

Symmetry-breaking Answer Set Solving

Christian Drescher¹, Oana Tifrea², and Toby Walsh³

¹ Vienna University of Technology, Austria

² Free University of Bozen-Bolzano, Italy

³ NICTA and University of New South Wales, Australia

Abstract. In the context of Answer Set Programming, this paper investigates symmetry-breaking to eliminate symmetric parts of the search space and, thereby, simplify the solution process. We propose a reduction of disjunctive logic programs to a coloured digraph such that permutational symmetries can be constructed from graph automorphisms. Symmetries are then broken by introducing symmetry-breaking constraints. For this purpose, we formulate a preprocessor that integrates a graph automorphism system. Experiments demonstrate its computational impact.

1 Introduction

Answer Set Programming (ASP; [4]) has been shown to be a useful approach for knowledge representation and non-monotonic reasoning in various applications that include difficult combinatorial search, among them graph theoretic problems, planning, model checking, and problems from bioinformatics. ASP combines an expressive but simple modelling language, able to encode all search problems within the first three levels of the polynomial hierarchy, with high-performance solving capacities [9]. In fact, ASP solvers have experienced dramatic improvements in their performance [12] and compete⁴ with the best Boolean Satisfiability (SAT; [5]) solvers.

However, many combinatorial search problems exhibit symmetries which can frustrate a search algorithm to fruitlessly explore independent symmetric subspaces. Various instance families, such as the *Pigeon Hole* problem, are known to require exponential time for resolution and backtracking algorithms [23]. Indeed, state-of-the-art ASP solvers take a very long time to solve those instances (Section 7). Once their symmetries are identified, it is possible to avoid redundant computational effort by pruning parts of the search space through symmetry-breaking.

Symmetry-breaking also addresses post-processing: Where symmetries induce equivalence classes in the solution space, symmetric solutions can be discarded. Problems like the *All-interval Series* taken from the CSPLib [14] have plenty symmetric solution. However, all solutions to the original problem can be reconstructed from the answer sets under symmetry-breaking.

⁴ <http://www.satcompetition.org/>

In this paper we break the problem of symmetry-breaking down into two parts: (1) identifying symmetries and (2) breaking the identified symmetries. We adopt existing theoretical foundations on introducing symmetry-breaking constraints (SBC) for SAT instances in conjunctive normal form (CNF) that exhibit symmetries [6,1,2,20]. As to SAT, the basic idea is to detect irredundant generators of the group of permutational symmetries using a reduction to coloured graph automorphism. For each such generator, an SBC is constructed and added to the original CNF formula.

The key contribution of our work is a reduction of symmetry detection for disjunctive logic programs to the automorphisms of a coloured digraph, and an ASP representation of SBC which is linear in the number of problem variables. Furthermore, we formulate Symmetry-breaking Answer Set Solving as preprocessing and demonstrate its computational impact on difficult combinatorial search problems.

The remainder of this paper is organized as follows. We start by giving the necessary background notions of ASP and group theory. In turn, Sections 4 and 5 cover symmetry detection and symmetry-breaking for ASP, respectively. In Section 6 we motivate partial symmetry-breaking, in Section 7 we empirically evaluate our approach. Section 8 draws conclusions.

2 Logical Background

A (*disjunctive*) *logic program* over a set of primitive propositions \mathcal{A} is a finite set of *rules* r of the form

$$a_1; \dots; a_l \leftarrow b_1, \dots, b_m, \sim c_1, \dots, \sim c_n \quad (1)$$

where and $a_i, b_j, c_k \in \mathcal{A}$ are *atoms* for $1 \leq i \leq l$, $1 \leq j \leq m$, and $1 \leq k \leq n$. A *literal* is an atom a or its default negation $\sim a$. Let $head(r) = \{a_1, \dots, a_l\}$ be the *head* of r and $body(r) = \{b_1, \dots, b_m, \sim c_1, \dots, \sim c_n\}$ the *body* of r . For a set of literals S , define $S^+ = \{a \mid a \in S\}$ and $S^- = \{a \mid \sim a \in S\}$. The set of atoms occurring in a logic program P is denoted by $atom(P)$, and the set of bodies in P is $body(P) = \{body(r) \mid r \in P\}$. For regrouping bodies sharing the same head atom a , define $body(a) = \{body(r) \mid r \in P, a \in head(r)\}$.

The semantics of a logic program is given by its answer sets. A set $M \subseteq \mathcal{A}$ is an *answer set* of a logic program P over \mathcal{A} , if M is a \subseteq -minimal model of the *reduct* [13]

$$P^M = \{head(r) \leftarrow body(r)^+ \mid r \in P, body(r)^- \cap M = \emptyset\}.$$

A rule of form (1) can be seen as a constraint on the answer sets of a program, stating that if b_{l+1}, \dots, b_m are in the answer set and none of c_{m+1}, \dots, c_n are included, then one of a_1, \dots, a_l must be in the set. Clearly, an answer set induces a truth assignment on the atoms in P .

The semantics of important extensions to logic programs, such as integrity constraints, is given through program transformations that introduce additional

propositions [21]. An *integrity constraint* of the form

$$\leftarrow b_1, \dots, b_m, \sim c_1, \dots, \sim c_n \quad (2)$$

is a short hand for a rule with an unsatisfiable head, and thus forbids its body to be satisfied in any answer set.

Example 1. Consider the logic programs P_1 and P_2 , both have two answer sets $\{a\}$ and $\{b\}$, where

$$P_1 = \left\{ \begin{array}{l} a \leftarrow \sim b \\ b \leftarrow \sim a \end{array} \right\}, \quad P_2 = \left\{ \begin{array}{l} a, b \leftarrow \\ \leftarrow a, b \end{array} \right\}.$$

Observe, that P_1 and P_2 remain invariant under a swap of atoms a and b which is what we call a symmetry. In this work we will only deal with symmetries that can be thought of as permutations of atoms.

Example 2. The *All-interval Series problem* is to find a permutation of the n integers from 0 to $n - 1$ such that the difference of adjacent numbers are also all-different. We encode the All-interval Series problem introducing propositional variables $v_{i,j}$ and $d_{k,l}$ for the $i \in 1..n$ integer variables taking values $j \in 0..(n-1)$, and $k \in 1..(n-1)$ auxiliary variables taking values $l \in 1..(n-1)$ to represent the differences between adjacent numbers, respectively. Furthermore, we require both sets of variables to have pairwise different values (all-different constraint).

$$\begin{array}{ll} v_{i,0}; v_{i,1}; \dots; v_{i,n-1} \leftarrow & i \in 1..n \\ \leftarrow v_{i,j}, v_{i,k} & j \neq k \\ d_{i,|j-k|} \leftarrow v_{i,j}, v_{i+1,k} & i \in 1..(n-1) \wedge j, k \in 0..(n-1) \\ \leftarrow d_{i,j}, d_{i,k} & j \neq k \end{array}$$

Note that above encoding for the all-different constraint corresponds to its *support encoding* [10]. It remains invariant under complex permutation of atoms (see Example 5).

3 Group Theoretic Background

Intuitively, a symmetry of a discrete object is a transformation of its components that leaves the object unchanged. By a symmetry of an answer set program we mean a permutation of its atoms that does not change the logic program, in particular, maps rules to rules. In principle, such a permutation can affect arbitrarily many atoms at once, for instance, as in the case of a complete cyclic shift.

Symmetries are studied in terms of groups. A *group* is an abstract algebraic structure $(G, *)$, where G is a set closed under a binary associative operation $*$ such that there is a *unit* element and every element has a unique *inverse*. Often, we abuse notation and refer to the group G , rather than to the structure $(G, *)$. A subset H of G is referred to as a *subgroup* of G if H is closed under the binary

operation of G . A set of group elements $H \subset G$ such that any other group element in G can be expressed in terms of their product is called a *generating set* and the elements of H are called *generators* of G . A generator is *redundant* if it can be expressed in terms of other generators. An *irredundant* generating set does not contain redundant generators, and provides an extremely compact representation of the group. In fact, representing groups by sets of generators always ensures exponential compression.

Theorem 3 (Exponential Compression [20]). *Any irredundant generating set for a finite group G , such that $|G| > 1$, contains at most $\log_2 |G|$ elements.*

A *permutation* of a set A is a bijection $\pi : A \rightarrow A$. Indeed, the set of permutations form a group under composition, denoted as $S(A)$. It is easy to see that the composition of two permutations is a permutation, that the composition of permutations is associative, that the composition with the *identity* never changes a permutation, and that every permutation has a unique inverse.

The image of $a \in A$ under a permutation π is denoted as a^π , and for $S \subseteq A$ define $S^\pi = \{a^\pi \mid a \in S\}$. The *orbits* of S under π are the set of elements of A to which S can be mapped by (repeatedly) applying π . Orbits under a permutation define an equivalence relation on A . Analogously, for vectors $v = (v_1, v_2, \dots, v_k) \in A^k$ define $v^\pi = (v_1^\pi, v_2^\pi, \dots, v_k^\pi)$, and sets of sets $S = \{S_1, S_2, \dots, S_k\}$ such that $S_i \subseteq A$ for $1 \leq i \leq k$ define $S^\pi = \{S_1^\pi, S_2^\pi, \dots, S_k^\pi\}$.

For a logic program P , we define the *symmetric group* of P , $S(\text{atom}(P))$, to be the group of all permutations of the atoms that occur in P . We will make use of the *cycle notation* where a permutation is a product of disjoint cycles. A cycle $(1 \ 2 \ 3 \ \dots \ n)$ means that the permutation maps 1 to 2, 2 to 3, and so on, finally n back to 1. An element that does not appear in any cycle is understood as being mapped to itself. Table 2 provides some examples. Finally, we define the *support* [18] of a permutation as those elements that are not mapped to themselves.

As to Symmetry-breaking Answer Set Solving, given a logic program P , we are interested in the subgroup of the symmetric group of P which elements leave P unchanged. Obviously, a symmetry of a logic program preserves answer sets.

Definition 4 (Symmetry of a Logic Program). *A symmetry of a logic program P is a permutation of its atoms that does not change P .*

Example 5. There are four symmetries in the All-interval Series problem: (1) the identity, (2) reversing the series (variable symmetry), (3) reflecting the series by subtracting each element from $n - 1$ (value symmetry), and (4) doing both. It is easy to see that (2) and (3) form a group of generators. Indeed, we can find

both symmetries in our encoding (see Example 2) given in cycle notation below.

$$\begin{aligned}
\pi_2 = & (v_{1,0} \ v_{n,0}) (v_{1,1} \ v_{n,1}) \ \dots \ (v_{1,n-1} \ v_{n,n-1}) \\
& \dots \\
& (v_{\lfloor n/2 \rfloor, 0} \ v_{\lceil n/2 \rceil, 0}) (v_{\lfloor n/2 \rfloor, 1} \ v_{\lceil n/2 \rceil, 1}) \ \dots \ (v_{\lfloor n/2 \rfloor, n-1} \ v_{\lceil n/2 \rceil, n-1}) \\
& (d_{1,1} \ d_{n-1,1}) (d_{1,2} \ d_{n-1,2}) \ \dots \ (d_{1,n-1} \ d_{n-1,n-1}) \\
& \dots \\
& (d_{\lfloor (n-1)/2 \rfloor, 1} \ d_{\lceil (n-1)/2 \rceil, 1}) (d_{\lfloor (n-1)/2 \rfloor, 2} \ d_{\lceil (n-1)/2 \rceil, 2}) \\
& \dots \ (d_{\lfloor (n-1)/2 \rfloor, n-1} \ d_{\lceil (n-1)/2 \rceil, n-1}) \\
\\
\pi_3 = & (v_{1,0} \ v_{1,n-1}) (v_{1,1} \ v_{1,n-2}) \ \dots \ (v_{n, \lfloor (n-1)/2 \rfloor} \ v_{n, \lceil (n-1)/2 \rceil}) \\
& \dots \\
& (v_{n,0} \ v_{n,n-1}) (v_{n,1} \ v_{n,n-2}) \ \dots \ (v_{n, \lfloor (n-1)/2 \rfloor} \ v_{n, \lceil (n-1)/2 \rceil})
\end{aligned}$$

Intuitively, the circles in the first three lines of π_2 simply swap the first and the last variable, the second and the last but one variable, etc., value by value to reverse the series, where the remaining circles adjust the auxiliary variables, i.e., swap the differences value by value, respectively. The circles in π_3 swap the values 0 and $n-1$, 1 and $n-2$, etc., for each variable to reflect the series. Obviously, the permutations π_2 and π_3 represent (2) and (3), respectively, and do not change the logic program.

4 ASP Symmetries via Graph Automorphism

Our approach for detecting symmetries of a logic program is through reduction to, and solution of, an associated Graph Automorphism problem (GAP).

Given a graph $G = (V, E)$, where V is a set of vertices and $E \subseteq V \times V$ is a set of edges. An *automorphism* (symmetry) of G is a permutation of its vertices that maps edges to edges, and non-edges to non-edges. Edge orientation must be preserved in case G is a directed graph. A further extension considers vertex colourings, where symmetries must map each vertex into a vertex with the same colour. More formally, given a partition of the vertices $\pi(V) = \{V_1, V_2, \dots, V_k\}$, the *automorphism group* of G [18] is $Aut(G, \pi) = \{\gamma \in S(V) \mid (V^\gamma, E^\gamma) = (V, E), \pi^\gamma = \pi\}$. We will think of the partition π as a *colouring* of the vertices. The *(Coloured) Graph Automorphism problem* (GAP) is to find all symmetries of a given graph, for instance, in terms of generators. It is not known to have any polynomial time solution, and is conjectured to be strictly between the complexity classes P and NP [3], thus potentially easier than computing answer sets. A practical algorithm for graph automorphism has been implemented in *nauty* [18] and significantly improved in the systems *saucy* [7,8].

A quite natural GAP encoding for detecting symmetries of logic programs is based on their body-atom graph. The *body-atom graph* $G_P = (V, E_0 \cup E_1, E_2)$ of a logic program P is a directed graph with vertices $V = body(P) \cup atom(P)$, and labelled edges $E_0 = \{(\beta, a) \mid a \in atom(P), \beta \in body(a)\}$, $E_1 = \{(a, \beta) \mid \beta \in body(P), a \in \beta^+\}$, and $E_2 = \{(a, \beta) \mid \beta \in body(P), a \in \beta^-\}$. The body-atom graph has been shown to be a suitable representation of a logic program [17].

However, we modify the body-atom graph by introducing additional vertices for negated atoms to circumvent labelled edges.

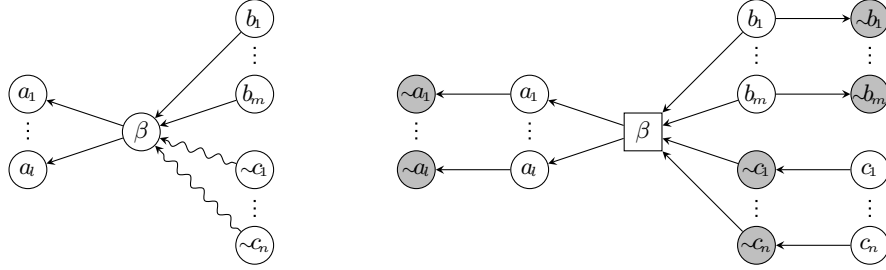


Fig. 1. The left picture shows a rule r of the form (1) as a body-atom-graph, where β is the body vertex. Straight lines represent edges in $E_0 \cup E_1$, curly lines represent edges in E_2 . On the right is the general structure of a 3-coloured graph construction of r . Vertices of color 1, 2, and 3 are represented by empty circles, filled circles, and empty squares, respectively.

In our GAP encoding every atom in $atom(P)$ is represented by two vertices of colour 1 and 2 that correspond to the positive and negative literals, respectively. Every rule is represented by a (body) vertex of colour 3, a set of directed edges that connect the vertices of the literals that appear in the rule's body to its body vertex, and a set of directed edges that connect the body vertex to the vertices of the atoms (positive literals) that appear in the head of the rule. To ensure consistency, that is, a maps to b if and only if $\sim a$ maps to $\sim b$ for any atoms a and b , vertices of opposite literals are mated by a directed edge from the positive literal to the negative literal. The choice of three vertex colours insures that body vertices can only be mapped to body vertices, and positive (negative) literal vertices can only be mapped to positive (negative) literal nodes. To conclude, given a logic program P consisting of m rules and l literals over n atoms, the GAP encoding for detecting symmetries of P is constructed by $m + 2n$ vertices and $l + n$ edges. Examples are given in Fig. 2.

Since graph automorphism algorithms are sensitive to the number of vertices of an input graph, our construction can be optimised to reduce the number of graph vertices while preserving its automorphism group. A first simplification is achieved by modelling rules with an empty body and a single head atom, so-called *facts*, by a (forth) colour for the vertex corresponding to the head atom instead of using (empty) body vertices. Furthermore, rules with a single head atom and a 1-literal body are modelled using a directed edge from the vertex corresponding to the literal of the body to the vertex corresponding to the head atom. Observe that this optimisation may connect a literal vertex to a positive literal vertex, where consistency edges connect positive literal vertices to their negative mates. Therefore, unintended mappings between 1-literal body edges

and consistency edges are impossible. For the special case of a 1-literal body and an empty head, we connect the literal vertex to the special node ' \perp '.

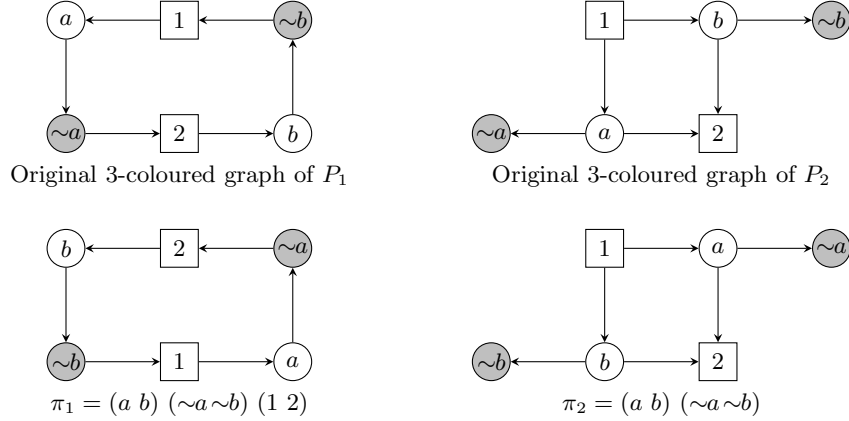


Fig. 2. 3-coloured graph constructions and resulting symmetries for the example logic programs P_1 and P_2 .

5 Symmetry-breaking Constraints

Recall that symmetries of a logic program P induce equivalence classes in the solution space (orbits). Given an answer set of P , all sets to which it can be mapped by symmetries, must be answer sets of P . Similarly, symmetries always map non-answer sets to non-answer sets. Therefore, it is sufficient to reason about one representative from every equivalence class. Symmetry-breaking amounts to selecting some representatives from every equivalence class and constructing rules, composed into a symmetry-breaking constraint, that is only satisfied on those representatives. A *full* SBC selects exactly one representative from each orbit, otherwise we call an SBC *partial*. The most common approach is to order all elements from the solution space lexicographically, and to select the lexicographically smallest element, the *lex-leader*, from each orbit as its representative (cf. [6,1,2,20]). A *lex-leader symmetry-breaking constraint* is an SBC that is satisfied only on the lex-leaders of orbits.

We will assume a total ordering on the atoms a_1, a_2, \dots, a_n of a logic program P and consider the induced lexicographic ordering on the truth assignments, i.e. their interpretation as unsigned integers. The most common approach for accomplishing the construction of a lex-leader SBC is by encoding a *permutation*

constraint (PC) for every permutation π , where

$$PC(\pi) = \bigwedge_{1 \leq i \leq n} \left[\bigwedge_{1 \leq j \leq i-1} (a_j = a_j^\pi) \right] \rightarrow (a_i \leq a_i^\pi).$$

A careful analysis reveals some possibilities to reduce the size of permutation constraints (cf. [20]). The first corresponds to atoms that are mapped to themselves by the permutation, i.e., $a_i^\pi = a_i$. This makes the consequent of the implication unconditionally true. For sparse symmetries, one can significantly reduce the size of the permutation constraint with a restriction of the PC construction to only those atoms that are in the support of π . A second possibility corresponds to the lexicographically biggest atom in each cycle of π . Assume a cycle $(a_s \dots a_e)$ on the atoms of some index set J . Using equality propagation on the portion of the permutation constraint where $i = e$, we get $(a_s = a_e) \rightarrow (a_e \leq a_s)$ which is tautologous. Hence, we can further restrict the index set in the PC by excluding the lexicographically biggest atom in each cycle.

Through *chaining* which includes additional atoms, we achieve a PC representation that is linear in the number of atoms (cf. [1]):

$$\begin{aligned} c_{\pi,1} &\equiv (a_1 \leq a_1^\pi) \wedge c_{\pi,2} \\ c_{\pi,i} &\equiv (a_{i-1} \geq a_{i-1}^\pi) \rightarrow (a_i \leq a_i^\pi) \wedge c_{\pi,i+1} \quad i = 2, \dots, n \\ c_{\pi,n+1} &\equiv 1 \end{aligned}$$

Finally, we encode above permutation constraint in ASP that is satisfied for the lex-leader in the orbit induced by π as follows

$$\begin{aligned} &\leftarrow a_1, \sim a_1^\pi \\ &\leftarrow c_{\pi,2} \\ c_{\pi,i} &\leftarrow a_{i-1}, a_i, \sim a_i^\pi \\ c_{\pi,i} &\leftarrow a_i, \sim a_{i-1}^\pi, \sim a_i^\pi \\ c_{\pi,i} &\leftarrow a_{i-1}, c_{\pi,i+1} \\ c_{\pi,i} &\leftarrow c_{\pi,i+1}, \sim a_{i-1}^\pi \\ c_{\pi,n+1} &\leftarrow \end{aligned}$$

where $i = 2, \dots, n$. The lex-leader symmetry-breaking constraint that breaks every symmetry in a logic program can now be constructed by conjoining all of its permutation constraints.

Example 6. We illustrate our PC encoding on the symmetries detected for the previous examples P_1 and P_2 . Since both permutations π_1 and π_2 (see Fig. 2) map a to b and vice versa, they share the same lex-leader SBC which is as simple as follows (assuming a is lexicographically greater than b):

$$\leftarrow b, \sim a$$

Observe that the ordering on the atoms of a logic program P induces a preference relation on the answer sets of P under symmetry-breaking. Here, the ordering selects $\{a\}$ as the representative of the set of all answer sets symmetric to $\{a\}$, hence, eliminating the answer set $\{b\}$.

6 Partial Symmetry-breaking

Breaking all symmetries may not speed up search because there are often exponentially many of them. A better trade-off may be provided by breaking *enough* symmetries [6]. Irredundant generators are good candidates because they can not be expressed in terms of each other, and implicitly represent all symmetries. Hence, breaking all symmetries in a generating set can eliminate all problem symmetries. However, this does not hold in general. In fact, it has been shown that even when breaking all symmetries is polynomial, there exists cases where for which SBCs based on any irredundant generating set fail to break all symmetry [15].

We can further reduce the size of symmetry-breaking constraints by restricting the construction of permutation constraints up to the k -th atom in each permutation [1].

Example 7. Consider the All-interval Series problem encoded as in Example 2 and the generators π_2 and π_3 from Example 5. The symmetry-breaking constraint, where both permutation constraints are restricted to the second atom, is given through the following, where c_0, \dots, c_3 are new atoms.

$$\begin{array}{ll}
\leftarrow v_{1,0}, \sim v_{1,n-1} & \leftarrow v_{1,0}, \sim v_{n,0} \\
\leftarrow c_0 & \leftarrow c_2 \\
c_0 \leftarrow v_{1,0}, v_{1,1}, \sim v_{1,n-2} & c_2 \leftarrow v_{1,0}, v_{1,1}, \sim v_{n,1} \\
c_0 \leftarrow v_{1,1}, \sim v_{1,n-1}, \sim v_{1,n-2} & c_2 \leftarrow v_{1,1}, \sim v_{n,0}, \sim v_{n,1} \\
c_0 \leftarrow v_{1,0}, c_1 & c_2 \leftarrow v_{1,0}, c_3 \\
c_0 \leftarrow c_1, \sim v_{1,n-1} & c_2 \leftarrow c_3, \sim v_{n,0} \\
c_1 \leftarrow & c_3 \leftarrow
\end{array}$$

7 Experiments

Our approach to Symmetry-breaking Answer Set Solving has been implemented within the preprocessor *sbass*⁵. It accepts a logic program in *smodels*⁶ format [22] produced by a grounder, e.g. *lpase* and *gringo*, and incorporates the graph automorphism tool *saucy*⁷ (2.1) for detecting irredundant generators of the group of permutational symmetries. In return, *sbass* outputs the given program together with symmetry-breaking constraints, again in *smodels* format, which can be applied to any suitable answer set solver, e.g. *smodels* and *clasp*. Note that *sbass* provides several options, for instance, to print detected generators in cycle notation or statistics.

To evaluate our approach, we conducted experiments on ASP encodings of several difficult combinatorial search problems. Experiments consider the answer set solver *clasp* (1.3.2) on instances with symmetry-breaking in terms of generators, i.e., instances preprocessed by *sbass*, and without symmetry-breaking. To

⁵ <http://potassco.sourceforge.net/> provides *clasp*, *gringo*, and *sbass*

⁶ <http://www.tcs.hut.fi/Software/smodels/> provides *lpase* and *smodels*

⁷ <http://vlsicad.eecs.umich.edu/BK/SAUCY/>

explore the impact of partial SBC, we also tried restrictions on the construction of permutation constraints up to the k -th support in a permutation, denoted as $clasp_k^\pi$. All tests were run on a 2.00 GHz PC under Linux, where each run was limited to 600 s time and 1 GB RAM, preprocessing excluded. However, we also report the runtime for *sbass* and give the number of generators. The latter gives an impression about the size of the search space implicitly pruned through symmetry-breaking. In the experiments below we generally compare the runtime for testing the existence of an answer set to a given problem.

7.1 Pigeon Hole Problems

The *Pigeon Hole problem* is to show that it is impossible to put n pigeons into $n - 1$ holes if each pigeon must be put into a distinct hole. This problem is provably exponentially hard for any resolution based method, but is tractable using symmetries (all the pigeons are interchangeable and all the holes are interchangeable). We encoded the Pigeon Hole problem based on the support encoding for the all-different constraint [10], as follows, where p_{ij} is taken to mean that pigeon i is assigned hole j :

$$\begin{array}{ll} p_{i,1}; p_{i,2}; \dots; p_{i,n-1} & \leftarrow \quad i \in 1..n \\ & \leftarrow p_{i,j}, p_{k,j} \quad i \neq k \end{array}$$

The runtimes for various sizes of n are shown in Table 1. Although symmetry-breaking has a positive impact, the runtime even with $clasp_\infty^\pi$ is still exponentially growing with the number of pigeons. Here, symmetry-breaking on the generating set returned by *saucy* does not break all problem symmetries. We enforced *saucy* to compute a different set of generators, denoted as $clasp_\infty^\theta$, and got a polynomial runtime. On such problems, full SBCs are essential.

Table 1. Runtime results in seconds for Pigeon Hole problems.

#n	#gen.	<i>sbass</i>	$clasp_1^\pi$	$clasp_5^\pi$	$clasp_\infty^\pi$	$clasp_\infty^\theta$	<i>clasp</i>
11	18	0.05	0.38	0.15	0.06	0.02	0.62
12	20	0.08	4.09	0.07	0.22	0.03	5.99
13	22	0.11	30.57	0.43	0.32	0.03	53.39
14	24	0.16	272.72	4.95	1.73	0.04	448.98
15	26	0.23	—	62.61	3.02	0.04	—
16	28	0.32	—	—	23.01	0.07	—
17	30	0.44	—	—	130.87	0.10	—

7.2 Ramsey's Theorem

Ramsey's Theorem states that for any pair of positive integers (k, m) there exists a least positive integer n such that, no matter how we color the edges of the clique

with n vertices, K_n , using two colours, say blue and red, there is a sub-clique with k nodes of colour blue or a sub-clique with m nodes of colour red. We used the encoding from [16], denoted as $R(k, m, n)$, to determine whether n is not an integer for which the theorem holds.

In formerly hard cases, namely $R(3, 5, 14)$ and $R(4, 5, 24)$, symmetry-breaking lead to significant pruning of the search space and yield solutions in a considerably short amount of time. The results presented in Table 2 suggest full SBCs for unsatisfiable instances, but small, partial SBCs for satisfiable instances.

Table 2. Average time for completed runs in seconds and the number of timeouts on Ramsey’s Theorem instances, each shuffled 5 times. The *asterisk denotes instances that have no answer sets.

		<i>sbass</i>		<i>clasp</i> π_1^{π}		<i>clasp</i> π_5^{π}		<i>clasp</i> π_{∞}^{π}		<i>clasp</i>	
	#gen.	time	time	#t.out	time	#t.out	time	#t.out	time	#t.out	
$R(3, 5, 13)$	11	0.06	0.01		0.01		0.03		0.01		
$R(3, 5, 14)^*$	12	0.10	3.58		1.23		0.49		354.25		
$R(3, 6, 17)$	15	1.18	0.12		0.12		0.14		0.11		
$R(3, 6, 18)^*$	16	1.87	—	5	—	5	—	5	—	5	
$R(4, 4, 17)$	15	0.26	0.73		0.12		0.50		0.07		
$R(4, 4, 18)^*$	16	0.37	—	5	—	5	—	5	—	5	
$R(4, 5, 23)$	21	5.43	4.23		2.29		2.05		1.32		
$R(4, 5, 24)$	22	7.15	77.64		208.66	1	180.96	3	—	5	
$R(4, 5, 25)^*$	23	9.54	—	5	—	5	—	5	—	5	

7.3 Graceful Graphs

A labelling f of the vertices of a graph (V, E) is *graceful* if f assigns a unique label $f(v)$ from $\{0, 1, \dots, |E|\}$ to each vertex $v \in V$ such that, when each edge $(v, w) \in E$ is assigned the label $|f(v) - f(w)|$, the resulting edge labels are distinct. The problem of determining the existence of a graceful labelling of a graph has been modelled as a CSP in [19], and is an interesting application for Symmetry-breaking Answer Set Solving because the symmetries are different for each instance and can not be modelled a-priori in general. Our experiments consider graphs DW_n and K_nP_m . The *double wheel* graph DW_n is composed of two copies of a cycle with n vertices, each connected to a central hub. The two wheels W_n , each have rotation and reflection symmetries. The labels of the two cycles can also be interchanged. The graph K_nP_m is the cross-product of the clique K_n and the path P_m . It consists of n copies of K_n , with corresponding vertices in the m cliques also forming the vertices of a path P_m . Symmetries of the graph are simultaneous rotations of the cliques and inter-clique permutations.

As can be seen in Table 3, we achieve speed-up on the unsatisfiable instance DW_3 . For the other instances, all of which are satisfiable, the branching heuristic used in our approach sometimes appears to be misled by the extra variables

introduced in $clasp_k^\pi$. That explains some of the variability in the runtimes. However, the difficult instances show symmetry-breaking to be outperforming.

Table 3. Average time for completed runs in seconds and the number of timeouts on Graceful Graph instances, each shuffled 5 times. The *asterisk denotes instances that have no answer sets.

		<i>sbass</i>	<i>clasp</i> ₁ ^π	<i>clasp</i> ₅ ^π	<i>clasp</i> _∞ ^π	<i>clasp</i>		
	#gen.	time	time	#t.out	time	#t.out	time	#t.out
<i>DW</i> ₃ *	5	0.02	4.24		1.45		1.32	5.40
<i>DW</i> ₆	5	0.17	0.46		0.56		1.09	0.57
<i>DW</i> ₈	5	0.48	28.81		5.47		17.11	4.30
<i>DW</i> ₁₀	5	1.21	191.86		66.18		61.59	27.04 2
<i>DW</i> ₁₂	5	3.34	145.89		202.18	1	111.96	1 112.38 4
<i>K</i> ₃ <i>P</i> ₃	3	0.04	0.08		0.08		0.07	0.08
<i>K</i> ₄ <i>P</i> ₂	4	0.07	0.20		0.10		0.54	0.19
<i>K</i> ₄ <i>P</i> ₃	4	0.29	24.68		29.06		198.57	24.01
<i>K</i> ₅ <i>P</i> ₂	5	0.37	274.85	3	334.55	3	312.56	1 226.03 3

7.4 Answer Set Enumeration

Finally, we want to test the impact of symmetry-breaking on the number of answer sets. We modelled the All-interval Series problem (AllInt) as described in Example 2, using a direct representation for n integer variables and auxiliary variables to represent the differences between adjacent numbers (cf. [10]), and required both sets of variables to be all-different.

As expected we observe that symmetry-breaking significantly reduces the number of solutions, and therefore, reduces the time necessary for post-processing solutions (see Table 4). Clearly, $clasp_k^\pi$ for an increasing number k discards more solutions (eliminating up to 90 per cent of the solution space).

As in all previous benchmarks, it seems safe to assume that the detection of symmetries in logic programs through reduction to graph automorphism is computationally quite feasible using today’s GAP tools such as *saucy*.

We should also note that a given problem can be encoded in many equivalent logic programs, and with each different encoding our techniques may detect a different generating set. For instance, we tried symmetry detection and symmetry-breaking on logic programs that were preprocessed, i.e. simplified. The key idea of preprocessing logic programs is to identify equivalences among its relevant constituents. These equivalences are then used for building a compact representation of the program [11]. Sometimes, we observed significant better results in terms of time and number of answer sets (eliminating up to 95 per cent of the solution space).

Table 4. Results on computing all answer sets of selected instances. Runtime, number of solutions, and the maximum compression achieved using full SBC are shown.

		<i>sbass</i>	<i>clasp</i> ₁ ^π		<i>clasp</i> ₅ ^π		<i>clasp</i> _∞ ^π		<i>clasp</i>		
	#gen.	time	time	#sol.	time	#sol.	time	#sol.	time	#sol.	compr.
<i>AllInt</i> ₈	2	0.01	0.15	39	0.11	15	0.17	14	0.14	40	65%
<i>AllInt</i> ₉	2	0.01	0.78	119	0.60	60	0.93	40	0.77	120	67%
<i>AllInt</i> ₁₀	2	0.01	4.60	295	3.43	148	5.69	107	4.08	296	64%
<i>AllInt</i> ₁₁	2	0.01	23.26	647	22.82	372	32.70	238	24.40	648	63%
<i>AllInt</i> ₁₂	2	0.01	161.90	1327	147.17	862	211.27	442	160.32	1328	67%
<i>DW</i> ₄	5	0.07	282.36	9472	168.03	5152	85.65	1150	314.15	11264	90%
<i>K</i> ₃ <i>P</i> ₃	3	0.05	229.15	5704	119.99	2836	126.25	1487	268.80	6816	76%
<i>K</i> ₄ <i>P</i> ₂	4	0.08	119.66	1080	67.96	552	27.72	146	145.13	1440	90%

8 Conclusions

We have investigated symmetry-breaking in the context of Answer Set Programming. In particular, we proposed a reduction from symmetry detection of disjunctive logic programs to the automorphisms of a coloured digraph. Our techniques were formulated as a completely automated flow that (1) starts with a logic program, (2) detects all of its permutational symmetries, (3) represents all symmetries implicitly and always with exponential compression, (4) adds symmetry-breaking constraints that do not affect the existence of answer sets, and (5) can be applied to any existing ASP system without changing its code, which allows for programmers to select the solvers that best fit their needs.

We have empirically evaluated symmetry-breaking on difficult combinatorial search problems and got promising results. In many cases, SBC lead to significant pruning of the search space and yield solutions to problems which are otherwise intractable. We also observe a significant compression of the solution space which makes symmetry-breaking attractive whenever all solutions have to be post-processed.

Motivated by this success future work concerns an extension to choice rules and weight constraints [21]. However, one should not expect Symmetry-breaking Answer Set Solving to give improvement on all benchmark classes. Many ASP benchmarks⁸ have large numbers of symmetries, but can be solved so quickly that the symmetry detection and -breaking overhead is not justified.

Furthermore, it is often reasonable to assume that the symmetries for a problem are known. For particular symmetries, there are more efficient breaking methods (cf. [24]). This is also target to future work.

Acknowledgements The work of Christian Drescher is partially funded by the Austrian Science Fund (FWF) under grant number P20841 and the Vienna Science and Technology Fund (WWTF) under grant ICT08-020.

⁸ <http://asparagus.cs.uni-potsdam.de/>

References

1. Aloul, F.A., Markov, I.L., Sakallah, K.A.: Shatter: efficient symmetry-breaking for boolean satisfiability. In: Proceedings of DAC'03. pp. 836–839. ACM (2003)
2. Aloul, F.A., Ramani, A., Markov, I.L., Sakallah, K.A.: Solving difficult instances of boolean satisfiability in the presence of symmetry. *IEEE Transactions on CAD of Integrated Circuits and Systems* 22(9), 1117–1137 (2003)
3. Babai, L.: Automorphism groups, isomorphism, reconstruction. In: Graham, R.L., Grötschel, M., Lovász, L. (eds.) *Handbook of Combinatorics*, vol. 2, pp. 1447–1540. Elsevier (1995)
4. Baral, C.: *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press (2003)
5. Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.): *Handbook of Satisfiability*. IOS Press (2009)
6. Crawford, J., Ginsberg, M., Luks, E., Roy, A.: Symmetry-breaking predicates for search problems. In: Proceedings of KR'96. pp. 148–159. Morgan Kaufmann (1996)
7. Darga, P.T., Liffiton, M.H., Sakallah, K.A., Markov, I.L.: Exploiting structure in symmetry detection for cnf. In: Proceedings of DAC'04. pp. 530–534. ACM Press (2004)
8. Darga, P.T., Sakallah, K.A., Markov, I.L.: Faster symmetry discovery using sparsity of symmetries. In: Proceedings of DAC'08. pp. 149–154. ACM (2008)
9. Drescher, C., Gebser, M., Grote, T., Kaufmann, B., König, A., Ostrowski, M., Schaub, T.: Conflict-driven disjunctive answer set solving. In: Proceedings of KR'08. pp. 422–432. AAAI Press (2008)
10. Drescher, C., Walsh, T.: A translational approach to constraint answer set solving. In: Proceedings of ICLP'10. Cambridge University Press (2010), To appear
11. Gebser, M., Kaufmann, B., Neumann, A., Schaub, T.: Advanced preprocessing for answer set solving. In: Proceedings of ECAI'08. pp. 15–19. IOS Press (2008)
12. Gebser, M., Kaufmann, B., Schaub, T.: The conflict-driven answer set solver clasp: Progress report. In: Proceedings of LPNMR'09. pp. 509–514. Springer (2009)
13. Gelfond, M., Lifschitz, V.: Classical negation in logic programs and disjunctive databases. *New Generation Computing* 9, 365–385 (1991)
14. Gent, I.P., Walsh, T.: CSPLIB: A benchmark library for constraints. In: Proceedings of CP'99. pp. 480–481. Springer (1999)
15. Katsirelos, G., Narodytska, N., Walsh, T.: Breaking generator symmetry (2009)
16. Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The dlv system for knowledge representation and reasoning. *ACM Transactions on Computational Logic* 7, 499–562 (2002)
17. Linke, T.: Suitable graphs for answer set programming. In: Proceedings of ASP'03. pp. 15–28 (2003)
18. McKay, B.: Practical graph isomorphism. In: *Numerical mathematics and computing*. pp. 45–87 (1981)
19. Petrie, K.E., Smith, B.M.: Symmetry breaking in graceful graphs. In: Proceedings of CP'03. pp. 930–934. Springer (2003)
20. Sakallah, K.A.: Symmetry and satisfiability. In: Biere et al. [5], pp. 289–338
21. Simons, P., Niemelä, I., Sooinen, T.: Extending and implementing the stable model semantics. *Artificial Intelligence* 138(1-2), 181–234 (2002)
22. Syrjänen, T.: *Lparse 1.0 user's manual*
23. Urquhart, A.: Hard examples for resolution. *Journal of ACM* 34(1), 209–219 (1987)
24. Walsh, T.: Symmetry breaking using value precedence. In: Proceedings of ECAI'06. pp. 168–172. IOS Press (2006)