 2012; Lifschitz, 2019] for more detailed explanations.

Syntax. An ASP program P is a set of *rules* r of the form:

$$a_0 \leftarrow a_1, \ldots, a_m, not\, a_{m+1}, \ldots, not\, a_n$$

where not stands for *default negation* and a_i, for $0 \leq i \leq n$, are atoms. An *atom* is an expression of the form $p(\bar{t})$, where p is a predicate, $\bar{t}$ is a possibly empty vector of terms, and the predicate $\bot$ (with an empty vector of terms) represents the constant *false*. Each *term* t in $\bar{t}$ is either a variable or a constant, and a *literal* l is either an atom a_i (positive) or its negation $not\, a_i$ (negative). The atom a_0 is the *head* of a rule r, denoted by $H(r) = a_0$, and the *body* of r includes the positive or negative, respectively, body atoms $B^+(r) = \{a_1, \ldots, a_m\}$ and $B^-(r) = \{a_{m+1}, \ldots, a_n\}$. A rule r is called a *fact* if $B^+(r) \cup B^-(r) = \emptyset$, and a *constraint* if $H(r) = \bot$.

Semantics. The semantics of an ASP program P is given in terms of its *ground instantiation* P_{grd}, replacing each rule $r \in P$ with instances obtained by substituting the variables in r by constants occurring in P. Then, an *interpretation* $\mathcal{I}$ is a set of (*true*) ground atoms occurring in P_{grd} that does not contain $\bot$. An interpretation $\mathcal{I}$ *satisfies* a rule $r \in P_{grd}$ if $B^+(r) \subseteq \mathcal{I}$ and $B^-(r) \cap \mathcal{I} = \emptyset$ imply $H(r) \in \mathcal{I}$, and $\mathcal{I}$ is a *model* of P if it satisfies all rules $r \in P_{grd}$. A model $\mathcal{I}$ of P is *stable* if it is a subset-minimal model of the reduct $\{H(r) \leftarrow B^+(r) \mid r \in P_{grd}, B^-(r) \cap \mathcal{I} = \emptyset\}$, and we denote the set of all stable models of P by $AS(P)$.

2.2 Symmetry Breaking

Modern approaches detect symmetries of a given object by representing it, e.g., as an instance of the *graph automorphism problem*,[1] which is solved using methods from the *group theory*, see [Sakallah, 2009] for an overview.

Let X be a set of n elements $x_1, ..., x_n$, then we can define a *permutation* of X as a bijection ϕ that rearranges its elements. The *symmetric group* $G = \langle X_p, \phi \rangle$ is defined by the set X_p of all possible permutations of X closed under ϕ, and its subgroups $G' = \langle X'_p, \phi \rangle$ are called *permutation groups*, where $X'_p \subseteq X_p$. In *cycle notation*, we represent a permutation π as a product of disjoint cycles, where each cycle $(x_1\ x_2\ x_3\ \ldots\ x_k)$ means that the element x_1 is mapped to x_2, x_2 to x_3 and so on, until x_k is mapped back to x_1; the elements mapped to themselves are not contained in the cycles. Let $g \in X_p$ be an element of a group G, then g is a *generator* for G if any other element of the set X_p can be obtained by a finite number of applications of ϕ. A set of generators $A_X \subset X_p$ of a symmetric group G *generates* a subgroup G' of G if G' is the smallest subgroup containing A_X. If G' is all of G, then A_X is a *group generator*. A generator is *redundant* if it can be expressed in terms of other generators. The goal of graph automorphism algorithms is to find a set A_X of *irredundant* generators, i.e., a set that contains no redundant generators. Given a total order of the elements in X, an irredundant group generator can be used to introduce a set of constraints that eliminates all permutations of its elements: this approach is called *lex-leader* and produces *Symmetry Breaking Constraints* (SBCs). Considering a set of irredundant generators of G is an effective heuristic, since they allow for representing G compactly.

Symmetry-Breaking Answer Set Solving (SBASS) [Drescher *et al.*, 2011] detects and eliminates syntactic symmetries in ASP by adding ground SBCs to an input ground program P_{grd}. A symmetry of P_{grd} is given by a permutation π of ground atoms that that keeps the programs syntactically equivalent, i.e., P^{π}_{grd} has the same rules as P_{grd}, where P^{π}_{grd} is the set of rules obtained by applying π to the head and body literals of rules in P_{grd}. In the first step, SBASS transforms P_{grd} to a colored graph $\mathcal{G}_P$ such that permutation groups of $\mathcal{G}_P$ and their generators correspond one-to-one to those of P_{grd}. In the second step, it uses SAUCY [Darga *et al.*, 2004; Codenotti *et al.*, 2013] to find a set of group generators

[1] It consists of finding all the symmetries of a graph in terms of its generators; it is an attractive problem for the reduction since it can be solved efficiently for many graphs.

for $\mathcal{G}_P$. Finally, for each found generator SBASS constructs a set of SBCs and appends them to P_{grd}. Given the modified ground program, an ASP solver can efficiently avoid symmetric answer sets.

2.3 Inductive Logic Programming

Inductive Logic Programming [Cropper *et al.*, 2020] is a form of machine learning whose goal is learning a logic program that explains a set of observations in the context of some pre-existing knowledge. Since its foundation, the majority of the research in the topic covers Prolog semantic [Muggleton, 1995; Srinivasan, 2004; Cropper and Muggleton, 2016], even though applications in other logic paradigms appeared in the last years. The most important ILP system for ASP is Inductive Learning of Answer Set Programs (ILASP) [Law *et al.*, 2014; Law *et al.*, 2021], for whom several releases have been developed, extending its learning expressiveness.

The learning task for ILP $\langle P, E, H_M \rangle$ is defined by three elements: a background knowledge P, a set of (positive and negative) examples E, and a hypothesis space H_M, which defines all the rules that can be learned. The learned hypothesis is a subset of the hypothesis space that satisfies a specified learning setting: for ILASP, the setting is *learning from interpretation* [Cropper and Dumančić, 2020]. Before defining it, we need the terminology that ILASP's authors introduced. A *Partial Interpretation* (PI) is a pair of sets of atoms, $e_{pi} = \langle T, F \rangle$, called *inclusions* ($T$) and *exclusions* ($F$), respectively. Given a (total) interpretation $\mathcal{I}$ and a PI e_{pi}, we say that $\mathcal{I}$ *extends* e_{pi} if $T \subseteq \mathcal{I}$ and $F \cap \mathcal{I} = \emptyset$. We can augment e_{pi} with an ASP program C to obtain a *Context Dependent Partial Interpretation* (CDPI) $e = \langle e_{pi}, C \rangle$. Given a program P, a CDPI $\langle e_{pi}, C \rangle$, and an interpretation $\mathcal{I}$, we say that $\mathcal{I}$ is an *accepting answer set* of e with respect to P if $\mathcal{I} \in AS(P \cup C)$ such that $\mathcal{I}$ extends e_{pi}.

A learning task for ILASP is given by an ASP program P as background knowledge, two sets of CDPIs, E^+ and E^-, as positive and negative examples, and the hypothesis space H_M defined by a language bias M, which limits the potentially learnable rules. The learned hypothesis $H \subseteq H_M$ must respect the following criteria: *(i)* for each positive example $e \in E^+$, there is some accepting answer set of e with respect to $P \cup H$; and *(ii)* for any negative example $e \in E^-$, there is no accepting answer set of e with respect to $P \cup H$. If multiple hypotheses satisfy the conditions, the system returns one of the shortest. In the article [Law *et al.*, 2018], the authors extended the expressiveness of ILASP, by allowing noisy examples. With this setting, if an example e is not covered (i.e, there is an accepting answer set for e if it is negative, and none if it is positive) then, the corresponding weight is counted as a penalty. Therefore, the learning task becomes an optimization problem with two goals: minimize the length of H and minimize the total penalties for the uncovered examples.

Now, we will define the syntax of ILASP necessary for our work and refer the reader to the system's manual [Law *et al.*, 2021] for further details. A CDPI is expressed as follow:

```
#type(ID@W,{Inc},{Exc},{C}).
```

where `type` is either `pos` or `neg`; `ID` is an unique identifier for the example; `W` is a positive integer representing the example's weight (if not defined, the weight is infinite); `Inc` and `Exc` are two sets of atoms; and `C` is an ASP program. The language bias can be specified by *mode declarations*, which define the predicates that may appear in a rule, their argument types and their frequency. Since in our work we aim to learn constraints, we restrict the search space just to rules r_i with $H(r_i) = \bot$. To do so, we need to specify only the mode declaration for the body of a rule, expressed by `#modeb(R,P,(E))` where `R` and `E` are optional and `P` is a ground atom whose arguments are placeholders of type `var(t)` for some constant term `t`. In the learned rules, the placeholders will be replaced by variables of type `t`. The optional element `R` is a positive integer called *recall* which specifies the maximum number of times that the mode declaration can be used in each rule. Lastly, `E` is a construct, which further restricts the hypothesis search. We limit our interest to the `anti reflexive` option that works with predicates of arity 2. Using it, the predicate `P` should be generated with two distinguished variables.

Choosing an appropriate language bias is still one of the major challenges for modern ILP systems; if the bias does not provide enough limitations, the problem becomes intractable; on the other hand, a strong bias may remove the solution from the search space [Cropper and Dumančić, 2020].

3 Approach

We tackle combinatorial problems modeled in ASP such that the instances of a logic program P are generated by a discrete and often stationary stochastic process. Such situations occur, e.g., in industrial settings where the encoding of a manufacturing system is fixed and production orders vary. In this case, every problem instance can be seen as an outcome of the generation process. We assume that any instance *(i)* specifies the (*true*) atoms of unary domain predicates $p_1, \ldots, p_k$ in P, where c_i is the number of atoms that hold for each p_i; and *(ii)* the satisfiability of the instance depends on the number of atoms for each domain predicate, but not on the values of their terms. Thus, without loss of generality, we consider the terms for each p_i to be consecutive integers from 1 to c_i.

Our method exploits *instance-specific* SBCs on a representative set of instances and utilizes them to generate examples for an ILP task. Solving the task yields first-order constraints that remove symmetries in the analyzed problem instances as much as possible while preserving the instances' satisfiability. We consider two following learning modes: *(i)* the *default* mode is cautious by preserving all answer sets that are not filtered out by the ground SBCs; and *(ii)* the *sat* mode aims to learn tighter constraints by only guaranteeing that the learned constraints do not eliminate all answer sets for an instance.

To compute the examples, our approach relies on small satisfiable instances (i.e., with a low value for each c_j), subdivided into two parts: S and Gen. Each instance $g \in Gen$ defines a positive example with empty inclusions and exclusions, and g as context. These examples, denoted by Ex_{Gen}, guarantee that the learned constraints generalize for the target distribution since they force the constraints to preserve some solution for each $g \in Gen$. The instances $i \in S$ are used to obtain positive and negative examples, representing

Algorithm 1: Approach to lift SBCs with ILP

input : P, ABK, H_M, Gen, S, m

1 $Ex_{Gen} \leftarrow \emptyset$;
2 $Ex_S \leftarrow \emptyset$;
3 **foreach** $g \in Gen$ **do**
4 $\quad Ex_{Gen} \leftarrow Ex_{Gen} \cup \{pos(\emptyset, \emptyset, g)\}$;
5 **foreach** $i \in S$ **do**
6 $\quad IG \leftarrow$ Set of irredundant generators for i;
7 $\quad$ **foreach** $\mathcal{I} \in AS(P \cup i \cup ABK)$ **do**
8 $\quad\quad T \leftarrow atoms(IG) \cap \mathcal{I}$;
9 $\quad\quad F \leftarrow atoms(IG) \setminus \mathcal{I}$;
10 $\quad\quad$ **if** $lexLead(\langle T, F \rangle, IG)$ **then**
11 $\quad\quad\quad Ex_S \leftarrow Ex_S \cup \{neg(T, F, i)\}$;
12 $\quad\quad$ **else if** *m = default* **then**
13 $\quad\quad\quad Ex_S \leftarrow Ex_S \cup \{pos(\mathcal{I}, \emptyset, i)\}$;
14 $\quad\quad$ **else**
15 $\quad\quad\quad Ex_S \leftarrow Ex_S \cup \{pos(\emptyset, \emptyset, i)\}$;
16 $C \leftarrow$ Solve $\langle P \cup ABK, Ex_{Gen} \cup Ex_S, H_M \rangle$;
17 $ABK \leftarrow ABK \cup C$;

answer sets of $P \cup i$ to be preserved or filtered out, respectively, by corresponding SBCs. We denote their union by Ex_S, where positive examples represent whole answer sets in *default* mode, or, like instances in Gen, consist of empty inclusions and exclusions along with the context i in *sat* mode.

An ILP task further requires background knowledge and a hypothesis space H_M. Both of them are defined by the user (for a possible instantiation, see Section 4.1). The former consists of P along with *Active Background Knowledge*, denoted by ABK in Algorithm 1, including auxiliary predicate definitions and constraints learned so far. The latter contains the mode declarations, and we assume it to be general enough to entail ground SBCs by learned first-order constraints. The remaining inputs of Algorithm 1 consist of the instances in Gen and S as well as the learning mode m. For each answer set $\mathcal{I}$ of an instance $i \in S$ to be analyzed, the predicate $lexLead(\langle T, F \rangle, IG)$ at line 10 evaluates to *true* if $\mathcal{I}$ is dominated, i.e., $\mathcal{I}$ can be mapped to a lexicographically smaller, symmetric answer set by means of some irredundant generator in IG. In this case, the negative example $neg(T, F, i)$ is added to Ex_S in order to eliminate $\mathcal{I}$, while $pos(\mathcal{I}, \emptyset, i)$ or $pos(\emptyset, \emptyset, i)$ is taken as the positive example otherwise, depending on whether *default* or *sat* mode is selected. Positive examples of the form $pos(\emptyset, \emptyset, g)$ are also gathered in Ex_{Gen} for instances $g \in Gen$, and solving the ILP task at line 16 gives new constraints C to extend ABK.

4 Implementation

The implementation of our approach relies on CLINGO (consisting of the grounding and solving components GRINGO and CLASP), SBASS and ILASP, and is available at [Tarzariol *et al.*, 2021]. Figure 1 shows the pipeline to generate the examples for a given instance $i \in S$ (the for-loop at line 5 of Algorithm 1). First, P, i, and ABK are grounded with GRINGO to get the ground program P_{grd} in SMODELS format.

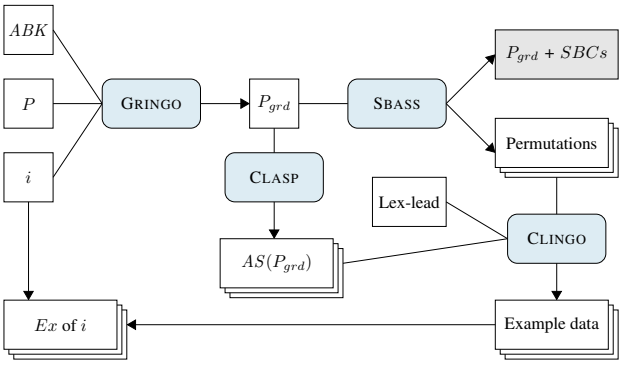

Figure 1: Pipeline to compute examples from instance i.

Then, the solver CLASP enumerates all its answer sets, obtaining $AS(P_{grd})$. Independently, SBASS is run on P_{grd} with the option `--show` to output the set of permutation group generators. This set contains the vertex permutations of $\mathcal{G}_P$, expressed in cycle notation. We extract the cycles defined by vertices representing atoms of P_{grd} and transform them from SMODELS format back into their original symbolic representation (by a predicate and terms), using the integer values of terms as lexicographic ordering criterion. Next, for efficiency reasons, we partition the permutations into a set C of clusters according to the involved atoms. More precisely, two permutations belong to the same cluster if they share a common atom; otherwise, they are considered separately. For each cluster $c \in C$, we identify the symmetric answer sets in $AS(P_{grd})$ according to c, by using an ASP encoding to evaluate SBCs based on the *lex-leader predicate* given in [Sakallah, 2009]. The encoding returns the undominated atom assignments concerning the atoms in c. For each $\mathcal{I} \in AS(P_{grd})$, we check whether an assignment according to $\mathcal{I}$ leads to unsatisfiability. In this case, $\mathcal{I}$ is a symmetric answer set and, therefore, produces a negative example. We assign a unique identifier to each negative example and a weight of 100. Due to the weights, ILASP returns a set of constraints even if some negative examples are not covered; moreover, we use uniform weights so that all negative examples have the same relevance and as many as possible are to be eliminated. Since we consider several clusters of permutations, the same answer set $\mathcal{I}$ may be symmetric for more than one cluster. If $\mathcal{I}$ is symmetric for n clusters, our system produces n negative examples for it, with a weight of $\lceil \frac{100}{n} \rceil$ for each. Lastly, answer sets that were not found to be dominated for any of the clusters in C yield positive examples according to the selected mode. Such positive examples are unweighted; thus, the learned hypothesis must cover all of them.

4.1 ILP Learning Task

After considering the example generation, we specify components of the ILP learning task that are suitable for the learning of constraints. The idea is to encode the predicates used by *lex-leader* symmetry breaking to order atoms and extract the maximal values for domain predicates. Since the mode declarations of ILASP (v4.0.0) do not support arithmetic built-ins such as $<$, we provide auxiliary predicates

in ABK to simulate them. Presupposing the presence of unary domain predicates $p_1, \ldots, p_k$ with integers from 1 to c_i for each p_i, ABK defines the auxiliary predicates $\texttt{max}p_i(c_i)$ for each p_i and $\texttt{lessThan}(t_1, t_2)$ for each pair of integers $1 \leq t_1 < t_2 \leq \max\{c_i \mid i = 1, \ldots, k\}$. These two predicates are minimal for overcoming limitations of ILASP to learn lex-leader SBCs. The selection of small yet representative instances for S and Gen depends on the considered problem. Regarding S, we pursued the strategy that empirically determines instances for which SBASS yields a manageable number of permutation group generators. The instances in Gen are usually larger yet still solvable in a short running time to check that the learned constraints generalize.

Our language bias includes $\texttt{\#modeb(2,}p_i(\texttt{var}(t_i))\texttt{)}$ and $\texttt{\#modeb(1,max}p_i(\texttt{var}(t_i))\texttt{)}$ as mode declarations for each domain predicate p_i, where t_i is a placeholder indicating the domain for which each p_i holds. Moreover, for each (non-auxiliary) predicate P appearing in some of the generators computed for instances in S, we use #modeb(2,P), where the domains of variables in atoms of P are indicated by a vector of the placeholders in $\{t_i \mid i = 1, \ldots, k\}$, depending on the role of P in the given program P. In addition, we include mode declarations $\texttt{\#modeb(2,lessThan(var(}t_i\texttt{),var(}t_j\texttt{)))}$ for all $i, j = 1, \ldots, k$, with the option anti reflexive in case $i = j$.

Iterative learning. Inspired by the *lifelong learning* approach [Cropper *et al.*, 2020], we apply our framework incrementally to a split learning task. In every iteration, we exploit the constraints learned so far to tackle the remaining symmetries. To this end, we divide the hypothesis space for programs with three or more types of variables in the language bias: in the first ILP run, the mode declarations are restricted to two types of variables, say t_1 and t_2, and then they are progressively extended to further types from t_3 to t_k. This iterative approach can speed up the learning procedure and required less than a minute of running time for each iteration on the combinatorial problems investigated in Section 5.

Example. To illustrate a feasible outcome of our ILP approach, let us inspect constraints learned for the pigeon-hole problem, which is about checking whether p pigeons can be placed into h holes such that each hole contains at most one pigeon. We use the following ASP encoding for this problem:

```
pigeon(X-1) :- pigeon(X), X > 1.
hole(X-1) :- hole(X), X > 1.
{p2h(P,H) : hole(H)} = 1 :- pigeon(P).
:- p2h(P1,H), p2h(P2,H), P1 != P2.
```

Assume that S consists of the instance with three pigeons and three holes only, which has six answer sets. Analyzing the instance with SBASS gives four generators identifying five symmetric answer sets. These are encoded as negative examples:

```
#neg(id1@100, {p2h(2,3), p2h(1,2), p2h(3,1)},
{p2h(2,1), p2h(1,1), p2h(3,3), p2h(1,3),
 p2h(3,2), p2h(2,2)}, {pigeon(3). hole(3).}).
#neg(id3@100, {p2h(2,1), p2h(3,2), p2h(1,3)},
{p2h(1,1), p2h(3,3), p2h(3,1), p2h(2,2),
 p2h(2,3), p2h(1,2)}, {pigeon(3). hole(3).}).
#neg(id4@100, {p2h(2,3), p2h(1,1), p2h(3,2)},
{p2h(2,1), p2h(3,3), p2h(3,1), p2h(1,3),
 p2h(2,2), p2h(1,2)}, {pigeon(3). hole(3).}).
#neg(id5@100, {p2h(2,1), p2h(3,3), p2h(1,2)},
{p2h(1,1), p2h(3,1), p2h(1,3), p2h(3,2),
 p2h(2,3), p2h(2,2)}, {pigeon(3). hole(3).}).
#neg(id6@100, {p2h(1,1), p2h(3,3), p2h(2,2)},
{p2h(2,1), p2h(3,1), p2h(1,3), p2h(3,2),
 p2h(2,3), p2h(1,2)}, {pigeon(3). hole(3).}).
```

The single positive example in *default* mode is the following:

```
#pos(id2, {p2h(3,1), p2h(2,2), p2h(1,3)},
{}, {pigeon(3). hole(3).}).
```

After running ILASP, the learned first-order constraints are:

```
:- p2h(X,Y), lessThan(Z,Y), maxpigeon(X).
 % do not assign the pigeon with the max
 % label to a hole other than the first one
:- p2h(X,Y), lessThan(X,Y), lessThan(Y,Z).
 % for all but the last hole, do not assign
 % a pigeon with a smaller label to the hole
```

5 Experiments

To evaluate our approach and the implementation design, we applied our framework to a series of combinatorial search problems. For each considered problem, we compared the running time of the original encoding, the version extended with our learned constraints, and the *instance-specific* approach of SBASS. The learned constraints depend on several aspects, e.g., the selected inputs or whether and how we apply the iterative learning approach. In the following, we report results for the constraints learned applying the definitions of Section 4.[2] We ran our tests on an Intel® Xeon® E5520 machine under Linux (Ubuntu 18.04.3), where each run was limited to 900 seconds time and 20 GB memory. In Table 1 to Table 4, the satisfiable instances are shown in grey rows, while the white rows contain unsatisfiable instances. The column BASE refers to CLINGO (v5.4.0) run on the original encoding, while ABK-DEF and ABK-SAT report results for the original encoding augmented with first-order constraints learned in the *default* or *sat* mode, respectively. The time required by SBASS to compute ground SBCs is given in the corresponding column, and CLASP$^\pi$ provides the solving time obtained with these ground SBCs. Runs that did not finish within the time limit of 900 seconds are indicated by TO entries.

We first tested the pigeon-hole problem, working without any division and iterative analysis of the language bias, and the two learning modes led to similar constraints. The running time comparison in Table 1 shows that the *default* and the *sat* mode of our framework bring about a similar speedup for solving satisfiable as well as unsatisfiable instances. In fact, the vast problem symmetries are cut by the learned first-order constraints, which is particularly important in case of unsatisfiability, where runs on the original encoding without additional constraints do not finish within the time limit. While SBASS also manages to handle the two smallest instances, the computation of permutation group generators becomes too expensive when the instance size grows, in which case we cannot run CLASP$^\pi$ with ground SBCs from SBASS.

Next, we tested two extensions of the pigeon-hole problem, adding color and owner assignments. The pigeon-hole problem with colors associates a color with each pigeon and requires pigeons placed into neighboring holes to be of the same color. The version with colors and owners additionally assigns an owner to each pigeon and imposes the

[2] Detailed settings are provided at [Tarzariol *et al.*, 2021].

	ABK-DEF	ABK-SAT	BASE	SBASS	CLASP$^\pi$
p50-h49	0.115	0.157	TO	54.581	2.978
p50-h50	0.117	0.116	0.272	54.521	2.334
p100-h99	1.139	1.188	TO	TO	–
p100-h100	1.200	1.301	1.845	TO	–
p200-h199	8.648	8.443	TO	TO	–
p200-h200	8.713	9.495	16.523	TO	–
p300-h299	30.643	30.237	TO	TO	–
p300-h300	30.510	30.333	55.606	TO	–
p400-h399	72.561	72.571	TO	TO	–
p400-h400	73.701	73.554	135.031	TO	–

Table 1: Runtime in seconds for pigeon-hole problem.

	ABK-DEF	ABK-SAT	BASE	SBASS	CLASP$^\pi$
c1-p12-h11	2.643	0.005	TO	0.758	0.018
c1-p52-h52	0.339	0.223	0.342	63.749	1.622
c2-p12-h12	9.512	0.012	TO	0.085	TO
c2-p52-h53	0.783	0.437	TO	103.843	TO
c3-p12-h13	6.442	0.021	TO	0.239	TO
c3-p52-h54	6.892	0.972	2.814	653.216	TO
c4-p12-h14	6.229	0.023	TO	0.675	TO
c4-p52-h55	5.130	1.579	TO	633.469	TO
c5-p12-h15	7.928	0.043	TO	1.076	TO
c5-p52-h56	27.226	2.383	8.447	TO	–

Table 2: Runtime in seconds for pigeon-hole problem with colors.

	ABK-DEF	ABK-SAT	BASE	SBASS	CLASP$^\pi$
o1-c3-p12-h13	2.327	0.015	TO	0.200	TO
o1-c3-p52-h54	2.818	0.858	2.760	394.272	TO
o2-c3-p12-h13	1.865	0.017	TO	0.395	TO
o2-c3-p52-h54	1.463	1.294	TO	745.101	TO
o3-c1-p12-h13	2.335	0.014	TO	0.320	TO
o3-c1-p52-h54	2.619	1.151	2.732	392.267	TO
o4-c4-p12-h14	1.270	0.041	TO	1.238	TO
o4-c4-p52-h55	6.316	3.036	10.041	TO	–
o5-c5-p12-h15	0.988	0.090	TO	2.418	TO
o5-c5-p52-h56	20.040	5.031	18.074	TO	–

Table 3: Runtime in seconds for pigeon-hole problem with colors and owners.

	ABK-DEF	ABK-SAT	BASE	SBASS	CLASP$^\pi$
p2-c6-t13	0.723	0.042	292.056	0.092	18.672
p2-c80-t160	7.861	8.498	10.274	TO	–
p3-c6-t13	0.555	0.059	344.526	0.217	102.544
p3-c80-t160	21.588	22.331	29.295	TO	–
p4-c6-t13	0.609	0.063	303.573	0.728	145.049
p4-c80-t160	41.551	43.521	59.380	TO	–
p5-c6-t13	0.705	0.072	329.670	1.174	582.101
p5-c80-t160	69.009	72.756	99.911	TO	–
p4-c7-t15	18.770	0.464	TO	1.060	TO
p15-c15-t30	9.187	10.323	6.851	TO	–

Table 4: Runtime in seconds for house-configuration problem.

same constraint as with the colors for owners as well. For the pigeon-hole problem with color assignments, we divided the language bias into two parts: the first limiting to predicates whose atoms exclusively include variables of the types `pigeon` and `hole`, while the second part allows variables to be of the type `color` too. Likewise, the problem version with owners and colors required a third language bias extension to variables of the type `owner`. For both extensions of the pigeon-hole problem, the first-order constraints learned in *default* mode turned out to be weaker than those obtained in *sat* mode, while either kind of constraints helped to improve the search for solutions. Table 2 and Table 3 show similar results: the constraints learned with the *sat* mode lead to the fastest running times for both satisfiable and unsatisfiable instances. For small unsatisfiable instances, the ground SBCs from SBASS lead to better performance than the constraints learned with the *default* mode. However, as soon as the color (or owner) dimension grows, the runs of CLASP$^\pi$ reach the timeout. This behavior is due to the redundancy of the ground SBCs, which slow down the search instead of facilitating it. For some of the satisfiable instances, finding a solution with the constraints learned in *default* mode takes longer than with the original encoding alone, but the latter also has timeouts that do not occur with our learned first-order constraints.

Lastly, we applied our framework to the house-configuration problem [Friedrich *et al.*, 2011], which consists of assigning t things of p persons to c cabinets, where each cabinet has a capacity limit of two things that must belong to the same owner. The running times in Table 4 exhibit the same trend as observed on the previous problems that our first-order constraints help the search, especially those learned with the *sat* mode. In some cases, the original encoding is quicker to solve satisfiable instances, but it takes considerably longer for unsatisfiable ones. On the other hand, SBASS brings a moderate speedup for unsatisfiable instances, but its performance suffers a lot when the problem size grows.

6 Conclusions

This paper introduces a new method to lift the SBCs of combinatorial problems for a target distribution of instances. Our framework addresses the limitations of common *instance-specific* approaches, like SBASS, since: *(i)* the knowledge is transferable, as learned constraints preserve the satisfiability for the considered instance distribution; *(ii)* the first-order constraints are easier to interpret than ground SBCs; *(iii)* the SBCs are computed offline, allowing for addressing large-scale instances, as shown in our experiments; and *(iv)* the learned constraints are non-redundant, avoiding performance degradation due to an excessive ground representation size.

In the future, we aim to investigate whether the learning of SBCs can be readily applied or further adapted to advanced industrial problems, such as the *Partner Unit Problem* [Dodaro *et al.*, 2016], as well as complex combinatorial problems with specific instance distributions, like the identification of *Graceful Graphs* [Petrie and Smith, 2003]. For such application scenarios, the language bias may be enriched, possibly extending the background knowledge with additional predicates. Moreover, we intend to provide automatic mechanisms to select suitable instances for S and Gen from instance collections, support lifelong learning, and further optimize the grounding and solving efficiency of learned constraints.

Acknowledgments

This work was partially funded by KWF project 28472, cms electronics GmbH, FunderMax GmbH, Hirsch Armbänder GmbH, incubed IT GmbH, Infineon Technologies Austria AG, Isovolta AG, Kostwein Holding GmbH, and Privatstiftung Kärntner Sparkasse. We thank the anonymous reviewers for helpful comments.

References

[Codenotti *et al.*, 2013] P. Codenotti, H. Katebi, K. Sakallah, and I. Markov. Conflict analysis and branching heuristics in the search for graph automorphisms. In *IEEE 25th International Conference on Tools with Artificial Intelligence*, pages 907–914. IEEE Computer Society, 2013.

[Cohen *et al.*, 2006] D. Cohen, P. Jeavons, C. Jefferson, K. Petrie, and B. Smith. Symmetry definitions for constraint satisfaction problems. *Constraints*, 11(2-3):115–137, 2006.

[Cropper and Dumančić, 2020] A. Cropper and S. Dumančić. Inductive logic programming at 30: A new introduction. https://arxiv.org/abs/2008.07912, 2020.

[Cropper and Muggleton, 2016] A. Cropper and S. Muggleton. Metagol. https://github.com/metagol/metagol, 2016. Accessed: 2021-05-21.

[Cropper *et al.*, 2020] A. Cropper, S. Dumančić, and S. Muggleton. Turning 30: New ideas in inductive logic programming. In *29th International Joint Conference on Artificial Intelligence*, pages 4833–4839. ijcai.org, 2020.

[Darga *et al.*, 2004] P. Darga, H. Katebi, M. Liffiton, I. Markov, and K. Sakallah. Saucy. http://vlsicad.eecs.umich.edu/BK/SAUCY, 2004. Accessed: 2021-05-21.

[Devriendt *et al.*, 2016] J. Devriendt, B. Bogaerts, M. Bruynooghe, and M. Denecker. On local domain symmetry for model expansion. *Theory and Practice of Logic Programming*, 16(5-6):636–652, 2016.

[Dodaro *et al.*, 2016] C. Dodaro, P. Gasteiger, N. Leone, B. Musitsch, F. Ricca, and K. Schekotihin. Combining answer set programming and domain heuristics for solving hard industrial problems. *Theory and Practice of Logic Programming*, 16(5-6):653–669, 2016.

[Drescher *et al.*, 2011] C. Drescher, O. Tifrea, and T. Walsh. Symmetry-breaking answer set solving. *AI Communications*, 24(2):177–194, 2011.

[Erdem *et al.*, 2016] E. Erdem, M. Gelfond, and N. Leone. Applications of ASP. *AI Magazine*, 37(3):53–68, 2016.

[Falkner *et al.*, 2018] A. Falkner, G. Friedrich, K. Schekotihin, R. Taupe, and E. Teppan. Industrial applications of answer set programming. *Künstliche Intelligenz*, 32(2-3):165–176, 2018.

[Friedrich *et al.*, 2011] G. Friedrich, A. Ryabokon, A. Falkner, A. Haselböck, G. Schenner, and H. Schreiner. (Re)configuration using answer set programming. In *IJCAI 2011 Workshop on Configuration*, pages 17–24. CEUR-WS.org, 2011.

[Gebser *et al.*, 2012] M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub. *Answer Set Solving in Practice*. Morgan and Claypool Publishers, 2012.

[Gelfond and Lifschitz, 1991] M. Gelfond and V. Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9:365–385, 1991.

[Law *et al.*, 2014] M. Law, A. Russo, and K. Broda. Inductive learning of answer set programs. In *14th European Conference on Logics in Artificial Intelligence*, pages 311–325. Springer-Verlag, 2014.

[Law *et al.*, 2018] M. Law, A. Russo, and K. Broda. Inductive learning of answer set programs from noisy examples. *Advances in Cognitive Systems*, 7:57–76, 2018.

[Law *et al.*, 2021] M. Law, A. Russo, and K. Broda. Ilasp. http://www.ilasp.com, 2021. Accessed: 2021-05-21.

[Lifschitz, 2019] V. Lifschitz. *Answer Set Programming*. Springer-Verlag, 2019.

[Margot, 2010] F. Margot. Symmetry in integer linear programming. In *50 Years of Integer Programming 1958-2008*, pages 647–686. Springer-Verlag, 2010.

[Mears *et al.*, 2008] C. Mears, M. García de la Banda, M. Wallace, and B. Demoen. A novel approach for detecting symmetries in CSP models. In *5th International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, pages 158–172. Springer-Verlag, 2008.

[Muggleton, 1995] S. Muggleton. Inverse entailment and Progol. *New Generation Computing*, 13(3-4):245–286, 1995.

[Petrie and Smith, 2003] K. Petrie and B. Smith. Symmetry breaking in graceful graphs. In *9th International Conference on Principles and Practice of Constraint Programming*, pages 930–934. Springer-Verlag, 2003.

[Puget, 2005] J. Puget. Automatic detection of variable and value symmetries. In *11th International Conference on Principles and Practice of Constraint Programming*, pages 475–489. Springer-Verlag, 2005.

[Sakallah, 2009] K. Sakallah. Symmetry and satisfiability. In *Handbook of Satisfiability*, pages 289–338. IOS Press, 2009.

[Srinivasan, 2004] A. Srinivasan. The Aleph manual. https://www.cs.ox.ac.uk/activities/programinduction/Aleph, 2004. Accessed: 2021-05-21.

[Tarzariol *et al.*, 2021] A. Tarzariol, M. Gebser, and K. Schekotihin. ILP symmetry breaking. https://github.com/prosysscience/Symmetry_Breaking_with_ILP, 2021. Accessed: 2021-05-21.

[Walsh, 2012] T. Walsh. Symmetry breaking constraints: Recent results. In *26th AAAI Conference on Artificial Intelligence*, pages 2192–2198. AAAI Press, 2012.