Learning to Break Symmetries for Efficient Optimization in Answer Set Programming

Alice Tarzariol[1], **Martin Gebser**[1,2], **Konstantin Schekotihin**[1], **Mark Law**[3]

[1] University of Klagenfurt, Klagenfurt, Austria
[2] Graz University of Technology, Graz, Austria
[3] ILASP Limited, Grantham, UK
{alice.tarzariol, martin.gebser, konstantin.schekotihin}@aau.at, mark@ilasp.com

Abstract

The ability to efficiently solve hard combinatorial optimization problems is a key prerequisite to various applications of declarative programming paradigms. Symmetries in solution candidates pose a significant challenge to modern optimization algorithms since the enumeration of such candidates might substantially reduce their optimization performance. This paper proposes a novel approach using Inductive Logic Programming (ILP) to lift symmetry-breaking constraints for optimization problems modeled in Answer Set Programming (ASP). Given an ASP encoding with optimization statements and a set of small representative instances, our method augments ground ASP programs with auxiliary normal rules enabling the identification of symmetries using existing tools, like SBASS. Then, the obtained symmetries are lifted to first-order constraints with ILP. We prove the correctness of our method and evaluate it on real-world optimization problems from the domain of automated configuration. Our experiments show significant improvements of optimization performance due to the learned first-order constraints.

Introduction

Combinatorial optimization problems appear in various practical applications, including transportation, manufacturing, healthcare, or power generation and distribution. An efficient and elegant approach to tackle these problems is to model them in a declarative programming paradigm such as Answer Set Programming (ASP) (Lifschitz 2019). However, finding optimal solutions to the vast majority of real-world problems is challenging since their search spaces include many symmetric solution candidates, i.e., an isomorphic candidate can be obtained by permuting elements of a known solution. Therefore, to be effective, the problem encoding must include symmetry-breaking constraints (SBCs) that prune the search space of solution candidates by removing symmetric ones. In the recent decades, several techniques have been designed to improve performance by automatically identifying and discarding redundant solutions (Margot 2010; Sakallah 2009; Walsh 2012), based on embedding symmetry breaking in search algorithms or adding SBCs to a given problem encoding or instance, respectively.

In the context of ASP, Drescher, Tifrea, and Walsh (2011) suggested an instance-specific approach to symmetry breaking, called SBASS. This approach identifies symmetries of a ground ASP program by *(i)* representing the input program as a colored graph, *(ii)* finding symmetric vertex permutations in the graph using SAUCY (Codenotti et al. 2013; Darga et al. 2004), and *(iii)* constructing ground SBCs from the permutations. Recently, Tarzariol, Gebser, and Schekotihin (2021) suggested an approach that learns first-order constraints by lifting ground SBCs of SBASS using Inductive Logic Programming (ILP) (Cropper, Dumančić, and Muggleton 2020). Their method applies SBASS to a set of representative problem instances and then uses the obtained permutations to define an ILP task. The learning phase returns a set of first-order constraints that remove symmetric solutions (classified as negative examples) while preserving the representative solutions (positive examples).

The main shortcoming of existing methods (Tarzariol, Gebser, and Schekotihin 2022; Tarzariol et al. 2022), which learn constraints from SBCs of ASP programs, is their inability to deal with optimization statements. A solver uses the latter to assign a weight to every solution while searching for a solution with the minimal weight. Consequently, two solution candidates symmetric in the absence of optimization statements may give rise to different optimization values in their presence. If a non-optimal solution happens to be lexicographically smallest and is taken as a witness by previous methods to discard symmetric solutions, then learned constraints might preclude a solver from finding optimal solutions.

In this paper, we extend the learning approaches of Tarzariol et al. (2022); Tarzariol, Gebser, and Schekotihin (2022) to optimization problems and make the following contributions:

- We propose a method that augments a ground ASP program P with auxiliary rules, O_P, that do not change the solutions but tighten the applicable symmetries for optimization problems. We prove that SBASS on $P \cup O_P$ partitions solutions of P in a way that respects their quality determined by optimization statements. That is, each cell of the partition is guaranteed to consist of symmetric solutions sharing the same optimization value.
- We enhance the previous learning approaches to support the learning of effective first-order constraints for opti-

mization problems.

- Lastly, we test the proposed framework on combinatorial configuration problems incorporating optimization statements. Our results show that the learning performance is comparable between ILP tasks defined with or without taking optimization statements into account. Moreover, the learned first-order constraints prune the symmetries of optimization problems and thus speed up the computation of (representative) optimal solutions.

Background

Answer Set Programming

Answer Set Programming (ASP) is a declarative programming paradigm based on non-monotonic reasoning and the stable model semantics (Gelfond and Lifschitz 1991). We briefly present the syntax and semantics of ASP, and refer the reader to (Gebser et al. 2012; Lifschitz 2019) for details.

Syntax. An ASP program P is a finite set of *(normal) 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. A *literal* l is an atom a_i (positive) or its negation $not\, a_i$ (negative). The atom a_0 is the *head* of 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 $B^-(r) = \{a_{m+1}, \ldots, a_n\}$. The rule r is called a *fact* if $B^+(r) \cup B^-(r) = \emptyset$, and a *constraint* if $H(r) = \bot$. As syntactic sugar, we admit writing *choice rules* c of the form:

$$b\{a_1; \ldots; a_k\}u \leftarrow a_{k+1}, \ldots, a_m, not\, a_{m+1}, \ldots, not\, a_n.$$

where a_i, for $1 \leq i \leq n$, are atoms, and b and u are non-negative integers, which default to 0 or k, respectively. The set $H(c) = \{a_1, \ldots, a_k\}$ is the *head* of choice rule c. We can extend an ASP program with optimization statements or weak constraints. A *weak constraint* has the following form:

$$:\sim a_1, \ldots, a_m, not\, a_{m+1}, \ldots, not\, a_n.\ [w@p, \bar{t}]$$

where a_i, for $1 \leq i \leq n$, are atoms, w and p are integers representing the *weight* and *priority level* of the weak constraint, and $\bar{t}$ is a tuple of terms.

Semantics. The semantics of an ASP program P is given in terms of its *ground instantiation* P_{grd}, obtained by mapping each rule $r \in P$ to ground instances producible by substituting the variables in r with constants occurring in P, and likewise for choice rules and weak constraints. 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 $r \in P_{grd}$ (or choice rule $c \in P_{grd}$) if $B^+(r) \subseteq \mathcal{I}$ (or $B^+(c) \subseteq \mathcal{I}$) and $B^-(r) \cap \mathcal{I} = \emptyset$ (or $B^-(c) \cap \mathcal{I} = \emptyset$) imply $H(r) \in \mathcal{I}$ (or $b \leq |H(c) \cap \mathcal{I}| \leq u$). The interpretation $\mathcal{I}$ is a *model* of P if it satisfies all rules $r \in P_{grd}$ and all choice rules $c \in P_{grd}$, and $\mathcal{I}$ is *stable* if it is a subset-minimal model of the reduct $\{H(r) \leftarrow B^+(r) \mid r \in P_{grd}, B^-(r) \cap \mathcal{I} = \emptyset\} \cup \{a_0 \leftarrow B^+(c) \mid c \in P_{grd}, B^-(c) \cap \mathcal{I} = \emptyset, a_0 \in H(c) \cap \mathcal{I}\}$. We denote the set of all stable models, also called *answer sets*, of P by $AS(P)$. Weak constraints in P do not change $AS(P)$, but induce a preference among answer sets. If $\{a_1, \ldots, a_m\} \subseteq \mathcal{I}$ and $\{a_{m+1}, \ldots, a_n\} \cap \mathcal{I} = \emptyset$ for a (ground) weak constraint in P_{grd}, the vector $\bar{t}$ of terms is associated with the weight w at priority level p. Summing the weights w of term tuples $\bar{t}$ at each priority level p yields a vector of integers, where an *optimal* answer sets incurs the smallest sum of weights at the highest priority level, then at the second highest priority level in case of a tie, and so on.

Example 1. Let us consider the (ground) ASP program P_1:

```
1 {a;b;c} 1.
:~ a. [3@1]   :~ b. [2@1]   :~ c. [3@1]
```

The choice rule in the first line gives rise to three answer sets, namely $AS(P_1) = \{\{\texttt{a}\}, \{\texttt{b}\}, \{\texttt{c}\}\}$. Three weak constraints in the second line attribute the weight 3 to answer sets including `a` or `c`, and 2 to an answer set comprising `b`. Hence, the optimal answer set $\{\texttt{b}\}$ yields the smallest sum 2 of weights at the single priority level 1.

`Smodels` Format. In the `smodels` format (Syrjänen 2001) for ground ASP programs, a weak constraint is expressed in terms of a single atom a that, if contained in an interpretation, accounts for the weight w at priority level p.[1] Thus, we denote by $weak(P)$ the set of all triples $\langle w, p, a\rangle$ standing for weak constraints in the `smodels` format representation of a (ground) ASP program P. Without loss of generality, we assume that levels p are denoted by consecutive integers from 1 up to the number of distinct priority levels, and also that there is at most one weight w per atom a at the same level p, since multiple weights can be summed up.

Inductive Logic Programming

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 (Cropper, Dumančić, and Muggleton 2020). A *learning task* is given by a triple $\langle B, E, H_M\rangle$, where an ASP program B defines the *background knowledge*, the set E comprises two disjoint subsets E^+ and E^- of *positive* and *negative examples*, and the *hypothesis space* H_M is defined by a language bias M, which limits the potentially learnable rules. The most expressive ILP system for ASP is *Inductive Learning of Answer Set Programs* (ILASP), which can solve a variety of ILP tasks (Law, Russo, and Broda 2014, 2015). Each example $e \in E$ is a pair $\langle e_{pi}, C\rangle$ called *Context Dependent Partial Interpretation* (CDPI), where *(i)* e_{pi} is a *Partial Interpretation* (PI) 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 PI e_{pi}. A

[1] To be precise, weights w are always positive in the `smodels` format, while a (single) negative literal $not\, a$ may be associated with w. In the latter case, we here consider an equivalent representation in terms of the atom a and negative weight $-w$. While positive weights express penalties, negative ones can be viewed as rewards, and weak constraints with weight $w = 0$ may be dropped.

(total) interpretation $\mathcal{I}$ *extends* e_{pi} if $T \subseteq \mathcal{I}$ and $F \cap \mathcal{I} = \emptyset$. Given an ASP program P, an interpretation $\mathcal{I}$, and a CDPI $e = \langle e_{pi}, C \rangle$, 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}.

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 $B \cup H$. Law, Russo, and Broda (2018) extend the expressiveness of ILASP by allowing noisy examples. In this setting, if an example e is not covered (i.e., there is an accepting answer set for e if e is negative, or none if e is positive), the corresponding weight is taken as penalty. The learning task thus becomes an optimization problem with two objectives: minimize the size of H as well as the penalties for uncovered examples.

Inductive Learning from Symmetries

Tarzariol, Gebser, and Schekotihin (2021) presented an approach to lift ground SBCs for ASP programs using ILP. Their system takes four kinds of inputs: *(i)* an ASP program P modeling a combinatorial problem; *(ii)* two sets S and Gen of small yet representative satisfiable instances of the problem to be solved using P; *(iii)* a hypothesis space H_M; and *(iv)* an Active Background Knowledge ABK, i.e., an ASP program comprising auxiliary predicate definitions and constraints learned so far. For each instance g from the *generalization set* Gen, a general positive example $\langle \langle \emptyset, \emptyset \rangle, g \rangle$ with empty inclusions and exclusions requires learned constraints to preserve some answer set, in order to increase the likelihood that the learned constraints generalize beyond the training examples. Each instance i from the *training set* S is grounded together with P and then passed to SBASS for identifying its symmetries, denoted by $\Pi(i)$. Such symmetries are represented as a set of permutation group *generators* (also called *permutations*) characterizing groups of symmetric answer sets for the analyzed ground program. The framework by Tarzariol, Gebser, and Schekotihin (2021) uses this information to define the positive and negative examples for an ILP task solved by applying ILASP. The negative examples, corresponding to symmetric answer sets that can be mapped to a lexicographically smaller representative by means of the permutations in $\Pi(i)$, are associated with weights, so that as many, but not necessarily all of them are to be eliminated by learned constraints.

In subsequent work, Tarzariol, Gebser, and Schekotihin (2022) investigate four approaches to generate training examples for ground programs P_i, obtained by instantiating P with the instances $i \in S$. In particular, the *full-SBCs* approach exploits the CLINGO API (Gebser et al. 2019) to interleave the computation of candidate answer sets $\mathcal{I} \in AS(P_i)$ with the analysis of their symmetric solutions. Exploiting the properties of permutation groups (Sakallah 2009), *full-SBCs* determines all symmetric answer sets by repeatedly applying the permutations in $\Pi(i)$ to $\mathcal{I}$. This leads to a partition of the answer sets in $AS(P_i)$ whose cells gather symmetric solutions. For each obtained cell, *full-SBCs* labels the lexicographically smallest answer set as a positive example and all remaining ones as negative examples. Letting $atoms(\Pi(i))$ denote the set of atoms occurring in $\Pi(i)$, an example consists of $\mathcal{I} \cap atoms(\Pi(i))$ and $atoms(\Pi(i)) \setminus \mathcal{I}$ as inclusions or exclusions, respectively, and the context is the instance i.

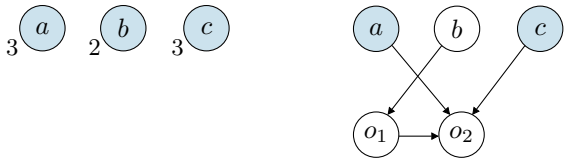
Example 2. Reconsider the program P_1 introduced in Example 1. When SBASS is run to analyze the choice rule in the first line of P_1, it determines permutations yielding that the answer sets in $AS(P_1) = \{\{\mathtt{a}\}, \{\mathtt{b}\}, \{\mathtt{c}\}\}$ are symmetric. Hence, starting from either $\{\mathtt{a}\}$, $\{\mathtt{b}\}$, or $\{\mathtt{c}\}$ as the first solution found, the *full-SBCs* approach produces the remaining two answer sets and includes all three in a single cell. Taking the lexicographically smallest answer set, $\{\mathtt{a}\}$, as the representative for the cell, $\langle \langle \{\mathtt{a}\}, \{\mathtt{b}, \mathtt{c}\} \rangle, P_1 \rangle$ is classified as positive, while $\langle \langle \{\mathtt{b}\}, \{\mathtt{a}, \mathtt{c}\} \rangle, P_1 \rangle$ and $\langle \langle \{\mathtt{c}\}, \{\mathtt{a}, \mathtt{b}\} \rangle, P_1 \rangle$ are negative examples. That is, ILASP's task is to learn some constraint(s), e.g., `:- not a.`, such that the optimal answer set $\{\mathtt{b}\}$ with the smallest sum 2 of weights gets eliminated.

Tarzariol, Gebser, and Schekotihin (2022) evaluate their methods on variants of the pigeon-hole problem and the house-configuration problem (Friedrich et al. 2011). In (Tarzariol et al. 2022), the authors extend their framework further to address complex application problems such as the Partner Units Problem (Aschinger et al. 2011; Teppan, Friedrich, and Gottlob 2016), where small yet representative training instances S are unavailable. To this end, the *scalable full-SBCs* approach is parametrized to sample (at most) n cells from the partition of $AS(P_i)$, thus limiting the examples to be included in an ILP task for ILASP to a manageable amount when all solutions are out of reach.

Symmetries for Optimization Problems

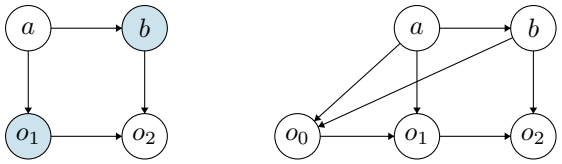
In view of lacking support for weak constraints by SBASS, we aim to introduce a set of auxiliary rules, O_P, characterizing the solution quality determined by weak constraints in a ground ASP program P. The new program part O_P is devised to be stratified (Przymusinski 1988) and define auxiliary atoms that do not occur in P, so that O_P is a conservative extension (Lifschitz and Turner 1994) for which $AS(P \cup O_P)$ and $AS(P)$ are in one-to-one correspondence.

In the following, we consider ground ASP programs P in `smodels` format and let j denote the number of distinct priority levels incorporated in $weak(P)$. Then, for $1 \leq p \leq j$, $\mathcal{W}_p = \{w \mid \langle w, p, a \rangle \in weak(P), w \neq 0\}$ is the set of (non-zero) weights in weak constraints at level p, and $\mathcal{A}_p^w = \{a \mid \langle w, p, a \rangle \in weak(P)\}$ is the set of atoms with the weight $w \in \mathcal{W}_p$ at p. The basic idea for the rules in O_P is to, for $1 \leq p \leq j$, $w \in \mathcal{W}_p$, and $i = |\{w' \in \mathcal{W}_p \mid w' \leq w\}| + \sum_{1 \leq p' < p} |\mathcal{W}_{p'}|$, define a fresh auxiliary atom o_i by (normal) rules $o_i \leftarrow a.$, for all $a \in \mathcal{A}_p^w$, along with $o_i \leftarrow o_{i-1}.$ if $i > 1$. This construction introduces auxiliary atoms o_i for $1 \leq i \leq k$, where $k = \sum_{1 \leq p \leq j} |\mathcal{W}_p|$, and connects program atoms $a \in \mathcal{A}_p^w$ as well as o_{i-1}, if $i > 1$, to o_i by directed edges in (a simplified version of) the *dependency graph* used by SBASS to identify symmetries. Since a symmetry π is a permutation of atoms such that the dependency graph contains an edge from $\pi(a)$ to $\pi(b)$ if and only if a has an edge to b, π preserves the structure of paths. Hence, any



(a) Dependency graph of P_1. (b) Dependency graph of $P_1 \cup O_{P_1}$.

Figure 1: Dependency graphs for programs in Example 3.



(a) Dependency graph of $P_2 \cup O_{P_2}$. (b) Dependency graph with o_0.

Figure 2: Dependency graphs illustrating the addition of o_0.

mapping such that $\pi(o_i) = o_{i'} \neq o_i$ is no symmetry, as the paths from o_i and $o_{i'} = \pi(o_i)$ to o_k are of different lengths.

Example 3. Reconsider the program P_1 introduced in Example 1. Figure 1(a) illustrates the dependency graph resulting from the choice rule in the first line of P_1, while weak constraints and the indicated weights of atoms are not taken into account by SBASS. Hence, each permutation of the atoms `a`, `b`, and `c` yields a symmetry, and the three symmetric atoms are highlighted in blue. By adding the rules

```
o2 :- a.  o2 :- c.  o2 :- o1.  o1 :- b.
```

as O_{P_1} to P_1, we obtain the dependency graph in Figure 1(b) for $P_1 \cup O_{P_1}$. Now only the atoms `a` and `c` with the same weight 3 are symmetric, as both are connected to the auxiliary atom `o2`. The atom `b` with the weight 2 is connected to `o1` instead, and the graph structure makes sure that symmetries of $P_1 \cup O_{P_1}$ map `b` as well as `o1` and `o2` to themselves.

While the rules in O_P are constructed such that auxiliary atoms o_i cannot be swapped with each other by symmetries, it can still happen that program atoms from weak constraints interact in a way making them symmetric to o_i for $P \cup O_P$.

Example 4. Consider the program and auxililiary rules $P_2 \cup O_{P_2}$ that yield the dependency graph in Figure 2(a):

P_2:
```
{a}.
b :- a.
:~ b. [2@1]
:~ a. [1@1]
```

O_{P_2}:
```
o2 :- b.
o2 :- o1.
o1 :- a.
```

In view of corresponding incoming and outgoing edges, the program atom `b` and the auxiliary atom `o1` can be swapped by a symmetry, thus mixing the original with introduced atoms. In order to avoid such phenomena, let us include an additional auxiliary atom `o0` along with the rules

```
o1 :- o0.  o0 :- a.  o0 :- b.
```

connecting `o0` to `o1` as well as all program atoms from weak constraints, regardless of their weights and priority levels, to `o0`. The resulting dependency graph in Figure 2(b) disambiguates the program atoms that are subject to weak constraints from auxiliary atoms o_i, since the former have non-trivial paths to each o_i, including o_0, while at least one such non-trivial path is missing when starting from any atom o_i.

The following definition generalizes the idea in Example 4 to construct O_P using an additional auxiliary atom o_0.

Definition 1. Let P be a ground ASP program in `smodels` format. The auxiliary rules O_P for P with fresh auxiliary atoms $o_0, \ldots, o_k$, where $k = \sum_{1 \leq p \leq j} |\mathcal{W}_p|$, are defined as:

$$O_P = \left\{ \begin{array}{l} o_0 \leftarrow a. \\ o_i \leftarrow a. \\ o_i \leftarrow o_{i-1}. \end{array} \;\middle|\; \begin{array}{l} 1 \leq p \leq j, w \in \mathcal{W}_p, a \in \mathcal{A}_p^w, \\ i = |\{w' \in \mathcal{W}_p \mid w' \leq w\}| \\ \quad + \sum_{1 \leq p' < p} |\mathcal{W}_{p'}| \end{array} \right\}$$

To characterize the properties of auxiliary rules, given a symmetry $\pi = (a_{1_1}\, a_{2_1}\, \ldots\, a_{n_1}) \ldots (a_{1_m}\, a_{2_m}\, \ldots\, a_{n_m})$ of P in cycle notation, let $C(\pi) = \{\{a_{1_1}, a_{2_1}, \ldots, a_{n_1}\}, \ldots, \{a_{1_m}, a_{2_m}, \ldots, a_{n_m}\}\}$ denote the sets of atoms belonging to (non-trivial) cycles in π. Then, the first property of interest is that the auxiliary atoms $o_0, \ldots, o_k$ can neither occur jointly nor share any cycle with program atoms from weak constraints in symmetries for $P \cup O_P$, where the introduction of o_0 is crucial for the latter condition to hold.

Proposition 1. Let P be a ground ASP program in `smodels` format and π be a symmetry of $P \cup O_P$. Then, for any $C \in C(\pi)$, we have that $C \cap \{o_0, \ldots, o_k\} = \emptyset$ or $|C \cap (\{a \mid 1 \leq p \leq j, w \in \mathcal{W}_p, a \in \mathcal{A}_p^w\} \cup \{o_0, \ldots, o_k\})| = 1$.

Proof. Assume that $C \cap \{o_0, \ldots, o_k\} \neq \emptyset$. By the construction of O_P, each $o_i \in \{o_0, \ldots, o_k\}$ has a single path of length $k-i$ to o_k in the dependency graph of $P \cup O_P$. Hence, any permutation π' such that $\pi'(o_i) = o_{i'} \neq o_i$ is not a symmetry of $P \cup O_P$, which implies that $C \cap \{o_0, \ldots, o_k\} = \{o_i\}$ for some $0 \leq i \leq k$. Moreover, any atom $b \in \{a \mid 1 \leq p \leq j, w \in \mathcal{W}_p, a \in \mathcal{A}_p^w\}$ has a path of length $k+1 > k-i$ to o_k in the dependency graph of $P \cup O_P$, so that any permutation π' such that $\pi'(b) = o_i$ is not a symmetry of $P \cup O_P$. Thus, we conclude that $C \cap (\{a \mid 1 \leq p \leq j, w \in \mathcal{W}_p, a \in \mathcal{A}_p^w\} \cup \{o_0, \ldots, o_k\}) = \{o_i\}$. □

As the auxiliary atoms $o_0, \ldots, o_k$ are on a path in the dependency graph of $P \cup O_P$, a symmetry π of $P \cup O_P$ contains either zero or k cycles $C \in C(\pi)$ with $C \cap \{o_0, \ldots, o_k\} \neq \emptyset$. In the latter case that there are k such cycles, π reproduces the structure of the conservative extension to P made by O_P. However, when, e.g., augmenting the program P_2 in Example 4 with new atoms and rules mimicking the structure of the dependency graph in Figure 2(b), a symmetry may map a and b to new atoms that are not subject to weak constraints. Hence, symmetries of $P \cup O_P$ such that $o_0, \ldots, o_k$ belong to (non-trivial) cycles should not be taken as symmetries of P.

As shown in the following, when symmetries of $P \cup O_P$ map the auxiliary atoms $o_0, \ldots, o_k$ to themselves, they are guaranteed to preserve the sum(s) of weights determined by $weak(P)$ for symmetric answer sets.

Theorem 1. Let P be a ground ASP program in `smodels` format and π be a symmetry of P such that $\{o_0, \ldots, o_k\} \cap \bigcup_{C \in C(\pi)} C = \emptyset$. Then, π is a symmetry of $P \cup O_P$ if and only if $C \cap \mathcal{A}_p^w = \emptyset$ or $C \cap \mathcal{A}_p^w = C$ for any $C \in C(\pi)$, priority level $1 \leq p \leq j$, and weight $w \in \mathcal{W}_p$.

Proof. ($\Rightarrow$) Assume that $\emptyset \subset C \cap \mathcal{A}_p^w \subset C$ for some $C \in C(\pi)$, priority level $1 \leq p \leq j$, and weight $w \in \mathcal{W}_p$. Then,

for $i = |\{w' \in \mathcal{W}_p \mid w' \leq w\}| + \sum_{1 \leq p' < p} |\mathcal{W}_{p'}|$, there is some atom $a \in C$ such that $o_i \leftarrow a.$ is contained in O_P, but not $o_i \leftarrow \pi(a)$. That is, the dependency graph of $P \cup O_P$ contains an edge from a to o_i, but $\pi(a)$ has no edge to $\pi(o_i) = o_i$, so that π is not a symmetry of $P \cup O_P$.

($\Leftarrow$) Assume that π is not a symmetry of $P \cup O_P$. Then, there is some $o_i \in \{o_1, \ldots, o_k\}$ and $C \in C(\pi)$ with an atom $a \in C$ such that $o_i \leftarrow a.$ is contained in O_P, but not $o_i \leftarrow \pi(a)$. That is, for $p = \min\{p'' \mid 1 \leq p'' \leq j, i \leq \sum_{1 \leq p' \leq p''} |\mathcal{W}_{p'}|\}$ and $w = \min\{w'' \in \mathcal{W}_p \mid i \leq |\{w' \in \mathcal{W}_p \mid w' \leq w''\}| + \sum_{1 \leq p' < p} |\mathcal{W}_{p'}|\}$, we have that $\emptyset \subset \{a\} \subseteq C \cap \mathcal{A}_p^w \subseteq C \setminus \{\pi(a)\} \subset C$. □

Corollary 1. Let P be a ground ASP program in `smodels` format and π be a symmetry of $P \cup O_P$ such that $\{o_0, \ldots, o_k\} \cap \bigcup_{C \in C(\pi)} C = \emptyset$. Then, for any interpretation $\mathcal{I}$ and priority level $1 \leq p \leq j$, we have that $\sum_{w \in \mathcal{W}_p, a \in \mathcal{A}_p^w \cap \mathcal{I}} w = \sum_{w \in \mathcal{W}_p, a' \in \mathcal{A}_p^w \cap \{\pi(a) \mid a \in \mathcal{I}\}} w$.

Corollary 1 is obtained from the fact that, given $\{o_0, \ldots, o_k\} \cap \bigcup_{C \in C(\pi)} C = \emptyset$, any symmetry π of $P \cup O_P$ is likewise a symmetry of P, while Theorem 1 establishes that π cannot permute any atoms with diverging levels or weights. Hence, symmetries identified on $P \cup O_P$ can be applied to the original program P without affecting the sum(s) of weights determined by weak constraints in P. In general, the rules in O_P separate atoms having syntactically different occurrences in optimization statements, where weak constraints constitute standard expressions (Calimeri et al. 2019) for which ASP systems provide built-in support. Since our symmetry breaking approach is syntactic, other proposed forms of optimization statements like, e.g., Answer Set Optimization (ASO) programs (Brewka, Niemelä, and Truszczyński 2003) and Logic Programs with Ordered Disjunction (LPODs) (Brewka, Niemelä, and Syrjänen 2004) could similarly be addressed using the O_P construction.

Learning Method for Optimization Problems

After introducing the auxiliary rules described in the previous section, we can apply the framework that performs inductive learning from ground symmetries. Given an ASP program P, an instance $i \in S$, and the set $\Pi(i)$ of its symmetries, below abbreviated as Π for brevity, the learning approach *full-SBCs* picks a positive example from each cell, obtained by permutations in Π, in the partition of $AS(P_i)$. However, this method is inappropriate for handling optimization problems as it also preserves suboptimal solutions. For this reason, we devise an extension of the approach suited for programs incorporating optimization statements.

Algorithm 1 shows the modified version of *full-SBCs* for addressing optimization problems. It takes as input P, i, Π, and opt, where the latter is the best optimization value obtained by running CLINGO on $P \cup i$ in a pre-solving step. When $i \in S$ is a small enough (representative) instance, a short pre-solving time, like five seconds taken in our experiments reported below, usually suffices to compute an optimal answer set of $P \cup i$ along with its opt value.

Algorithm 1: Method *full-SBCs* to compute examples for an instance $i \in S$ of the optimization problem modeled by P

Input: P, i, Π, opt

```
1:  cnt ← CLINGO.init(P ∪ i)
2:  while (I, c_I) ← cnt.get_new_solution() do
3:      Q ← ∅
4:      Q' ← {I}
5:      while Q' ≠ ∅ do
6:          Q ← Q ∪ Q'
7:          Q' ← {{π(a) | a ∈ I} | I ∈ Q', π ∈ Π} \ Q
8:      min ← {I ∈ Q | ∄I' ∈ Q : I' < I}
9:      create_examples(neg, Q \ min, Π, i)
10:     if c_I ≤ opt then
11:         create_examples(pos, min, Π, i)
12:     else
13:         create_examples(neg, min, Π, i)
14:     cnt.ignore_solution(Q)
```

In line 1 of Algorithm 1, a search control object cnt is created by using the CLINGO API. This object keeps track of already identified solutions and provides the get_new_solution method, which returns either a new answer set $\mathcal{I}$ together with its cost $c_\mathcal{I}$, or *false* if all solutions have been exhausted. Provided that the while-loop from line 2 is entered with a new answer set $\mathcal{I}$, the sets Q and Q' of already explored or newly obtained (symmetric) solutions, respectively, are set to $\emptyset$ and $\{\mathcal{I}\}$ in line 3 and line 4. The while-loop from line 5 then repeatedly applies the generators in Π until the cell of solutions symmetric to $\mathcal{I}$ is fully explored and the answer sets are collected in Q. In line 8, we determine the lexicographically smallest (according to some atom ordering) and thus representative solution in Q and store it in the (singleton) set min. The subsequent generation of examples is expressed in terms of the procedure create_examples(type, sol, Π, i), where the type is either pos(itive) or neg(ative) and sol is a set of interpretations. For each $\mathcal{I} \in sol$, the CDPI $\langle\langle \mathcal{I} \cap atoms(\Pi), atoms(\Pi) \setminus \mathcal{I}\rangle, i\rangle$ then constitutes a corresponding example of the selected type. Negative examples are in line 9 created for all symmetric, but not representative, solutions in the computed cell Q, and depending on whether the cost $c_\mathcal{I}$ is at least as good as the opt value, the representative solution in min is taken as a positive example in line 11 or as a negative example in line 13, respectively. Lastly, before querying cnt for the next solution, all answer sets belonging to the explored cell Q are excluded from the search space of cnt in line 14.

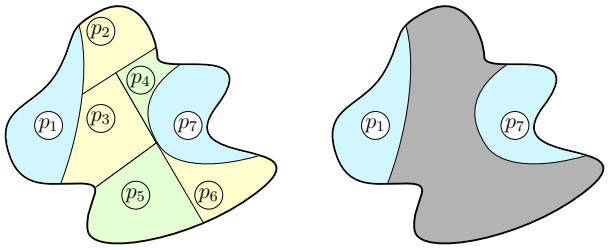
Figure 3(a) illustrates a space of examples obtained with the *full-SBCs* method, where the explored partition of $AS(P_i)$ consists of seven cells with their representative solutions indicated by points $p_1, \ldots, p_7$. The colors of cells represent different optimization values associated with the contained answer sets, and the two blue cells with p_1 and p_7 comprise optimal solutions. Hence, the *full-SBCs* method generates positive examples for p_1 and p_7, while all other solutions are taken as negative examples.

In case the number of answer sets in $AS(P_i)$ is large, as for the combinatorial problems investigated in (Tarzariol et al. 2022), the *full-SBCs* method fails to exhaustively

enumerate all solutions for training instances. For this reason, an alternative approach called *scalable full-SBCs* has been designed to sample at most n cells and generate at most max_cell negative examples as well as one positive example per cell. We adopt this approach to optimization problems by restricting the number of iterations of the while-loop from line 2 in Algorithm 1 to n and letting the procedure create_examples(type, sol, Π, i) generate $\min\{|sol|, max\_cell\}$ many examples in each execution.

When dealing with training instances whose answer sets are partitioned into plenty cells such that many of them comprise suboptimal solutions, it is important to still explore at least one cell of optimal answer sets. Otherwise, all generated examples were negative and the resulting ILP task has the trivial constraint $\bot \leftarrow .$ as solution eliminating all answer sets, including those that are optimal. For focusing on cells with optimal solutions only, the *opt-scalable full-SBCs* method restricts the search space of the cnt object to answer sets with the best optimization value opt. Figure 3(b) illustrates this approach, now just exploring the two blue instead of all cells as in Figure 3(a). While p_1 and p_7 yield positive examples as before, *opt-scalable full-SBCs* generates negative examples for symmetric solutions in their cells only, but does not explore suboptimal solutions in the remaining cells.

Example Orderings. The learning framework introduced by Tarzariol, Gebser, and Schekotihin (2021) utilizes general positive examples $\langle\langle\emptyset,\emptyset\rangle, g\rangle$ for instances g from the generalization set Gen, so that learned constraints are required to preserve some arbitrary answer set. However, when addressing optimization problems, this requirement must be strengthened to the preservation of at least one optimal answer set. To this end, the ILASP developers provided us with a new system version supporting orderings on the accepting answer sets of examples. For imposing upper bounds on the sum(s) of weights admitted for an accepting answer set, we use `#brave_ordering(p_id,[bounds],<=)` declarations supplied by the new version of ILASP, where `p_id` identfies a general positive example $\langle\langle\emptyset,\emptyset\rangle, g\rangle$ and `bounds` is the vector of optimization values by priority levels for an optimal answer set of $P \cup g$. Like the opt values for training instances S taken by Algorithm 1, the best optimization values `bounds` are computed in a (short) pre-solving step.



(a) *full-SBCs* (b) *opt-scalable full-SBCs*

Figure 3: Examples of *full-SBCs* and *opt-scalable full-SBCs*.

Experiments

We evaluate the devised learning framework for optimization problems on optimization versions of the combinatorial problems addressed in (Tarzariol, Gebser, and Schekotihin 2022; Tarzariol et al. 2022) as well as on the fastfood problem (Denecker et al. 2009). Our experiments were performed on an Intel® i7-3930K machine under Linux (Debian GNU/Linux 11), running ILASP (v4.3.1) as ILP and CLINGO (v5.5.2) as ASP system. The benchmarks and settings for reproducing our experiments can be found in https://github.com/prosysscience/Symmetry_Breaking_with_ILP/tree/optimization.

Fastfood Problem. The fastfood problem has been contributed as a benchmark to ASP system competitions. An instance specifies a collection of highway restaurants, where each fact `restaurant(id,km)` provides the identifier `id` of a restaurant and its location `km` in terms of kilometers along a highway. The goal is to select a subset of n restaurants serving as depots such that the sum of supply distances, i.e., the distance of each restaurant to its nearest depot, is minimized. We use the following ASP encoding:

```
N {depot(ID,KM) : restaurant(ID,KM)} N
    :- number_depots(N).
:~ restaurant(ID,KM), D = #min{|MK-KM| :
     depot(DI,MK)}. [D@1,ID]
```

An example instance can be specified by facts as follows:

```
restaurant(1,10).   restaurant(3,5).
restaurant(2,10).   number_depots(2).
```

This instance has three solutions, {`depot(1,10)`, `depot(2,10)`}, {`depot(1,10)`, `depot(3,5)`}, and {`depot(2,10)`, `depot(3,5)`}, the last two of which are optimal. However, without considering the weak constraint on supply distances, all three solutions are deemed symmetric, so that {`depot(1,10)`, `depot(2,10)`} may be taken as representative and thus a positive example.

Following (Tarzariol, Gebser, and Schekotihin 2022), we introduce an auxiliary predicate `less(ID,ID')` in the background knowledge, which holds for each pair of restaurants with identifiers `ID` $<$ `ID'` located at the same kilometer `KM`. Then, running ILASP with a language bias containing the predicate `depot` and the auxiliary predicate `less` leads to two learned constraints as follows:

```
:- depot(Y,Z), less(X,Y), not depot(X,Z).
:- depot(X,Z), less(X,Y), not depot(Y,Z).
```

These constraints eliminate the two optimal solutions for the above example instance. Unlike this, the extended learning framework introduced in the previous sections determines the optimal solution {`depot(1,10)`, `depot(3,5)`} as positive example, and the learned constraint

```
:- depot(Y,Z), less(X,Y), not depot(X,Z).
```

correctly discards its single symmetric (and also optimal) solution {`depot(2,10)`, `depot(3,5)`}.

Combinatorial Problems. The combinatorial problems investigated by Tarzariol, Gebser, and Schekotihin (2022) consist of the pigeon-hole problem, its extensions with color and owner assignments, and the house-configuration problem. We derived optimization versions of these four problems by incorporating weak constraints that distinguish as-

Problem	BASELINE $\#o/\#s$	min	max	avg	$stdev$	BASELINE + CONSTRAINTS $\#o/\#s$	min	max	avg	$stdev$	
pigeon-hole	0/10	–	–	–	–	10/10	0.02	15.17	1.92	4.68	
pigeon-color	1/ 4	248.75	248.75	248.75	–	10/10	0.08	258.58	52.15	83.47	
pigeon-owner	2/ 5	1.41	66.26	33.83	45.86	10/10	0.02	180.66	34.24	58.91	
house-config.	0/10	–	–	–	–	10/10	0.04	155.37	30.13	49.65	BB
fastfood	1/10	277.25	277.25	277.25	–	10/10	0.07	12.12	2.81	4.27	
pup-double	0/10	–	–	–	–	10/10	0.03	192.49	37.17	67.72	
pup-doublev	0/10	–	–	–	–	10/10	0.04	70.58	15.47	23.33	
pup-triple	1/10	19.54	19.54	19.54	–	2/10	0.05	0.83	0.44	0.56	
pigeon-hole	0/ 0	–	–	–	–	1/ 1	20.95	20.95	20.95	–	
pigeon-color	0/ 5	–	–	–	–	10/10	0.07	67.53	20.63	26.96	
pigeon-owner	2/ 7	0.18	1.66	0.92	1.05	10/10	0.01	82.43	17.15	27.70	
house-config.	0/ 0	–	–	–	–	10/10	0.03	265.14	63.80	87.05	USC
fastfood	0/ 0	–	–	–	–	2/ 2	0.06	0.17	0.11	0.07	
pup-double	0/ 0	–	–	–	–	8/ 8	0.05	276.60	37.29	96.82	
pup-doublev	0/ 0	–	–	–	–	10/10	0.06	189.52	29.19	58.85	
pup-triple	1/ 1	25.52	25.52	25.52	–	2/ 2	0.04	2.76	1.40	1.92	

Table 1: Runtime results for CLINGO on: *(i)* baseline encodings; and *(ii)* baseline encodings augmented with learned constraints. $\#o$ and $\#s$ indicate the number of instances finished with a proven or unproven optimal solution, respectively. min, max, avg, and $stdev$ provide the minimal, maximal, and average runtime in seconds along with the standard deviation over instances counted in $\#o$ (finished with a proven optimal solution). The upper part of the table provides results for CLINGO with model-guided optimization (i.e., *branch and bound*), while core-guided optimization (i.e., *unsatisfiable core*) is run in the lower part.

signed values, i.e., holes, colors, owners, or racks, and give preference to smaller values. For the Partner Units Problem (PUP) addressed in (Tarzariol et al. 2022), we included weak constraints minimizing the number of assigned units, thus switching to the PUP optimization version by Aschinger et al. (2011), who also introduced the instance families denoted by *double*, *doublev*, and *triple*.

Learning Performance. We first compared the runtime required to learn first-order constraints between decision and optimization versions of the considered benchmark problems. This comparison includes the *full-SBCs* and *scalable full-SBCs* methods: for decision problems, we applied them unchanged from (Tarzariol, Gebser, and Schekotihin 2022; Tarzariol et al. 2022), while the modified versions introduced above were run on optimization problems. Since already the small training instances of PUP yield too many solutions to enumerate them all, we resort to *scalable full-SBCs* on PUP, and use the exhaustive *full-SBCs* method for the other problems. As it turns out, the learning time required for decision and optimization versions of the three pigeon-hole problem variants, the house-configuration problem, and the fastfood problem never exceeded five seconds. In the following, we report more detailed results for the way more complex PUP.

As training sets S for the *double*, *doublev*, and *triple* instance families of PUP, we used two small yet representative instances per family. One of the training instances admits a minimal number of units only, so that each of its solutions is optimal, while the decision and optimization versions of *scalable full-SBCs* differ on the second instance offering one more unit than needed for satisfiability. Given that *scalable full-SBCs* samples a subset of solutions and the obtained examples depend on the search heuristics of CLINGO, we repeat each learning run 120 times with different seeds and a time limit of one hour per run. The unchanged decision version of *scalable full-SBCs* successfully finished runs with 104 seeds on the training set for *double*, 82 runs for *doublev*, but not a single run for *triple*. Somewhat unexpectedly, such a decline does not arise with *scalable full-SBCs* modified to optimization problems, where 116, 84, and 108 learning runs are successfully finished for *double*, *doublev*, and *triple*. This can be explained by the addition of auxiliary rules distinguishing atoms from weak constraints, as they reduce the symmetries identified by SBASS and thus also the size of cells that must be explored to determine positive examples.

To also compare the *scalable full-SBCs* and *opt-scalable full-SBCs* methods, where the latter explores cells with optimal solutions only, we turned the training sets for *double*, *doublev*, and *triple* into singletons, comprising just the instance that admits one more unit than needed. Moreover, we used weak constraints taking the integer label instead of the uniform weight 1 as penalty for an assigned unit, which leads to fewer symmetries and a finer partition of answer sets than before. This helps *scalable full-SBCs* and *opt-scalable full-SBCs* to finish all of their 120 learning runs within the time limit, where *opt-scalable full-SBCs* always yields optimal solutions as positive examples. With *scalable full-SBCs*, just 5 for *double*, 42 for *doublev*, and 17 runs for *triple* led to some positive example needed for a non-trivial ILP task.

Solving Performance. Second, we compare the optimization performance of CLINGO on the baseline problem encodings to the same encodings augmented with learned first-order constraints, using a benchmark set of 10 (more challenging than the training) instances per problem with a time limit of 300 seconds for each run. The $\#o/\#s$ columns in Table 1 provide the number of runs that finished with a proven optimal solution in the time limit or led to some optimal solution without necessarily proving its optimality, re-

Problem	PUP-ADVANCED #o/#s	min	max	avg	stdev	BASELINE + CONSTRAINTS #o/#s	min	max	avg	stdev	
pup-double	3/10	0.22	167.00	56.17	95.98	10/10	0.03	192.49	37.17	67.72	
pup-doublev	4/10	0.38	70.01	21.49	33.00	10/10	0.04	70.58	15.47	23.33	BB
pup-triple	2/10	0.04	2.12	1.08	1.47	2/10	0.05	0.83	0.44	0.56	
pup-double	3/ 3	0.17	239.75	80.23	138.15	8/ 8	0.05	276.60	37.29	96.82	
pup-doublev	3/ 3	0.98	61.77	33.00	30.53	10/10	0.06	189.52	29.19	58.85	USC
pup-triple	2/ 2	0.03	1.67	0.85	1.15	2/ 2	0.04	2.76	1.40	1.92	

Table 2: Runtime results for CLINGO on: *(i)* an advanced PUP encoding (Dodaro et al. 2016) incorporating hand-crafted static symmetry breaking; and *(ii)* the baseline encoding augmented with learned constraints. The parameters reported are the same as in Table 1.

spectively. Moreover, min, max, and avg runtimes along with $stdev$ over instances finished with a proven optimal solution give an impression of the search efforts of CLINGO.

The upper part of Table 1 contains the results obtained by CLINGO with model-guided optimization (i.e., *branch and bound*), while the lower part reflects CLINGO with core-guided optimization (i.e., *unsatisfiable core*). For the combinatorial problems in the first five rows, the baseline encodings prohibit CLINGO to prove optimal solutions for most of the instances, although some optimal solution is found for many of them when using model-guided optimization. By adding the learned constraints, CLINGO succeeds to find and prove optimal solutions for all instances, where the required search efforts vary between the problems. We further notice that, on the pigeon-hole and fastfood problems, core-guided optimization struggles to find optimal solutions, while model-guided optimization applied to the augmented encoding does not exhibit any such issues.

The rows starting with **pup-** in Table 1 and Table 2 summarize results obtained with the most efficient first-order constraints from some of the 120 learning runs per instance family *double*, *doublev*, and *triple* of PUP. While the baseline encoding with model-guided optimization leads to optimal solutions for all instances, CLINGO manages to prove the optimality for a single *triple* instance only. A more advanced PUP encoding (Dodaro et al. 2016) that, independent of the particular instance family, incorporates static symmetry breaking on the integer labels of units brings about a noticeable increase of runs finished with a proven optimal solution. However, as specifically learned constraints also take the structure of instances into account, they prune the search space even better and aid CLINGO to prove optimality for all instances of the *double* and *doublev* families. In contrast, the instances of the *triple* family remain hard, and the limited success of proving optimality for just two of them reminds that symmetries are one, but not the only complexity factor.

Conclusion

This paper extends the framework of Tarzariol, Gebser, and Schekotihin (2021) to support the learning of effective first-order constraints for optimization problems. The introduced auxiliary rules are crucial for SBASS to identify symmetries applicable to ASP programs with weak constraints. We have proven the correctness of this approach and devised suitable methods to generate ILP tasks for optimization problems. Experiments with the new framework show that the learned constraints can help to significantly speed up the computation of (representative) optimal solutions. In future work, we plan to investigate alternative instance-specific approaches for the construction of ground SBCs and their lifting to first-order constraints. For instance, we will evaluate BREAKID system (Devriendt and Bogaerts 2016), which extends the graph representation of a problem instance used by SBASS to handle weight rules and minimize statements. This is an alternative approach to the one suggested in this paper that might detect more symmetries in comparison to SBASS. By using BREAKID we hope that our approach will get an additional performance boost while solving instances of optimization problems.

Acknowledgements

This work was 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.

References

Aschinger, M.; Drescher, C.; Friedrich, G.; Gottlob, G.; Jeavons, P.; Ryabokon, A.; and Thorstensen, E. 2011. Optimization Methods for the Partner Units Problem. In *International Conference on Integration of Artificial Intelligence and Operations Research Techniques in Constraint Programming*, volume 6697 of *Lecture Notes in Computer Science*, 4–19. Springer.

Brewka, G.; Niemelä, I.; and Syrjänen, T. 2004. Logic Programs with Ordered Disjunction. *Computational Intelligence*, 20(2): 335–357.

Brewka, G.; Niemelä, I.; and Truszczyński, M. 2003. Answer Set Optimization. In Gottlob, G.; and Walsh, T., eds., *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence (IJCAI'03)*, 867–872. Morgan Kaufmann Publishers.

Calimeri, F.; Faber, W.; Gebser, M.; Ianni, G.; Kaminski, R.; Krennwallner, T.; Leone, N.; Maratea, M.; Ricca, F.; and Schaub, T. 2019. ASP-Core-2 Input Language Format. *Theory and Practice of Logic Programming*, 20(2): 294–309.

Codenotti, P.; Katebi, H.; Sakallah, K.; and Markov, I. 2013. Conflict Analysis and Branching Heuristics in the Search for Graph Automorphisms. In *IEEE 25th International Con-*

ference on Tools with Artificial Intelligence, 907–914. IEEE Computer Society.

Cropper, A.; Dumančić, S.; and Muggleton, S. 2020. Turning 30: New ideas in Inductive Logic Programming. In *International Joint Conference on Artificial Intelligence*, 4833–4839. ijcai.org.

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

Denecker, M.; Vennekens, J.; Bond, S.; Gebser, M.; and Truszczyński, M. 2009. The Second Answer Set Programming Competition. In Erdem, E.; Lin, F.; and Schaub, T., eds., *Proceedings of the Tenth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'09)*, volume 5753 of *Lecture Notes in Artificial Intelligence*, 637–654. Springer.

Devriendt, J.; and Bogaerts, B. 2016. BreakID: Static symmetry breaking for ASP (system description). *arXiv preprint arXiv:1608.08447*.

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

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

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

Gebser, M.; Kaminski, R.; Kaufmann, B.; and Schaub, T. 2012. *Answer Set Solving in Practice*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan and Claypool Publishers.

Gebser, M.; Kaminski, R.; Kaufmann, B.; and Schaub, T. 2019. Multi-shot ASP solving with clingo. *Theory and Practice of Logic Programming*, 19(1): 27–82.

Gelfond, M.; and Lifschitz, V. 1991. Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Computing*, 9: 365–385.

Law, M.; Russo, A.; and Broda, K. 2014. Inductive Learning of Answer Set Programs. In *European Conference on Logics in Artificial Intelligence*, volume 8761, 311–325.

Law, M.; Russo, A.; and Broda, K. 2015. The ILASP system for learning Answer Set Programs. www.ilasp.com. Accessed: 2023-03-03.

Law, M.; Russo, A.; and Broda, K. 2018. Inductive Learning of Answer Set Programs from Noisy Examples. *Advances in Cognitive Systems*, 7: 57–76.

Lifschitz, V. 2019. *Answer Set Programming*. Springer.

Lifschitz, V.; and Turner, H. 1994. Splitting a logic program. In *Proceedings of the Eleventh International Conference on Logic Programming*, 23–37. MIT Press.

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

Przymusinski, T. 1988. On the Declarative Semantics of Deductive Databases and Logic Programs. In Minker, J., ed., *Foundations of Deductive Databases and Logic Programming*, 193–216. San Mateo, CA: Morgan Kaufmann Publishers.

Sakallah, K. 2009. Symmetry and Satisfiability. In Biere, A.; Heule, M.; van Maaren, H.; and Walsh, T., eds., *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, chapter 10, 289–338. IOS Press.

Syrjänen, T. 2001. Lparse 1.0 User's Manual. http://www.tcs.hut.fi/Software/smodels/lparse.ps.gz. Accessed: 2023-03-03.

Tarzariol, A.; Gebser, M.; and Schekotihin, K. 2021. Lifting Symmetry Breaking Constraints with Inductive Logic Programming. In *International Joint Conference on Artificial Intelligence*, 2062–2068. ijcai.org.

Tarzariol, A.; Gebser, M.; and Schekotihin, K. 2022. Lifting Symmetry Breaking Constraints with Inductive Logic Programming. *Machine Learning*.

Tarzariol, A.; Schekotihin, K.; Gebser, M.; and Law, M. 2022. Efficient Lifting of Symmetry Breaking Constraints for Complex Combinatorial Problems. *Theory and Practice of Logic Programming*.

Teppan, E. C.; Friedrich, G.; and Gottlob, G. 2016. Tractability frontiers of the partner units configuration problem. *Journal of Computer and System Sciences*, 82(5): 739–755.

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