# A Model-Oriented Approach for Lifting Symmetries in Answer Set Programming*

Alice Tarzariol
Alpen-Adria-Universität
Klagenfurt, Austria
alice.tarzariol@aau.at

When solving combinatorial problems, pruning symmetric solution candidates from the search space is essential. 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 since the computed SBCs are propositional and, therefore, can neither be meaningfully interpreted nor transferred to other instances. As a result, a time-consuming recomputation of SBCs must be done before every invocation of a solver. 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 a form of machine learning called Inductive Logic Programming. After targeting simple combinatorial problems, we aim to extend our method to be applied also for advanced decision and optimization problems.

## 1 Introduction and Problem Description

A common approach for solving combinatorial problems is modelling them using declarative programming paradigms, e.g., Answer Set Programming (ASP) [14, 15, 2]. In general, defining such models is relatively simple, and the obtained encodings are easy to understand. However, although correct, a trivial encoding might become useless because of its performance when solving non-trivial instances. Indeed, the solving phase turns infeasible when the size of input instances and, correspondingly, the number of possible solution candidates start to grow [10]. In many cases, these candidates are symmetric, i.e., one candidate can easily be obtained from another by renaming constants. Therefore, the ability to encode *Symmetry Breaking Constraints* (SBCs) in a program becomes an essential skill for programmers as they prune a consistent part of the search space. Nevertheless, identifying symmetric solutions and formulating constraints that remove only them might be a time-consuming and challenging task. As a result, various tools emerged for avoiding the computation of symmetric solutions. A popular approach consists in automatically detecting and introducing a set of SBCs using properties of permutation groups [24]. The system SBASS [11] implements this type of approach for ground ASP programs.

Unfortunately, the computational advantages derived from SBASS or, more generally, from any *instance-specific* symmetry breaking approach, do not carry forward to large-scale instances or advanced encodings. Indeed, instance-specific approaches often require as much time as it takes to solve the original problem. Moreover, ground SBCs generated approaches are *(i)* not transferable, since the knowledge obtained is limited to a single instance; *(ii)* usually hard to interpret and comprehend because they are not expressed with a symbolic representation; *(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. In particular, when solving instances sharing similar characteristics, the

*This PhD work is conducted under the supervision of professors Martin Gebser and Konstantin Schekotihin.

Y. Lierler, J.F. Morales, C. Dodaro, V. Dahl, M. Gebser, T. Tekle (Eds.):
International Conference on Logic Programming 2022 (ICLP'22).
EPTCS 364, 2022, pp. 200–210, doi:10.4204/EPTCS.364.35

identified symmetries follow the same structure for the whole set. Thus, the instance-specific approaches require extra computation time to identify the same symmetries applied to every single instance. Assuming we consider a combinatorial ASP program and a distribution of similar problem instances, in our work, we aim to overcome the limitations of instance-specific approaches by lifting ground symmetries using a form of machine learning called Inductive Logic Programming (ILP) [6]. The resulting first-order constraints can be applied to any instance drawn from the considered distribution, and they should speed up the identification of satisfiable/unsatisfiable instances.

## 2 Background and Existing Literature

In this section, we will briefly introduce the concepts considered in our work, i.e., Answer Set Programming, Inductive Logic Programming and Symmetry Breaking techniques.

### 2.1 Answer Set Programming

ASP is a declarative programming paradigm that applies non-monotonic reasoning and relies on the stable model semantics [16]. Over the past decades, it has attracted considerable interest thanks to its elegant syntax, expressiveness, and efficient system implementations. It showed promising results in numerous domains, including industrial, robotics, or biomedical applications [12].

**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 $p(\overline{t})$, where $p$ is a predicate, $\overline{t}$ 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 $\overline{t}$ is either a variable or a constant. A *literal* $l$ is 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* $P_{grd}$, which is obtained by replacing each rule $r \in P$ with its instances obtained by substituting the variables in $r$ by constants occurring in $P$. Then, an *interpretation* $\mathscr{I}$ is a set of (*true*) ground atoms occurring in $P_{grd}$ that does not contain $\bot$. An interpretation $\mathscr{I}$ *satisfies* a rule $r \in P_{grd}$ if $B^+(r) \subseteq \mathscr{I}$ and $B^-(r) \cap \mathscr{I} = \emptyset$ imply $H(r) \in \mathscr{I}$, and $\mathscr{I}$ is a *model* of $P$ if it satisfies all rules $r \in P_{grd}$. A model $\mathscr{I}$ of $P$ is *stable* if it is a subset-minimal model of the reduct $\{H(r) \leftarrow B^+(r) \mid r \in P_{grd}, B^-(r) \cap \mathscr{I} = \emptyset\}$, and we denote the set of all stable models, also called answer sets, of $P$ by $AS(P)$.

### 2.2 Inductive Logic Programming

ILP is a form of machine learning whose goal is to learn a logic program that explains a set of observations in the context of some pre-existing knowledge. The most expressive ILP system for ASP is *Inductive Learning of Answer Set Programs* (ILASP) [20, 21], which can be used to solve a variety of ILP tasks. A learning task $\langle B, E^+, E^-, H_M \rangle$ is defined by four elements: a background knowledge $B$, a

set of positive and negative examples, respectively $E^+$ and $E^-$, and lastly a hypothesis space $H_M$, which defines the rules that can be learned. Each example $e \in E^+ \cup E^-$ is a pair $\langle e_{pi}, C \rangle$ called *Context Dependent Partial Interpretation*, where *(i)* $e_{pi}$ is a *Partial Interpretation* defined as pair of sets of atoms $\langle T, F \rangle$, called *inclusions* ($T$) and *exclusions* ($F$), respectively, and *(ii)* $C$ is an ASP program defining the *context* of $e_{pi}$. Given a (total) interpretation $\mathscr{I}$ of a program $P$ and a partial interpretation $e_{pi}$, we say that $\mathscr{I}$ *extends* $e_{pi}$ if $T \subseteq \mathscr{I}$ and $F \cap \mathscr{I} = \emptyset$. Given an ASP program $P$, an interpretation $\mathscr{I}$, and an example $e = \langle e_{pi}, C \rangle$, we say that $\mathscr{I}$ is an *accepting answer set* of $e$ with respect to $P$ if $\mathscr{I} \in AS(P \cup C)$ such that $\mathscr{I}$ extends $e_{pi}$.

Each hypothesis $H \subseteq H_M$ learned by ILASP must respect the following criteria: *(i)* for each positive example $e \in E^+$, there is some accepting answer set of $e$ with respect to $B \cup H$; and *(ii)* for any negative example $e \in E^-$, there is no accepting answer set of $e$ with respect to $B \cup H$. If multiple hypotheses satisfy the conditions, the system returns one of those with the lowest cost. By default, the cost $c_r$ of each rule $r \in H_M$ corresponds to its number of literals [18]; however, the user can define a custom scoring function for defining the rule costs. ILASP allows to define learning tasks with noisy examples [19]. With this setting, if an example $e$ is not covered, i.e., there is an accepting answer set for $e$ if it is negative, or none if $e$ is positive, the corresponding weight is counted as a penalty. If no dedicated weight is specified, the example's weight is infinite, thus forcing the system to cover the example. Therefore, the learning task becomes an optimization problem with two goals: minimize the cost of $H$ and minimize the total penalties for the uncovered examples. In our work, we use the most recent version of ILASP (v4.1.2), which implements the search approach *Conflict Driven ILP* [17].

## 2.3 Symmetry Breaking

Modern symmetry breaking approaches can be split into two families: *instance-specific* and *model-oriented* approaches [24, 30]. The former identify symmetries for a particular instance at hand by obtaining a ground program, computing ground SBCs, composing a new extended program, and solving it [23, 4, 11]. The system SBASS [11] implements this type of approach for ground ASP programs. As mentioned in Section 1, when applied to large-scale instances or advanced encodings, the *instance-specific* symmetry breaking approaches may struggle to compute the symmetries in a reasonable time, or the resulting SBCs are redundant, leading to more drawbacks than benefits for the solver. Moreover, the ground SBCs are often difficult to understand as they are not expressed in symbolic representation; e.g., the SBCs produced by SBASS are represented in SMODELS format [25].

In contrast, *model-oriented* approaches aim to find general SBCs that depend less on a particular instance. The method presented in [9] 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. The method of [22] computes SBCs by generating small instances of parametrized constraint programs, and then finds candidate symmetries using SAUCY [3, 8] – 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.

# 3 Current Research

This section describes the research goals targeted for my PhD, and the current state of my work.

## 3.1 Goal of the Research

To the best of our knowledge, currently there are no *model-oriented* systems that lift ground SBCs for ASP programs. Therefore, we aim to introduce a novel *model-oriented* method that generalizes the process of discarding redundant solution candidates for ASP instances of a target domain using ILP. More precisely, we identify and lift SBCs of small problem-instances, obtaining a set of interpretable first-order constraints. Such constraints 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. After targeting simple combinatorial problems, we aim to extend our method to be applied also for advanced decision and optimization problems.

The research goals of this work are the following:

**RG 1** Given an ASP combinatorial program and a target instances distribution, define a learning framework capable of obtaining first-order constraints that speed up the solving of satisfiable and unsatisfiable instances.

**RG 2** Develop an approach capable of applying the learning framework iteratively.

**RG 3** Investigate how the framework can be extended to enable learning first-order constraints for advanced combinatorial problems.

**RG 4** Design and implement systems that automate parameter selection for the framework for guiding the learning of first-order constraints that speed up solving.

**RG 5** Extend the expressiveness of the learning framework to analyse the symmetries on optimization problems.

For the research goal **RG 1**, we assume that the instances analyzed for a given ASP combinatorial problem $P$ follow a specific distribution. Moreover, we can easily provide a set of simple instances (i.e., such that the total number of solutions can be managed by SBASS, CLINGO[1] and ILASP) that entail the symmetries of the whole target distribution. Our framework defines the set of examples of an ILP task such that ILASP learns first-order constraints that remove symmetric solutions while preserving the satisfiability of the instances in the considered distribution. To do so, the framework relies on SBASS to compute the SBCs of a set of small, satisfiable, and representative problem-instances, identified with $S$. More precisely, for each instance $i \in S$, we find its symmetries expressed as a set of irredundant generators, $IG(i)$, and we enumerate the set of its answer sets, $AS(i)$. Then, for each interpretation $\mathscr{I} \in AS(i)$, we define an example where $IG(i) \cap \mathscr{I}$ and $IG(i) \setminus \mathscr{I}$ are the inclusions and exclusions of a partial interpretation and using $i$ as context. If $\mathscr{I}$ is dominated, i.e., $\mathscr{I}$ can be mapped to a lexicographically smaller, symmetric answer set by means of some irredundant generator in $IG(i)$, the example is labelled as negative, otherwise, positive. Together with $S$, we use another set of instances $Gen$, where each $g \in Gen$ defines a single positive example with empty inclusions and exclusions and $g$ as context. These examples 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 two sets of instances, $S$ and $Gen$, and the language bias for the ILP task are defined by the user, while $P$ is used as background knowledge.

[1] In our work, to find the solutions of ASP programs, we use the system CLINGO, consisting of the grounding and solving components GRINGO and CLASP.

After the learning phase, a validation set of satisfiable instances, $V$, is used to check whether the learn constraints in *ABK* are correct. If there is a satisfiable instance $v \in V$ for which no solution is found, we discard the learned constraints and add $v$ in *Gen*. Subsequently, we rerun our framework and repeat the procedure until all the instances in the validation set are satisfiable.

The research goal **RG 2** consists of identifying techniques that can speed up the learning phase for the tasks analysed in **RG 1** as the learning time represent a critical aspect on ILP [7]. To do so, we define a procedure that applies our framework iteratively to learn the first-order constraints incrementally. More precisely, we outline a criterion for splitting the framework inputs to create sub-learning tasks. This approach speeds up the computation of first-order constraints, especially when the program contains symmetries independent from each others. To do so, we introduce an auxiliary ASP file called *Active Background Knowledge* or *ABK*, containing the constraints learned so far. By including *ABK* in the background knowledge, we can rerun our framework taking into account the constraints previously learned.

The current definition of our framework yields a number of examples proportional to the number of solutions for each problem-instance in $S$. Therefore, if it gets difficult to compute all the solutions for an instance in $S$ to analyze, the resulting formulation of the ILP task to learn constraints can become prohibitive. The research goal **RG 3** consists of overcoming the limitations of the current framework, in order to apply it to advanced combinatorial problems. With the term "advanced", we refer to problems whose solutions rely on atoms of multi-dimensional instead of just unary predicates, so that there might be no trivial instances to analyse. An example of this kind of problems is the *Partner Units Problem* (PUP) [1, 29], which is an abstract representation of configuration problems occurring in railway safety or building security systems. Considering the smallest PUP instance representing a class of building security systems named *double* by [1], CLINGO finds 145368 solutions, 98.9% of which can be identified as symmetric by SBASS (for instance, by renaming the units of a solution). Thus, the enumeration of symmetries for PUP instances is problematic, even for the smallest and simplest ones. To overcome this problem, we need to revise the framework's approach such that it manages to cope with any number of solutions of the analyzed instances, for example, by sampling a subset of answer sets. To be effective, the sample size must be small while containing an adequate number of positive and negative examples. Besides, two further limitations need to be addressed concerning ILASP's searching technique. The former is the inefficiency of the default ILASP's conflict analysis techniques[2] when applied to positive examples producing many solutions (i.e., the examples generated from the PUP instances in *Gen*). The second issues concerns the optimal criterion for the learned hypothesis, which considers only the length of the constraints and not the nature of the predicates. To overcome both issues, we aim to devise a specific conflict analysis technique that exploit the nature of the learning task (namely, constraints learning) and a custom scoring function for ILASP that provides further information for learning efficient[3] constraints.

For the research goal **RG 4**, we aim to identify appropriate inputs to our framework automatically. So far the inputs must be chosen by the user, however, we would like to provide guidelines or automate the process of selecting the framework inputs. First, the selection method of our framework needs to assess the properties of candidate inputs. Then, the method should determine parameters leading to correct and performant first-order constraints. That is, it aims to find constraints preserving at least one solution for satisfiable instances and cutting down the solving time for unsatisfiable instances.

[2] A key component of *Conflict-Driven ILP*.

[3] Namely, constraints with a limited number of variables and, possibly, containing some predicates that are simplified during grounding.

Lastly, the research goal **RG 5** is to extend the applicability of our framework to optimization problems. The tool SBASS, used in our framework, can analyse most of the ASP rules as normal rules, choice rules, aggregates, and hard constraints; however, it does not support weak constraints. Therefore, we aim to define a reduction from programs with weak constraints to an equivalent representation that can be processed by SBASS. More specifically, we aim to introduce new (normal) rules that respect the symmetries of the optimisation rules. As a consequence, when running SBASS on the extended program, we get a finer partition of the solutions. Namely, it could be that two solutions, which were considered symmetric from the analysis of the original program, can be identified as non-symmetric after the introduction of the new rules.

## 3.2 Results Accomplished

We devised and implemented the learning framework of **RG 1** and a rough idea of **RG 2** in a conference paper [27]; subsequently, we formalised the method to split the learning task in a journal paper [28]. We applied the framework to simple ASP programs, namely, the pigeon-hole problem and two its extensions that consider also the assignments of colors and owners. Moreover, we tested the house-configuration problem [13]. For all the addressed problems, we suggested some guidelines to define the framework inputs, $S$, $Gen$, $H_M$, and $ABK$. Table 1 contains the solving times for the house-configuration problem; the satisfiable instances are shown in grey rows, while the white rows contain unsatisfiable instances. The column **BASE** refers to CLINGO (v5.5.0) run on the original encoding, while **ABK** reports results for the original encoding augmented with first-order constraints learned with our framework. 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. Therefore, the total time required for the online usage of SBASS is the sum of **BASE** and **CLASP**$^\pi$. Runs that did not finish within the time limit of 900 seconds are indicated by TO entries.

The running times in the table show the limits of SBASS both in the pre-solving phase, when computing the symmetries (obtaining a timeout for all the satisfiable instances), and when solving a program extended with redundant constraints (the performance degradation is visible with the instance `p5-c6-t13`). The **BASE** encoding is quicker than **SBASS+CLASP**$^\pi$ to solve satisfiable instances, although it takes considerably longer for unsatisfiable ones. On the other hand, the first-order constraints learned with our framework helped the search for satisfiable and, especially, unsatisfiable instances. Similar results have also been observed for the pigeon-hole problems analysed. The repository containing the implementation and complete experiments can be found at the following link: `https://github.com/prosysscience/Symmetry_Breaking_with_ILP/tree/extended`

We addressed the extension mentioned in **RG 3** in a paper presented at ICLP 2022 [26]. In this paper, we revised several parts of our framework in order to target PUP instances supplied by [1], studying the *double*, *doublev*, and *triple* instance collections. Instances of the same type represent buildings of similar topology with scaling parameters that follow a common distribution. Although the benchmark instances are synthetic, they represent a relevant configuration problem concerning safety and security issues in public buildings, like administration offices or museums. In addition, the scalable synthetic benchmarks are easy to generate and analyze.

Table 2 contains the solving time for the PUP instances in *double*, and the table follows the same structure as the previous, but with a timeout of 600 seconds. Moreover, it also considers the computational time obtained by running CLINGO on the advanced encoding **SYMM**[4] which incorporates handcrafted static symmetry breaking as well as an ordered representation [5] of assigned units. From the

[4]The encoding used for **SYMM** is taken from the paper [10], where it is called ENC2. From the same paper, we also take

| | ABK | BASE | SBASS | CLASP$^{\pi}$ |
|---|---|---|---|---|
| p2-c6-t13 | 0.329 | 219.753 | 0.095 | 12.951 |
| p2-c80-t160 | 5.024 | 6.583 | TO | – |
| p3-c6-t13 | 0.424 | 254.065 | 0.242 | 73.041 |
| p3-c80-t160 | 14.110 | 20.724 | TO | – |
| p4-c6-t13 | 0.349 | 221.784 | 0.453 | 105.145 |
| p4-c80-t160 | 27.299 | 40.121 | TO | – |
| p5-c6-t13 | 0.397 | 236.961 | 0.890 | 405.461 |
| p5-c80-t160 | 49.645 | 68.167 | TO | – |
| p4-c7-t15 | 14.229 | TO | 0.729 | TO |
| p15-c15-t30 | 2.525 | 4.155 | TO | – |

Table 1: Runtime in seconds for house-configuration problem.

| | ABK | SYMM | BASE | SBASS | CLASP$^{\pi}$ |
|---|---|---|---|---|---|
| dbl-10 | 0.01 | 0.01 | 0.02 | 0.04 | 0.02 |
| un-dbl-10 | 0.01 | 0.16 | 505.91 | 0.03 | TO |
| dbl-20 | 0.08 | 0.53 | 1.06 | 0.31 | 1.40 |
| un-dbl-20 | 0.34 | TO | TO | 0.28 | TO |
| dbl-30 | 0.46 | 2.21 | 1.70 | 3.12 | 0.91 |
| un-dbl-30 | 19.97 | TO | TO | 3.19 | TO |
| dbl-40 | 5.50 | 11.50 | TO | 14.58 | 482.17 |
| un-dbl-40 | 65.54 | TO | TO | 9.52 | TO |
| dbl-50 | 54.89 | 542.09 | TO | 57.91 | TO |
| un-dbl-50 | 61.27 | TO | TO | 48.63 | TO |

Table 2: Runtimes for PUP double

table, we can observe that **SYMM** leads to more robust CLINGO performance than the simpler **BASE** encoding, and the ground SBCs computed from SBASS (obtained by summing the time in **SBASS** with **CLASP**$^{\pi}$). Moreover, when comparing **SYMM** to **ABK**, we observe further significant performance improvements thanks to our approach, particularly on the unsatisfiable instances. That is, the learned *ABK* enables CLINGO to solve the considered PUP instances and efficiently prunes the search space, which must be fully explored in case of unsatisfiability. Similar results have been observed also for the other two type of instances, *doublev* and *triple*. The repository containing the implementation and complete experiments can be found at the following link: `https://github.com/prosysscience/Symmetry_Breaking_with_ILP/tree/pup`

### 3.3 Open Issues and Expected Achievements

The research goals that we still need to tackle are **RG 4** and **RG 5**. For the former, we would like to help the user on deciding the elements in *S* and *Gen*, and automatically identify a set of constraints which performs relatively fast, while preserving the satisfiability of the target instances. A more ambitious target that can be developed for this research goal is the identification of predicates to use in the language bias to learn the constraints. Moreover, from the experiments with the PUP instances, we observed that the labelling instance impacted the symmetries identified by SBASS. Thus, we hope to introduce automatic

the basic encoding ENC1 and use it as **BASE**.

(re-)labeling schemes for constants appearing in instances to exploit common problem structure in a less input-specific way. For **RG 5** we aim to extend the applicability of our framework to programs containing weak constraints. Targeting optimization problems can lead to relevant results as optimization involves solving unsatisfiable subproblem(s) on attempting (and failing) to improve an optimal answer set, where symmetry breaking is particularly crucial for the performance.

## References


[1] M. Aschinger, C. Drescher, G. Friedrich, G. Gottlob, P. Jeavons, A. Ryabokon & E. Thorstensen (2011): *Optimization Methods for the Partner Units Problem*. In: *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, *Lecture Notes in Computer Science* 6697, Springer, pp. 4–19, doi:10.1007/978-3-642-21311-3_4.

[2] G. Brewka, T. Eiter & M. Truszczyński (2011): *Answer Set Programming at a Glance*. *Communications of the ACM* 54(12), pp. 92–103, doi:10.1145/2043174.2043195.

[3] P. Codenotti, H. Katebi, K. Sakallah & I. Markov (2013): *Conflict Analysis and Branching Heuristics in the Search for Graph Automorphisms*. In: *IEEE 25th International Conference on Tools with Artificial Intelligence*, IEEE Computer Society, pp. 907–914, doi:10.1109/ICTAI.2013.139.

[4] D. Cohen, P. Jeavons, C. Jefferson, K. Petrie & B. Smith (2006): *Symmetry Definitions for Constraint Satisfaction Problems*. *Constraints* 11(2-3), pp. 115–137, doi:10.1007/s10601-006-8059-8.

[5] J. Crawford & A. Baker (1994): *Experimental Results on the Application of Satisfiability Algorithms to Scheduling Problems*. In: *Proceedings of the twelfth national conference on Artificial Intelligence*, pp. 1092–1097.

[6] A. Cropper, S. Dumančić & S. Muggleton (2020): *Turning 30: New Ideas in Inductive Logic Programming*. In C. Bessiere, editor: *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence (IJCAI'20)*, ijcai.org, pp. 4833–4839, doi:10.24963/ijcai.2020/673.

[7] A. Cropper & S. Dumančić (2020): *Inductive Logic Programming at 30: A New Introduction*. https://arxiv.org/abs/2008.07912. arXiv:2008.07912.

[8] P. Darga, H. Katebi, M. Liffiton, I. Markov & K. Sakallah (2004): *Saucy*. http://vlsicad.eecs.umich.edu/BK/SAUCY/.

[9] J. Devriendt, B. Bogaerts, M. Bruynooghe & M. Denecker (2016): *On Local Domain Symmetry for Model Expansion*. *Theory and Practice of Logic Programming* 16(5-6), pp. 636–652, doi:10.1017/S1471068416000508.

[10] C. Dodaro, P. Gasteiger, N. Leone, B. Musitsch, F. Ricca & K. Schekotihin (2016): *Combining Answer Set Programming and Domain Heuristics for Solving Hard Industrial Problems*. *Theory and Practice of Logic Programming* 16(5-6), pp. 653–669, doi:10.1017/S1471068416000284.

[11] C. Drescher, O. Tifrea & T. Walsh (2011): *Symmetry-breaking Answer Set Solving*. *AI Communications* 24(2), pp. 177–194, doi:10.3233/AIC-2011-0495.

[12] E. Erdem, M. Gelfond & N. Leone (2016): *Applications of ASP*. *AI Magazine* 37(3), pp. 53–68, doi:10.1609/aimag.v37i3.2678.

[13] G. Friedrich, A. Ryabokon, A. Falkner, A. Haselböck, G. Schenner & H. Schreiner (2011): *(Re)configuration using Answer Set Programming*. In: *IJCAI 2011 Workshop on Configuration*, CEUR-WS.org, pp. 17–24.

[14] M. Gebser, R. Kaminski, B. Kaufmann & T. Schaub (2012): *Answer Set Solving in Practice*. Synthesis Lectures on Artificial Intelligence and Machine Learning, Morgan and Claypool Publishers, doi:10.2200/S00457ED1V01Y201211AIM019.

[15] M. Gelfond & V. Lifschitz (1988): *The Stable Model Semantics for Logic Programming*. In R. Kowalski & K. Bowen, editors: *Proceedings of the Fifth International Conference and Symposium of Logic Programming (ICLP'88)*, MIT Press, pp. 1070–1080.

[16] M. Gelfond & V. Lifschitz (1991): *Classical Negation in Logic Programs and Disjunctive Databases*. *New Generation Computing* 9, pp. 365–385, doi:10.1007/BF03037169.

[17] M. Law (2022): *Conflict-driven Inductive Logic Programming*. *Theory and Practice of Logic Programming*, pp. 1–28, doi:10.1017/S1471068422000011.

[18] M. Law, A. Russo & K. Broda (2014): *Inductive Learning of Answer Set Programs*. In E. Fermé & J. Leite, editors: *Proceedings of the Fourteenth European Conference on Logics in Artificial Intelligence (JELIA'14)*, *Lecture Notes in Artificial Intelligence* 8761, Springer-Verlag, pp. 311–325, doi:10.1007/978-3-319-11558-0_22.

[19] M. Law, A. Russo & K. Broda (2018): *Inductive Learning of Answer Set Programs from Noisy Examples*. *Advances in Cognitive Systems* 7, pp. 57–76.

[20] M. Law, A. Russo & K. Broda (2020): *The ILASP System for Inductive Learning of Answer Set Programs*. The Association for Logic Programming Newsletter. Available at https://www.cs.nmsu.edu/ALP/2020/04/the-ilasp-system-for-inductive-learning-of-answer-set-programs/.

[21] M. Law, A. Russo & K. Broda (2021): *ILASP*. www.ilasp.com.

[22] C. Mears, M. García de la Banda, M. Wallace & B. Demoen (2008): *A Novel Approach for Detecting Symmetries in CSP Models*. In L. Perron & M. Trick, editors: *Proceedings of the Fifth International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR'08)*, *Lecture Notes in Computer Science* 5015, Springer-Verlag, pp. 158–172, doi:10.1007/978-3-540-68155-7_14.

[23] J. Puget (2005): *Automatic Detection of Variable and Value Symmetries*. In P. van Beek, editor: *Proceedings of the Eleventh International Conference on Principles and Practice of Constraint Programming (CP'05)*, *Lecture Notes in Computer Science* 3709, Springer-Verlag, pp. 475–489, doi:10.1007/11564751_36.

[24] K. Sakallah (2009): *Symmetry and Satisfiability*. In A. Biere, M. Heule, H. van Maaren & T. Walsh, editors: *Handbook of Satisfiability*, chapter 10, *Frontiers in Artificial Intelligence and Applications* 185, IOS Press, pp. 289–338, doi:10.3233/978-1-58603-929-5-289.

[25] T. Syrjänen (2001): *Lparse 1.0 User's Manual*. Available at http://www.tcs.hut.fi/Software/smodels/lparse.ps.gz.

[26] A. Tarzariol, M. Gebser, M. Law & K. Schekotihin (2022): *Efficient lifting of symmetry breaking constraints for complex combinatorial problems*, doi:10.48550/ARXIV.2205.07129. Available at https://arxiv.org/abs/2205.07129.

[27] A. Tarzariol, M. Gebser & K. Schekotihin (2021): *Lifting Symmetry Breaking Constraints with Inductive Logic Programming*. In Z. Zhi-Hua, editor: *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence (IJCAI'21)*, ijcai.org, pp. 2062–2068, doi:10.24963/ijcai.2021/284.

[28] A. Tarzariol, M. Gebser & K. Schekotihin (2022): *Lifting Symmetry Breaking Constraints with Inductive Logic Programming*. *Machine Learning* 111(4), pp. 1303 – 1326, doi:10.1007/s10994-022-06146-3.

[29] Erich Christian Teppan, Gerhard Friedrich & Georg Gottlob (2016): *Tractability frontiers of the partner units configuration problem*. *J. Comput. Syst. Sci.* 82(5), pp. 739–755, doi:10.1016/j.jcss.2015.12.004.

[30] T. Walsh (2012): *Symmetry Breaking Constraints: Recent Results*. In J. Hoffmann & B. Selman, editors: *Proceedings of the Twenty-Sixth National Conference on Artificial Intelligence (AAAI'12)*, AAAI Press, pp. 2192–2198.

# Appendix A Inductive Learning from Symmetries - Example

Here we will illustrate an example of our ILP framework; for more details, see [28]. Let us consider 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. An encoding in ASP of this problem is:

```
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.
```

It takes as input the ground facts `pigeon(`$p$`).` and `hole(`$h$`).` For example, solving the instance with $p = 3$ and $h = 3$ leads to six answer sets:

$AS_1$ = {p2h(1,1), p2h(2,2), p2h(3,3)} = 100010001
$AS_2$ = {p2h(1,1), p2h(2,3), p2h(3,2)} = 010100001
$AS_3$ = {p2h(1,2), p2h(2,1), p2h(3,3)} = 100001010
$AS_4$ = {p2h(1,2), p2h(2,3), p2h(3,1)} = 001100010
$AS_5$ = {p2h(1,3), p2h(2,1), p2h(3,2)} = 010001100
$AS_6$ = {p2h(1,3), p2h(2,2), p2h(3,1)} = 001010100

where the binary integer given on the right corresponds to the value that will be considered for the lexicographic order. Using SBASS with this instance produces the following set of generators:

$\pi_1$ = (p2h(3,2) p2h(3,3)) (p2h(2,2) p2h(2,3)) (p2h(1,2) p2h(1,3))
$\pi_2$ = (p2h(3,1) p2h(3,3)) (p2h(2,1) p2h(2,3)) (p2h(1,1) p2h(1,3))
$\pi_3$ = (p2h(2,3) p2h(3,3)) (p2h(2,2) p2h(3,2)) (p2h(2,1) p2h(3,1))
$\pi_4$ = (p2h(1,1) p2h(3,3)) (p2h(2,1) p2h(2,3)) (p2h(1,3) p2h(3,1))
(p2h(1,2) p2h(3,2))

Applying a generator to an answer set returns a symmetric solution. For example, $\pi_1(AS_6) = AS_4$. For each answer set $AS_i$, we apply all the generators to it and check whether there is a generator $\pi_j$ such that $AS_i \geq \pi_j(AS_i)$. If there exists such $\pi_j$, then $AS_i$ will define a negative example, otherwise a positive one.

As a result, we create one positive example with $AS_6$ (since it is the only answer set that is not mapped into a smaller interpretation) and five negative examples with the other answer sets. The resulting ILP task is as follows:

```
%% Input encoding
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.

%% Active Background Knowledge
lessThan(X,Y) :- pigeon(X), pigeon(Y), X < Y.
lessThan(X,Y) :- hole(X), hole(Y), X < Y.
maxpigeon(X) :- pigeon(X), not pigeon(X+1).
maxhole(X) :- hole(X), not hole(X+1).

%% 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).}).

%% Positive example
#pos(id2, {p2h(3,1), p2h(2,2), p2h(1,3)}, {},
 {pigeon(3). hole(3).}).

%% Language bias
#modeb(2,p2h(var(pigeon),var(hole))).
#modeb(2,pigeon(var(pigeon))).
#modeb(2,hole(var(hole))).
#modeb(1,maxhole(var(hole))).
#modeb(1,maxpigeon(var(pigeon))).
#modeb(2,lessThan(var(hole),var(hole)),(anti_reflexive)).
#modeb(2,lessThan(var(pigeon),var(pigeon)),(anti_reflexive)).
#modeb(2,lessThan(var(hole),var(pigeon))).
#modeb(2,lessThan(var(pigeon),var(hole))).
```

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
```