TPLP **23** (2): 415–443, 2023.
doi:10.1017/S1471068422000096 First published online 21 March 2022

*On the Configuration of More and Less Expressive Logic Programs**

CARMINE DODARO
University of Calabria, Rende, Italy
(*e-mail:* dodaro@mat.unical.it)

MARCO MARATEA
University of Genoa, Genoa, Italy
(*e-mail:* marco.maratea@unige.it)

MAURO VALLATI
University of Huddersfield, Huddersfield HD1 3DH, UK
(*e-mail:* m.vallati@hud.ac.uk)

submitted 23 December 2020; revised 2 March 2022; accepted 3 March 2022

Abstract

The decoupling between the representation of a certain problem, that is, its knowledge model, and the reasoning side is one of main strong points of model-based artificial intelligence (AI). This allows, for example, to focus on improving the reasoning side by having advantages on the whole solving process. Further, it is also well known that many solvers are very sensitive to even syntactic changes in the input. In this paper, we focus on improving the reasoning side by taking advantages of such sensitivity. We consider two well-known model-based AI methodologies, SAT and ASP, define a number of syntactic features that may characterise their inputs, and use automated configuration tools to reformulate the input formula or program. Results of a wide experimental analysis involving SAT and ASP domains, taken from respective competitions, show the different advantages that can be obtained by using input reformulation and configuration.

KEYWORDS: SATisfiability, answer set programming, knowledge configuration

1 Introduction

Model-based reasoning is one of the most prominent areas of research in artificial intelligence (AI). In model-based AI approaches, solvers accept input instances written in a given logical language and automatically compute their solutions (Geffner 2018).

A pillar of model-based AI is the decoupling between the knowledge model and the reasoning side, which is usually referred to as domain-independent reasoning. This supports the use of knowledge engineering approaches that separate the modelling part from the reasoning part. The main advantage of such separation is that it is possible to "optimise" one of the two parts without changing the other for obtaining overall advantage in the whole process.

*Mauro Vallati was supported by a UKRI Future Leaders Fellowship [grant number MR/T041196/1].

We follow this path, further evidencing that this modular approach also supports the use of reformulation and configuration techniques which can automatically re-formulate, re-represent or tune the knowledge model, while keeping the same input language, in order to increase the efficiency of a general solver (see, e.g. (Vallati *et al*. 2015) still for the case of automated planning, where reformulation techniques have been widely applied). The idea is to make these techniques to some degree independent of domain knowledge and solver (that is, applicable to a range of domains and solvers technology), and use them to form a wrapper around a solver, improving its overall performance for the problem to which it is applied.

In this paper, we investigate how the configuration of knowledge models, that is, the order in which elements are listed in the considered model, can affect the performance of general automated solvers in the wider field of logic programming. In particular, we focus on two areas that can be considered to end of the spectrum: Propositional Satisfiability (SAT) and Answer Set Programming (ASP) (Baral 2003; Gelfond and Lifschitz 1988; 1991; Brewka *et al*. 2011). In both areas, configuration has been traditionally exploited to modify the behaviour of solvers to improve performance on a considered (class of) instance(s) (Eggensperger *et al*. 2019). With regard to the knowledge models characteristics, SAT, particularly in its CNF connotation, has a limited expressivity in terms of the syntax of the models. On the other hand, ASP has instead a great level of expressivity, having a rich syntax (Calimeri *et al*. 2020) aiming for better readability.

In fact, the two approaches, while sharing similar aspects, for example (i) presence of a somehow similar input format for the propositional part, (ii) the reasoning part of state-of-the-art SAT and ASP solvers employ variations of the CDCL algorithm (Mitchell 2005; Gebser *et al*. 2012), and (iii) the existence of linear techniques for rewriting ASP programs falling in a certain syntactic class (cf. tight, (Erdem and Lifschitz 2003)) and solvers that exploit such property to use SAT as ASP solver, are also very different on several respects. Among others: (a) ASP is a first-order language allowing for variables, that are eliminated during grounding, and during grounding some kind of reformulation already happens, (b) ASP rules are somehow more "constrained" than CNF clauses, since they need to preserve the head-body structure, (c) ASP allows for a number of additional constructs, like aggregates, and (d) propositional ASP is strictly more expressive than SAT.

Building on the experience gained in our prior work on SAT (Vallati and Maratea 2019), the large experimental analysis presented in this paper provides a collection of results that help to understand the impact of knowledge model configuration on automated solvers from these two subareas of the logic programming field, and to provide valuable support to knowledge engineers. In particular, in this work we:

1. define a number of SAT and ASP syntactic features useful for analysing the structure of the formula/program at hand,
2. introduce a framework that, leveraging on the introduced features, allows the automated reconfiguration of the input formula/program,
3. employ SMAC (Hutter *et al*. 2011) as a configuration tool to reformulate the input formula/program, and
4. compare the performance of state-of-the-art SAT and ASP solvers on basic and reformulated formulae/programs coming from well-known benchmark domains taken

https://doi.org/10.1017/S1471068422000096 Published online by Cambridge University Press

for respective competitions (see. e.g. the reports of the last competitions available (Heule *et al*. 2019; Gebser *et al*. 2020)).

Results show that SAT solvers can greatly benefit from such reformulation, being able to solve a consistent number of additional instances or in shorter time, and that same happens to ASP solvers, to a lesser degrees, despite the limitations (a) and (b), and the wider degrees of parameters to analyse, cf. (c).

This paper is organised as follows. First, in Section 2, we present needed preliminaries about SAT and ASP, their input languages, and the configuration techniques we are going to exploit in the paper. Then, Section 3 is devoted to the configuration of SAT formulae and ASP programs, by defining the syntactic features we are going to employ for reformulation. Further, Section 4 presents the results of our experimental analysis on both SAT and ASP domains. The paper ends in Sections 5 and 6 with the analysis of related literature and conclusions, respectively.

2 Preliminaries

This section provides the essential background with regard to SAT and ASP fields, and with regard to automated configuration techniques.

2.1 SAT formulae and answer set programming

We define (ground) disjunctive ASP programs and SAT formulae so as to underline similarities, in order to make it easier in later sections to compare the presented techniques.

Syntax. Let $\mathcal{A}$ be a propositional signature. An element $p \in \mathcal{A}$ is called *atom* or *positive literal*. The negation of an atom p, in symbols $\neg p$, is called *negative literal*. Given a literal l, we define $\bar{l} = p$, if $l = p$ and $\bar{l} = \neg p$, if $l = p$ for some $p \in \mathcal{A}$. Given a set of literals M, we denote by M^+ the set of positive literals of M, by M^- the set of negative literals of M, and by $\overline{M}$ the set $\{\bar{l} \ : \ l \in M\}$. A *clause* is a finite set of literals (seen as a disjunction), and a *SAT formula* is a finite set of clauses (seen as a conjunction).

Example 1
Let φ_{run} be the following SAT formula:

$$\begin{array}{ll} c_1: & \{p_1, \neg p_3\} \\ c_2: & \{p_2, p_3, \neg p_1, \neg p_4\} \\ c_3: & \{\neg p_5, \neg p_4\}, \end{array}$$

where c_1, c_2, c_3 are clauses. ◁

An aggregate atom is of the form:

$$\text{SUM}\{w_1 : l_1, \ldots, w_n : l_n\} \geq b, \tag{1}$$

where $n \geq 1$, $l_1, \ldots, l_n$ are distinct literals, and $b, w_1, \ldots, w_n$ are positive integers. For an atom p of the form (1), $elem(p) := \{(w_i, l_i) | i \in [1..n]\}$, $lits(p) := \{l | (w, l) \in elem(p)\}$, and $bound(p) := b$. Moreover, $\text{COUNT}\{l_1, \ldots, l_n\} \geq b$ denotes a shortcut for $\text{SUM}\{w_1 : l_1, w_2 : l_2, \ldots, w_n : l_n\} \geq b$ where $w_1 = w_2 = \ldots = w_n = 1$.

https://doi.org/10.1017/S1471068422000096 Published online by Cambridge University Press

An ASP *program* Π is a finite set of rules of the following form:

$$p_1 \vee \cdots \vee p_m \leftarrow \neg p_{m+1}, \ldots, \neg p_k, p_{k+1}, \ldots, p_n, \tag{2}$$

where $n > 0$ and $n \geq m$, $p_1, \ldots, p_m$ are atoms, $p_{m+1}, \ldots, p_n$ are atoms or aggregate atoms. For a rule r of the form (2), let $H(r)$ denote the set $\{p_1, \ldots, p_m\}$ of head atoms, and $B(r)$ denote the set $\{\neg p_{m+1}, \ldots, \neg p_k, p_{k+1}, \ldots, p_n\}$ of body literals. A rule r of the form (2) is said to be *disjunctive* if $m \geq 2$, *normal* if $m = 1$, and a *constraint* if $m = 0$. Moreover, a rule r of the form $\{p_1, p_2, \ldots, p_m\} \leftarrow \neg p_{m+1}, \ldots, \neg p_k, p_{k+1}, \ldots, p_n$ is called *choice rule* and defined here as a shortcut for the following rules: $p_1 \vee p'_1 \leftarrow B(r)$, $p_2 \vee p'_2 \leftarrow B(r)$, $\ldots$, $p_m \vee p'_m \leftarrow B(r)$, where $p'_1, \ldots, p'_m$ are fresh atoms not appearing in other rules. Note that modern ASP solvers do not usually create any auxiliary atom to handle choice rules. For an expression (SAT formula or ASP program) γ, $atoms(\gamma)$ denotes the set of (aggregate) atoms occurring in γ.

Example 2
Let Π_{run} be the following program:

$$\begin{aligned} r_1: &\quad p_1 \vee p_4 \leftarrow \\ r_2: &\quad p_2 \vee p_3 \leftarrow \neg p_4, p_1 \\ r_3: &\quad p_5 \leftarrow \text{SUM}\{1 : p_1, 2 : p_2, 4 : p_4\} \geq 7. \end{aligned}$$

Note that r_1 and r_2 are disjunctive rules, r_3 is a normal rule, and $\text{SUM}\{1 : p_1, 2 : p_2, 4 : p_4\} \geq 7$ is an aggregate atom. ◁

The *dependency graph* G_Π of Π has nodes $atoms(\Pi)$, and an arc xy, where x and y are (aggregate) atoms, for each rule $r \in \Pi$ such that $x \in H(r)$ and $y \in B(r)$. An atom is *recursive* in Π if it is involved in a cycle of G_Π. In the following, every program Π is assumed to have no recursive aggregate atoms. Note that Π_{run} has such a property.

Semantics. An interpretation I is a set of (aggregate) atoms. Given an interpretation I, relation $\models$ is defined as follows:

- for an atom p, $I \models p$ if $p \in I$; while $I \models \neg p$ if $p \notin I$;
- for an aggregate atom p of the form (1), $I \models p$ if $\sum_{(w,l) \in elem(p),\ I \models l} w \geq bound(p)$; while $I \models \neg p$ if $\sum_{(w,l) \in elem(p),\ I \models l} w < bound(p)$;
- for a clause c, $I \models c$ if $I \models l$ for some $l \in c$;
- for a rule r of the form (2), $I \models B(r)$ if $I \models l$ for all $l \in B(r)$, $I \models H(r)$ if $I \models p$ for some $p \in H(r)$, and $I \models r$ if $I \models H(r)$ whenever $I \models B(r)$;
- for a SAT formula φ, $I \models \varphi$ if $I \models c$ for all $c \in \varphi$;
- for a program Π, $I \models \Pi$ if $I \models r$ for all $r \in \Pi$.

For an expression (SAT formula or ASP program) γ, I is a model of γ if $I \models \gamma$.

Example 3
Consider φ_{run} of Example 1 and Π_{run} of Example 2, and $I = \{p_4\}$. $I \models \varphi_{run}$ and $I \models \Pi_{run}$. ◁

The *reduct* Π^I of a program Π with respect to an interpretation I is $\{H(r) \leftarrow B(r) : r \in \Pi, I \models B(r)\}$ (Faber *et al.* 2011). A model I is a *stable model* of a program Π if there is no model J of Π^I such that $J \subset I$.

https://doi.org/10.1017/S1471068422000096 Published online by Cambridge University Press

```
p cnf 5 3
1 -3 0
2 3 -1 -4 0
-5 -4 0
```

Fig. 1. SAT formula φ_{run} encoded in the DIMACS format.

```
8 2 2 3 0 0
8 2 4 5 2 1 2 3
5 6 7 3 0 3 5 2 1 2 4
1 7 1 0 6
⋮
```

Fig. 2. ASP program Π_{run} encoded in the lparse format.

Example 4
Consider Π_{run} of Example 2 and $I = \{p_4\}$. The reduct Π^I_{run} is equal to $p_1 \vee p_4 \leftarrow$. Thus, I is a stable model. ◁

2.2 Input format

Modern SAT and ASP solvers usually take as input CNF formulae and ASP programs represented by means of a numeric format. Concerning SAT, the numeric format is called *DIMACS*. Figure 1 shows the representation of the formula φ_{run} of Example 1 in the DIMACS format. The first line, starting by p, gives information about the formula: the instance is in CNF, and the numbers of atoms and clauses, respectively, are provided. In the DIMACS format each atom is uniquely identified by a number. After the initial descriptive line, clauses are listed. Each clause is a sequence of distinct non-null numbers ending with `0` on the same line. Positive numbers denote the corresponding positive literals, while negative numbers represent negative literals.

Concerning ASP programs, they are usually represented in the *lparse format* (Syrjänen 2002). As in the DIMACS format, each atom is uniquely identified by a number and each rule is represented by a sequence of numbers. Rules are listed, and each rule starts with an identifier of the rule type, as follows:

- 1 represents normal rules and constraints;
- 2 represents aggregate atoms of the type COUNT;
- 3 represents choice rules;
- 5 represents aggregate atoms of the type SUM;
- 8 represents disjunctive rules.

Figure 2 shows the representation of the program Π_{run} of Example 2 in the lparse format. In particular, consider line `8 2 4 5 2 1 2 3` representing the disjunctive rule r_2. The first number, `8`, is the identifier of the rule type, the number `2` represents $|H(r_2)|$, and `4` and `5` are the numeric identifiers of atoms p_2 and p_3, respectively. Then, `2` and `1` represent $|B(r_2)|$ and $|B(r_2)^-|$, respectively. Finally, `2` and `3` are the identifiers of the atoms p_4 and p_1, respectively. Concerning the aggregate atom appearing in rule r_3, this

https://doi.org/10.1017/S1471068422000096 Published online by Cambridge University Press

is represented by the line `5 6 7 3 0 3 5 2 1 2 4`, where `5` is the identifier of the rule type, the number `6` represents the numeric identifier of the aggregate atom, `7` is the bound of the aggregate atom, then `3` and `0` represent the number of literals and negative literals in $elem()$, respectively. Then, literals p_1, p_2, and p_4 are listed followed by their corresponding weights. Note that rule r_3 is represented by `1 7 1 0 6`, where `7` and `6` are the identifiers of p_5 and of the aggregate atom, respectively.

2.3 Automated configuration techniques

Many algorithms have parameters that can be adjusted to optimise performance (in terms of, e.g. solution cost, or runtime to solve a set of instances). Formally, this problem can be stated as follows: given a parameterised algorithm with possible configurations $\mathcal{C}$, a benchmark set Π, and a performance metric $m(c, \pi)$ that measures the performance of a configuration $c \in \mathcal{C}$ on an instance $\pi \in \Pi$ (the lower the better), find a configuration $c \in \mathcal{C}$ that minimises m over Π, that is, that minimises

$$f(c) = \frac{1}{|\Pi|} \sum_{\pi \in \Pi} m(c, \pi). \tag{3}$$

The AI community has developed dedicated algorithm configuration systems to tackle this problem Hutter *et al.* (2009); Ansótegui *et al.* (2009); Yuan *et al.* (2010). In this work, we exploit the sequential model-based algorithm configuration method SMAC (Hutter *et al.* 2011), which represents the state of the art of configuration tools and, differently from other existing tools, can handle continuous parameters. SMAC uses predictive models of algorithm performance (Hutter *et al.* 2014) to guide its search for good configurations. It uses previously observed ⟨configuration, performance⟩ pairs $\langle c, f(c)\rangle$ and supervised machine learning (in particular, random forests (Breiman 2001)) to learn a function $\hat{f} : \mathcal{C} \to \mathbb{R}$ that predicts the performance of arbitrary parameter configurations, and is used to select a promising configuration. Random forests are collections of regression trees, which are similar to decision trees but have real values (here: CPU-time performance) rather than class labels at their leaves. Regression trees are known to perform well for categorical input data. Random forests share this benefit and typically yield more accurate predictions; they also allow to quantify the uncertainty of a prediction. The performance data to fit the predictive models are collected sequentially.

In a nutshell, after an initialisation phase, SMAC iterates the following three steps: (1) use the performance measurements observed so far to fit a random forest model $\hat{f}$; (2) use $\hat{f}$ to select a promising configuration $c \in \mathcal{C}$ to evaluate next, trading off exploration of new parts of the configuration space and exploitation of parts of the space known to perform well; and (3) run c on one or more benchmark instances and compare its performance to the best configuration observed so far.

In order to save time in evaluating new configurations, SMAC first evaluates them on a single training instance; additional evaluations are only carried out (using a doubling schedule) if, based on the evaluations to date, the new configuration appears to outperform SMAC's best known configuration. Once the same number of runs has been evaluated for both configurations, if the new configuration still performs better then SMAC updates its best known configuration accordingly.

SMAC is an *anytime algorithm* (or *interruptible algorithm*) that interleaves the exploration of new configurations with additional runs of the current best configuration to

https://doi.org/10.1017/S1471068422000096 Published online by Cambridge University Press

yield both better and more confident results over time. As all anytime algorithms, SMAC improves performance over time, and for finite configuration spaces it is guaranteed to converge to the optimal configuration in the limit of infinite time.

SMAC has been used for configuring knowledge models in the fields of AI Planning (Vallati *et al*. 2015; Vallati and Serina 2018) and Abstract argumentation (Cerutti *et al*. 2018).

3 Knowledge configuration

In both SAT formulae and ASP programs, clauses, and rules are usually not ordered following a principled approach, but they are ordered according to the way in which the randomised generator has been coded, or following the way in which information from the application domain has been collected, or deliberately shuffled to prevent potential biases. This is also generally true for the order in which literals of a given clause are presented in the formula, or in the program, with some differences among SAT clauses and ASP rules. However, we should consider that rules must preserve the head-body structure, so only rule's bodies are amenable to configuration; moreover, literals in the positive and negative parts of the body can not be mixed. On the other hand, ASP programs contain further degrees of freedom given that the ASP language allows for a number of additional constructs, like aggregates.

In this section, we focus on the following question: *given the set of clauses/rules, and the set of corresponding literals, in which order should they be listed to maximise the performance of a given solver, taking into account for ASP existing constraints above-mentioned and more constructs?* The underlying hypothesis is that the order in which clauses, rules, and literals are listed can be tuned to highlight elements that are important for satisfying, or demonstrating the unsatisfability, of the considered instance by the considered solver. To answer the above question, here we explain how we have configured SAT formulae and ASP programs, that is, what features have been considered, and how related scores have been computed. Noteworthy, there is a significant body of work in both SAT and ASP that deal with features selection and computation, and they are outlined in Section 5; they are mainly concerned at instance-level, while our goal is to analyse the structure also at clause/rule level, for ordering among those elements.

3.1 Configuration of SAT formulae

The CNF configuration has to be performed online: as soon as a new formula is provided as input, the formula has to be configured before being presented to the solver. In a nutshell, given a set of parameters that can be used to modify the ordering of some aspect of the CNF formula, and given the value assigned to each parameter, the online configuration is performed by re-ordering clauses and literals accordingly. Notably, the value of each parameter has to be provided and can be identified via an appropriate off-line learning step.

Given the depicted online scenario, we are restricted to information about the CNF that can be quickly gathered and that are computationally cheap to extract. Furthermore, the configuration must consider only general aspects that are common to any CNF. As it is apparent, the use of a computationally expensive configuration of a single CNF, that considers elements that are specific to the given CNF, would nullify the potential

https://doi.org/10.1017/S1471068422000096 Published online by Cambridge University Press

performance improvement, by drastically reducing the time available for the solver to find a solution (or to demonstrate unsatisfiability).

In this work, we consider the possibility to order *clauses* according to the following criteria, denoted as $\mathcal{F}_c$:

(c1) the number of literals of the clause (*size*);
(c2) the fact that the clause is binary (*bin*);
(c3) the fact that the clause is ternary (*ter*);
(c4) the number of positive literals of the clause (*positive*);
(c5) the number of negative literals of the clause (*negative*);
(c6) the fact that the clause is binary, and both literals are negative (*bin_neg*);
(c7) the fact that the clause has only one negative literal (*only_one_neg*).

Atoms can be listed in clauses according to the following criteria, denoted as $\mathcal{F}_m$:

(m1) the number of clauses in which the atom appears (*occ*);
(m2) the average size of the clauses in which the atom is involved (*occ_avg*);
(m3) the number of binary clauses in which the atom in involved (*occ_bin*);
(m4) the number of ternary clauses in which the atom is involved (*occ_ter*);
(m5) the number of times the atom appears in clauses as positive (*occ_pos*);
(m6) the number of times the atom appears in clauses as negative (*occ_neg*);
(m7) the number of times the atom is involved in clauses where all literals are positive (*occ_all_pos*);
(m8) the number of times the atom is involved in clauses where all literals are negative (*occ_all_neg*).

Moreover, we also include two additional categorical selectors, denoted as $\mathcal{F}_s$:

(s1) to enable/disable the ordering of literals in the clauses (*ord_lit*);
(s2) to order clauses according to the ordering (direct or inverse) followed by the involved literals (*ord_cl*);

The set of proposed ordering criteria, denoted as $\mathcal{F} = \mathcal{F}_c \cup \mathcal{F}_m \cup \mathcal{F}_s$, is aimed at being as inclusive as possible, so that different characterising aspects of clauses and atoms can be taken into account, at the same time, for the configuration process.

It is easy to notice that many of the introduced criteria focus on aspects of binary and ternary clauses. This is due to their importance in the search process. For instance, binary clauses are responsible, to a great degree, of unit propagation. There are also criteria that aim at identifying potentially relevant aspects. For instance, criterion (c7) aims at identifying clauses that may represent implication relations between literals.

There are different ways for encoding the identified degrees of freedom in CNFs as parameters. This is due to the fact that orders are not natively supported by general configuration techniques (Hutter *et al*. 2011; Kadioglu *et al*. 2010a). Results presented by Vallati *et al*. (2015) suggest that purely categorical parametrisations are not indicated for the configuration of models, as they tend to fragment the configuration space and to introduce discontinuities. Those combined aspects make the exploration of the configuration space particularly challenging for learning approaches. For this reason, here we generate 7 continuous parameters for configuring the order of clauses and 8 continuous parameters for configuring the order of variables in clauses. Each parameter corresponds to one of the aforementioned criteria, and they have to be combined to generate different

https://doi.org/10.1017/S1471068422000096 Published online by Cambridge University Press

φ_{run}:
```
p cnf 5 3
1 -3 0
2 3 -1 -4 0
-5 -4 0
```

φ_{conf}:
```
p cnf 5 3
3 -1 -4 2 0
-4 -5 0
1 -3 0
```

Fig. 3. The example CNF formula non configured (φ_{run}), and the configured version (φ_{conf}). Configuration has been done by listing clauses according to their length and the number of negative literals. Literals are listed following the number of clauses they are involved.

possible orderings of clauses and literals in CNFs. Each continuous parameter in $\mathcal{F}_c$ and $\mathcal{F}_m$ has associated a real value in the interval $[-10.0, +10.0]$ which represents (in absolute value) the *weight* given to the corresponding ordering criterion. Concerning selectors, *ord_lit* can assume a Boolean value, 0 or 1, whereas *ord_cl* can be 0 if clauses must be ordered using only the features of the literals appearing in the clause, 1 if clauses must be ordered using only the features of the clause, 2 if clauses must be ordered using both the features of the literals and the features of the clauses. Thus, the configuration space is $\mathcal{C} = [-10.0, +10.0]^{15} \times \{0, 1\} \times \{0, 1, 2\}$. A (total) function $\omega : \mathcal{F} \mapsto [-10.0, +10.0]$ maps parameters in $\mathcal{F}$ to a weight, where $\omega(ord\_lit)$ is restricted to be in $\{0, 1\}$ and $\omega(ord\_cl)$ is restricted to be in $\{0, 1, 2\}$, respectively.

The configuration criteria mentioned above can be used to order any CNF. In particular, given a CNF φ and a weight function ω, the corresponding configuration of the formula is obtained as follows. For each atom p occurring in φ, an ordering score of p is defined as:

$$O_{at}(p, \varphi, \omega) = \sum_{c \in \mathcal{F}_m} (value(p, \varphi, c) \cdot \omega(c)), \tag{4}$$

where c is a criterion for configuring literals' order in the set $\mathcal{F}_m$ (i.e., from (m1) to (m8)), and $value(p, \varphi, c)$ is the numerical value of the corresponding aspect for the atom p. If $\omega(ord\_lit) = 1$, then, for every clause, the involved literals are ordered (in descending order) following the score O_{at}. Ties are broken following the order in the original CNF configuration. As it is apparent from equation (4), a positive (negative) value of $\omega(c)$ can be used to indicate that the aspect corresponding to the parameter c is important for the SAT solver, and that literals with that aspect should be listed early (late) in the clause to improve performance. If $\omega(ord\_lit) = 0$, literals follow the order as in the provided initial CNF.

Similarly to what is presented in equation (4) for literals, clauses are ordered according to a corresponding score $O_{cl}(cl, \varphi, \omega)$, defined as follows:

$$O_{cl}(cl, \varphi, \omega) = \begin{cases} \sum_{p \in cl^+ \cup \overline{cl^-}} O_{at}(p, \varphi, \omega) & \text{if } \omega(ord\_cl) = 0 \\ \sum_{c \in \mathcal{F}_c} (value(cl, \varphi, c) \cdot \omega(c)) & \text{if } \omega(ord\_cl) = 1 \\ \sum_{p \in cl^+ \cup \overline{cl^-}} O_{at}(p, \varphi, \omega) + \sum_{c \in \mathcal{F}_c} value(cl, \varphi, c) & \text{if } \omega(ord\_cl) = 2. \end{cases} \tag{5}$$

Example 5

Let us consider again the CNF φ_{run} of Example 1 and reported, using the DIMACS format, in Figure 3. Suppose that we are interested in listing clauses according to their length (criterion (c1)) and to the number of involved negative literals (criterion (c5)). Similarly, we are interested in listing the literals of a clause according to the number of

https://doi.org/10.1017/S1471068422000096 Published online by Cambridge University Press

clauses in which they appear (criterion (m1)). In this case, we have to set $\omega(size) = 10.0$, $\omega(negative) = 10.0$, $\omega(occ) = 10.0$, $\omega(ord\_lit) = 1$, and $\omega(ord\_cl) = 1$, whereas $\omega(c) = 0.0$ for all other criterion c in $\mathcal{F}$. Then, $O_{cl}(\text{“}\texttt{2 3 -1 -4 0}\text{”}, \varphi_{run}, \omega) = 60.0$, since it involves 4 literals, and 2 of them are negative, thus $4 \cdot 10.0 + 2 \cdot 10.0 = 60.0$. According to the same criteria, $O_{cl}(\text{“}\texttt{1 -3 0}\text{”}, \varphi_{run}, \omega) = 30.0$. In a similar way, but considering the corresponding criterion, the score of literals can be calculated, and literals are then ordered accordingly in each clause. Result is φ_{conf} reported in Figure 3. Note that the first line of the considered CNF formula is unmodified, as the DIMACS format require it to be the first, and to present information in a given order. ◁

The way in which the considered ordering criteria are combined, via equations (4) and (5), gives a high degree of freedom for encoding and testing different configurations. Very specific aspects can be prioritised: for instance, it would be possible to present first clauses that are binary, and where both literals are positive, by penalising criterion (c5) and giving a high positive weight to criterion (c2). Furthermore, additional criteria can be added, with no need to modify or update the overall configuration framework.

3.2 Configuration of ASP programs

In this subsection, instead, we turn our attention to the configuration of ASP programs. Similarly to SAT, we generate 23 continuous parameters for configuring the order of rules and aggregates. Each parameter corresponds to a feature that is syntactic and easy to compute, and they have to be combined to generate different possible orderings of rules and aggregates. Each continuous parameter has an associated real value in the interval $[-10.0, +10.0]$ which represents (in absolute value) the *weight* given to the corresponding ordering criterion. The continuous parameters are detailed in the following:

(k1) occurrences of a literal in heads (*head_occ*)
(k2) occurrences of a literal in bodies (*body_occ*)
(k3) occurrences of a literal in positive part of bodies (*pos_body_occ*)
(k4) occurrences of a literal in negative part of bodies (*neg_body_occ*)
(k5) occurrences of a literal in bodies of “short” size (*short_body_occ*)
(k6) occurrences of a literal in positive part of bodies of “short” size (*short_pos_body_occ*)
(k7) occurrences of a literal in negative part of bodies of “short” size (*short_neg_body_occ*)
(k8) occurrences of a literal in aggregates (*aggregate_occ*)

(k9) constraints (*constraints*)
(k10) normal rules (*normal*)
(k11) disjunctive rules (*disjunctive*)
(k12) choice rules (*choice*)
(k13) literals in the body (*body*)
(k14) literals in the positive part of the body (*p_body*)
(k15) literals in the negative body of the body (*n_body*)
(k16) ratio between positive and negative body literals (*ratio_pos_neg*)
(k17) Horn bodies (*horn*)

https://doi.org/10.1017/S1471068422000096 Published online by Cambridge University Press

Function O_l(Literal l, Program Π, Weight function ω)

s := 0;
s += $|\{r \ : \ r \in \Pi, l \in H(r)\}| * \omega(head\_occ)$;
s += $|\{r \ : \ r \in \Pi, l \in B(r)\}| * \omega(body\_occ)$;
s += $|\{r \ : \ r \in \Pi, l \in B(r)^+\}| * \omega(pos\_body\_occ)$;
s += $|\{r \ : \ r \in \Pi, l \in B(r)^-\}| * \omega(neg\_body\_occ)$;
s += $|\{r \ : \ r \in \Pi, l \in B(r), |B(r)| \leq 2\}| * \omega(short\_body\_occ)$;
s += $|\{r \ : \ r \in \Pi, l \in B(r)^+, |B(r)| \leq 2\}| * \omega(short\_pos\_body\_occ)$;
s += $|\{r \ : \ r \in \Pi, l \in B(r)^-, |B(r)| \leq 2\}| * \omega(short\_neg\_body\_occ)$;
s += $|\{p \ : \ p \in atoms(\Pi), p \ is \ an \ aggregate \ atom, l \in lits(p)\}| * \omega(aggregate\_occ)$;
return s;

Function O_r(Rule r, Program Π, Weight function ω)

s := 0;
if $|H(r)| = 0$ **then** s += $\omega(constraint)$;
if $|H(r)| = 1$ **then** s += $\omega(normal)$;
if $|H(r)| > 1$ **then** s += $\omega(disjunctive)$;
if r *is choice* **then** s += $\omega(choice) * t_1$;
s += $|B(r)| * \omega(body) + |B(r)^+| * \omega(p\_body) + |B(r)^-| * \omega(n\_body)$;
if $|B(r)^-| \neq 0$ **then** s += $(|B(r)^+| \div |B(r)^-|) * \omega(ratio\_pos\_neg)$;
if $|B(r)^+| = 1$ **then** s += $\omega(horn)$;
s += $|\{p \in H(r)|\ p \ is \ recursive\}| * \omega(rec\_head)$;
s += $|\{l \in B(r)|\ l \ is \ recursive\}| * \omega(rec\_body)$;
if $|H(r)| + |B(r)| \geq 2$ ***and*** $|H(r)| + |B(r)| \leq 3$ **then** s+= $\omega(short)$;
s += $(\sum_{l \in H(r) \cup B(r)} O_l(\Pi, l, \omega)) \div (|H(r)| + |B(r)|)$;
return s;

(k18) recursive atoms in heads (*rec_head*)
(k19) recursive atoms in bodies (*rec_body*)
(k20) binary or ternary rules (*short*)

(k21) aggregates (*aggregate*)
(k22) aggregate size (*aggregate_size*)
(k23) ratio between aggregate size and bound (*aggregate_ratio_bound_size*)

Given the structure of ASP programs, richer than SAT formulae, it is not as straightforward and compact to calculate scores as for SAT formulae; in order to calculate the final score, we have introduced three functions O_l, O_r, and O_a for calculating scores for literals, rules and aggregate atoms, respectively, that take as input a program, an element (a literal, a rule or an aggregate atom, respectively), and a weight function $\omega : \mathcal{F} \mapsto [-10.0, 10.0]$, where $\mathcal{F} = \{head\_occ, body\_occ, \ldots, aggregate\_ratio\_bound\_size\}$, that is, it includes all the features reported from (k1) to (k23). The output of the three functions is the score of the element, computed as a sum of individual contributions brought by the features linked to the element. The score of rules and aggregates is later on used to order the rules of ASP programs.

https://doi.org/10.1017/S1471068422000096 Published online by Cambridge University Press

Function O_a(Aggregate atom p, Program Π, Weight function ω)

1 s := $t_2 * \omega(aggregate) + |lits(p)| * \omega(aggregate\_size)$;
2 s += $(bound(p) \div \sum_{(w,l) \in elem(p)} w) * \omega(aggregate\_ratio\_bound\_size)$;
3 s += $(\sum_{l \in lits(p)} O_l(\Pi, l, \omega)) \div |lits(p)|$;
4 **return** s;

Π_{run} :

```
8 2 2 3 0 0
8 2 4 5 2 1 2 3
5 6 7 3 0 3 5 2 1 2 4
1 7 1 0 6
⋮
```

Π_{conf} :

```
5 6 7 3 0 3 5 2 1 2 4
8 2 2 3 0 0
8 2 4 5 2 1 2 3
1 7 1 0 6
⋮
```

Fig. 4. The example ASP program non configured (Π_{run}), and the configured version (Π_{conf}). Configuration has been done by preferring aggregates and rules with literals occurring in many negative bodies.

Function O_l computes the score of a given literal l by summing up, from line 2 to 9, all single contributions of features (k1)–(k8), by multiplying the number of times l "falls" in the category described by the respective feature to the weight of the feature. As an example, line 4, related to feature *pos_body_occ*, gives a contribution to s obtained by multiplying the number of times literal l occurs in positive bodies of the program Π and the weight of the feature.

Function O_r computes the score of a rule r. Depending of whether r is a constraint, a normal or disjunctive, or a choice rule, one of the lines from 2 to 5 is activated. If r is a choice, an additional factor t_1 is considered, which is an arbitrary large value, set to 10^5 in our experiments, and (from the configuration side) means to put priorities to such rules. Then, lines from 6 to 12 contribute further to the score, as a bonus, for features (k13)–(k20): lines 6, 9, and 10 work similarly as within function O_l, while lines 7, 8, and 11 behave similarly to lines 2–5 in this function for the respective feature. Finally, line 12 employs function O_l to compute a score that is later on divided by the number of literals appearing in the rule.

Function O_a computes the score for an aggregate atom a. In particular, line 1 takes into account features (k21) and (k22), related to the presence of aggregates and its size, giving a high reward (value t_2 set to 10^5) to the presence of aggregates as for choice rules before, line 2 considers the ratio between bound and size of the aggregate, while line 3 has a similar behaviour as of line 12 of the function O_r.

Example 6
Consider again the program Π_{run} of Example 2 and its lparse representation, reported in Figure 4. Suppose that we are interested in ordering the program by giving a high priority to aggregates and then to give additional priorities to rules according to the atoms that occur in negative bodies. This can be done by leaving all the parameters to the default value 0.0, but $\omega(aggregate)$ and $\omega(p\_body)$ that are both set to 10.0. In particular, the atom with id 2 occurs in the negative body of the second rule of Π_{run},

https://doi.org/10.1017/S1471068422000096 Published online by Cambridge University Press

while other atoms do not occur in the negative body. Thus, $O_l(2, \Pi_{run}, \omega)$ returns 10.0, whereas $O_l(l, \Pi_{run}, \omega)$ returns 0.0 for $l \in \{1, 3, 4, 5, 6, 7\}$. Then, O_r("8 2 2 3 0 0", Π_{run}, ω) and O_r("8 2 4 5 2 1 2 3", Π_{run}, ω) return 5.0 and 2.5, respectively, whereas O_r("1 7 1 0 6", Π_{run}, ω) returns 0.0, and O_a("5 6 7 3 0 3 5 2 1 2 4", Π_{run}, ω) returns 100003.33. Result is Π_{conf} reported in Figure 4. ◁

4 Experimental analysis

This experimental analysis aims at evaluating the impact of the proposed automated approach for performing the configuration of knowledge models, on state-of-the-art domain-independent solvers' performance from SAT and ASP.

4.1 Experimental settings

In this work, we use the state-of-the-art SMAC (Hutter *et al.* 2011) configuration approach for identifying a configuration of the knowledge model that aims at improving the number of solved instances and the PAR10 performance of a given solver. PAR10 is the average runtime where unsolved instances count as 10× cutoff time. PAR10 is a metric commonly exploited in machine learning and algorithm configuration techniques, as it allows to consider coverage and runtime at the same time (Eggensperger *et al.* 2019).

For each solver, a benchmark-set specific configuration was generated using SMAC 2.08. A dedicated script, either in Python 2.7 or in C++, is used as a wrapper for extracting information from a knowledge model and, according to the parameters' values, reconfigure it and provide it as input for the solver.

Experiments, on both SAT and ASP instances, were run on a machine executing Linux Ubuntu 4.4.0-104 and equipped with Intel Xeon 2.50 Ghz processors. Each SMAC configuration process, that is, for each pair $\langle solver, benchmark\ set \rangle$, has been given a budget of 7 sequential CPU-time days, and run on a dedicated processor.

To compare performance, as mentioned we rely on the number of solved instances, the PAR10, and the IPC score. For a solver $\mathcal{R}$ and an instance p to be solved, $Score(\mathcal{R}, p)$ is defined as:

$$Score(\mathcal{R}, p) = \begin{cases} 0 & \text{if } p \text{ is unsolved} \\ \frac{1}{1+\log_{10}\left(\frac{T_p(\mathcal{R})}{T_p^*}\right)} & \text{otherwise,} \end{cases}$$

where T_p^* is the minimum amount of time required by any compared system to solve the instance, and $T_p(\mathcal{R})$ denotes the CPU time required by $\mathcal{R}$ to solve the instance p. Higher values of the score indicate better performance, where the best performing solver obtains a score equals to 1.

All the executables, benchmarks, instances, and generators used in the experiments are available at `https://www.mat.unical.it/~dodaro/research/aspsatconfig`.

4.2 Configuration of SAT formulae

We selected 3 SAT solvers, based on their performance in recent SAT competitions and their widespread use: CADICAL version sc17 (Biere 2017), GLUCOSE 4.0 (Audemard *et al.* 2013), and LINGELING version bbc (Biere 2017).

https://doi.org/10.1017/S1471068422000096 Published online by Cambridge University Press

In designing this part of the experimental analysis, we followed the Configurable SAT Solver Challenge (CSSC) (Hutter *et al*. 2017). The competition aimed at evaluating to which extent SAT solvers' performance can be improved by algorithm configuration for solving instances from a given class of benchmarks. In that, the CSSC goals are similar to the goals of this experimental analysis, that is, assessing how performance can be improved via configuration, thus their experimental settings are deemed to be appropriate for our analysis. However, CSSC focused on the configuration of SAT solvers' behaviour by modifying exposed parameters of solvers. In this work, we do not directly manipulate the behaviour of SAT solvers via exposed parameters, but we focus on the impact that the configuration of a CNF formula can have on solvers.

Following CSSC settings, a cut-off of 5 CPU-time minutes, and a memory limit of 8 GB of RAM, has been set for each solver run on both training and testing instances. This is due to the fact that many solvers have runtime distributions with heavy tails (Gomes *et al*. 2000), and that practitioners often use many instances and relatively short runtimes to benchmark solvers for a new application domain (Hutter *et al*. 2017). There is also evidence that rankings of solvers in SAT competitions would remain similar if shorter runtimes are enforced (Hutter *et al*. 2010).

We chose benchmark sets from the CSSC 2014 edition (Hutter *et al*. 2017), and the benchmarks used in the Agile track of the 2016 SAT competition.[1] These two competitions provide benchmarks that can highlight the importance of configuration (CSSC) even though a different type of configuration than the one considered in this paper and that include instances that have to be solved quickly (Agile). In particular, CSSC benchmarks can allow us to compare the impact of the proposed CNF configuration with regard to the solvers' configuration.

Selected CSSC 2014 benchmark sets include: Circuit Fuzz (Industrial track), 3cnf, K3 (Random SAT+UNSAT Track), and Queens and Low Autocorrelation Binary Sequence (Crafted track).[2] Benchmark sets were selected in order to cover most of the tracks considered in CSSC, and by checking that at least 20% of the instances were solvable by considered solvers, when run on the default CNFs. Benchmarks were randomly divided into training and testing instances, aiming at having between 150 and 300 instances for testing purposes, and a similar amount of benchmarks for training. The size of each testing set is shown in Table 1.

Table 1 summarises the results of the selected SAT solvers on the considered benchmark sets. Results are presented in terms of the number of timeouts on testing instances, achieved by solvers run using either the default or the configured CNFs. Indeed, all of the considered solvers benefited from the configuration of the CNFs. Improvements vary according to the benchmark sets: the Agile16 set is, in general, the set where the solvers gained more by the use of configured CNFs. Remarkably, the improvements observed in Table 1 are comparable to those achieved in CSSC 2013 and 2014, that were achieved by configuring the solvers' behaviour (Hutter *et al*. 2017). In fact, these results may confirm our intuition that the way in which clauses and literals are ordered has an impact on the way in which solvers explore the search space. Listing "important" clauses earlier may lead the solver to tackle complex situations early in the search process, making it

[1] https://baldur.iti.kit.edu/sat-competition-2016/.
[2] http://aclib.net/cssc2014/benchmarks.html.

https://doi.org/10.1017/S1471068422000096 Published online by Cambridge University Press

Table 1. *Number of solved instances of the selected solvers on the considered benchmark set when running on the default and on the configured CNFs. Bold indicates the best result*

		CADICAL		GLUCOSE		LINGELING	
Problem	#	Def.	Conf.	Def.	Conf.	Def.	Conf.
K3	150	61	**66**	78	**81**	74	**75**
3cnf	250	31	**34**	116	**119**	37	**40**
Queens	150	140	**141**	124	**125**	126	**127**
Low autocorrelation	300	182	**184**	185	**191**	177	**180**
Circuit Fuzz	185	166	**168**	176	176	173	**175**
Agile16	250	219	**221**	226	**231**	195	**202**
Total	1285	799	**814**	905	**923**	782	**799**

Table 2. *Results of the selected solvers on the considered benchmark sets. For each solver and benchmark, we show the IPC score achieved when running on the default and on the configured CNFs. Bold indicates the best result. Results of different solvers cannot be directly compared*

	CADICAL		GLUCOSE		LINGELING	
Problem	Def.	Conf.	Def.	Conf.	Def.	Conf.
K3	56.7	**59.9**	71.3	**76.3**	67.8	**68.6**
3cnf	27.3	**31.6**	106.6	**107.0**	33.6	**35.9**
Queens	136.5	**137.6**	119.3	**121.1**	120.6	**122.9**
Low autocorrelation	171.8	**173.4**	177.2	**183.7**	171.0	**175.3**
Circuit Fuzz	156.3	**160.8**	175.2	**175.3**	161.3	**164.3**
Agile16	208.1	**211.3**	209.1	**215.9**	188.6	**196.6**
Total	756.7	**774.6**	858.7	**879.3**	742.9	**763.6**

then easier to find a solution. In that, it may be argued that a solver's behaviour can be controlled internally, by modifying its exposed parameters, and externally by ordering the CNF in a suitable way.

Interestingly, the overall results (last row of Table 1) indicate that the CNF configuration does not affect all the solvers in a similar way, and that can potentially lead to rank inversions in competitions or comparisons. This is the case of LINGELING (on configured formulae) and CADICAL on "default" formulae. This may suggest that current competitions could benefit by exploiting a solver-specific configuration, in order to mitigate any implicit bias due to the particular CNF configuration exploited. Randomly listing clauses and variables may of course remove some bias, but it can also be the case that different biases are introduced. In that sense, allowing solvers to be provided with a specifically-configured CNF may lead to a better comparison of performance. Finally, it is worth noting that the way in which the CNFs are configured varies significantly between solvers, as well as according to the benchmark set. In other words, there is not a single ordering that allows to maximise the performance of all the SAT solvers at once.

In Tables 2 and 3, the performance of a solver run on the default and configured formulae are compared in terms of IPC score and PAR10. Results indicate that the configuration provides, for most of the benchmark sets, a noticeable improvement.

https://doi.org/10.1017/S1471068422000096 Published online by Cambridge University Press

Table 3. *Results of the selected solvers on the considered benchmark sets. For each solver and benchmark, we show the PAR10 score achieved when running on the default and on the configured CNFs. Bold indicates the best result*

	CADICAL		GLUCOSE		LINGELING	
Problem	Def.	Conf.	Def.	Conf.	Def.	Conf.
K3	1788.9	**1692.8**	1448.7	**1391.1**	1538.8	**1521.5**
3cnf	2640.7	**2601.8**	1660.5	**1629.2**	2569.9	**2534.7**
Queens	217.2	**196.3**	526.3	**507.6**	495.2	**476.1**
Low autocorrelation	1184.8	**1166.3**	1155.0	**1099.6**	1236.6	**1208.9**
Circuit Fuzz	324.7	**294.1**	**159.4**	159.5	216.1	**199.3**
Agile16	417.7	**391.0**	327.2	**277.1**	707.6	**628.8**

To shed some light on the most relevant aspects of the SAT formula configuration, we assessed the importance of parameters in the considered configurations using the fANOVA tool (Hutter *et al*. 2014). We observed that in most of the cases, improvements are mainly due to the effect of the correct configuration of a single criterion, rather than to the interaction of two or more criteria together. In terms of clauses, parameters controlling the weight of criteria (c4) and (c5) are deemed to be the most important: in other words, the number of positive (or negative) literals that are involved in a clause are a very important aspect for the performance of SAT solvers. The solver that can gain the most by ordering the clauses is LINGELING. In particular, this solver shows best performance when clauses with a large number of negative literals are listed early.

Parameters related to criteria (m2), (m6), and (m8) have shown to have a significant impact with regard to the literals' ordering in clauses. For GLUCOSE and CADICAL, criterion (m2), that is, the average size of the clauses in which the literal is involved, is the most important single criterion that has to be correctly configured. However, it is a bit hard to derive some general rules, as their impact on orderings vary significantly with regard to the solver and the benchmark set.

Generally speaking, also in the light of the criteria that are most important for clauses, the ordering of literals appears to be the most important in a CNF formulae: this is also because, in many cases, clauses are ordered according to the (separately-calculated) weight of the involved literals. This behaviour can be due to the way in which data structures are generated by solvers: usually literals are the main element, that is also the focus of heuristic search used by SAT solvers. Instead, clauses from the CNF tend to have a less marked importance during the exploration of the search space, as they are related to literals mostly via lists, and are exploited only for checking satisfiability and performing unit propagation. Clauses learnt during the search process are not included in our analysis, as they are not part of the CNF formula–but are generated online by the solver.

Finally, we want to test if there is a single general configuration that improves the performance of a solver on any formula, despite of the benchmark and underlying structure. Therefore, we trained each of the considered solvers on a training set composed by an equal proportion of instances from each of the 6 benchmark sets. As for previous configurations, we gave 5 days of sequential CPU-time for each learning process and obtained configurations have been tested on an independent testing set that includes instances from all the benchmark sets. Results are presented in Table 4.

https://doi.org/10.1017/S1471068422000096 Published online by Cambridge University Press

Table 4. *Results achieved by the selected solvers on the general testing set. For each solver, we show the number of solved instances and IPC score achieved when running on the default and on the CNFs configured using the general configuration. Bold indicates the best result*

	Solved		IPC score	
Solver	Def.	Conf.	Def.	Conf.
CADICAL	1184	**1187**	**172.7**	172.5
GLUCOSE	1205	**1207**	207.5	**211.2**
LINGELING	1176	**1177**	190.7	**191.7**

Results on the independent testing set indicate that this sort of configuration has a very limited impact on solvers' performance. This seems to confirm our previous intuition that solvers require differently configured formulae according to the underlying structure of the benchmark: it is therefore the case that structurally different sets of instances require a very different configuration. Intuitively, this seems to point to the fact that, in different structures, the characteristics that identify challenging elements to deal with, vary. Solvers, when dealing with different sets of benchmarks, are then sensitive to different aspects of the CNF formulae, that should be appropriately highlighted and configured. On the one hand, this result may be not fully satisfying, as it suggests that there is not a quick way to improve the performance of SAT solvers. On the other hand, the results of the other experiments indicate that, for real-world applications of SAT where instances share some underlying structure, there is the possibility to further improve the SAT solving process by identifying a specific configuration for the solver at hand. As a further observation, we remark that the presented results, achieved on testing set instances, are comparable to those observed on the training set used by SMAC for the configuration process. This confirms the ability of the learned knowledge to generalise on different instances.

4.3 Configuration of ASP programs

We selected 3 ASP solvers, based on their performance in recent competitions, on the different approaches implemented, and for their widespread use: CLASP, (Gebser *et al.* 2012), LP2SAT (Janhunen 2018), and WASP (Alviano *et al.* 2015).

We chose benchmark sets used in ASP competitions for which either a sufficiently large number of instances or a generator is available. Selected benchmark sets include: Graceful Graphs, Graph Colouring, Hamiltonian, Incremental Scheduling, and Sokoban. Concerning Incremental Scheduling and Sokoban, we use all the instances submitted to the 2015 ASP Competition (Gebser *et al.* 2017), whereas for Graceful Graphs, Graph Colouring and Hamiltonian instances were randomly generated. Benchmarks were randomly divided into training and testing instances, aiming at having between 50 and 100 instances for testing purposes, and a 200–500 instances for training. In this setting, we considered a cutoff of 10 CPU-time minutes, and a memory limit of 8 GB of RAM. Table 5 summarises all the ASP constructs that are available for each tested benchmark. Moreover, concerning LP2SAT, there are several levels of configurations. In particular, it

https://doi.org/10.1017/S1471068422000096 Published online by Cambridge University Press

Table 5. *List of ASP constructs available for each considered benchmark*

Benchmark	Choice rules	Recursive atoms	Count	Sum
Graceful graphs	✓		✓	
Graph colouring				
Hamiltonian	✓	✓	✓	
Incremental scheduling	✓		✓	✓
Sokoban	✓		✓	

Table 6. *Results of the selected solvers on the considered benchmark set. For each solver and benchmark, we show the number of solved instances when running on the default and on the configured ASP programs. Bold indicates the best result*

		CLASP		LP2SAT		WASP	
Problem	#	Def.	Conf.	Def.	Conf.	Def.	Conf.
Graph colouring	50	48	48	49	49	43	**44**
Graceful graphs	50	24	24	24	24	20	**22**
Hamiltonian	50	50	50	0	0	50	50
Incremental scheduling	50	38	**39**	25	25	36	**37**
Sokoban	100	44	44	40	**44**	40	**42**
Total	300	204	**205**	138	**142**	189	**195**

is possible *(i)* to configure the ASP input and then to configure the SAT formula, *(ii)* to configure only the SAT formula, *(iii)* to configure only the ASP input. We have conducted a preliminary experiment analysis and in the following we report only the results of *(iii)* since they are the ones that obtained the best performance. Note that no domain contains (stratified) disjunction, so in this restricted setting ASP is as expressive as SAT (i.e., point (d) in the introduction is not leveraged). We are not aware of any publicly available benchmark containing disjunctive rules with a huge number of instances and/or generators, therefore we could not extend our analysis to this kind of programs.

Table 6 summarises the results of the selected solvers on the considered benchmark sets. Results are presented in terms of number of solved testing instances, achieved by solvers run using either the default or the configured ASP program. It is interesting to notice that, also for ASP, solvers are differently affected by the use of the configured knowledge. On the one hand, CLASP does not benefit by the configuration in terms of coverage. On the other hand, the configuration has a widespread beneficial impact on the performance of WASP. LP2SAT sits between the two sides of the spectrum: the configuration provided a significant improvement to the coverage performance on a single domain, Sokoban.

In Tables 7 and 8, the performance of a solver run on the default and configured programs are compared in terms of IPC score and PAR10, respectively. Here it is possible to observe that the configuration has a beneficial impact on the considered solvers in most of the benchmark domains. When considering the performance of CLASP in Hamiltonian, for instance, we observed an average runtime drop from 32 to 26 CPU-time seconds. Similar improvements have been observed also for the other solvers. The presented results

https://doi.org/10.1017/S1471068422000096 Published online by Cambridge University Press

Table 7. *Results of the selected solvers on the considered benchmark sets. For each solver and benchmark, we show the IPC score achieved when running on the default and on the configured ASP programs. Bold indicates the best result*

	CLASP		LP2SAT		WASP	
Problem	Def.	Conf.	Def.	Conf.	Def.	Conf.
Graph colouring	47.3	**47.9**	49.0	49.0	41.6	**44.0**
Graceful graphs	18.8	**22.3**	24.0	24.0	18.6	**20.0**
Hamiltonian	44.8	**46.0**	0.0	0.0	50.0	50.0
Incremental scheduling	37.7	**38.9**	24.6	**24.9**	35.4	**35.6**
Sokoban	44.0	44.0	37.6	**39.2**	38.9	**40.5**
Total	192.6	**199.1**	135.2	**137.1**	184.5	**190.1**

Table 8. *Results of the selected solvers on the considered benchmark sets. For each solver and benchmark, we show the PAR10 score achieved when running on the default and on the configured ASP programs. Bold indicates the best result*

	CLASP		LP2SAT		WASP	
Problem	Def.	Conf.	Def.	Conf.	Def.	Conf.
Graph colouring	262.3	**261.7**	4.6	4.6	850.8	**734.8**
Graceful graphs	3138.5	**3136.4**	3126.2	3126.2	3618.7	**3487.6**
Hamiltonian	32.3	**26.3**	6000	6000	63.9	63.9
Incremental scheduling	1466.1	**1352.7**	3020.6	**3015.3**	2089.6	**1981.1**
Sokoban	3401.2	3401.2	3621.0	**3410.7**	3620.5	**3502.5**

indicate that the configuration of ASP programs can improve the runtime performance of ASP solvers.

Finally, we also test if there is a single general configuration that improves the performance of a solver on any ASP program, despite of the benchmark and underlying structure. As in the SAT counterpart, we trained each of the considered solvers on a training set composed by an equal proportion of instances from each of the benchmark sets. It should come as no surprise that the results indicate that this sort of configuration has no significant impact on ASP solvers' performance. For no one of the considered solvers it has been possible to identify a configuration able to improve the average performance.

Also for ASP solvers, we used the fANOVA tool to identify the most relevant aspects of the configuration process. The lack of a general configuration that allows to improve average performance of the solvers, suggests that the importance of a configuration parameter depends on the benchmark domain. In other words, the same element can be more important in a domain, but almost irrelevant in another, according the the structure of the instances to be solved. Looking at the configured ASP program identified for LP2SAT in the Sokoban domain, it appears that features related to occurrences of literals in heads (k1), and to occurrences of literals in bodies are among the most relevant criterion. In particular, the 5 most important parameters identified by the fANOVA tool are (k1), (k23), (k7), (k15), and (k13). By analysing the configurations obtained for WASP it is possible to derive that, for this solver, most of the important criteria are different than

https://doi.org/10.1017/S1471068422000096 Published online by Cambridge University Press

those of LP2SAT. Considering a domain where the configuration allowed WASP to obtain a significant improvement, Graph Colouring, the 5 most important parameters identified by the fANOVA tool are: (k16), (k1), (k13), (k14), and (k9). Only (k1), that focuses on the importance of the occurrences of a literal in heads, is shared between the configuration of LP2SAT and WASP. This suggests that different solvers are more sensitive to different aspects of the ASP program, and therefore require different configurations. To shed some light into the relevant parameters for the same solver across different domains, we analysed the configuration of WASP on the Graceful Graphs domain. In this case, the 5 most important parameters identified by the fANOVA tool are: (k8), (k2), (k9), (k18), (k7). In this case, it is easy to notice that (k1) is not deemed to be relevant for improving the performance of the solver, and only (k9) is relevant for WASP on both Graceful Graphs and Graph Colouring. It is worth reminding that the fANOVA analysis does not provide information on the actual value of the parameters, but only on the impact that a parameter can have on the performance of a solver. On this regards, we observed that also in cases where the same parameter is identified as very important by the fANOVA analysis, its best selected value by SMAC can be different in different domains.

Note that it is difficult to find a high level explanation on why the solvers benefit of this different order, since they present a complex structure, where each (small) change in the order of input might have an impact on several different components. However, in our view, this represents one of the strengths of our approach, since tools for automatic configuration might understand some hidden properties of the instances that are not immediately visible even to developers.

Discussion. ASP solvers are more complex than SAT solvers, since they have to deal with many additional aspects that are not present in SAT, such as the minimality property of answer sets, and additional constructs, as for example the aggregates. Albeit this additional complexity, we observed that reordering is beneficial for both LP2SAT and WASP, while the impact on the performance of CLASP is less noticeable. We conducted an additional analysis on the implementation of the solvers and we identified several reasons that can explain the results:

- The considered solvers do not work directly on the input program, since they use a technique, called Clark's Completion, that basically produces a propositional SAT formula which is then used to compute the answer sets of the original program. In this context, the impact of the reorder might be mitigated by the additional transformation made by the solvers.
- Data structures employed by the ASP solvers might deactivate some of the parameters used for the configuration. As an example, WASP uses a dedicated data structure for storing binary clauses which are then checked before other clauses, independently from their order in the input program. CLASP extends this special treatment also to ternary clauses. Concerning SAT solvers, as far as we know, only GLUCOSE has a similar data structure, whereas LINGELING and CADICAL have an efficient memory management of binary clauses but they do not impose an order among the clauses.
- Our tool considers the solvers as black boxes and uses their default configurations. Most of them employ parameters that were tuned on instances without specific

https://doi.org/10.1017/S1471068422000096 Published online by Cambridge University Press

reordering. Different results might be obtained by employing other configurations or by tuning the heuristics with instances processed by our reordering tool. However, from our perspective, the configuration tool can be already incorporated into grounders/preprocessors as an additional feature, to have a configured instance given to the solvers.

- Differently from other ASP and SAT solvers, the default configuration of CLASP uses the Maximum Occurrence of clauses of Minimum size (MOMS) heuristic to initialize its heuristic parameters. This led to a more uniform behaviour when it is executed on the same instance with different orders. We observed that the behaviour of WASP is much more dependent on the order of the instances as confirmed in our experiment and as also shown in Section 4.4.

4.4 Synthetic experiment

In this section, we report the results of an experimental analysis conducted on a synthetic benchmark. Goal of the experiment is to give an explanation of the different performance between CLASP and WASP, and investigate the different qualitative results achieved in SAT and ASP. As a side effect, we show that it is possible to improve the performance of the ASP solver WASP using a proper ordering of the input program.

In particular, we focused on the following (synthetic) problem:

$$
\begin{array}{lll}
r_1: & \{in(i,j) \mid j \in \{1,\ldots,h\}\} \leftarrow & \forall i \in \{1,\ldots,p\} \\
r_2: & \leftarrow \#count\{in(i,j) \mid j \in \{1,\ldots,h\}\} \neq 1 & \forall i \in \{1,\ldots,p\} \\
r_3: & \leftarrow in(i_1,j), in(i_2,j), i_1 < i_2 & \forall i_1,i_2 \in \{1,\ldots,p\}, j \in \{1,\ldots,h\} \\
r_4: & col(i,c_1) \vee col(i,c_2) \vee \ldots \vee col(i,c_k) \leftarrow & \forall i \in \{1,\ldots,n\} \\
r_5: & \leftarrow edge(i_1,i_2), col(i_1,c), col(i_2,c) & \forall i_1,i_2 \in \{1,\ldots,n\}, c \in \{c_1,c_2,\ldots,c_k\} \\
r_6: & edge(i,j) \leftarrow & \forall i,j \in \{1,\ldots,n\}, i \neq j,
\end{array}
$$

where rules r_1, r_2 and r_3 encode the pigeonhole problem with p pigeons and h holes, rules r_4 and r_5 encode the k-graph colouring problem, and r_6 encodes a complete graph with n nodes provided as input to the graph colouring problem. It is possible to observe that r_1, r_2, and r_3 admit no stable model when $p > h$, whereas r_4, r_5, and r_6 admit no stable model when $n > k$. Concerning the pigeonhole problem, it is well known that the performance of CDCL and resolution-based solvers are poor when $p > h$ and p is greater than a given threshold (Biere *et al*. 2009; Haken 1985). For instance, CLASP terminates after 1.51 s when $h = 9$ and $p = 10$, after 17.81 s when $h = 10$ and $p = 11$, and it does not terminate within 5 min when $h = 11$ and $p = 12$. Similarly, concerning the k-graph colouring problem, large values of k (e.g. $k \geq 10$) with $n > k$ are associated with poor performance of the solver. Such properties are important in our case, since we are now able to control the hardness of the instances by properly selecting values of h, p, k, and n. In particular, if the two sub-problems are combined, that is, when the solver is executed on rules from r_1 to r_6, then we are able to create hard and easy sub-programs. For instance, if we consider the case with $h = 11$, $p = 12$, $k = 5$, and $n = 100$, then we have that the rules from r_1 to r_6 admit no stable model, which is hard to prove for rules from r_1 to r_3 and easy to prove for rules from r_4 to r_6. In this case the performance of the solver depends on the sub-program considered at hand. As noted in Section 4, the

https://doi.org/10.1017/S1471068422000096 Published online by Cambridge University Press

heuristic of CLASP is not dependent on the ordering of the input program and, in this case, it is able to automatically focus on the easy subprogram. On the other hand, we observed that the performance of WASP depends on the processed order of the variables. This behaviour of WASP can be explained by looking at its branching heuristic, which first selects literals with the lowest ids and then it focuses on the sub-problem related to such literals. Clearly, this might lead to poor performance on the programs described above if the hard sub-problem is considered first.

In the following we show that the performance of WASP can be improved by performing an additional step after that the program has been configured, that is, ids of the literals can be sorted according to the value of O_l in descending order. In particular, we report the results of an experimental analysis conducted on instances of the rules r_1–r_6, where $h = 10$, $k = 5$, $p = [20, \ldots, 40]$ and $n = [7, \ldots, 29]$. Overall, we considered 483 instances, where 433 were used for the configuration and 50 were used for the testing. Results show that WASP without configuration solves 33 instances out of 50 with a PAR10 equal to 2108.63, whereas WASP after the configuration solves 39 instances out of 50 with a PAR10 equal to 1351.94.

It is important to emphasise that the results are obtained without changing the implementation of the solver. Such changes might be directly included in the grounders (e.g. as additional parameter of the system DLV (Alviano *et al*. 2017)) or in specific tools dedicated to preprocess the input programs.

5 Related work

In both SAT and ASP there have been numerous papers where machine-learning-based configuration techniques based on features computation have been employed. Traditionally, such approaches aimed at modifying the behaviour of the solvers by either configuring their exposed parameters, or by combining different solvers into portfolios. In this work we consider an orthogonal perspective, where we configure the way in which the input knowledge, that is, formula or program, is presented to the solver. This is done with the idea that the way in which instances are formulated and ordered can carry some knowledge about the underlying structure of the problem to be solved. In the following, we present main related literature in SAT and ASP, in two different paragraphs.

SAT. A significant amount of work in the area has focused on approaches for configuring the exposed parameters of SAT solvers in order to affect their behaviour (Eggensperger *et al*. 2019). Well-known examples include the use of PARAMILS for configuring SAPS and SPEAR (Hutter *et al*. 2009), and of REACTR for configuring LINGELING (Fitzgerald *et al*. 2015). Some approaches also looked into the generation of instance-specific configurations of solvers (Kadioglu *et al*. 2010b). This line of work led to the development of dedicated tools, such as SPYSMAC (Falkner *et al*. 2015) and CAVE (Biedenkapp *et al*. 2018), and to the organisation of the dedicated Configurable SAT Solver Challenge (Hutter *et al*. 2017). It also lead to the design and development of solvers, such as SATENSTEIN (KhudaBukhsh *et al*. 2016), that are very modular and natively support the use of configuration to combine all the relevant modules. A large body of works also focused on techniques for automatically configuring portfolios of solvers, such as SATZILLA (Xu *et al*. 2008), based on the use of empirical prediction models of the

https://doi.org/10.1017/S1471068422000096 Published online by Cambridge University Press

performance of the considered solvers on the instance to be solved (Hutter *et al*. 2014). Tools for assessing the contribution of different solvers to a portfolio has been introduced (Xu *et al*. 2012). With regard to portfolio generation, the interested reader is referred to Hurley *et al*. (2016). The use of configuration techniques to generate portfolios of solvers has also been extensively investigated: HYDRA (Xu *et al*. 2010) builds a set of solvers with complementary strengths by iteratively configuring new algorithms; AUTOFOLIO (Lindauer *et al*. 2015) uses algorithm selection to optimise the performance of algorithm selection systems by determining the best selection approach and its hyperparameters; finally, in Lindauer *et al*. (2017) an approach based on algorithm configuration for the automated construction of parallel portfolios has been introduced. The approach we follow relates also to the problem of discovering a backdoor, that is, an ordering that will allow the problem to be solved faster, see, for example Kilby *et al*. (2005). However, it has to be noted that this is not a characteristic of our approach to configuration, but common to many approaches mentioned above.

ASP. Inspired by the solver SATZILLA in the area of SAT, the CLASPFOLIO system (Gebser *et al*. 2011; Hoos *et al*. 2014) uses support vector regression to learn scoring functions approximating the performance of several CLASP variants in a training phase. Given an instance, CLASPFOLIO then extracts features and evaluates such functions in order to pick the most promising CLASP variant for solving the instance. This algorithm selection approach was particularly successful in the Third ASP Competition (Calimeri *et al*. 2014), held in 2011, where CLASPFOLIO won the first place in the NP category and the second place overall (without participating in the BeyondNP category). The MEASP system (Maratea *et al*. 2014; 2015b) goes beyond the solver-specific setting of CLASPFOLIO and chooses among different grounders as well as solvers. Grounder selection traces back to Maratea *et al*. (2013), and similar to the QBF solver AQME (Pulina and Tacchella 2009), MEASP uses a classification method for performance prediction. Notably, "bad" classifications can be treated by adding respective instances to the training set of MEASP (Maratea *et al*. 2015a), which enables an adjustment to new problems or instances thereof. Some of the parameters reported in Section 3.2 were also adopted by CLASPFOLIO and MEASP, since they were recognised to be important to discriminate the properties of the input program.In the Seventh ASP Competition (Gebser *et al*. 2017), the winning system was I-DLV+S (Calimeri *et al*. 2020) that utilises I-DLV (Calimeri *et al*. 2017) for grounding and automatically selects back-ends for solving through classification between CLASP and WASP. Going beyond the selection of a single solving strategy from a portfolio, the ASPEED system (Hoos *et al*. 2015) indeed runs different solvers, sequentially or in parallel, as successfully performed by PPFOLIO. Given a benchmark set, a fixed time limit per instance, and performance results for candidate solvers, the idea of ASPEED is to assign time budgets to the solvers such that a maximum number of instances can be completed within the allotted time. In other words, the goal is to divide the total runtime per computing core among solvers such that the number of instances on which at least one solver successfully completes its run is maximised. The portfolio then consists of all solvers assigned a non-zero time budget along with a schedule of solvers to run on the same computing core. Calculating such an optimal portfolio for a benchmark set is an Optimisation problem addressed with ASP in ASPEED. In Dingess and Truszczynski (2020), an approach for

https://doi.org/10.1017/S1471068422000096 Published online by Cambridge University Press

encoding selection was presented as a strategy for improving the performance of answer set solvers. In particular, the idea is to create an automated process for generating alternative encodings. The presented tool, called Automated Aggregator, was able to handle non-ground count aggregates. Other automatic non-ground program rewriting techniques are presented in Hippen and Lierler (2019).

Further, there has been a recent growing interest in techniques for knowledge model configuration in the areas of AI Planning and Abstract Argumentation. In AI Planning, it has been demonstrated that the configuration of either the domain model (Vallati *et al.* 2015; 2021) or the problem model (Vallati and Serina 2018) can lead to significant performance improvement for domain-independent planning engines. On the argumentation side, it has been shown that even on syntactically simple models that represent directed graphs, the configuration process can lead to performance improvements (Cerutti *et al.* 2018).

6 Conclusions

In this paper, we proposed an approach for exploiting the fact that the order in which the main elements of CNF formulae and ASP programs, that is, literals, clauses, and rules, are listed carries some information about the structure of the problem to be solved, and therefore affect the performance of solvers. The proposed approach allows to perform the automated configuration of formulae and programs. In SAT, we considered as configurable the order in which clauses are listed and the order in which literals are listed in the clauses, while for ASP we considered as configurable similar entities, that is, literals and rules, but taking into account that some rule's structure has to be maintained, and that other powerful constructs like aggregates come into play. In our experimental analysis we configured formulae and programs for improving number of solved instances and PAR10 performance of solvers. The performed analysis, aimed at investigating how the configuration affects the performance of state-of-the-art solvers: (i) demonstrates that the automated configuration has a significant impact on solvers' performance, more evident for SAT but also significant for ASP, despite the constraints on rule's structure; (ii) indicates that the configuration should be performed on specific set of benchmarks for a given solver; and (iii) highlights what are the main features and aspects of formulae and programs that have a potentially strong impact on the performance of solvers. Such features can be taken into account by knowledge engineers to encode formulae or programs in a way that supports solvers that will reason upon them.

Our findings can have implications on both solving and encoding side. Given the positive results obtained, solver's developers should consider rearranging internally the structure of the formula/program in order to optimise their performance, and/or users could their-selves consider these hints while writing the encoding. However, given that such positive results are obtained per-domain, and varies with solvers, care should be taken in doing so.

We see several avenues for future work. We plan to evaluate the impact of configuration on optimisation variants of SAT and ASP, that is, weighted max SAT, or ASP including soft constraints, where the weight of the elements can provide another important information to the configuration process. We are also interested in evaluating if ordering

https://doi.org/10.1017/S1471068422000096 Published online by Cambridge University Press

clauses (and literals) that are learnt during the search process of a SAT solver can be beneficial for improving performance, given that all of the solvers employed are based on (variant of) the CDCL algorithm. In this paper, we focused on exact solvers based on CDCL, as future work it can be interesting to consider also SAT solvers based on local search. Concerning ASP solvers, it can be also interesting to check if reordering the non-ground program can have a positive impact on the performance. This would also open the combination of reordering and systems based on lazy-grounding (Weinzierl *et al*. 2020). Another interesting future work can be to investigate the joint tuning of ordering and solver parameters. Moreover, in this paper we focused on parameters that are based on our knowledge of existing solvers, involving parameters, among others, that proved to be important to characterised input programs; of course, the inclusion of additional parameters is possible, for example, taking into account theoretical properties of the input programs (see, e.g., (Fichte *et al*. 2015; Janhunen 2006; Lifschitz *et al*. 2001; Gebser and Schaub 2013)), or adding symmetric counterparts of current criteria, for example, (c6) and (c7) of the input formula. Finally, we plan to incorporate the re-ordering into existing approaches for configuring portfolios of SAT solvers, such as SATenstein, which works in a similar way as ASPEED, but differently from ASPEED is not tailored on different configurations of the same solver, in order to further improve performance, and to investigate the concurrent configuration of formulae/programs and solvers.

Conflicts of interest

The authors declare none.

References

Alviano, M., Calimeri, F., Dodaro, C., Fuscà, D., Leone, N., Perri, S., Ricca, F., Veltri, P. and Zangari, J. 2017. The ASP system DLV2. In *LPNMR*. Lecture Notes in Computer Science, vol. 10377. Springer, 215–221.

Alviano, M., Dodaro, C., Leone, N. and Ricca, F. 2015. Advances in WASP. In *Logic Programming and Nonmonotonic Reasoning - 13th International Conference, LPNMR 2015, Lexington, KY, USA, 27–30 September 2015. Proceedings*, F. Calimeri, G. Ianni and M. Truszczynski, Eds. Lecture Notes in Computer Science, vol. 9345. Springer, 40–54.

Ansótegui, C., Sellmann, M. and Tierney, K. 2009. A gender-based genetic algorithm for the automatic configuration of algorithms. In *Principles and Practice of Constraint Programming - CP 2009, 15th International Conference, CP 2009, Lisbon, Portugal, 20–24 September 2009, Proceedings*, I. P. Gent, Ed. Lecture Notes in Computer Science, vol. 5732. Springer, 142–157.

Audemard, G., Lagniez, J. and Simon, L. 2013. Improving glucose for incremental SAT solving with assumptions: Application to MUS extraction. In *Theory and Applications of Satisfiability Testing - SAT 2013 - 16th International Conference, Helsinki, Finland, 8–12 July 2013. Proceedings*, M. Järvisalo and A. V. Gelder, Eds. Lecture Notes in Computer Science, vol. 7962. Springer, 309–317.

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

Biedenkapp, A., Marben, J., Lindauer, M. and Hutter, F. 2018. CAVE: Configuration assessment, visualization and evaluation. In *Learning and Intelligent Optimization - 12th International Conference, LION*. 115–130.

https://doi.org/10.1017/S1471068422000096 Published online by Cambridge University Press

Biere, A. 2017. Cadical, lingeling, plingeling, treengeling and yalsat entering the SAT competition 2017. In *SAT Competition 2017, Solver and Benchmark Descriptions*.

Biere, A., Heule, M., van Maaren, H. and Walsh, T., Eds. 2009. *Handbook of Satisfiability*. Frontiers in Artificial Intelligence and Applications, vol. 185. IOS Press.

Breiman, L. 2001. Random forests. *Machine Learning 45,* 1, 5–32.

Brewka, G., Eiter, T. and Truszczynski, M. 2011. Answer set programming at a glance. *Communications of the ACM 54,* 12, 92–103.

Calimeri, F., Dodaro, C., Fuscà, D., Perri, S. and Zangari, J. 2020. Efficiently coupling the I-DLV grounder with ASP solvers. *Theory and Practice of Logic Programming 20,* 2, 205–224.

Calimeri, F., Faber, W., Gebser, M., Ianni, G., Kaminski, R., Krennwallner, T., Leone, N., Maratea, M., Ricca, F. and Schaub, T. 2020. Asp-core-2 input language format. *Theory and Practice of Logic Programming 20,* 2, 294–309.

Calimeri, F., Fuscà, D., Perri, S. and Zangari, J. 2017. I-DLV: The new intelligent grounder of DLV. *Intelligenza Artificiale 11,* 1, 5–20.

Calimeri, F., Ianni, G. and Ricca, F. 2014. The third open answer set programming competition. *Theory and Practice of Logic Programming 14,* 1, 117–135.

Cerutti, F., Vallati, M. and Giacomin, M. 2018. On the impact of configuration on abstract argumentation automated reasoning. *International Journal of Approximate Reasoning 92*, 120–138.

Dingess, M. and Truszczynski, M. 2020. Automated aggregator - rewriting with the counting aggregate. In *Proceedings 36th International Conference on Logic Programming (Technical Communications), ICLP Technical Communications 2020, (Technical Communications) UNICAL, Rende (CS), Italy, 18–24 September 2020*, F. Ricca, A. Russo, S. Greco, N. Leone, A. Artikis, G. Friedrich, P. Fodor, A. Kimmig, F. A. Lisi, M. Maratea, A. Mileo and F. Riguzzi, Eds. EPTCS, vol. 325, 96–109.

Eggensperger, K., Lindauer, M. and Hutter, F. 2019. Pitfalls and best practices in algorithm configuration. *Journal of Artificial Intelligence Research 64*, 861–893.

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

Faber, W., Pfeifer, G. and Leone, N. 2011. Semantics and complexity of recursive aggregates in answer set programming. *Artificial Intelligence 175,* 1, 278–298.

Falkner, S., Lindauer, M. and Hutter, F. 2015. Spysmac: Automated configuration and performance analysis of SAT solvers. In *Theory and Applications of Satisfiability Testing - SAT 2015*, 215–222.

Fichte, J. K., Truszczynski, M. and Woltran, S. 2015. Dual-normal logic programs - the forgotten class. *Theory and Practice of Logic Programming 15,* 4-5, 495–510.

Fitzgerald, T., Malitsky, Y. and O'Sullivan, B. 2015. Reactr: Realtime algorithm configuration through tournament rankings. In *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, 25–31 July 2015*, Q. Yang and M. J. Wooldridge, Eds. AAAI Press, 304–310.

Gebser, M., Kaminski, R., Kaufmann, B., Schaub, T., Schneider, M. T. and Ziller, S. 2011. A portfolio solver for answer set programming: Preliminary report. In *Logic Programming and Nonmonotonic Reasoning - 11th International Conference, LPNMR 2011, Vancouver, Canada, 16–19 May 2011. Proceedings*, J. P. Delgrande and W. Faber, Eds. Lecture Notes in Computer Science, vol. 6645. Springer, 352–357.

Gebser, M., Kaufmann, B. and Schaub, T. 2012. Conflict-driven answer set solving: From theory to practice. *Artificial Intelligence 187*, 52–89.

Gebser, M., Maratea, M. and Ricca, F. 2017. The sixth answer set programming competition. *Journal of Artificial Intelligence Research 60*, 41–95.

Gebser, M., Maratea, M. and Ricca, F. 2020. The seventh answer set programming competition: Design and results. *Theory and Practice of Logic Programming 20,* 2, 176–204.

Gebser, M. and Schaub, T. 2013. Tableau calculi for logic programs under answer set semantics. *ACM Transactions on Computational Logic 14,* 2, 15:1–15:40.

Geffner, H. 2018. Model-free, model-based, and general intelligence. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, 13–19 July 2018, Stockholm, Sweden*, J. Lang, Ed. ijcai.org, 10–17.

Gelfond, M. and Lifschitz, V. 1988. The stable model semantics for logic programming. In *Logic Programming, Proceedings of the Fifth International Conference and Symposium, Seattle, Washington, USA, 15–19 August 1988 (2 Volumes)*, R. A. Kowalski and K. A. Bowen, Eds. MIT Press, 1070–1080.

Gelfond, M. and Lifschitz, V. 1991. Classical negation in logic programs and disjunctive databases. *New Generation Computing 9,* 3/4, 365–386.

Gomes, C. P., Selman, B., Crato, N. and Kautz, H. A. 2000. Heavy-tailed phenomena in satisfiability and constraint satisfaction problems. *Journal of Automated Reasoning 24,* 1/2, 67–100.

Haken, A. 1985. The intractability of resolution. *Theoretical Computer Science 39*, 297–308.

Heule, M. J. H., Järvisalo, M. and Suda, M. 2019. SAT competition 2018. *Journal on Satisfiability, Boolean Modeling and Computation 11,* 1, 133–154.

Hippen, N. and Lierler, Y. 2019. Automatic program rewriting in non-ground answer set programs. In *Practical Aspects of Declarative Languages - 21th International Symposium, PADL 2019, Lisbon, Portugal, 14–15 January 2019, Proceedings*, J. J. Alferes and M. Johansson, Eds. Lecture Notes in Computer Science, vol. 11372. Springer, 19–36.

Hoos, H. H., Kaminski, R., Lindauer, M. and Schaub, T. 2015. aspeed: Solver scheduling via answer set programming. *Theory and Practice of Logic Programming 15,* 1, 117–142.

Hoos, H. H., Lindauer, M. and Schaub, T. 2014. claspfolio 2: Advances in algorithm selection for answer set programming. *Theory and Practice of Logic Programming 14,* 4-5, 569–585.

Hurley, B., Kotthoff, L., Malitsky, Y., Mehta, D. and O'Sullivan, B. 2016. Advanced portfolio techniques. In *Data Mining and Constraint Programming - Foundations of a Cross-Disciplinary Approach*. Springer, 191–225.

Hutter, F., Hoos, H. H. and Leyton-Brown, K. 2010. Tradeoffs in the empirical evaluation of competing algorithm designs. *Annals of Mathematics and Artificial Intelligence 60,* 1-2, 65–89.

Hutter, F., Hoos, H. H. and Leyton-Brown, K. 2011. Sequential model-based optimization for general algorithm configuration. In *Learning and Intelligent Optimization - 5th International Conference, LION 5, Rome, Italy, 17–21 January 2011. Selected Papers*, C. A. C. Coello, Ed. Lecture Notes in Computer Science, vol. 6683. Springer, 507–523.

Hutter, F., Hoos, H. H. and Leyton-Brown, K. 2014. An efficient approach for assessing hyperparameter importance. In *Proceedings of the 31th International Conference on Machine Learning, ICML 2014, Beijing, China, 21–26 June 2014*. JMLR Workshop and Conference Proceedings, vol. 32. JMLR.org, 754–762.

Hutter, F., Hoos, H. H., Leyton-Brown, K. and Stützle, T. 2009. Paramils: An automatic algorithm configuration framework. *Journal of Artificial Intelligence Research 36*, 267–306.

Hutter, F., Lindauer, M., Balint, A., Bayless, S., Hoos, H. H. and Leyton-Brown, K. 2017. The configurable SAT solver challenge (CSSC). *Artificial Intelligence 243*, 1–25.

Hutter, F., Xu, L., Hoos, H. H. and Leyton-Brown, K. 2014. Algorithm runtime prediction: Methods & evaluation. *Artificial Intelligence 206*, 79–111.

Janhunen, T. 2006. Some (in)translatability results for normal logic programs and propositional theories. *Journal of Applied Non-Classical Logics 16,* 1-2, 35–86.

Janhunen, T. 2018. Cross-translating answer set programs using the ASPTOOLS collection. *Künstliche Intelligenz 32,* 2-3, 183–184.

Kadioglu, S., Malitsky, Y., Sellmann, M. and Tierney, K. 2010a. ISAC - instance-specific algorithm configuration. In *ECAI 2010 - 19th European Conference on Artificial Intelligence, Lisbon, Portugal, 16–20 August 2010, Proceedings*, H. Coelho, R. Studer and M. J. Wooldridge, Eds. Frontiers in Artificial Intelligence and Applications, vol. 215. IOS Press, 751–756.

Kadioglu, S., Malitsky, Y., Sellmann, M. and Tierney, K. 2010b. Isac-instance-specific algorithm configuration. In *Proceedings of the European Conference on AI*, vol. 215, 751–756.

KhudaBukhsh, A. R., Xu, L., Hoos, H. H. and Leyton-Brown, K. 2016. Satenstein: Automatically building local search SAT solvers from components. *Artificial Intelligence 232*, 20–42.

Kilby, P., Slaney, J. K., Thiébaux, S. and Walsh, T. 2005. Backbones and backdoors in satisfiability. In *Proceedings of the Twentieth National Conference on Artificial Intelligence, AAAI 2005*, M. M. Veloso and S. Kambhampati, Eds. AAAI Press/The MIT Press, 1368–1373.

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

Lindauer, M., Hoos, H. H., Hutter, F. and Schaub, T. 2015. Autofolio: An automatically configured algorithm selector. *Journal of Artificial Intelligence Research 53*, 745–778.

Lindauer, M., Hoos, H. H., Leyton-Brown, K. and Schaub, T. 2017. Automatic construction of parallel portfolios via algorithm configuration. *Artificial Intelligence 244*, 272–290.

Maratea, M., Pulina, L. and Ricca, F. 2013. Automated selection of grounding algorithm in answer set programming. In *AI*IA 2013: Advances in Artificial Intelligence - XIIIth International Conference of the Italian Association for Artificial Intelligence, Turin, Italy, 4–6 December 2013. Proceedings*, M. Baldoni, C. Baroglio, G. Boella and R. Micalizio, Eds. Lecture Notes in Computer Science, vol. 8249. Springer, 73–84.

Maratea, M., Pulina, L. and Ricca, F. 2014. A multi-engine approach to answer-set programming. *Theory and Practice of Logic Programming 14,* 6, 841–868.

Maratea, M., Pulina, L. and Ricca, F. 2015a. Multi-engine ASP solving with policy adaptation. *Journal of Logic and Computation 25,* 6, 1285–1306.

Maratea, M., Pulina, L. and Ricca, F. 2015b. Multi-level algorithm selection for ASP. In *Logic Programming and Nonmonotonic Reasoning - 13th International Conference, LPNMR 2015, Lexington, KY, USA, 27–30 September 2015. Proceedings*, F. Calimeri, G. Ianni and M. Truszczynski, Eds. Lecture Notes in Computer Science, vol. 9345. Springer, 439–445.

Mitchell, D. G. 2005. A SAT solver primer. *Bulletin of the EATCS 85*, 112–132.

Pulina, L. and Tacchella, A. 2009. A self-adaptive multi-engine solver for quantified boolean formulas. *Constraints - An International Journal 14,* 1, 80–116.

Syrjänen, T. 2002. Lparse 1.0 user's manual.

Vallati, M., Chrpa, L., McCluskey, T. L. and Hutter, F. 2021. On the importance of domain model configuration for automated planning engines. *Journal of Automated Reasoning 65,* 6, 727–773.

Vallati, M., Hutter, F., Chrpa, L. and McCluskey, T. L. 2015. On the effective configuration of planning domain models. In *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, 25–31 July 2015*, Q. Yang and M. J. Wooldridge, Eds. AAAI Press, 1704–1711.

Vallati, M. and Maratea, M. 2019. On the configuration of SAT formulae. In *AI*IA 2019 - Advances in Artificial Intelligence - XVIIIth International Conference of the Italian Association for Artificial Intelligence, Rende, Italy, 19–22 November 2019, Proceedings*, M. Alviano, G. Greco and F. Scarcello, Eds. Lecture Notes in Computer Science, vol. 11946. Springer, 264–277.

Vallati, M. and Serina, I. 2018. A general approach for configuring PDDL problem models. In *Proceedings of the Twenty-Eighth International Conference on Automated Planning and*

Scheduling, ICAPS 2018, Delft, The Netherlands, 24–29 June 2018, M. de Weerdt, S. Koenig, G. Röger and M. T. J. Spaan, Eds. AAAI Press, 431–436.

Weinzierl, A., Taupe, R. and Friedrich, G. 2020. Advancing lazy-grounding ASP solving techniques - restarts, phase saving, heuristics, and more. *Theory and Practice of Logic Programming 20,* 5, 609–624.

Xu, L., Hoos, H. H. and Leyton-Brown, K. 2010. Hydra: Automatically configuring algorithms for portfolio-based selection. In *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2010, Atlanta, Georgia, USA, 11–15 July 2010*, M. Fox and D. Poole, Eds. AAAI Press.

Xu, L., Hutter, F., Hoos, H. H. and Leyton-Brown, K. 2008. Satzilla: Portfolio-based algorithm selection for SAT. *Journal of Artificial Intelligence Research 32*, 565–606.

Xu, L., Hutter, F., Hoos, H. H. and Leyton-Brown, K. 2012. Evaluating component solver contributions to portfolio-based algorithm selectors. In *Theory and Applications of Satisfiability Testing - SAT 2012*, 228–241.

Yuan, Z., Stützle, T. and Birattari, M. 2010. Mads/f-race: Mesh adaptive direct search meets f-race. In *Trends in Applied Intelligent Systems - 23rd International Conference on Industrial Engineering and Other Applications of Applied Intelligent Systems, IEA/AIE 2010, Cordoba, Spain, 1–4 June 2010, Proceedings, Part I*, N. García-Pedrajas, F. Herrera, C. Fyfe, J. M. Benítez and M. Ali, Eds. Lecture Notes in Computer Science, vol. 6096. Springer, 41–50.