BreakID: Static Symmetry Breaking for ASP (System Description)

Jo Devriendt[1] and Bart Bogaerts[2,1]

[1] KU Leuven, Department of Computer Science
Celestijnenlaan 200A, Leuven, Belgium
firstname.lastname@cs.kuleuven.be
[2] Helsinki Institute for Information Technology HIIT, Aalto University
FI-00076 AALTO, Finland

Abstract. Symmetry breaking has been proven to be an efficient preprocessing technique for satisfiability solving (SAT). In this paper, we port the state-of-the-art SAT symmetry breaker BreakID to answer set programming (ASP). The result is a lightweight tool that can be plugged in between the grounding and the solving phases that are common when modelling in ASP. We compare our tool with sbass, the current state-of-the-art symmetry breaker for ASP.

1 Introduction

Answer set programming (ASP) has always benefited from progress in the satisfiability solving (SAT) community. In fact, many (if not all) modern ASP solvers [17, 4, 12] are based on conflict-driven clause learning (CDCL) [25], a technique first developed for SAT. Some people even go as far as to claim that ASP is just "a good modelling language for SAT".

Thus, now and then it makes sense to check which developments in SAT can be transferred to ASP. In this paper, we port a recently introduced symmetry breaking preprocessor, BreakID[3] [15] to ASP. A first version of BreakID was developed in 2013 [14]. This version won a gold medal on the 2013 SAT competition [7] in combination with the SAT solver Glucose 2.1 [5]. The latest version of BreakID was developed in 2015 [15] and won the gold medal in the *no limit* track of the 2016 SAT competition [20] in combination with the SAT solver CoMiniSatPS [28].

Symmetry breaking has been around for quite a long time. The most well-known symmetry breaker for SAT is Shatter [2]. Shatter works by *(i)* transforming a SAT theory into a colored graph whose automorphisms correspond to symmetries of the theory, *(ii)* calling a graph automorphism detection tool such as Saucy [22] to obtain a set of generators of the symmetry group, and *(iii)* adding constraints that break each of the generators returned by the previous step. BreakID is based on the same ideas, but improves the last step of the symmetry breaking process. In particular, the set of symmetry breaking

[3] Pronounced "break it".

constraints is parametrized by both the chosen set of generator symmetries, as well as a chosen order on the set of propositional variables. Good choices for these two parameters strongly influence the breaking power and the size of the resulting constraints [15] . As such, BreakID

- searches for more structure in the symmetry breaking group, essentially detecting subgroups of the symmetry group that can be broken completely without additional overhead,
- reduces the size of symmetry breaking constraints,
- chooses an order on the variables that "matches" the generators to break,
- actively searches for small (size 2) symmetry breaking clauses.

This resulted in significant improvements with respect to Shatter.

The techniques from Shatter have been ported to ASP by the ASP symmetry breaker sbass [16]. The main contribution there was to develop an alternative for the first step in the ASP context, i.e., to transform an ASP program $\mathcal{P}$ into a colored graph such that graph isomorphisms correspond to symmetries of $\mathcal{P}$.

In this paper, we show how to use BreakID as a symmetry breaker for ASP. Our implementation extends (and slightly modifies) the graph encoding used in sbass. The extensions serve to support a richer language: we provide support for so-called *weight rules* and *minimize statements*. The modification serves to ensure more effective breaking; we present several simple symmetric examples where BreakID detects the symmetry while sbass does not. For the third step (generating symmetry breaking constraints), we use all of the improvements that are already present in the SAT-version of BreakID [15].

We evaluate performance of BreakID and sbass on a number of benchmarks with symmetries and conclude that also in an ASP context, BreakID's improvements have a significant effect.

The rest of this paper is structured as follows. In Section 2 we recall preliminaries regarding answer set programming and symmetries. Afterwards, in Section 3 we present our symmetry breaker for ASP. We evaluate this tool in Section 4 and conclude in Section 5.

2 Preliminaries

2.1 Answer Set Programming

A *vocabulary* is a set of symbols, also called *atoms*; vocabularies are denoted by σ, τ. A *literal* is an atom or its negation. An interpretation I of the vocabulary Σ is a subset of Σ. Propositional formulas and the satisfaction relation between formulas and interpretations are defined as usual. A *logic program* $\mathcal{P}$ over vocabulary Σ is a set of *rules* r of form

$$h_1 \vee \cdots \vee h_l \leftarrow a_1 \wedge \cdots \wedge a_n \wedge \neg b_1 \wedge \cdots \wedge \neg b_m, \tag{1}$$

where h_i's, a_i's, and b_i's are atoms in Σ. The formula $h_1 \vee \cdots \vee h_l$ is called the *head* of r, denoted $head(r)$, and the formula $a_1 \wedge \cdots \wedge a_n \wedge \neg b_1 \wedge \cdots \wedge \neg b_m$ the *body* of r, denoted $body(r)$. A program is called *normal* (resp. *positive*) if $l = 1$

(resp. $m = 0$) for all rules in $\mathcal{P}$. If $n = m = 0$, we simply write $h_1 \vee \cdots \vee h_l$. If $l = 0$, we call r a *constraint*.

An interpretation I is a *model* of a logic program $\mathcal{P}$ if, for all rules r in $\mathcal{P}$, whenever $body(r)$ is satisfied by I, so is $head(r)$. The *reduct* of $\mathcal{P}$ with respect to I, denoted $\mathcal{P}^I$, is a positive program that consists of rules $h_1 \vee \cdots \vee h_l \leftarrow a_1 \wedge \cdots \wedge a_n$ for all rules of the form (1) in $\mathcal{P}$ such that $b_i \notin I$ for all i. An interpretation I is a *stable model* or an *answer set* of $\mathcal{P}$ if it is a $\subseteq$-minimal model of $\mathcal{P}^I$ [19]. Checking whether a logic program has a stable model is an Σ_2^P-complete task in general and an NP-complete task for normal programs; hence, logic programs can be used to encode search problems. This observation gave birth to the field of answer set programming [24, 27, 23].

This paper sometimes uses syntactic sugar such as choice rules, constraints, cardinality atoms and weight atoms. A *cardinality atom*

$$g \leq \#\{l_1, \ldots, l_f\} \leq k$$

(with $l_1, \ldots, l_f$ being literals and $f, g, k \in \mathbb{N}$) is satisfied by I if

$$g \leq \#\{i \mid I \models l_i\} \leq k.$$

A *weight atom*

$$g \leq \text{sum}\{l_1 = w_i, \ldots, l_f = w_f\} \leq k$$

(with $l_1, \ldots, l_f$ being literals and $f, g, k, w_i \in \mathbb{N}$) is satisfied if

$$g \leq \sum_{\{i \mid I \models l_i\}} w_i \leq k.$$

Sometimes, the "$g \leq$" or "$\leq k$" parts of cardinality or weight atoms are dropped. In this case, the obvious semantics applies: the atom is satisfied for any lower/upper bound. For instance, the cardinality atom

$$g \leq \#\{l_1, \ldots, l_f\}$$

is satisfied whenever

$$g \leq \#\{i \mid I \models l_i\}.$$

A *choice rule* is a rule with a cardinality atom in the head, i.e., a rule of the form

$$g \leq \#\{l_1, \ldots, l_f\} \leq k \leftarrow a_1 \wedge \cdots \wedge a_n \wedge \neg b_1 \wedge \cdots \wedge \neg b_m.$$

I satisfies a constraint c if it does not satisfy $body(c)$. These language constructs can all be translated to normal rules [8].

2.2 Symmetry

Let π be a permutation of Σ. We extend π to literals: $\pi(\neg a) = \neg(\pi(a))$, to rules:

$$\begin{aligned}&\pi(h_1 \vee \cdots \vee h_l \leftarrow a_1 \wedge \cdots \wedge a_n \wedge \neg b_1 \wedge \cdots \wedge \neg b_m) = \\ &\quad \pi(h_1) \vee \cdots \vee \pi(h_l) \leftarrow \pi(a_1) \wedge \cdots \wedge \pi(a_n) \wedge \neg\pi(b_1) \wedge \cdots \wedge \neg\pi(b_m),\end{aligned}$$

to logic programs: $\pi(\mathcal{P}) = \{\pi(r) \mid r \in \mathcal{P}\}$, and to interpretations: $\pi(I) = \{\pi(p) \mid p \in I\}$. A *symmetry* of a program $\mathcal{P}$ is a permutation π of Σ that *preserves stable models* of $\mathcal{P}$ (i.e., $\pi(I)$ is a stable model of $\mathcal{P}$ if and only if I is a stable model of $\mathcal{P}$). A permutation of atoms for which $\pi(\mathcal{P}) = \mathcal{P}$ is a *syntactical* symmetry of $\mathcal{P}$.[4] Typically, only syntactical symmetry is exploited, since this type of symmetry can be detected with relative ease. The practical techniques presented in this paper are no exception.

Symmetry breaking aims at eliminating symmetry, either by *statically* posting a set of constraints that invalidate symmetric interpretations, or by altering the search space *dynamically* to avoid symmetric search paths. In this paper, we focus on static symmetry breaking. If Π is a symmetry group, then a set of constraints ψ is *sound* if for each interpretation I there exists at least one symmetry $\pi \in \Pi$ such that $\pi(I)$ satisfies ψ; ψ is *complete* if for each assignment α there exists at most one symmetry $\pi \in \Pi$ such that $\pi(\alpha)$ satisfies ψ [35].

2.3 Graph Automorphisms

A *colored graph* is a tuple $G = (V, E, c)$, where V is a set, whose elements we call *nodes*, E is a binary relation on V; elements of E are called edges and c is a mapping $V \to C$ for some set C. The elements of C are called *colors*.

An automorphism of G is a mapping $\pi : V \to V$ such that the following two conditions hold:

- $(u, v) \in E$ if and only if $(\pi(u), \pi(v)) \in E$ for each $u, v \in V$, and
- $c(v) = c(\pi(v))$ for each $v \in V$.

The graph automorphism problem is the task of finding all graph automorphisms of a given (colored) graph. The complexity of this problem is conjectured to be strictly in between P and NP [6]. Several tools are available to tackle this problem, including Saucy [22], nauty [26] and bliss [21].

3 Symmetry Breaking for ASP

As mentioned in the introduction, symmetry breaking typically consists of three steps.

- First, a colored graph G is created such that the automorphisms of G correspond to symmetries of $\mathcal{P}$.
- Second, a set $\mathcal{G}$ of generators of the automorphism group of G is derived.
- Third, a set of symmetry breaking constraints is added to the original problem, based on $\mathcal{G}$.

As for the second step, any graph automorphism tool can be used. Shatter [2], sbass and BreakID make use of Saucy for this task. We discuss how we tackle the first and last step in the following subsections.

[4] In some texts, the term *symmetry* is reserved for what we call *syntactical symmetry*.

3.1 Colored Graph Encoding

As for encoding a logic program as a colored graph, our approach is very close to the one introduced by SBASS [16] (we discuss and justify the differences below). We use four colors $\{1, \ldots, 4\}$; our graph consists of the following nodes:

- For each atom $p \in \Sigma$, two nodes, referred to as p and $\neg p$ below. Node p is colored as 1, node $\neg p$ is colored 2.
- For each rule $r \in \mathcal{P}$, two nodes, referred to as $\mathit{head}(r)$ and $\mathit{body}(r)$ below. Node $\mathit{head}(r)$ is colored 3 and node $\mathit{body}(r)$ is colored 4.

Our graph is *undirected* and the edges are as follows:

- Each node p is connected to $\neg p$.
- For each rule r of the form (1) in $\mathcal{P}$, each node h_i is connected to $\mathit{head}(r)$, each node a_i is connected to $\mathit{body}(r)$ and each node $\neg b_i$ is connected to $\mathit{body}(r)$. The complete encoding of this type of rule is illustrated in Figure 1.

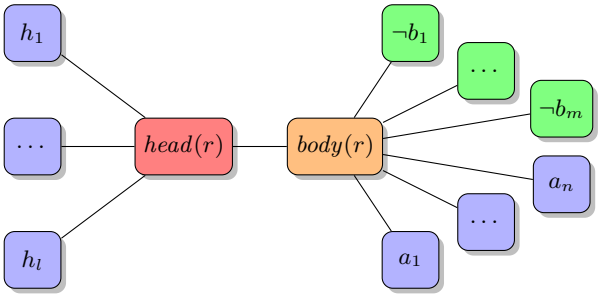

Fig. 1. Encoding of a rule r of the form (1).

It can be seen that there is a one-to-one correspondence between automorphisms of this graph and syntactic symmetries of the logic program. Since atoms p are the only nodes colored in color 1, an automorphism induces a permutation of Σ. The edge between p and $\neg p$ guarantee that an automorphism that maps to p to q also maps $\neg p$ to $\neg q$. Furthermore, the edges between $\mathit{head}(r), \mathit{body}(r)$ and the various literals occurring in a rule capture the full structure of the rule. As such, it can be verified that automorphisms of this graph must map rules to "syntactically symmetric" rules. Vice versa, each syntactic symmetry preserves the graph structure defined above.

Additionally, we extend this graph encoding to capture more language extensions such as, e.g., *choice rules*. We only support limited forms of these rules. Actually, our tool supports *exactly* the LPARSE-SMODELS intermediate format [32], extended with support for disjuctive rules. This means for instance that in cardinality rules, we assume there is no upper. The general language can easily be transformed to this format (in fact, that is what all modern ASP grounders do). For this, we introduce two new colors, extending the set of colors to $\{1, \ldots, 6\}$. In programs with these extensions, integer numbers can occur, either as bounds of

cardinality or weight constraint or as weights in a weight constraint or minimize statement (see below). We assume that for each integer n that occurs in such a program, there is a unique color $c_n \notin \{1, \ldots, 6\}$ available and extend our set of colors to

$$\{1, \ldots, 6\} \cup \{c_n \mid n \text{ occurs as weight or bound in } \mathcal{P}\}.$$

Cardinality rules Rules of the form

$$h \leftarrow g \leq \#\{a_1, \ldots, a_n, \neg b_1, \ldots, \neg b_m\}$$

are encoded as follows. The head of the rule is encoded as usual. The body node of the rule is colored in c_g, the color associated with the bound g. The body node is connected to each of the a_i and $\neg b_i$ and to the head node, as usual.

Choice rules Rules of the form

$$\#\{h_1, \ldots, h_l\} \leftarrow a_1 \wedge \cdots \wedge a_n \wedge \neg b_1 \wedge \cdots \wedge \neg b_m$$

are encoded exactly the same as rules of form (1), except that $\mathit{head}(r)$ is colored 5. This allows to differentiate between standard rules and choice rules.

Weight rules Rules of the form

$$h \leftarrow g \leq \text{sum}\{a_1 = w_1, \ldots, a_n = w_n, \neg b_1 = w_{n+1}, \ldots, \neg b_m = w_{n+m}\} \quad (2)$$

are encoded as follows. The node $\mathit{body}(r)$ is colored as for cardinality rules in c_g. For each occurrence of an expression $l_i = w_j$, we create one additional node, referred to as $l_i = w_j$. This node is colored in c_{w_j}, the color associated to the integer w_j and is connected to l_i. The body of this rule is connected to all the nodes $a_i = w_i$ and $\neg b_i = w_{n+i}$. $\mathit{body}(r)$ is connected to $\mathit{head}(r)$ and $\mathit{head}(r)$ to h, as usual. A visualisation of this encoding can be found in Figure 2.

Minimize statements Minimize statements are expressions of the form

$$\text{minimize}\{a_1 = w_1, \ldots, a_n = w_n, \neg b_1 = w_{n+1}, \ldots, \neg b_m = w_{n+m}\} \quad (3)$$

They are directions to the solver that the user is only interested in models such that the term

$$\sum_{\{\text{rules of the form (3) that occur in } \mathcal{P}\ \}} \sum_{\{i \mid I \models l_i\}} w_i.$$

is minimal (among all stable models). Such a statement is encoded analogously to the body of a weight rule, except that the body node is replaced by a *minimize node* with color 6.

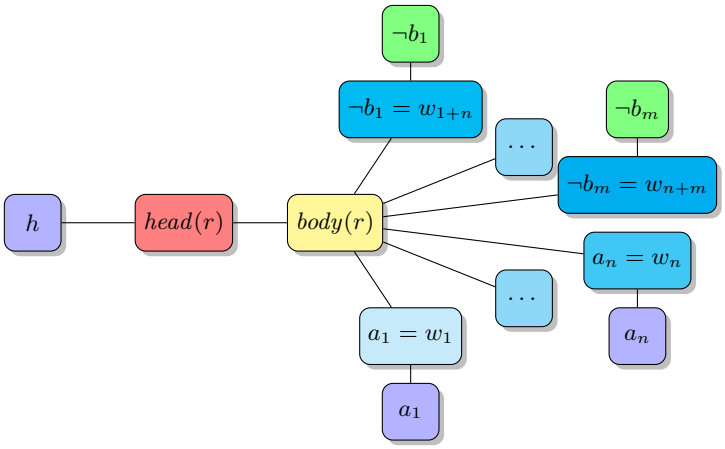

Fig. 2. Encoding of a rule r of the form (2). The color of $body(r)$ depends on the bound g. The color of each $l_j = w_i$ depends on the value w_i.

Comparison with SBASS Our graph encoding differs from the one used by SBASS in two respects.

First of all, we added support for *minimize statements* and *weight rules*. As a result, our tool supports the full LPARSE-SMODELS intermediate language as documented in [32] and additionally, the rule type "8" used by GRINGO [18] to represent disjunctive rules.

Second, we use *undirected graphs* whereas SBASS uses directed graphs. To be precise, when using directed graphs, all $head(r)$ nodes can be dropped by using edges from literals a_i and $\neg b_i$ to $body(r)$ and from $body(r)$ to h_i (the directionality thus distinguishes between head and body literals). We expect because of this that symmetry detection in BREAKID takes slightly more time than symmetry detection in SBASS (automorphism algorithms are sensitive to the number of nodes in the input graph). The reason why we did this is that we experimentally noticed that SAUCY does not always behave well with directed graphs. On very small examples already, symmetries are missed when using the directional encoding. SBASS, which uses SAUCY with directed graphs, shows the same kind of behaviour, as we show in the following list of examples. The examples illustrate that BREAKID is more stable than SBASS with respect to symmetry detection on different encodings.

Example 1.

$$\mathcal{P}_1 = \left\{ \begin{array}{l} 0 \leq \#\{p\} \leq 1. \\ 0 \leq \#\{q\} \leq 1. \end{array} \right\}.$$

It is clear that p and q are interchangeable in $\mathcal{P}_1$. By this we mean that the mapping $\sigma\colon \{p, q\} \to \{p, q\}\colon p \mapsto q, q \mapsto p$ is a symmetry of $\mathcal{P}_1$. BREAKID detects this, while SBASS detects (and breaks) no symmetry.

Example 2. Consider the logic program

$$\mathcal{P}_2 = \left\{ \begin{array}{l} r \leftarrow p \wedge q. \\ 0 \leq \#\{p\} \leq 1. \\ 0 \leq \#\{q\} \leq 1. \end{array} \right\}.$$

It is clear that p and q are interchangeable in $\mathcal{P}_2$. In this case, both sbass and BreakID detect (and break) this interchangeability.

Example 3. Consider the logic program

$$\mathcal{P}_3 = \left\{ \begin{array}{l} \leftarrow p \wedge q. \\ 0 \leq \#\{p\} \leq 1. \\ 0 \leq \#\{q\} \leq 1. \end{array} \right\}.$$

It is clear that p and q are interchangeable in $\mathcal{P}_3$. BreakID detects this, while sbass detects (and breaks) no symmetry.

Example 4. Consider the logic program

$$\mathcal{P}_4 = \left\{ \begin{array}{l} p \vee q \leftarrow p \wedge q. \\ 0 \leq \#\{p\} \leq 1. \\ 0 \leq \#\{q\} \leq 1. \end{array} \right\}.$$

It is clear that p and q are interchangeable in $\mathcal{P}_4$. BreakID detects this, while sbass detects (and breaks) no symmetry.

Example 5. Consider the logic program

$$\mathcal{P}_5 = \left\{ \begin{array}{l} p. \\ q. \end{array} \right\}.$$

It is clear that p and q are interchangeable in $\mathcal{P}_4$. BreakID detects this, while sbass detects no symmetry.

In the last example, the fact that this symmetry is not detected does not have any practical consequences. Indeed, since these are only interchangeable facts, exploiting the symmetry will not help the solver. However, in examples such as Example 1, the difference is more important. If there are interchangeable atoms in choice rules, symmetry breaking can cut out exponentially large parts of the search space.

3.2 Symmetry Breaking Constraints

A classic approach to static symmetry breaking is to construct *lex-leader constraints*.

Definition 1 (Lex-leader constraint [10]). *Let ϕ be a formula over Σ, π a symmetry of ϕ, $\prec_\Sigma$ an order on Σ and $\prec$ the induced lexicographic order on the set of interpretations over Σ. A formula LL_π over $\Sigma' \supseteq \Sigma$ is a* lex-leader constraint *for π if for each Σ-interpretation I, there exists a Σ'-extension of I that satisfies LL_π iff $I \prec \pi(I)$.*

In other words, each interpretation whose symmetric image under π is smaller is invalidated by LL_π. It is easy to see that the conjunction of LL_π for all π in some $\Pi' \subseteq \Pi$ is a sound (but not necessarily complete) symmetry breaking constraint for Π.

An efficient encoding of the lex-leader constraint LL_π as a conjunction of propositional clauses is given by Aloul et al. [1]. Such clauses translate directly to ASP constraints. In order to make such an encoding, an order on Σ needs to be chosen. The classic approach to symmetry breaking is to use the numbers as they occur in the numerical DIMACS CNF format or LPARSE-SMODELS intermediate format and to post for each symmetry in the set of generators a lex-leader constraint. This is the approach that is taken for instance in the SAT symmetry breaking tool SHATTER and in SBASS.

The drawback of this approach is discussed extensively in the work that introduced BREAKID [15]. There, we show that this phase leaves room for several optimisations.

- The set $\mathcal{G}$ of generators is often not good for symmetry breaking. Manipulating this set can result in symmetry breaking constraints with much stronger breaking power.
- The order given by the numerical format is often suboptimal for symmetry breaking. The order should "match" the symmetries in $\mathcal{G}$.
- If both of the above optimisations are executed, we effectively obtain a situation in which exponentially large subgroups of the group generated by $\mathcal{G}$ are broken *completely* (without increasing the size of the symmetry breaking constraints).
- Instead of breaking on the generators, one can generate a large number of very small (size 2) symmetry breaking clauses, resulting in stronger propagation power.
- A slightly better encoding of the lex-leader constraints can be used [30].

All of these optimisations are used in the ASP version of BREAKID as well.

4 Experiments

In this section, we experimentally compare SBASS and BREAKID, using CLASP 3.1.4 as solver and GRINGO 4.5.5 as grounder. We compare the number of solved instances for a set of four symmetric decision problems, and a set of two symmetric optimization problems from the 2013's ASP competition [3].

4.1 Setup

The four decision problems in our benchmark set are **pigeons**, **crew**, **graceful** and **queens**.

pigeons is a set of 16 unsatisfiable pigeonhole instances where n pigeons must be placed in $n-1$ different holes. n takes values from $\{5, 6, \ldots, 14, 15, 20, 30, 50, 70, 100\}$. The pigeons and holes are interchangeable, leading to a large symmetry group.

crew is a set of 42 unsatisfiable airline crew scheduling instances, where optimality has to be proven for a minimal crew assignment given a moderately complex flight plan. The instances are generated by hand, with the number of crew members ranging from 5 to 25. Crew members have different attributes, but depending in the instance, multiple crew members exist with the same exact attribute set, making these crew members interchangeable.

graceful consists of 60 satisfiable and unsatisfiable graceful graph instances, taken from 2013's ASP competition [3]. These instances require to label a graph's vertices and edges such that all vertices have a different label, all edges have a different label, and each edge's label is the difference of the labels of the vertices it connects. The labels used are $\{0, 1, \ldots, n\}$, with n the number of edges. Any symmetry exhibited by the input graph is present, as well as a symmetry mapping each vertex' label l to $n - l$.

queens is a set of 4 large satisfiable N-Queens instances trying to fit n queens on an n by n chessboard so that no queen threatens another. n takes values from $\{50, 100, 150, 200\}$. The symmetries present in **queens** are the rotational and reflective symmetries of the chessboard.

The two optimization problems are **valves** and **still**. Both problems' models and instances are taken from 2013's ASP competition [3], but manual symmetry breaking constraints were removed from the ASP specification.

valves models connected pipelines in urban hydraulic networks, and features interchangeable valves. Our instance set counts 50 instances.

still models a fixpoint connected cell configuration in Conway's game of life. The game board exhibits rotational and reflective symmetry. Our instance set counts 21 instances.

All experiments were run on on an Intel® Xeon® E3-1225 CPU with Ubuntu 14.04 Linux kernel 3.13 as operating system. ASP specifications, instances, shell scripts and detailed experimental results are available online[5].

The decision problems had 6GB RAM and 1000s timeout as resource limits, and the results exclude grounding time, as this is the same for each solving configuration, but include any time needed to detect symmetry and construct symmetry breaking constraints. By default, BreakID limits the size of the symmetry breaking formula to 50 auxiliary variables for a given symmetry. To keep the comparison as fair as possible, sbass was given the same limit.

The optimization problems had 8GB RAM and 50000s timeout as resource limits, and the results exclude identical grounding time and negligable symmetry breaking time. No comparison with sbass was made as sbass does not support optimization statements.

4.2 Decision problem results

Table 1 summarizes the results of the decision problems. Analyzing the results on **pigeons**, clasp gets lost in symmetric parts of the search tree, solving only 8 instances (up to 12 pigeons). sbass can only solve three more instances (up

[5] `bitbucket.org/krr/breakid_asp`

	CLASP	SBASS					BREAKID				
	#	#	t	π	r	a	#	t	π	r	a
pigeons (16)	8	11	51.0	43.5	7729	1932	**14**	12.5	90.0	7383	2235
crew (42)	32	36	0.0	7.8	734	183	**41**	0.0	100.8	654	184
graceful (60)	**33**	28	0.7	5.5	1085	271	32	2.1	102.7	1161	347
queens (4)	4	4	33.7	2.0	392	98	4	41.1	126.0	426	100

Table 1. Experimental results of (i) CLASP without symmetry breaking preprocessor, (ii) with SBASS, and (iii) with BREAKID. # represents the number of solved instances within resource bounds, t the average symmetry preprocessing time in seconds (including detection and construction of symmetry breaking constraints), π the average number of symmetry generators detected by SAUCY, r the average number of symmetry breaking rules constructed, and a the average number of auxiliary atoms introduced.

to 15 pigeons), as the derived symmetry generators do not suffice to construct strong symmetry breaking constraints. These results are consistent with the results of Drescher et al. [16]. BREAKID detects more structure, solving all but two instances (the largest being solved contains 50 pigeons). As far as symmetry preprocessing time goes, both SBASS and BREAKID spend a significant time detecting symmetry, especially on the larger instances with more than 20 pigeons. To our surprise, BREAKID requires only a quarter of the symmetry preprocessing time SBASS uses. This difference is entirely due to SAUCY needing much more time to detect automorphisms on the graph encoding of SBASS than on the graph encoding of BREAKID. Recall that, compared to BREAKID, SBASS has the smaller encoding graph for automorphism detection due to the use of directed edges. Hence, the most plausible hypothesis to explain the difference in symmetry detection time is simply that SAUCY is sensitive to differences in graph encodings, which gets magnified by large problem instances. Nonetheless, even for the largest instance with 100 pigeons, symmetry preprocessing by SBASS did not reach the timeout limit.

The results on **crew** are similar to **pigeons**: BREAKID outperforms SBASS, which in turn outperforms plain CLASP. On the other hand, symmetry preprocessing time is negligible for **crew**. This is mainly due to the sizes of the ground programs remaining relatively small; less than 3000 rules for the largest ground program in the **crew** instance set.

Continuing with **graceful**, it is striking that the number of solved instances is *reduced* by symmetry breaking. Upon closer inspection, this is only the case for satisfiable instances. For unsatisfiable **graceful** instances, SBASS and BREAKID both solve four instances, two more than CLASP. This discrepancy is not uncommon, as static symmetry breaking reduces the search space by sometimes removing easy-to-find solutions. These results are also consistent with those reported by Drescher et al. [16]. Focusing on symmetry preprocessing time, SBASS is faster than BREAKID. This is consistent with our previous findings [15], where we argued that deriving better symmetry breaking clauses incurs extra overhead.

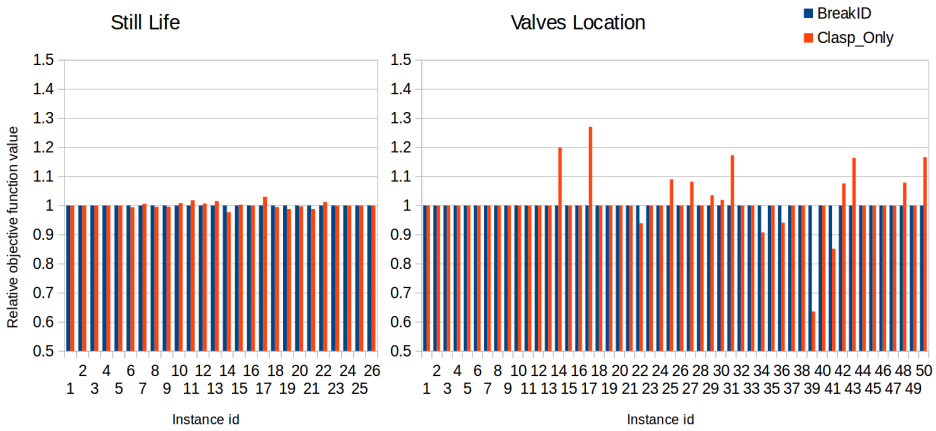

Fig. 3. The relative objective value of the best solution found after 50000s of search time for **still** and **valves**. Lower is better.

Lastly, for **queens**, all approaches solve all four instances easily. Again SBASS is faster than BreakID, which is explained by the same reason as for **graceful**. However, BreakID's preprocessing time remains well within timeout bounds.

It is also interesting to note that on all problem families SBASS and BreakID have about the same number of symmetry breaking constraints and auxiliary variables introduced, incurring roughly the same memory overhead for both approaches. However the number of symmetry generators detected by BreakID is much larger. Upon closer inspection, it turns out that BreakID often detects (trivial) interchangeability of *facts* in the program. There is no advantage in breaking such symmetry, but it does not hurt performance either. We are not sure why SBASS does not detect this kind of symmetries. Either it is because of the directional encoding, resulting essentially in a bug similar to Example 5, or it is because of an optimisation for facts we have no knowledge of.

We conclude that on the decision problem benchmark set, BreakID outperforms SBASS, especially for problems with interchangeable objects such as **pigeons** and **crew**. The price to be paid is a bit more symmetry preprocessing overhead, though the size of the symmetry breaking formula remains comparable between both approaches. This is to be expected, as BreakID compared similarly to Shatter for SAT problems.

4.3 Optimization problem results

Figure 3 shows the relative objective value of the best solution found after 50000s of search time for **still** and **valves**. Even though BreakID detects and breaks significant symmetry for both problems, and often infers interchangeability in **valves**, the resulting symmetry breaking constraints do not seem to vastly improve the final objective value for these two benchmark families. For **still**, the resulting objective value is virtually identical. For **valves**, the objective value

for BreakID's run is improved for 11 instances, while it has worsened for 5 instances, giving a small edge to BreakID compared to clasp. When looking at the number of instances for which optimality was proven, both approaches were able to prove optimality for 5 **still** instances and 15 **valves** instances.

We conclude that for optimization problems **still** and **valves**, our symmetry breaking approach detects and breaks symmetry, but any resulting speedups are small at best.

5 Discussion and Conclusion

In this paper, we describe our system BreakID, which performs *static*, *ground-level* symmetry breaking for ASP. This type of symmetry breaker is the "easiest" type to implement. It serves merely as a preprocessor for the solver. Other types of symmetry breaking exist as well. For instance, in SAT and constraint programming, *dynamic* symmetry breaking is well-studied [31, 14, 29]. The disadvantage of these approaches is that they require modifying the internals of the solver at hand. The advantage, however, is that such approaches are often more memory-friendly, since they do not require to create all the symmetry breaking constraints prior to search, are not dependent on any fixed variable order chosen a priori, and do not remove easy-to-find solutions from the search space. Another line of research in symmetry breaking is *first-order symmetry detection*, where symmetries are detected on the predicate level. In constraint programming, work has been done on composing the symmetry properties of global constraints into a symmetry group of the original problem [34]. However, most existing symmetry detection implementations also detect symmetry on the ground level.

Recently, we proposed first-order symmetry detection and breaking algorithms for the IDP system [11], a system whose input language relates closely to ASP. The main advantage of such an approach is that it allows to use the *structure* of the program to detect certain kinds of symmetries without the need for an automorphism tool. The drawbacks are that they cannot detect all types of symmetry and that they are required to interpret a much richer (first-order) language. Thus, such symmetry detection requires a significantly larger implementation effort. We believe it would definitely benefit the community to have a version of gringo, or another ASP grounder, with support for first-order symmetry detection.

The techniques we presented here are not bound to the stable semantics. They are equally applicable for instance to the well-founded semantics [33], and hence apply as well to the logic FO(ID) [13] (which combines logic programs under the well-founded semantics with propositional theories). The reason is that our techniques detect *syntactic* symmetries and hence are agnostic about the semantics at hand.

We presented experimental results comparing our tool to the existing ASP symmetry breaker sbass. For the employed decision problem benchmark set, BreakID outperforms sbass, with a small drawback in the form of greater

symmetry preprocessing overhead. BreakID also detects and breaks symmetry on optimization problems, though significant speedups have yet to be observed.

The result of this work is an easy-to-use tool that can significantly improve performance on non-optimised encodings and works with all modern ASP solvers. It can be downloaded at `bitbucket.org/krr/breakid`. Our expectation is that using this tool yields no (or very little) performance gain on highly optimised encodings, crafted by experts. However, for average users of ASP tools, BreakID can cut away large parts of the search space and is an improvement to sbass, both from an applicability as well as an efficiency point-of-view.

Acknowledgements

This research was supported by the project GOA 13/010 Research Fund KU Leuven and projects G.0489.10, G.0357.12 and G.0922.13 of FWO (Research Foundation - Flanders). Bart Bogaerts is supported by the Finnish Center of Excellence in Computational Inference Research (COIN) funded by the Academy of Finland (grant #251170).

References

1. Aloul, F., Ramani, A., Markov, I., Sakallah, K.: Solving difficult SAT instances in the presence of symmetry. In: Design Automation Conference, 2002. Proceedings. 39th. pp. 731–736 (2002)
2. Aloul, F.A., Sakallah, K.A., Markov, I.L.: Efficient symmetry breaking for Boolean satisfiability. IEEE Transactions on Computers 55(5), 549–558 (2006)
3. Alviano, M., Calimeri, F., Charwat, G., Dao-Tran, M., Dodaro, C., Ianni, G., Krennwallner, T., Kronegger, M., Oetsch, J., Pfandler, A., Pührer, J., Redl, C., Ricca, F., Schneider, P., Schwengerer, M., Spendier, L.K., Wallner, J.P., Xiao, G.: The fourth Answer Set Programming competition: Preliminary report. In: Cabalar and Son [9], pp. 42–53, `http://dx.doi.org/10.1007/978-3-642-40564-8_5`
4. Alviano, M., Dodaro, C., Faber, W., Leone, N., Ricca, F.: WASP: A native ASP solver based on constraint learning. In: Cabalar and Son [9], pp. 54–66, `http://dx.doi.org/10.1007/978-3-642-40564-8_6`
5. Audemard, G., Simon, L.: Predicting learnt clauses quality in modern SAT solvers. In: Boutilier, C. (ed.) IJCAI. pp. 399–404 (2009)
6. Babai, L.: Handbook of Combinatorics, chap. Automorphism groups, isomorphism, reconstruction, pp. 1447–1540. Elsevier (1995)
7. Balint, A., Belov, A., Heule, M.J., Järvisalo, M.: The 2013 international SAT competition. `satcompetition.org/2013` (2013)
8. Bomanson, J., Janhunen, T.: Normalizing cardinality rules using merging and sorting constructions. In: Cabalar and Son [9], pp. 187–199, `http://dx.doi.org/10.1007/978-3-642-40564-8_19`
9. Cabalar, P., Son, T.C. (eds.): Logic Programming and Nonmonotonic Reasoning, 12th International Conference, LPNMR 2013, Corunna, Spain, September 15-19, 2013. Proceedings, LNCS, vol. 8148. Springer (2013)
10. Crawford, J.M., Ginsberg, M.L., Luks, E.M., Roy, A.: Symmetry-Breaking Predicates for Search Problems. In: Principles of Knowledge Representation and Reasoning. pp. 148–159. Morgan Kaufmann (1996)

11. De Cat, B., Bogaerts, B., Bruynooghe, M., Janssens, G., Denecker, M.: Predicate logic as a modelling language: The IDP system. CoRR abs/1401.6312v2 (2016), http://arxiv.org/abs/1401.6312v2
12. De Cat, B., Bogaerts, B., Devriendt, J., Denecker, M.: Model expansion in the presence of function symbols using constraint programming. In: 2013 IEEE 25th International Conference on Tools with Artificial Intelligence, Herndon, VA, USA, November 4-6, 2013. pp. 1068–1075. IEEE Computer Society (2013), http://dx.doi.org/10.1109/ICTAI.2013.159
13. Denecker, M., Ternovska, E.: A logic of nonmonotone inductive definitions. ACM Trans. Comput. Log. 9(2), 14:1–14:52 (Apr 2008), http://dx.doi.org/10.1145/1342991.1342998
14. Devriendt, J., Bogaerts, B., Bruynooghe, M.: BreakIDGlucose: On the importance of row symmetry in SAT. In: Proceedings of the Fourth International Workshop on the Cross-Fertilization Between CSP and SAT (CSPSAT) (2014), https://lirias.kuleuven.be/handle/123456789/456639
15. Devriendt, J., Bogaerts, B., Bruynooghe, M., Denecker, M.: Improved static symmetry breaking for SAT. In: Creignou, N., Berre, D.L. (eds.) Theory and Applications of Satisfiability Testing - SAT 2016 - 19th International Conference, Bordeaux, France, July 5-8, 2016, Proceedings. Lecture Notes in Computer Science, vol. 9710, pp. 104–122. Springer (2016), http://dx.doi.org/10.1007/978-3-319-40970-2_8
16. Drescher, C., Tifrea, O., Walsh, T.: Symmetry-breaking answer set solving. AI Communications 24(2), 177–194 (2011)
17. Gebser, M., Kaufmann, B., Schaub, T.: Conflict-driven answer set solving: From theory to practice. Artif. Intell. 187, 52–89 (2012)
18. Gebser, M., Schaub, T., Thiele, S.: GrinGo: A new grounder for Answer Set Programming. In: Baral, C., Brewka, G., Schlipf, J.S. (eds.) LPNMR. Lecture Notes in Computer Science, vol. 4483, pp. 266–271. Springer (2007)
19. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In: Kowalski, R.A., Bowen, K.A. (eds.) ICLP/SLP. pp. 1070–1080. MIT Press (1988), http://citeseer.ist.psu.edu/viewdoc/summary?doi=10.1.1.24.6050
20. Heule, M., Järvisalo, M., Balyo, T.: The 2016 international SAT competition. http://baldur.iti.kit.edu/sat-competition-2016 (2016)
21. Junttila, T., Kaski, P.: Engineering an efficient canonical labeling tool for large and sparse graphs. In: Applegate, D., Brodal, G.S., Panario, D., Sedgewick, R. (eds.) Proceedings of the Ninth Workshop on Algorithm Engineering and Experiments and the Fourth Workshop on Analytic Algorithms and Combinatorics. pp. 135–149. SIAM (2007)
22. Katebi, H., Sakallah, K.A., Markov, I.L.: Symmetry and satisfiability: An update. In: Strichman, O., Szeider, S. (eds.) SAT. LNCS, vol. 6175, pp. 113–127. Springer (2010)
23. Lifschitz, V.: Answer set planning. In: De Schreye, D. (ed.) Logic Programming: The 1999 International Conference, Las Cruces, New Mexico, USA, November 29 - December 4, 1999. pp. 23–37. MIT Press (1999)
24. Marek, V., Truszczyński, M.: Stable models and an alternative logic programming paradigm. In: Apt, K.R., Marek, V., Truszczyński, M., Warren, D.S. (eds.) The Logic Programming Paradigm: A 25-Year Perspective, pp. 375–398. Springer-Verlag (1999), http://arxiv.org/abs/cs.LO/9809032
25. Marques-Silva, J.P., Sakallah, K.A.: GRASP: A search algorithm for propositional satisfiability. IEEE Transactions on Computers 48(5), 506–521 (1999)

26. McKay, B.D., Piperno, A.: Practical graph isomorphism, {II}. Journal of Symbolic Computation 60(0), 94 – 112 (2014), http://www.sciencedirect.com/science/article/pii/S0747717113001193
27. Niemelä, I.: Logic programs with stable model semantics as a constraint programming paradigm. Ann. Math. Artif. Intell. 25(3-4), 241–273 (1999), http://dx.doi.org/10.1023/A:1018930122475
28. Oh, C.: Improving SAT solvers by exploiting empirical characteristics of CDCL (2015), https://cs.nyu.edu/media/publications/oh_chanseok.pdf, PhD thesis
29. Sabharwal, A.: Symchaff: exploiting symmetry in a structure-aware satisfiability solver. Constraints 14(4), 478–505 (2009), http://dx.doi.org/10.1007/s10601-008-9060-1
30. Sakallah, K.A.: Symmetry and satisfiability. Handbook of Satisfiability 185, 289–338 (2009)
31. Schaafsma, B., Heule, M.J., Maaren, H.: Dynamic symmetry breaking by simulating zykov contraction. In: Proceedings of the 12th International Conference on Theory and Applications of Satisfiability Testing. pp. 223–236. SAT '09, Springer-Verlag, Berlin, Heidelberg (2009), http://dx.doi.org/10.1007/978-3-642-02777-2_22
32. Syrjänen, T.: Lparse 1.0 user's manual. http://www.tcs.hut.fi/Software/smodels/lparse.ps.gz
33. Van Gelder, A., Ross, K.A., Schlipf, J.S.: The well-founded semantics for general logic programs. J. ACM 38(3), 620–650 (1991), http://dx.doi.org/10.1145/116825.116838
34. Van Hentenryck, P., Flener, P., Pearson, J., Ågren, M.: Compositional derivation of symmetries for constraint satisfaction. In: Proceedings of the 6th International Conference on Abstraction, Reformulation and Approximation. pp. 234–247. SARA'05, Springer-Verlag, Berlin, Heidelberg (2005), http://dx.doi.org/10.1007/11527862_17
35. Walsh, T.: Symmetry breaking constraints: Recent results. CoRR abs/1204.3348 (2012)