Symmetry-breaking Answer Set Solving *

Christian Drescher [a], Oana Tifrea [b] and Toby Walsh [a]

[a] *NICTA and University of New South Wales*
Locked Bag 6016
University of New South Wales
Kensington 1466, Australia
E-mail:
{christian.drescher, toby.walsh}@nicta.com.au
[b] *Free University of Bozen-Bolzano*
Computer Science Faculty
Piazza Domenicani 3
39100 Bolzano, Italy
E-mail: oana.tifrea@unibz.it

We investigate the role of symmetry detection and symmetry breaking in answer set programming to eliminate symmetric parts of the search space and, thereby, simplify the solution process. We reduce symmetry detection to a graph automorphism problem which allows us to extract symmetries of a logic program from the symmetries of the constructed coloured graph. The correctness of our reduction is proven. We also propose an encoding of symmetry-breaking constraints in terms of permutation cycles and use only generators in this process to implicitly represent symmetries with exponential compression. These ideas are formulated as preprocessing and implemented in a completely automated flow that first detects symmetries from a given answer set program, adds symmetry-breaking constraints, and can be applied to any existing answer set solver. We demonstrate computational impact on benchmarks versus direct application of the solver.

Keywords: answer set programming, symmetry breaking, graph automorphism problem

*Part of this work was performed when Christian Drescher was studying at the Vienna University of Technology, Austria, supported by the Austrian Science Fund (FWF) under grant number P20841 and the Vienna Science and Technology Fund (WWTF) under grant ICT 08-020. Oana Tifrea was supported by the European Master's Program in Computational Logic (EMCL). NICTA is funded by the Department of Broadband, Communications and the Digital Economy, and the Australian Research Council.

1. Introduction

Answer set programming (ASP; [6]) is a promising approach for knowledge representation and nonmonotonic reasoning in various applications that include difficult combinatorial search, among them bioinformatics [7], crypto analysis [1], configuration [51], database integration [38], diagnosis [18], hardware design [22], model checking [31], planning [40], preference reasoning [10], semantic web [20], and as a highlight among these applications the high-level control of the space shuttle [44]. 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 [16]. In fact, ASP solvers have experienced dramatic improvements in their performance [24] and compete [49] with the best Boolean satisfiability (SAT; [9]) solvers.

However, many combinatorial search problems exhibit symmetries which can frustrate a search algorithm to fruitlessly explore independent symmetric subspaces. The relevance of symmetry to real world applications is very strong since it can prevent a solver to solve even small problems [48]. Indeed, various instance families, such as the *pigeon hole* problem, require exponential time for resolution and backtracking algorithms [55], and state-of-the-art ASP solvers take a very long time to solve these instances (see Section 7). Once their symmetries are identified, we can avoid redundant computational effort by pruning parts of the search space through symmetry breaking. Symmetry breaking also helps 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 [28] have plenty of symmetric solutions. However, all solutions to the original problem can be reconstructed from the answer sets under symmetry breaking.

This work breaks the problem of symmetry breaking down into two parts: (1) identifying symmetries and (2) breaking the identified symmetries. We adopt existing theoretical foundations from symmetry detection for constraint satisfaction problems [12] and present a reduction of symmetry detection to the graph automorphism problem (GAP; [43]). For SAT, this has been proposed by Crawford et al. in [13] and further refined by Aloul et al. in [2,3]. Detected symmetries can then be eliminated with symmetry-breaking constraints (SBCs). These constraints ensure that a search engine never visits two points in the search space that are equivalent under the symmetry they represent. Unfortunately, generating all SBCs is intractable since there might be an exponential number of symmetries, but partial symmetry breaking can be done in polynomial time (assuming that the associated GAP is tractable). While Crawford et al. construct a partial symmetry tree, Aloul et al. restrict to a set of irredundant generators of the symmetric group.

The key contribution of our work is a reduction of symmetry detection for the class of disjunctive and extended logic programs to the automorphisms of a coloured digraph which allows us to extract symmetries of a logic program from the symmetries of the constructed graph, and also propose a linear-sized ASP representation of SBC. In particular, we completely automate a flow that starts with a logic program and finds all of its symmetries within a very general class, including all *syntactic* symmetries, i.e., permutations that do not change the logic program. In our flow, all symmetries are captured implicitly, in terms of irredundant group generators, which guarantees exponential compression. The logic program is then preprocessed by adding symmetry-breaking constraints that do not affect the existence of answer sets. Any ASP solver can be applied to the preprocessed logic program without changing its code, which allows for programmers to select the solvers that best fit their needs.

The remaining material is organised as follows. First, we provide all necessary preliminaries to answer set programming, give group theoretic background and define what we mean by a symmetry in Section 2. In Section 3, we present our symmetry detection techniques for logic programs, and for their extensions. Sections 4 and 5 give our to symmetry-breaking methods. We implemented our techniques in a system (Section 6) which we evaluate in Section 7. Section 8 concludes our work.

2. Background

2.1. Answer Set Programming

As a form of declarative programming oriented towards combinatorial search problems, ASP comes with a simple modelling language.

Definition 2.1. *A* (disjunctive) logic program *over a set of primitive propositions $\mathcal{A}$ is a finite set of* rules r *of the form*

$$a_1; \dots ; a_\ell \leftarrow b_1, \dots , b_m, {\sim}c_1, \dots , {\sim}c_n \tag{1}$$

where $a_i, b_j, c_k \in \mathcal{A}$ are atoms *for $1 \leq i \leq \ell$, $1 \leq j \leq m$, and $1 \leq k \leq n$.*

A *default literal* $\hat{a}$ is an atom a or its default negation ${\sim}a$. Let $H(r) = \{a_1, \dots , a_\ell\}$ be the *head* of r and $B(r) = \{b_1, \dots , b_m, {\sim}c_1, \dots , {\sim}c_n\}$ the *body* of r. For a set S of default literals, 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 $B(P) = \{B(r) \mid r \in P\}$. For regrouping bodies sharing the same head atom a, define $B(a) = \{B(r) \mid r \in P, a \in H(r)\}$. If $|H(r)| = 1$ for all $r \in P$, i.e., all rules in the P have a single head atom, we call P a *normal* logic program.

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* [26]

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

A rule of form (1) can be seen as a constraint on the answer sets of a program, stating that if $b_1, \dots , b_m$ are in the answer set and none of $c_1, \dots , c_n$ are included, then one of $a_1, \dots , a_\ell$ must be in the set. Important extensions to logic programs are integrity constraints, choice rules, and cardinality constraints [50].

Definition 2.2. *Given an alphabet $\mathcal{A}$. An* integrity constraint *has the form*

$$\leftarrow b_1, \dots , b_m, {\sim}c_1, \dots , {\sim}c_n \tag{2}$$

where $b_j, c_k \in \mathcal{A}$, for $1 \leq j \leq m$ and $1 \leq k \leq n$.

We understand an integrity constraint as a short hand for a rule with an unsatisfiable head, and thus forbids its body to be satisfied in any answer set.

Example 2.1. *Consider the logic programs P_1 and P_2, both have two answer sets $\{a\}$ and $\{b\}$, given by*

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

To verify, for instance, answer set $\{a\}$, we consider the reduct $P_1^{\{a\}}$, $P_2^{\{a\}}$ respectively:

$$P_1^{\{a\}} = \{\, a \leftarrow \,\}, \qquad P_2^{\{a\}} = \{\, a; b \leftarrow \,\}.$$

The $\subseteq$-minimal model of $P_1^{\{a\}}$ is $\{a\}$. $P_2^{\{a\}}$ has three classical models, $\{a\}$, $\{b\}$, and $\{a,b\}$ where $\{a\}$ and $\{b\}$ are $\subseteq$-minimal. Therefore, $\{a\}$ is an answer set of both P_1 and P_2. 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.

Definition 2.3. *Given an alphabet $\mathcal{A}$. A* choice rule *has the form*

$$\{a_1, \ldots, a_\ell\} \leftarrow b_1, \ldots, b_m, {\sim}c_1, \ldots, {\sim}c_n \qquad (3)$$

where $a_i, b_j, c_k \in \mathcal{A}$, for $1 \le i \le \ell$, $1 \le j \le m$, and $1 \le k \le n$.

A choice rule allows for the nondeterministic choice over atoms in $\{a_1, \ldots, a_n\}$.

Definition 2.4. *Given an alphabet $\mathcal{A}$. A* cardinality constraint *has the form*

$$\leftarrow k\{\hat{a}_1, \ldots, \hat{a}_n\} \qquad (4)$$

where $a_i \in \mathcal{A}$, for $1 \le i \le n$ and $k \ge 0$ is an integer.

A cardinality constraint is interpreted as no answer set satisfies k or more default literals of the set $\{\hat{a}_1, \ldots, \hat{a}_n\}$.

More formally, the semantics of integrity constraints, choice rules, and cardinality constraints can be given through program transformations that introduce additional propositions [50].

ASP engineers usually use a *generate-and-test* technique [6] to model a problem, by producing the space of solution candidates in the *generate* component and defining rules that filter invalid solutions in the *test* component. For instance, the first line of our *all-interval series* problem encoding from Examples 2.2 generates an assignment to the problem variables. The remaining rules comprise the *test* component as they eliminate assignments that do not solve the problem.

Example 2.2. *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 $v_{i,k}$ for the integer variables (indexed $1 \le i \le n$) taking values $0 \le k < n$, and $d_{i,m}$ for the auxiliary variables (indexed $1 \le i < n$) taking values $1 \le m < n$ to represent the differences between adjacent numbers. Furthermore, we require both sets of variables to have pairwise different values.*

$$\begin{array}{rll} v_{i,0}; \ldots; v_{i,n-1} \leftarrow & & \\ \leftarrow & v_{i,k}, v_{j,k} & i < j \\ d_{i,|k-\ell|} \leftarrow & v_{i,k}, v_{i+1,\ell} & \\ \leftarrow & d_{i,m}, d_{j,m} & i < j \end{array}$$

Note that above encoding remains invariant under complex permutation of atoms. We refer to Example 2.6 for a detailed analysis.

2.2. *Translating Answer Set Programs into Propositional Logic*

As shown by Lee in [36], the answer sets of a logic program P correspond to the classical models of P that satisfy all loop formulas, where the classical models of P are represented by the set of formulas

$$\mathrm{RF}_P = \left\{ \left(\bigwedge_{b \in B(r)^+} b \wedge \bigwedge_{c \in B(r)^-} \neg c \right) \rightarrow \bigvee_{a \in H(r)} a \mid r \in P \right\}.$$

A nonempty set $L \subseteq \mathcal{A}$ is called a *loop* of P, if for all nonempty $K \subset L$, there is some $r \in P$ such that $H(r) \cap K \neq \emptyset$ and $B(r) \cap (L \setminus K) \neq \emptyset$ [25]. Note that every atom contained in $\mathcal{A}$, forms a loop of P, i.e. a singleton, and if all loops are singletons, then P is called *tight* [21]. For a loop L, let

$$\sup_P(L) = \{r \in P \mid H(r) \cap L \neq \emptyset, B(r) \cap L = \emptyset\}$$

be the set of rules from P that can externally support L. The *(disjunctive) loop formula* [36] of L, denoted $\mathrm{LF}_P(L)$, is:

$$\bigvee_{a \in L} a \rightarrow \bigvee_{r \in \sup_P(L)} \left(\bigwedge_{b \in B(r)^+} b \wedge \bigwedge_{c \in B(r)^-} \neg c \wedge \bigwedge_{d \in H(r) \setminus L} \neg d \right).$$

Finally, let $\mathrm{loop}(P)$ denote the set of all loops in P and $\mathrm{LF}_P = \{\mathrm{LF}_P(L) \mid \mathrm{loop}(P)\}$. Then, according to Lee, a set $M \subseteq \mathcal{A}$ is an answer set of a logic program P, if M is a model of $\mathrm{RF}_P \cup \mathrm{LF}_P$.

The influential Clark's completion [11] allows us a slightly different characterisation. The *completion $Comp(P)$ of a logic program P* over alphabet $\mathcal{A}$ is defined as $\mathrm{Comp}(P) = \mathrm{RF}_P \cup \bigcup_{a \in \mathcal{A}} \mathrm{LF}(\{a\})$, i.e. the set of rules in P, and the loop formulas for singletons. Hence, a set $M \subseteq \mathcal{A}$ is an answer set of a logic program P, if M is a model of $\mathrm{Comp}(P) \cup \mathrm{LF}_P$ [8,37].

2.3. Group Theoretic Background

Intuitively, a symmetry of a discrete object is a transformation of its components that leaves the object unchanged. 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 *identity* element and every element has a unique *inverse.* A *subgroup* is a subset of a group that is closed under the group operation, and is therefore a group itself. Often, we abuse notation and refer to the group G, rather than to the structure $(G, *)$. We denote the size of a group G as $|G|$.

In our context, the most important group is the group of permutations. A *permutation* of a set S is a bijection $\pi : S \rightarrow S$. Indeed, the set of permutations form a group under composition, denoted as $\Pi(S)$. 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 $a \in S$ under a permutation π is denoted as a^π. For a set $X = \{a_1, a_2, \ldots, a_k) \subseteq S$ define $X^\pi = \{a^\pi \mid a \in X\}$. Analogously, for vectors $s = (a_1, a_2, \ldots, a_k) \in S^k$ define $s^\pi = (a_1^\pi, a_2^\pi, \ldots, a_k^\pi)$. We will make use of the *cycle notation* where a permutation is a product of disjoint cycles. A cycle $(a_1\ a_2\ a_3\ \ldots\ a_n)$ means that the permutation maps a_1 to a_2, a_2 to a_3, and so on, finally a_n back to a_1. An element that does not appear in any cycle is understood as being mapped to itself. Furthermore, we define the *support* of a permutation [43] as those elements that are not mapped to themselves. The *orbit* of $a \in S$ under a permutation $\pi \in \Pi(S)$ are the set of elements of S to which a can be mapped by (repeatedly) applying π. Note that orbits define an equivalence relation on elements in S.

A compact representation of a group is given through generators.

Definition 2.5. *A set of group elements such that any other group element can be expressed in terms of their product is called a* generating set *or* set of generators, *and its elements are called* generators. *A generator is* redundant *if it can be expressed in terms of other generators. An* irredundant *generating set, by definition, has no strict subset that is also a set of generators.*

Example 2.3. *Consider the following set G of permutation of the elements in $\{a, b, c, d\}$: the idenity mapping id, $\pi_1 = (a\ b)$, $\pi_2 = (c\ d)$, and $\pi_3 = (a\ b)\ (c\ d)$. The identity fixes each element, π_1 interchanges a and b, and fixes the c and d, π_2 interchanges c and d, and fixes the others. The permutation π_3 is the composition of the previous two, i.e., exchanges a with b and c with d simultaneously. G forms a group under permutation multiplication, since $\pi_1\pi_1 = id$, $\pi_2\pi_2 = id$, $\pi_1\pi_2 = \pi_3$, and $\pi_3\pi 3 = id$. A generating set is given through $\{\pi_1, \pi_2, \pi_3\}$. However, one of either π_1, π_2 or π_3 is redundant, i.e., can be represented as a composition of the other two permutations. Hence, $\{\pi_1, \pi_2\}$, $\{\pi_1, \pi_3\}$, $\{\pi_2, \pi_3\}$, each is an irredundant generating set.*

An irredundant set of generators provides an extremely compact representation of a group. In fact, representing a group by a generating set ensures exponential compression.

Theorem 2.1 (Lagrange, from Elementary Group Theory; [30])**.** *The size of any subgroup of any finite group G must divide $|G|$.*

Corollary 2.2 (Aloul et al. [3])**.** *Any irredundant set of generators for a finite, nonempty group G contains at most $\log_2 |G|$ elements.*

To relate different groups, we recall some more notions from algebra. A mapping $f : G \rightarrow H$ between to groups $(G, *)$ and $(H, \circ)$ is a *homomorphism* iff for and $a, b \in G$ we have $f(a * b) = f(a) \circ f(b)$. A homomorphism for which an inverse exists that is also a homomorphism, is called an *isomorphism.* If an isomorphism exists, the two groups G and H are called *isomorphic.* An isomorphism of a group with itself is called an *automorphism.* Since we can describe groups in terms of generators, it is important to know that isomorphisms preserve generators.

Theorem 2.3 (Aloul et al. [3])**.** *Any group isomorphism maps sets of generators to sets of generators, and maps irredundant sets of generators to irredundant sets of generators.*

2.4. Graph Automorphism Problems

In graph theory, the symmetries are studied in terms of graph automorphisms. We consider directed graphs $G = (V, E)$, where V is a set of vertices and $E \subseteq V \times V$ is a set of directed edges. Intuitively, an automorphism of G is a permutation of its vertices that maps edges to edges, and non-edges to non-edges, preserving edge orientation. More formally, we define as follows.

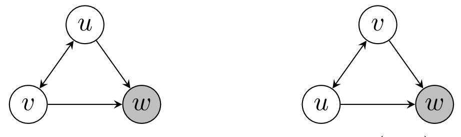
Definition 2.6. *An* automorphism *or a* symmetry *of a graph $G = (V, E)$ is a permutation $\pi \in \Pi(V)$ such that $(u, v) \in E$ iff $(u, v)^\pi \in E$.*

A further extension considers vertex colourings, where symmetries must map each vertex into a vertex with the same colour.

Definition 2.7. *Given a partition of the vertices $\rho(V) = \{V_1, V_2, \ldots, V_k\}$. An* automorphism *or a* symmetry of a coloured graph *G is a symmetry π of G such that $\rho(V)^\pi = \rho(V)$.*

We will think of the partition ρ as a *colouring* of the vertices.

Example 2.4. *Consider the graph $G = (\{u, v, w\}, \{(u, v), (u, w), (v, u), (v, w)\})$ and the partition ρ given through $\{\{u, v\}, \{w\}\}$. G's only nontrivial symmetry is $\pi = (u\ v)$.*



Original coloured graph $\pi = (u\ v)$

The *(coloured) graph automorphism* problem 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 [5], thus potentially easier than computing answer sets. Practical algorithms for computing graph automorphism groups have been implemented in the systems NAUTY [43], SAUCY [15], and BLISS [33].

2.5. Symmetry in Answer Set Programming

We will follow Krishnamurthy, who was one of the first to exploit symmetry [35] for SAT. He defined symmetry as a permutation of the variables leaving the set of clauses 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. For a rule r of the form (1) and a permutation π define r^π as

$$a_1^\pi; \ldots ; a_\ell^\pi \leftarrow b_1^\pi, \ldots , b_m^\pi, {\sim}c_1^\pi, \ldots , {\sim}c_n^\pi$$

For a set of rules P, i.e., a logic program, define $P^\pi = \{r^\pi \mid r \in P\}$.

Definition 2.8. *A* symmetry of a logic program *P is a permutation $\pi \in \Pi(atom(P))$ such that $P^\pi = P$.*

By definition, a symmetry of a logic program preserves answer sets.

Example 2.5. *Reconsider P_1 from Example 2.1, and $\pi = (a\ b)$. Since $P_1^\pi = P_1$, π is a symmetry of P_1.*

Example 2.6. *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 2.2) given in cycle notation below.*

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

$$\begin{aligned}
\pi_3 = {} & (v_{1,0}\ v_{1,n-1}) \ldots (v_{n,\lfloor\frac{(n-1)}{2}\rfloor}\ v_{n,\lceil\frac{(n-1)}{2}\rceil}) \\
& \ldots \\
& (v_{n,0}\ v_{n,n-1}) \ldots (v_{n,\lfloor\frac{(n-1)}{2}\rfloor}\ v_{n,\lceil\frac{(n-1)}{2}\rceil})
\end{aligned}$$

Intuitively, the cycles 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 cycles adjust the auxiliary variables, i.e., swap the differences value by value, respectively. The cycles 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.

3. Symmetry Detection

Our approach for detecting symmetries of a logic program is through reduction to, and solution of, an associated *graph automorphism* problem. Our techniques are based on the *body-atom graph* $(V, E_0 \cup E_1, E_2)$ of a logic program P, that is, a directed graph with vertices $V = B(P) \cup atom(P)$, and labelled edges

$$\begin{aligned} E_0 &= \{(\beta, a) \mid a \in atom(P), \beta \in B(a)\}, \\ E_1 &= \{(a, \beta) \mid \beta \in B(P), a \in \beta^+\}, \text{ and} \\ E_2 &= \{(a, \beta) \mid \beta \in B(P), a \in \beta^-\}. \end{aligned}$$

The body-atom graph has been shown to be a useful representation of a logic program [42]. However, we modify the body-atom graph by introducing additional vertices for negated atoms to circumvent labelled edges, and construct a 3-coloured graph as follows:

1. In our graph 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.
2. 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.
3. 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 bodies and l literals over n atoms, the graph encoding for detecting symmetries of P is constructed by $m + 2n$ vertices and $l + n$ edges. Fig. 1 illustrates the general structure of a rule r of the form (1) as a body-atom-graph (left), 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 colour 1, 2, and 3 are represented by empty circles, filled circles, and empty squares, respectively. Fig. 2 and 3 provide an examples.

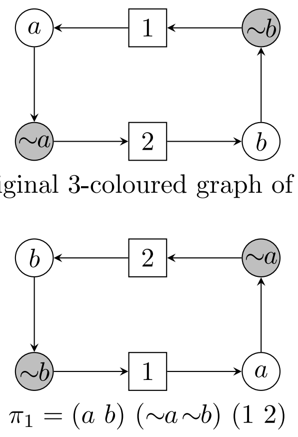


Fig. 2. 3-coloured graph constructions and resulting symmetries for the example logic programs P_1.

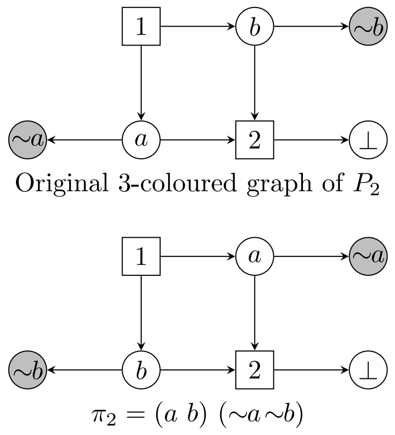


Fig. 3. 3-coloured graph construction and resulting symmetry for the example logic programs P_2.

Theorem 3.1. *The symmetries of a logic program correspond one-to-one to the symmetries of its 3-coloured graph encoding.*

Proof. ($\Rightarrow$) We begin by showing that any symmetry of a logic program corresponds to a symmetry of the constructed 3-coloured graph. Such a graph symmetry will map vertices of the same colour and edges to edges. In particular, if a maps to b, then $\sim a$ maps to $\sim b$, and the edge $(a, \sim a)$ maps to the edge $(b, \sim b)$. Since a and b, and $\sim a$ and $\sim b$, have the same colour, the symmetry is preserved. The same can be said about the other edges between vertices of different colours: In a logic program, a and b might also be connected with one or more body vertices. These connections would also be swapped at the respective vertices. Again, only vertices of the same colour are mapped one

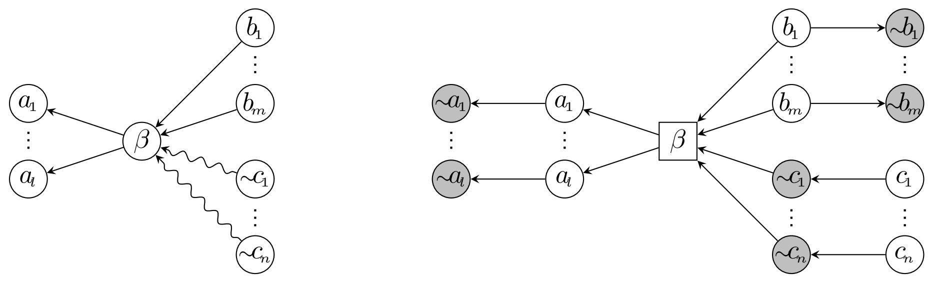


Fig. 1. General structure of a rule as a body-atom-graph and its 3-coloured graph encoding.

to another. Thus, a consistent mapping of atoms in the program, when carried over to the graph, must preserve the colours of the vertices.

($\Leftarrow$) We now show that every symmetry in the graph corresponds to a symmetry of the logic program. It is not hard to see because we use one colour for positive literals, one for negative literals, and one for bodies. Hence, a graph symmetry must map (1) positive literal vertices to other such, and negative literal vertices to negative literal vertices, and body vertices to body vertices, and (2) the body edges of a vertex to body edges of its mate. This is consistent with symmetries of the logic program mapping atoms to atoms, and bodies to bodies, i.e., rules to rules. To prove Boolean consistency, i.e., if a maps to b then $\sim a$ maps to $\sim b$, we recall that every edge from a vertex of colour 2 to a vertex of colour 1 is a Boolean consistency edge of the form $(a, \sim a)$. Since every such edge can only map to another such edge, a mapping a to b leaves no choice for $(a, \sim a)$ but to map to $(b, \sim b)$ because $(b, \sim b)$ is the only edge that connects b to another vertex of the same colour as $\sim a$. □

Theorem 3.2. *The symmetry groups of the logic program and its 3-coloured graph encoding are isomorphic.*

Proof Sketch. The proof consists of the straightforward verification that the one-to-one mapping constructed in the proof of Theorem 3.1 is a homomorphism. □

Corollary 3.3. *Sets of symmetry generators of the 3-coloured graph encoding correspond one-to-one to sets of symmetry generators of the logic program.*

Proof. By Theorem 2.3 and Theorem 3.2. □

Since GAP 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 symmetries. 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. Still, unintended mappings between 1-literal body edges and consistency edges remain impossible, since consistency edges connect positive literal vertices to their negative mates. For the special case of a 1-literal body and an empty head, we connect the literal vertex to the special node '$\perp$'.

We extend our graph encoding to integrity constraints, choice rules and cardinality constraints. No changes are necessary to cover integrity constraints. Also, the structure of a choice rule is encoded like a rule, i.e, is represented by a body vertex, a set of directed edges that connect the vertices of the literals that appear in the choice rule's body to its body vertex, and a set of directed edges that connect the body vertex to the vertices of the literals that appear in the head of the rule. To distinguish choice rules from rules a new colour 5 is introduced for their body vertices. An extension to cardinality constraints of the form (4) has to consider the bound k. Hence, we colour its body vertex by $k + 5$ to ensure that the literals of two cardinality constraints can be mated only if their bound is equal. Furthermore, each cardinality constraints is represented by a set of directed edges that connect the vertices of the lit-

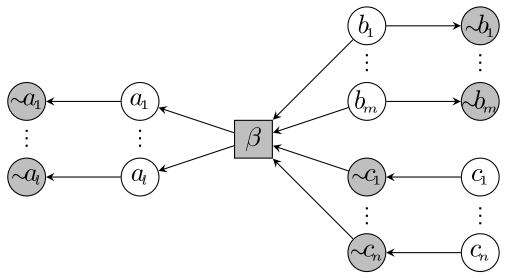


Fig. 4. The general structure of the coloured graph construction of a choice rule.

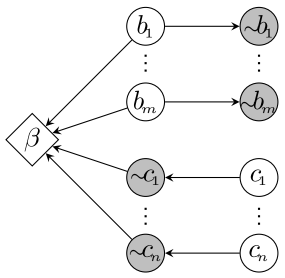


Fig. 5. The general structure of the coloured graph construction of a cardinality constraint.

erals $b_1, \ldots, b_m, \sim c_1, \ldots, \sim c_n$, that appear in its body, to its body vertex. Fig. 4 illustrates the general structure of a coloured graph construction of a choice rule of the form (3). Vertices of colour 1, 2, and 5 are represented by empty circles, filled circles, and filled squares, respectively. Fig. 5 shows the general structure of a coloured graph construction of a cardinality constraint of the form (4). Vertices of colour 1, 2, and $k+5$ are represented by empty circles, filled circles, and empty diamonds, respectively.

4. Symmetry Breaking

Recall that a symmetry π of a logic program P defines equivalence classes on the atoms in P (orbits). This naturally extends to truth assignments. Hence, symmetries induces equivalence classes in the solution space of a problem: 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 (c.f. [13,3,2]). A *lex-leader symmetry-breaking constraint* (LL-SBC) is an SBC that is satisfied only on the lex-leaders of orbits.

We will assume a total ordering on the atoms $a_1, a_2, \ldots, a_n$ of a logic program's alphabet $\mathcal{A}$ and consider the induced lexicographic ordering on the truth assignments, i.e., their interpretation as unsigned integers. The construction of a lex-leader SBC is accomplished by encoding a *permutation constraint* (PC) for every permutation π, denoted PC(π), given through:

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

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.

$$\text{LL-SBC}(\Pi) = \bigwedge_{\pi \in \Pi} \text{PC}(\pi)$$

Through chaining, which typically introduces additional atoms, we achieve a PC representation that is linear in the number of atoms [2], as follows, where $1 < i \le n$:

$$\begin{array}{ll} \text{PC}(\pi) & = (a_1 \le a_1^{\pi}) \wedge \neg c_{\pi,2} \\ \neg c_{\pi,i} & \leftrightarrow ((a_{i-1} \ge a_{i-1}^{\pi}) \rightarrow (a_i \le a_i^{\pi}) \wedge \neg c_{\pi,i+1}) \\ \neg c_{\pi,n+1} & \leftrightarrow \bot \end{array}$$

Theorem 4.1 (Crawford et al. [13])**.** *For a group of symmetries* Π*, the truth assignments that satisfy LL-SBC*(Π) *are the lexicographically smallest representatives from each class of truth assignments that can be mapped to each others by elements from* Π.

Finally, we encode above permutation constraint in a set of rules, denoted $P(\pi)$, that are satisfied for the lex-leader of the orbit induced by π as follows, where $1 < i \le n$:

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

Note that new atoms are introduced, thus extending the alphabet of P. Correctness is provided by the following theorem.

Theorem 4.2. *Let π be a symmetry of a logic program P. An answer sets of P satisfies PC(π) iff it satisfies the conditions expressed in $P(\pi)$.*

Proof. We prove in in two steps. First, we show that adding $P(\pi)$ to P does not impose conditions to an answer set other than Comp(P). Second, we show via induction on the structure of $P(\pi)$ that Comp($P(\pi)$) equals the formula $\top \leftrightarrow \mathrm{PC}(\pi)$.

To start with, since $P(\pi)$ is tight and $c_{\pi,i} \notin$ *atom*(P) for all $1 \leq i \leq n$, adding $P(\pi)$ to P does not introduce new loops that are not singletons. Hence, the conditions to an answer set imposed by $P(\pi)$ are fully captured by Comp(P). From rules 1–2, i.e., rules in $P(\pi)$ with empty head, we get

$$\begin{aligned} &\bot \leftarrow a_1 \wedge \neg {a_1}^\pi \\ &\bot \leftarrow c_{\pi,2} \end{aligned}$$

are in $\mathrm{RF}_{P(\pi)}$, and

$$\bot \rightarrow a_1 \wedge \neg {a_1}^\pi \vee c_{\pi,2}$$

is the only formula in $\mathrm{LF}_{P(\pi)}(\{\bot\})$. The conjunction of all three formulas is equivalent to

$$\top \leftrightarrow (\neg a_1 \vee {a_1}^\pi) \wedge \neg c_{\pi,2}.$$

From rules 3–6, i.e., rules in $P(\pi)$ with head $c_{\pi,i}$ for $2 \leq i \leq n$, we get

$$\begin{aligned} c_{\pi,i} &\leftarrow a_{i-1} \wedge a_i \wedge \neg {a_i}^\pi \\ c_{\pi,i} &\leftarrow \neg {a_{i-1}}^\pi \wedge a_i \wedge \neg {a_i}^\pi \\ c_{\pi,i} &\leftarrow a_{i-1} \wedge c_{\pi,i+1} \\ c_{\pi,i} &\leftarrow \neg {a_{i-1}}^\pi \wedge c_{\pi,i+1} \end{aligned}$$

are in $\mathrm{RF}_{P(\pi)}$, and

$$\begin{aligned} c_{\pi,i} \rightarrow\ & a_{i-1} \wedge a_i \wedge \neg {a_i}^\pi \vee \neg {a_{i-1}}^\pi \wedge a_i \wedge \neg {a_i}^\pi \\ & \vee a_{i-1} \wedge c_{\pi,i+1} \vee \neg {a_{i-1}}^\pi \wedge c_{\pi,i+1} \end{aligned}$$

is the formulas in $\mathrm{LF}_{P(\pi)}(\{c_{\pi,i}\})$. The conjunction of above formulas is equivalent to

$$\begin{aligned} \neg c_{\pi,i} \leftrightarrow ((\neg a_{i-1} \vee a^\pi_{i-1}) \rightarrow \\ (\neg a_i \vee a^\pi_i) \wedge \neg c_{\pi,i+1}). \end{aligned}$$

Finally, for the last rule, that is, the single rule in $P(\pi)$ with head $c_{\pi,n+1}$, we get

$$c_{\pi,n+1} \leftarrow \top$$

is in $\mathrm{RF}_{P(\pi)}$, and

$$c_{\pi,n+1} \rightarrow \top$$

is the only formula in $\mathrm{LF}_{P(\pi)}(\{c_{\pi,n+1}\})$. The conjunction of above formulas is equivalent to

$$\neg c_{\pi,n+1} \leftrightarrow \bot.$$

There are no further rules in $P(\pi)$. Substitution of the clauses of the form $\neg a \vee b$ by $a \leq b$ and conjoining all formulas results in $\top \leftrightarrow \mathrm{PC}(\pi)$. $\square$

A careful analysis reveals some possibilities to reduce the size of permutation constraints. The first corresponds to atoms that are mapped to themselves by the permutation, i.e., $a^\pi = a$. 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. Second, also for atoms a and b such that both appear in P as facts, and $a^\pi = b$, the consequent $a \leq a^\pi$ is satisfied.

A third possibility corresponds to the lexicographically largest atom in each cycle of π. Assume a cycle $(a_s \dots a_e)$ on the atoms of some index set $\{a, \dots, e\}$. 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 last atom in each cycle.

Example 4.1. *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 (Fig. 2 and 3) map a to b and vice versa, they share the same LL-SBC which is as simple as follows, assuming a is lexicographically smaller than b:*

$$\leftarrow a, \sim b$$

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 $\{b\}$ as the representative of the set of all answer sets symmetric to $\{b\}$, hence, eliminating the answer set $\{a\}$.

5. 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 [13]. We explore partial SBCs, i.e., we do not require that SBCs are satisfied by lex-leading assignments only (but we still require that all lex-leaders satisfy SBCs). Irredundant generators are good candidates because they cannot 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, e.g., different generating sets of the group of a logic program's symmetries may lead to different pruning [34].

Example 5.1. *Consider a logic program P with interchangeable atoms a_1, a_2, a_3, a_4, for instance*

$$\begin{aligned} \{a_1, a_2, a_3, a_4\} &\leftarrow \\ &\leftarrow a_1, a_2, a_3, a_4 \end{aligned}$$

An irredundant generating set for $\Pi(P)$ is the pair swap $(a_1\ a_2)$ and the rotation $(a_1\ a_2\ a_3\ a_4)$. To break the symmetry $(a_1\ a_2)$ we post the permutation constraint

$$\leftarrow a_1, \sim a_2$$

To break the symmetry $(a_1\ a_2\ a_3\ a_4)$ we post

$$\begin{array}{llll} & \leftarrow a_1, \sim a_2 & & \\ & \leftarrow c_0 & & \\ c_0 & \leftarrow a_1, a_2, \sim a_3 & c_1 & \leftarrow a_2, a_3, \sim a_4 \\ c_0 & \leftarrow a_1, c_1 & c_1 & \leftarrow a_2, c_2 \\ c_0 & \leftarrow \sim a_2, c_1 & c_1 & \leftarrow \sim a_3, c_2 \\ c_2 & \leftarrow & & \end{array}$$

However, these two permutation constraints do not eliminate all symmetries. For instance, they permit both answer sets $\{a_2, a_4\}$ and its symmetry $\{a_3, a_4\}$. There is an alternative irredundant generating set which breaks all symmetries, that is $\{(a_1\ a_2), (a_2\ a_3), (a_3\ a_4)\}$. We can break these three symmetries with

$$\begin{aligned} &\leftarrow a_1, \sim a_2 \\ &\leftarrow a_2, \sim a_3 \\ &\leftarrow a_3, \sim a_4 \end{aligned}$$

eliminating all symmetries of P.

We can further relax symmetry breaking to k supports from each permutation [2]. For $k \leq n$ and a_i is a support of permutation π, we define the *partial* permutation constraint:

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

By restricting the construction of permutation constraints this way, we further reduce the size of partial SBC.

Example 5.2. *Consider the* all-interval series *problem encoded from Example 2.2 and the generators π_2 and π_3 from Example 2.6. The symmetry-breaking constraint, where both permutation constraints are restricted to the second support, is given through the following, where $c_0, \ldots, c_3$ are new atoms.*

$$\begin{aligned} &\leftarrow v_{1,0}, \sim v_{1,n-1} \\ &\leftarrow c_0 \\ c_0 &\leftarrow v_{1,0}, v_{1,1}, \sim v_{1,n-2} \\ c_0 &\leftarrow v_{1,1}, \sim v_{1,n-1}, \sim v_{1,n-2} \\ c_0 &\leftarrow v_{1,0}, c_1 \\ c_0 &\leftarrow c_1, \sim v_{1,n-1} \\ c_1 &\leftarrow \end{aligned}$$

$$\begin{aligned} &\leftarrow v_{1,0}, \sim v_{n,0} \\ &\leftarrow c_2 \\ c_2 &\leftarrow v_{1,0}, v_{1,1}, \sim v_{n,1} \\ c_2 &\leftarrow v_{1,1}, \sim v_{n,0}, \sim v_{n,1} \\ c_2 &\leftarrow v_{1,0}, c_3 \\ c_2 &\leftarrow c_3, \sim v_{n,0} \\ c_3 &\leftarrow \end{aligned}$$

6. The sbass System

Our approach to symmetry-breaking answer set solving has been implemented within the preprocessor SBASS, available at [46]. The global architecture of SBASS is shown in Fig. 6. It accepts a logic program P in SMODELS format [52] produced by a grounder, e.g. LPARSE, available at [54], and GRINGO, available at [47]. A first component, the Program Reader, takes care of creating an internal representation and encodes symmetry de-

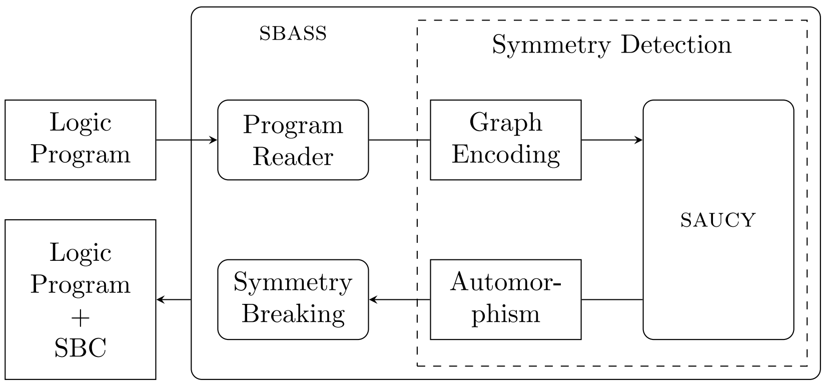


Fig. 6. Global architecture of SBASS.

tection as a graph automorphism problem. Notably, the Program Reader also checks for duplicate edges in the graph encoding of P which, otherwise, defect further processing. The actual search for an irredundant generating set of the group of symmetries of P, taking P's graph encoding as input, is performed by the graph automorphism program SAUCY (2.1), available at [53], which is incorporated into SBASS. SAUCY sequentially returns graph symmetry generators as soon as they are detected. Each such symmetry is used to construct a PC, all of which result in an SBC. In turn, SBASS prints P together with symmetry-breaking constraints, again in SMODELS format, which can be applied to any suitable answer set solver, e.g. SMODELS, available at [54], and CLASP, available at [47]. Note that SBASS provides several options, for instance, to print detected generators in cycle notation or statistics.

7. Experiments

To evaluate our approach, we conducted experiments on ASP encodings of several difficult combinatorial search problems. We use GRINGO (2.0.5) to generate our proposed encodings. Since our encodings are disjunctive, but tight, we make use of *shifting* [27] to provide an adequate encoding for the ASP solver CLASP, that are *normal* logic programs and its extensions. 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 explore the impact of partial PC, we restrict the construction of permutation constraints to k supports per permutation, denoted as CLASP^{π}_{k}, using SBASS' option `-size=k`.

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 allows careful conclusions to be drawn with respect to the size of the search space implicitly pruned through symmetry breaking. In the following experiments 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 exponentially hard for resolution based method [55], but is tractable using symmetries (all the pigeons are interchangeable and all the holes are interchangeable).

We chose a disjunctive encoding for the *pigeon hole* problem, where p_{ij} is taken to mean that a pigeon (indexed $1 \leq i \leq n$) is assigned a hole (indexed $1 \leq k < n$):

$$p_{i,1}; p_{i,2}; \ldots; p_{i,n-1} \leftarrow$$
$$\leftarrow p_{i,k}, p_{j,k} \qquad i < j$$

The runtimes for various sizes of n are shown in Table 1. Although symmetry breaking has a positive impact, the runtime even with full PC is still exponentially grows with the number of pigeons. Here, symmetry breaking on the generating set returned by SAUCY does not break all problem symmetries. At this point, we should note that a given problem can be encoded in many *equivalent* logic

Table 1
Runtime results in seconds for *pigeon hole* problems using the disjunctive encoding.

$\#n$	#gen.	SBASS	CLASP$^\pi_1$	CLASP$^\pi_5$	CLASP$^\pi$	CLASP
11	18	0.05	0.38	0.15	0.06	0.62
12	20	0.08	4.09	0.07	0.22	5.99
13	22	0.11	30.57	0.43	0.32	53.39
14	24	0.16	272.72	4.95	1.73	448.98
15	26	0.23	—	62.61	3.02	—
16	28	0.32	—	—	23.01	—
17	30	0.44	—	—	130.87	—

Table 2
Runtime results in seconds for *pigeon hole* problems using the support encoding.

$\#n$	#gen.	SBASS	CLASP$^\pi_1$	CLASP$^\pi_5$	CLASP$^\pi$	CLASP
11	19	0.03	8.70	0.02	0.02	47.28
12	21	0.04	66.57	0.03	0.03	397.01
13	23	0.07	540.26	0.09	0.03	—
14	25	0.08	—	0.77	0.04	—
15	27	0.12	—	5.91	0.05	—
16	29	0.17	—	47.98	0.06	—
17	31	0.22	—	520.39	0.13	—

programs [41,19], and with each different encoding our techniques may detect a different generating set. Therefore, we also tried an encoding of the *pigeon hole* problem based on the support encoding [17]:

$$\begin{aligned}
&\{p_{i,1}, \ldots, p_{i,n-1}\} \leftarrow \\
&\qquad \leftarrow {\sim}p_{i,1}, \ldots, {\sim}p_{i,n-1} \\
&\qquad \leftarrow 2\{p_{i,1}, \ldots, p_{i,n-1}\} \\
&\qquad \leftarrow p_{i,k}, p_{j,k} \qquad\qquad i < j
\end{aligned}$$

This caused SAUCY to compute a different, obviously better set of generators, which consequently breaks all symmetry resulting in a polynomial runtime. (Observe the change in the number of generators.) As can be seen in Table 2, full PCs are essential to tackle the *pigeon hole* problem.

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 colour the edges of the clique with n vertices, K_n, using two colours, say blue and red, there is a sub-clique with k vertices of colour blue or a sub-clique with m nodes of colour red. Symmetries in *Ramsey's Theorem* are between the colours and the vertices in the sub-clique. *Ramsey's Theorem* is discussed in many articles (see, for instance, [29]) and can be found in [4] and [14].

We use the encoding proposed by Leone et al. in [39], denoted as $R(k, m, n)$, to determine whether n is not an integer for which the theorem holds. The problem $R(3, 5, n)$ is encoded as follows, introducing propositional variables $b_{i,j}$ and $r_{i,j}$ representing the colouring (either blue or red) of the edge between nodes i and j, where $i, j, k, \ell, m \in 1..n$ and $i < j < k < \ell < m$:

$$\begin{aligned}
&b_{i,j}; r_{i,j} \leftarrow \\
&\qquad \leftarrow r_{i,j}, r_{i,k}, r_{j,k} \\
&\qquad \leftarrow b_{i,j}, b_{i,k}, b_{j,k}, \\
&\qquad\quad\; b_{i,\ell}, b_{j,\ell}, b_{k,\ell}, \\
&\qquad\quad\; b_{i,m}, b_{j,m}, b_{k,m}, b_{\ell,m}
\end{aligned}$$

Intuitively, the disjunctive rule guesses a colour for each edge. The first integrity constraint eliminates the colourings containing a red clique with 3 vertices, and the second integrity constraint eliminates the colourings containing a blue clique with 5 vertices.

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 3 suggest full PCs for unsatisfiable instances, but small, partial PCs for satisfiable instances.

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 $\{0, 1, \ldots, |E|\}$ 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 constraint satisfaction problem in [45], and is an interesting application for symmetry-breaking answer set solving because the symmetries are different for each instance and cannot be modelled a-priori in general.

As in previous experimental studies of symmetry breaking [45], our experiments consider

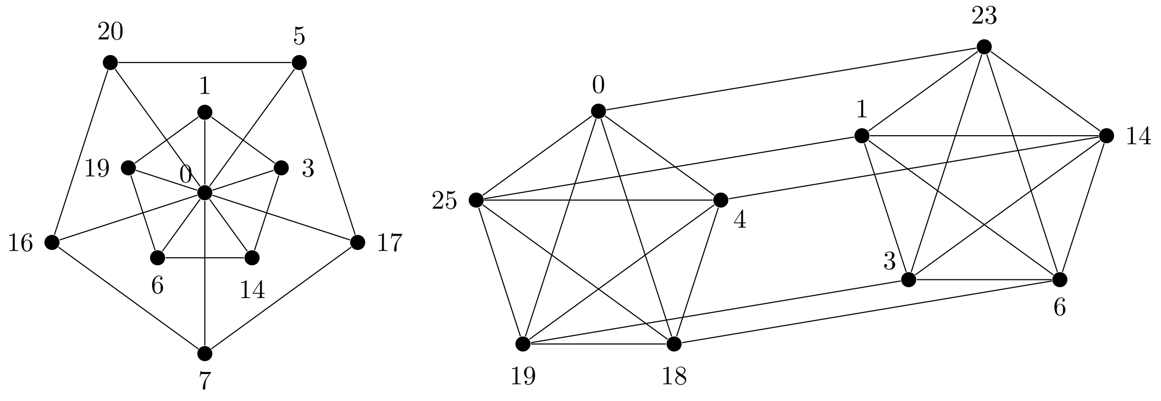


Fig. 7. A graceful labelling of the double wheel graph DW_5 (left) and the graph K_5P_2 (right).

Table 3

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

	SBASS	CLASP$^\pi_1$	CLASP$^\pi_5$	CLASP$^\pi$	CLASP
$R(3,5,13)$	0.06	0.01	0.01	0.03	0.01
$R(3,5,14)^*$	0.10	3.58	1.23	0.49	354.25
$R(3,6,17)$	1.18	0.12	0.12	0.14	0.11
$R(3,6,18)^*$	1.87	—(5)	—(5)	—(5)	—(5)
$R(4,4,17)$	0.26	0.73	0.12	0.50	0.07
$R(4,4,18)^*$	0.37	—(5)	—(5)	—(5)	—(5)
$R(4,5,23)$	5.43	4.23	2.29	2.05	1.32
$R(4,5,24)$	7.15	77.64	208.66(1)	180.96(3)	—(5)
$R(4,5,25)^*$	9.54	—(5)	—(5)	—(5)	—(5)

Table 4

Average time for completed runs in seconds and the number of timeouts on *graceful graph* instances, each shuffled 5 times. Timeouts, if any, are given in parenthesis. The *asterisk denotes instances that have no answer sets.

	SBASS	CLASP$^\pi_1$	CLASP$^\pi_5$	CLASP$^\pi$	CLASP
$DW_3{}^*$	0.02	4.24	1.45	1.32	5.40
DW_6	0.17	0.46	0.56	1.09	0.57
DW_8	0.48	28.81	5.47	17.11	4.30
DW_{10}	1.21	191.86	66.18	61.59	27.04(2)
DW_{12}	3.34	145.89	202.18(1)	111.96(1)	112.38(4)
K_3P_3	0.04	0.08	0.08	0.07	0.08
K_4P_2	0.07	0.20	0.10	0.54	0.19
K_4P_3	0.29	24.68	29.06	198.57	24.01
K_5P_2	0.37	274.85(3)	334.55(3)	312.56(1)	226.03(3)

graphs DW_n and K_nP_m (Fig. 7). 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 m 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 4, we achieve speed-up on the unsatisfiable instance DW_3. For the other instances, all of which are satisfiable, no complete traversal of the search space is necessary, and the branching heuristic used in our approach sometimes appears to be misled by the extra variables introduced in CLASP$^\pi_k$. That explains some of the variability in the runtimes. However, we still observe a substantial impact of our symmetry breaking techniques on the difficult instances.

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, considering SBASS' runtime.

7.4. Answer Set Enumeration

Finally, we want to test the impact of symmetry breaking on the number of answer sets. Our study considers instances from the *all-interval series* problem and *graceful graphs*. Recall, 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. It has been proposed as a benchmark domain for CP systems by Hoos in [32] and is part of the CSPlib [14]. We modelled the *all-interval series* problem (*AllInt*) as previously described in Example 2.2, using a direct representation for n integer variables and auxiliary variables to represent the differences between adjacent numbers, and required both sets of variables to be all-different.

As one might expect, we can observe that symmetry breaking significantly compresses the solution-space (see Table 5), and therefore, reduces the time necessary for post-processing solutions. Clearly, CLASP$^{\pi}_{k}$ discards more solutions (eliminating up to 90 per cent of the solution space) for an increasing number k.

Recall 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 [23]. 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.

8. Conclusions

Our work addresses solving combinatorial problems in ASP whose difficulty arise from symmetries and redundant search caused by them. We have shown a reduction of symmetry detection to a graph automorphism problem which allows us to extract symmetries of a logic program from the symmetries of the constructed coloured graph. Our techniques are formulated as a completely automated flow that (1) starts with a logic program, (2) detects all of its symmetries within a very general class, including all permutations that do not change the logic program, (3) represents all symmetries implicitly and always with exponential compression in terms of irredundant group generators, and (4) constructs a linear-sized symmetry-breaking constraint that does not affect existence of answer sets. This flow does not require source code modifications in ASP solvers. We successfully validated our implementation with CLASP and SMODELS (SMODELS results are not included in this paper). Experiments indicate that breaking just the symmetries in a generating set is an efficient and effective way to deal with large numbers of symmetries. In many cases, our techniques achieved significant pruning of the search space and yield solutions to problems which are otherwise intractable. We also observed a significant compression of the solution space which makes symmetry breaking attractive whenever all answer sets have to be post-processed.

Our techniques can be easily extended to constraint answer set programming using our translation based approach [17], where a constraint logic program is decomposed into a logic program under answer set semantics. Then generic symmetry detection and symmetry breaking can be applied.

However, we stress that the proposed flow may not be useful on ASP instances that are easy, or do not have symmetries. Many ASP benchmarks in [4] 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 (see, for instance, [56]). This is target of future work.

References

[1] L. Aiello and F. Massacci. Verifying security protocols as planning in logic programming. *ACM Transactions on Computational Logic*, 2(4):542–580, 2001.

[2] F. A. Aloul, I. L. Markov, and K. A. Sakallah. Shatter: efficient symmetry-breaking for Boolean satisfiability. In *Proceedings of DAC'03*, pages 836–839. ACM, 2003.

[3] F. A. Aloul, A. Ramani, I. L. Markov, and K. A. Sakallah. Solving difficult instances of Boolean satisfiability in the presence of symmetry. *Technical Report CSE-TR-463-02, University of Michigan*, 2002.

[4] http://asparagus.cs.uni-potsdam.de/.

[5] L. Babai. Automorphism groups, isomorphism, reconstruction. In R. L. Graham, M. Grötschel, and L. Lovász, editors, *Handbook of Combinatorics*, volume 2, pages 1447–1540. Elsevier, 1995.

[6] C. Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press, 2003.

[7] C. Baral, K. Chancellor, N. Tran, N. Tran, A. Joy, and M. Berens. A knowledge based approach for representing and reasoning about signaling networks. In *Proceedings of ISMB/ECCB'04*, pages 15–22, 2004.

Table 5

Results on computing all answer sets of selected instances.
Runtime and number of solutions are shown.

		SBASS	CLASP$^{\pi}_{1}$		CLASP$^{\pi}_{5}$		CLASP$^{\pi}$		CLASP		
	#gen.	time	time	#sol.	time	#sol.	time	#sol.	time	#sol.	compression
$AllInt_8$	2	0.01	0.15	39	0.11	15	0.17	14	0.14	40	65%
$AllInt_9$	2	0.01	0.78	119	0.60	60	0.93	40	0.77	120	67%
$AllInt_{10}$	2	0.01	4.60	295	3.43	148	5.69	107	4.08	296	64%
$AllInt_{11}$	2	0.01	23.26	647	22.82	372	32.70	238	24.40	648	63%
$AllInt_{12}$	2	0.01	161.90	1327	147.17	862	211.27	442	160.32	1328	67%
DW_4	5	0.07	282.36	9472	168.03	5152	85.65	1150	314.15	11264	90%
K_3P_3	3	0.05	229.15	5704	119.99	2836	126.25	1487	268.80	6816	76%
K_4P_2	4	0.08	119.66	1080	67.96	552	27.72	146	145.13	1440	90%

[8] R. Ben-Eliyahu and R. Dechter. Propositional semantics for disjunctive logic programs. *Annals of Mathematics and Artificial Intelligence*, 12(1-2):53–87, 1994.

[9] A. Biere, M. Heule, H. van Maaren, and T. Walsh, editors. *Handbook of Satisfiability.* IOS Press, 2009.

[10] G. Brewka and T. Eiter. Preferred answer sets for extended logic programs. In *Proceedings of KR'96*, pages 86–97. Morgan Kaufmann Publishers, 1996.

[11] K. Clark. Negation as failure. In *Logic and Data Bases*, pages 293–322. Plenum Press, 1978.

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

[13] J. Crawford, M. Ginsberg, E. Luks, and A. Roy. Symmetry-breaking predicates for search problems. In *Proceedings of KR'96*, pages 148–159. Morgan Kaufmann, 1996.

[14] http://http://www.csplib.org/.

[15] P. T. Darga, M. H. Liffiton, K. A. Sakallah, and I. L. Markov. Exploiting structure in symmetry detection for CNF. In *Proceedings of DAC'04*, pages 530–534. ACM Press, 2004.

[16] C. Drescher, M. Gebser, T. Grote, B. Kaufmann, A. König, M. Ostrowski, and T. Schaub. Conflict-driven disjunctive answer set solving. In *Proceedings of KR'08*, pages 422–432. AAAI Press, 2008.

[17] C. Drescher and T. Walsh. A translational approach to constraint answer set solving. *Theory and Practice of Logic Programming*, 10(4-6):465–480, 2010.

[18] T. Eiter, W. Faber, N. Leone, and G. Pfeifer. The diagnosis frontend of the dlv system. *AI Communications*, 12(1-2):99–111, 1999.

[19] T. Eiter and M. Fink. Uniform equivalence of logic programs under the stable model semantics. In *Proceedings of ICLP'03*, pages 224–238. Springer, 2003.

[20] T. Eiter, G. Ianni, T. Lukasiewicz, R. Schindlauer, and H. Tompits. Combining answer set programming with description logics for the semantic web. *Artificial Intelligence*, 172(12-13):1495–1539, 2008.

[21] E. Erdem and V. Lifschitz. Tight logic programs. *Theory and Practice of Logic Programming*, 3(4-5):499–518, 2003.

[22] E. Erdem and M. Wong. Rectilinear Steiner tree construction using answer set programming. In *Proceedings of ICLP'04*, pages 386–399. Springer, 2004.

[23] M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub. Advanced preprocessing for answer set solving. In *Proceedings of ECAI'08*, pages 15–19. IOS Press, 2008.

[24] M. Gebser, B. Kaufmann, and T. Schaub. The conflict-driven answer set solver clasp: Progress report. In *Proceedings of LPNMR'09*, pages 509–514. Springer, 2009.

[25] M. Gebser, J. Lee, and Y. Lierler. Elementary sets for logic programs. In *Proceedings of AAAI'06.* AAAI Press, 2006.

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

[27] M. Gelfond, V. Lifschitz, H. Przymusinska, and M. Truszczyński. Disjunctive defaults. In *Proceedings of KR'91*, pages 230–237. Morgan Kaufmann, 1991.

[28] I. P. Gent and T. Walsh. CSPLIB: A benchmark library for constraints. In *Proceedings of CP'99*, pages 480–481. Springer, 1999.

[29] R. L. Graham and B. L. Rothschild. Ramsey theory. *Studies in combinatorics*, 17:80–99, 1978.

[30] M. Hall. *Theory of Groups.* McMillan, 1959.

[31] K. Heljanko and I. Niemelä. Bounded LTL model checking with stable models. *Theory and Practice of Logic Programming*, 3(4-5):519–550, 2003.

[32] H. H. Hoos. *Stochastic Local Search - Methods, Models, Applications.* infix-Verlag, 1999.

[33] T. Junttila and P. Kaski. Engineering an efficient canonical labeling tool for large and sparse graphs. In *Proceedings of the Workshop on Algorithm Engineering and Experiments, ALENEX'07.* SIAM, 2007.

[34] G. Katsirelos, N. Narodytska, and T. Walsh. Breaking generator symmetry. In *Proceedings of Symcon'09*, 2009.

[35] B. Krishnamurthy. Short proofs for tricky formulas. *Acta Informatica*, 22(3):253–275, 1985.

[36] J. Lee. A model-theoretic counterpart of loop formulas. In *Proceedings of IJCAI'05*, pages 503–508. Professional Book Center, 2005.

[37] J. Lee and V. Lifschitz. Loop formulas for disjunctive logic programs. In *Proceedings of ICLP'03*, pages 451–465. Springer, 2003.

[38] N. Leone, G. Greco, G. Ianni, V. Lio, G. Terracina, T. Eiter, W. Faber, M. Fink, G. Gottlob, R. Rosati, D. Lembo, M. Lenzerini, M. Ruzzi, E. Kalka, B. Nowicki, and W. Staniszkis. The INFOMIX system for advanced integration of incomplete and inconsistent data. In *Proceedings of SIGMOD'05*, pages 915–917, 2005.

[39] N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, and F. Scarcello. The dlv system for knowledge representation and reasoning. *ACM Transactions on Computational Logic*, 7:499–562, 2002.

[40] V. Lifschitz. Answer set programming and plan generation. *Artificial Intelligence*, 138(1-2):39–54, 2002.

[41] V. Lifschitz, D. Pearce, and A. Valverde. Strongly equivalent logic programs. *ACM Transactions on Computational Logic*, 2(4):526–541, 2001.

[42] T. Linke. Suitable graphs for answer set programming. In *Proceedings of ASP'03*, pages 15–28, 2003.

[43] B. McKay. Practical graph isomorphism. In *Numerical mathematics and computing*, pages 45–87, 1981.

[44] M. Nogueira, M. Balduccini, M. Gelfond, R. Watson, and M. Barry. An A-prolog decision support system for the space shuttle. In *Proceedings of PADL'01*, pages 169–183. Springer, 2001.

[45] K. E. Petrie and B. M. Smith. Symmetry breaking in graceful graphs. In *Proceedings of CP'03*, pages 930–934. Springer, 2003.

[46] http://potassco.sourceforge.net/labs.html.

[47] http://potassco.sourceforge.net/.

[48] J.-F. Puget. On the satisfiability of symmetrical constrained satisfaction problems. In *Proceedings of ISMIS'93*, pages 350–361. Springer, 1993.

[49] http://www.satcompetition.org/.

[50] P. Simons, I. Niemelä, and T. Soininen. Extending and implementing the stable model semantics. *Artificial Intelligence*, 138(1-2):181–234, 2002.

[51] T. Soininen and I. Niemelä. Developing a declarative rule language for applications in product configuration. In *Proceedings of PADL'99*, pages 305–319. Springer, 1999.

[52] T. Syrjänen. Lparse 1.0 user's manual. http://www.tcs.hut.fi/Software/smodels/lparse.ps.gz.

[53] http://vlsicad.eecs.umich.edu/BK/SAUCY/.

[54] http://www.tcs.hut.fi/Software/smodels/.

[55] A. Urquhart. Hard examples for resolution. *Journal of the ACM*, 34(1):209–219, 1987.

[56] T. Walsh. Symmetry breaking using value precedence. In *Proceedings of ECAI'06*, pages 168–172. IOS Press, 2006.