*TLP* **14** (6): 841–868, 2014. © Cambridge University Press 2013
doi:10.1017/S1471068413000094 First published online 15 August 2013


# *A multi-engine approach to answer-set programming**

MARCO MARATEA
*DIBRIS, Università degli Studi di Genova, Viale F. Causa 15, 16145 Genova, Italy*
(*e-mail:* marco@dist.unige.it)

LUCA PULINA
*POLCOMING, Università degli Studi di Sassari, Viale Mancini 5, 07100 Sassari, Italy*
(*e-mail:* lpulina@uniss.it)

FRANCESCO RICCA
*Dipartimento di Matematica, Università della Calabria, Via P. Bucci, 87030 Rende, Italy*
(*e-mail:* ricca@mat.unical.it)

*submitted 12 July 2012; revised 18 December 2012; accepted 4 June 2013*

## Abstract

Answer-set programming (ASP) is a truly declarative programming paradigm proposed in the area of non-monotonic reasoning and logic programming, which has been recently employed in many applications. The development of efficient ASP systems is, thus, crucial. Having in mind the task of improving the solving methods for ASP, there are two usual ways to reach this goal: (i) extending state-of-the-art techniques and ASP solvers or (ii) designing a new ASP solver from scratch. An alternative to these trends is to build on top of state-of-the-art solvers, and to apply machine learning techniques for choosing automatically the "best" available solver on a per-instance basis.

In this paper, we pursue this latter direction. We first define a set of cheap-to-compute syntactic features that characterize several aspects of ASP programs. Then, we apply classification methods that, given the features of the instances in a *training* set and the solvers' performance on these instances, inductively learn algorithm selection strategies to be applied to a *test* set. We report the results of a number of experiments considering solvers and different training and test sets of instances taken from the ones submitted to the "System Track" of the Third ASP Competition. Our analysis shows that by applying machine learning techniques to ASP solving, it is possible to obtain very robust performance: our approach can solve more instances compared with any solver that entered the Third ASP Competition.

*KEYWORDS*: answer-set programming, automated algorithm selection, multi-engine solvers

* This is an extended and revised version of Maratea *et al.* (2012a, 2012b).

https://doi.org/10.1017/S1471068413000094 Published online by Cambridge University Press

## 1 Introduction

Answer-set programming (ASP) (Gelfond and Lifschitz 1988; Gelfond and Lifschitz 1991; Eiter *et al.* 1997; Marek and Truszczyński 1998; Niemelä 1998; Lifschitz 1999; Baral 2003) is a truly declarative programming paradigm proposed in the area of non-monotonic reasoning and logic programming. The idea of ASP is to represent a given computational problem by a logic program whose answer sets correspond to solutions, and then use a solver to find such solutions (Lifschitz 1999). The language of ASP is very expressive, indeed all problems in the second level of the polynomial hierarchy can be expressed in ASP (Eiter *et al.* 1997). Moreover, in the last years ASP has been employed in many applications (Nogueira *et al.* 2001; Baral 2003; Brooks *et al.* 2007; Friedrich and Ivanchenko 2008; Gebser *et al.* 2011; Balduccini and Lierler 2012), and even in industry (Ricca *et al.* 2009, 2010, 2012; Rullo *et al.* 2009; Marczak *et al.* 2010; Smaragdakis *et al.* 2011). The development of efficient ASP systems is, thus, a crucial task, made even more challenging by existing and new-coming applications.

Having in mind the task of improving the robustness, i.e., the ability to perform well across a wide set of problem domains, and the efficiency, i.e., the quality of solving a high number of instances, of solving methods for ASP, it is possible to extend existing state-of-the-art techniques implemented in ASP solvers, or design from scratch a new ASP system with powerful techniques and heuristics. An alternative to these trends is to build on top of state-of-the-art solvers, leveraging on a number of efficient ASP systems (Simons *et al.* 2002; Leone *et al.* 2006; Giunchiglia *et al.* 2006; Gebser *et al.* 2007; Mariën *et al.* 2008; Janhunen *et al.* 2009), and applying machine learning techniques for inductively choosing, among a set of available ones, the "best" solver on the basis of the characteristics, called *features*, of the input program. This approach falls in the framework of the *algorithm selection problem* (Rice 1976). Related approaches, following this per-instance selection, have been exploited for solving propositional satisfiability (SAT), (Xu *et al.* 2008), and Quantified SAT (QSAT) (Pulina and Tacchella 2007) problems. In ASP, an approach for selecting the "best" CLASP internal configuration is followed in Gebser *et al.* (2011), while another approach that imposes learned heuristics ordering to SMODELS is Balduccini (2011).

In this paper, we pursue this direction and propose a multi-engine approach to ASP solving. We first define a set of cheap-to-compute syntactic features that describe several characteristics of ASP programs, paying particular attention to ASP peculiarities. We then compute such features for the grounded version of all benchmarks submitted to the "System Track" of the Third ASP Competition (Calimeri *et al.* 2012) falling in the "*NP*" and "*Beyond NP*" categories of the competition: this track is well suited for our study given that (i) contains many ASP instances, (ii) the language specification, ASP-Core, is a common ASP fragment such that (iii) many ASP systems can deal with it.

Then, we apply classification methods that, starting from the features of the instances in a *training* set, and the solvers' performance on these instances, inductively learn general algorithm selection strategies to be applied to a *test* set. We consider

https://doi.org/10.1017/S1471068413000094 Published online by Cambridge University Press

six well-known multinomial classification methods, some of them considered in Pulina and Tacchella (2007). We perform a number of analyses considering different training and test sets. Our experiments show that it is possible to obtain a very robust performance, by solving many more instances than all the solvers that entered the Third ASP Competition and DLV (Leone *et al.* 2006).

This paper is structured as follows. Section 2 contains preliminaries about ASP and classification methods. Section 3 then describes our benchmark setting, in terms of dataset and solvers employed. Section 4 defines how features and solvers have been selected, and presents the classification methods employed. Section 5 is dedicated to the performance analysis, while Sections 6 and 7 end the paper with discussion about related work and conclusions, respectively.

## 2 Preliminaries

In this section, we recall some preliminary notions concerning ASP and machine learning techniques for algorithm selection.

### 2.1 Answer-set programming

In the following, we recall both the syntax and semantics of ASP. The presented constructs are included in ASP-Core (Calimeri *et al.* 2012), which is the language specification that was originally introduced in the Third ASP Competition (Calimeri *et al.* 2012) as well as the one employed in our experiments (see Section 3). Hereafter, we assume the reader is familiar with logic programming conventions and refer the reader to Gelfond and Lifschitz (1991), Baral (2003), and Gelfond and Leone (2002) for complementary introductory material on ASP, and to Calimeri *et al.* (2011) for obtaining the full specification of ASP-Core.

**Syntax.** A variable or a constant is a *term*. An *atom* is $p(t_1, \ldots, t_n)$, where $p$ is a *predicate* of arity $n$ and $t_1, \ldots, t_n$ are terms. A *literal* is either a *positive literal* $p$ or a *negative literal* `not` $p$, where $p$ is an atom. A *(disjunctive) rule* $r$ is of the form:

$$a_1 \;\vee\; \cdots \;\vee\; a_n \;\text{:-}\; b_1, \ldots, b_k,\; \texttt{not}\;\; b_{k+1}, \ldots,\; \texttt{not}\;\; b_m.$$

where $a_1, \ldots, a_n, b_1, \ldots, b_m$ are atoms. The disjunction $a_1 \vee \ldots \vee a_n$ is the *head* of $r$, while the conjunction $b_1, \ldots, b_k, \texttt{not}\ b_{k+1}, \ldots, \texttt{not}\ b_m$ is the *body* of $r$. We denote by $H(r)$ the set of atoms occurring in the head of $r$, and we denote by $B(r)$ the set of body literals. A rule s.t. $|H(r)| = 1$ (i.e., $n = 1$) is called a *normal rule*; if the body is empty (i.e., $k = m = 0$) it is called a *fact* (and the :– sign is omitted); if $|H(r)| = 0$ (i.e., $n = 0$) is called a *constraint*. A rule $r$ is *safe* if each variable appearing in $r$ also appears in some positive body literal of $r$.

An *ASP program* $\mathscr{P}$ is a finite set of safe rules. A `not`-free (resp., $\vee$-free) program is called *positive* (resp., *normal*). A term, an atom, a literal, a rule, or a program is *ground* if no variable appears in it.

**Semantics.** Given a program $\mathscr{P}$, the *Herbrand Universe* $U_{\mathscr{P}}$ is the set of all constants appearing in $\mathscr{P}$, and the *Herbrand Base* $B_{\mathscr{P}}$ is the set of all possible ground atoms

https://doi.org/10.1017/S1471068413000094 Published online by Cambridge University Press

which can be constructed from the predicates appearing in $\mathscr{P}$ with the constants of $U_{\mathscr{P}}$. Given a rule $r$, $Ground(r)$ denotes the set of rules obtained by applying all possible substitutions from the variables in $r$ to elements of $U_{\mathscr{P}}$. Similarly, given a program $\mathscr{P}$, the *ground instantiation* of $\mathscr{P}$ is $Ground(\mathscr{P}) = \bigcup_{r \in \mathscr{P}} Ground(r)$.

An *interpretation* for a program $\mathscr{P}$ is a subset $I$ of $B_{\mathscr{P}}$. A ground positive literal $A$ is true (resp., false) w.r.t. $I$ if $A \in I$ (resp., $A \notin I$). A ground negative literal $\texttt{not}\ A$ is true w.r.t. $I$ if $A$ is false w.r.t. $I$; otherwise $\texttt{not}\ A$ is false w.r.t. $I$.

The answer sets of a program $\mathscr{P}$ are defined in two steps using its ground instantiation: first, the answer sets of positive disjunctive programs are defined; then, the answer sets of general programs are defined by a reduction to positive ones and a stability condition.

Let $r$ be a ground rule, the head of $r$ is true w.r.t. $I$ if $H(r) \cap I \neq \emptyset$. The body of $r$ is true w.r.t. $I$ if all body literals of $r$ are true w.r.t. $I$, otherwise the body of $r$ is false w.r.t. $I$. The rule $r$ is *satisfied* (or true) w.r.t. $I$ if its head is true w.r.t. $I$ or its body is false w.r.t. $I$.

Given a *ground positive* program $P_g$, an *answer set* for $P_g$ is a subset-minimal interpretation $A$ for $P_g$ such that every rule $r \in P_g$ is true w.r.t. $A$ (i.e., there is no other interpretation $I \subset A$ that satisfies all the rules of $P_g$).

Given a *ground* program $P_g$ and an interpretation $I$, the (Gelfond–Lifschitz) *reduct* (Gelfond and Lifschitz 1991) of $P_g$ w.r.t. $I$ is the positive program $P_g^I$, obtained from $P_g$ by (i) deleting all rules $r \in P_g$ whose negative body is false w.r.t. $I$, and (ii) deleting the negative body from the remaining rules of $P_g$.

An answer set (or stable model) of a general program $\mathscr{P}$ is an interpretation $I$ of $\mathscr{P}$ such that $I$ is an answer set of $Ground(\mathscr{P})^I$.

As an example consider the program $\mathscr{P} = \{\ a \vee b \text{ :- } c.,\ b \text{ :- } \texttt{not}\ a, \texttt{not}\ c.,\ a \vee c \text{ :- } \texttt{not}\ b.,\ k \text{ :- } a.,\ k \text{ :- } b.\ \}$ and $I = \{b, k\}$. The reduct $\mathscr{P}^I$ is $\{a \vee b \text{ :- } c.,\ b.\ k \text{ :- } a.,\ k \text{ :- } b.\}$. $I$ is an answer set of $\mathscr{P}^I$, and for this reason it is also an answer set of $\mathscr{P}$.

### 2.2 Multinomial classification for algorithm selection

With regard to empirically hard problems, there is rarely a best algorithm to solve a given combinatorial problem, while it is often the case that different algorithms perform well on different problem instances. In this work, we rely on a per-instance selection algorithm in which, given a set of *features* – i.e., numeric values that represent particular characteristics of a given instance – it is possible to choose the best (or a good) algorithm among a pool of them – in our case, ASP solvers. In order to make such a selection in an automatic way, we model the problem using *multinomial classification* algorithms, i.e., machine learning techniques that allow automatic classification of a set of instances, given some instance features.

In more detail, in multinomial classification we are given a set of patterns, i.e., input vectors $X = \{\underline{x}_1, \ldots \underline{x}_k\}$ with $\underline{x}_i \in \mathbb{R}^n$, and a corresponding set of labels, i.e., output values $Y \in \{1, \ldots, m\}$, where $Y$ is composed of values representing the $m$ classes of the multinomial classification problem. In our modeling, the $m$ classes are $m$ ASP solvers. We think to the labels as generated by some unknown function

https://doi.org/10.1017/S1471068413000094 Published online by Cambridge University Press

Table 1. *Problems and instances*

| Problem | Class | No. of instances |
|---|---|---|
| DisjunctiveScheduling | $NP$ | 10 |
| GraphColouring | $NP$ | 60 |
| HanoiTower | $NP$ | 59 |
| KnightTour | $NP$ | 10 |
| MazeGeneration | $NP$ | 50 |
| Labyrinth | $NP$ | 261 |
| MultiContextSystemQuerying | $NP$ | 73 |
| Numberlink | $NP$ | 150 |
| PackingProblem | $NP$ | 50 |
| SokobanDecision | $NP$ | 50 |
| Solitaire | $NP$ | 25 |
| WeightAssignmentTree | $NP$ | 62 |
| MinimalDiagnosis | *Beyond* $NP$ | 551 |
| StrategicCompanies | *Beyond* $NP$ | 51 |
| Total | | 1462 |

$f : \mathbb{R}^n \rightarrow \{1,\ldots,m\}$ applied to the patterns, i.e., $f(\underline{x}_i) = y_i$ for $i \in \{1,\ldots,k\}$ and $y_i \in \{1,\ldots,m\}$. Given a set of patterns $X$ and a corresponding set of labels $Y$, the task of a multinomial classifier $c$ is to extrapolate $f$ given $X$ and $Y$, i.e., construct $c$ from $X$ and $Y$ so that when we are given some $\underline{x}^\star \in X$ we should ensure that $c(\underline{x}^\star)$ is equal to $f(\underline{x}^\star)$. This task is called *training* and the pair $(X, Y)$ is called the *training set*.

## 3 Benchmark data and settings

In this section, we report the benchmark settings employed in this work, which is needed for properly introducing the techniques described in the remainder of the paper. In particular, we report some data concerning: benchmark problems, instances, and ASP solvers employed, as well as the hardware platform, and the execution settings for reproducibility of experiments.

### *3.1 Dataset*

The benchmarks considered for the experiments belong to the suite of the Third ASP Competition (Calimeri *et al.* 2011). This is a large and heterogeneous suite of hard benchmarks encoded in ASP-Core, which was already employed for evaluating the performance of state-of-the-art ASP solvers. That suite includes planning domains, temporal and spatial scheduling problems, combinatorial puzzles, graph problems, and a number of application domains, i.e., databases, information extraction, and

https://doi.org/10.1017/S1471068413000094 Published online by Cambridge University Press

molecular biology field.[1] In more detail, we have employed the encodings used in the System Track of the competition, and all the problem instances made *available* (in form of facts) from the contributors of the problem submission stage of the competition, which are available from the competition website (Calimeri *et al.* 2011). Note that this is a superset of the instances actually selected for running (and thus *evaluated* in) the competition itself. Hereafter, with *instance* we refer to the complete input program (i.e., encoding+facts) to be fed to a solver for each instance of the problem to be solved.

The techniques presented in this paper are conceived for dealing with propositional programs; thus, we have grounded all the mentioned instances by using GRINGO (v.3.0.3) (Gebser *et al.* 2007) to obtain a setup very close to the one of the competition. We considered only computationally hard benchmarks, corresponding to all problems belonging to the categories *NP* and *Beyond NP* of the competition. The dataset is summarized in Table 1, which also reports the complexity classification and the number of available instances for each problem.

### *3.2 Executables and hardware settings*

We have run all the ASP solvers that entered the System Track of the Third ASP Competition (Calimeri *et al.* 2011) with the addition of DLV (Leone *et al.* 2006) (which did not participate in the competition since it is developed by the organizers of the event). In this way, we have covered – to the best of our knowledge – all the state-of-the-art solutions fitting the benchmark settings. In detail, we have run: CLASP (Gebser *et al.* 2007), CLASPD (Drescher *et al.* 2008), CLASPFOLIO (Gebser *et al.* 2011), IDP (Wittocx *et al.* 2008), CMODELS (Lierler 2005), SUP (Lierler 2008), SMODELS (Simons *et al.* 2002), and several solvers from both the LP2SAT (Janhunen 2006) and LP2DIFF (Janhunen *et al.* 2009) families, namely LP2GMINISAT, LP2LMINISAT, LP2LGMINISAT, LP2MINISAT, LP2DIFFGZ3, LP2DIFFLGZ3, LP2DIFFLZ3, and LP2DIFFZ3. More in detail, CLASP is a native ASP solver relying on conflict-driven nogood learning; CLASPD is an extension of CLASP that is able to deal with disjunctive logic programs, while CLASPFOLIO exploits machine learning techniques in order to choose the best-suited execution options of CLASP; IDP is a finite model generator for extended first-order logic theories, which is based on *MiniSatID* (Mariën *et al.* 2008); SMODELS is one of the first robust native ASP solvers that have been made available to the community; DLV (Leone *et al.* 2006) is one of the first systems able to cope with disjunctive programs; CMODELS exploits a SAT solver as a search engine for enumerating models, and also verifies model minimality with SAT, whenever needed; SUP exploits nonclausal constraints, and can be seen as a combination of the computational ideas behind CMODELS and SMODELS; the LP2SAT family employs several variants (indicated by the trailing G, L, and LG) of a translation strategy to SAT and resorts to MINISAT (Eén and Sörensson 2003) for actually computing the answer sets; the LP2DIFF family translates programs in

[1] An exhaustive description of the benchmark problems can be found in Calimeri *et al.* (2011).

https://doi.org/10.1017/S1471068413000094 Published online by Cambridge University Press

difference logic over integers (smt-lib-web 2011) and exploit *Z3* (de Moura and Bjørner 2008) as underlying solver (again, G, L, and LG indicate different translation strategies). DLV was run with default settings, while remaining solvers were run on the same configuration (i.e., parameter settings) as in the competition.

Concerning the hardware employed and the execution settings, all the experiments were carried out on CyberSAR (Masoni *et al.* 2009), a cluster comprised of 50 Intel Xeon E5420 blades equipped with 64 bit GNU Scientific Linux 5.5. Unless otherwise specified, the resources granted to the solvers are 600 s of CPU time and 2GB of memory. Time measurements were carried out using the `time` command shipped with GNU Scientific Linux 5.5.

## 4 Designing a multi-engine ASP solver

The design of a multi-engine solver involves several steps: (i) design of (syntactic) features that are both significant for classifying the instances and cheap-to-compute (so that the classifier can be fast and accurate); (ii) selection of solvers that are representative of the state of the art (to be able to possibly obtain the best performance in any considered instance); and (iii) selection of the classification algorithm, and fair design of training and test sets, to obtain a robust and unbiased classifier.

In the following, we describe the choices we have made for designing ME-ASP, which is our multi-engine solver for ground ASP programs.

### *4.1 Features*

Our features selection process started by considering a very wide set of candidate features that correspond, in our view, to several characteristics of an ASP program that, in principle, should be taken into account.

The features that we compute for each ground program are divided into four groups (such a categorization is borrowed from Nudelman *et al.* (2004)):

- **Problem size features**. Number of rules $r$, number of atoms $a$, ratios $r/a$, $(r/a)^2$, $(r/a)^3$ and ratios reciprocal $a/r$, $(a/r)^2$ and $(a/r)^3$. These types of features are considered to give an idea of what is the size of the ground program.
- **Balance features**. Ratio of positive and negative atoms in each body, and ratio of positive and negative occurrences of each variable; fraction of unary, binary and ternary rules. These features can help to understand what is the "structure" of the analyzed program.
- **"Proximity to horn" features**. Fraction of horn rules and number of atoms occurrences in horn rules. These features can give an indication on "how much" a program is close to be horn: this can be helpful, since some solvers may take advantage of this setting (e.g., minimum or no impact of completion (Clark 1978) when applied).
- **ASP peculiar features**. Number of true and disjunctive facts, fraction of normal rules and constraints, head sizes, occurrences of each atom in heads, bodies and

https://doi.org/10.1017/S1471068413000094 Published online by Cambridge University Press

rules, occurrences of true negated atoms in heads, bodies and rules; Strongly Connected Components (SCC) sizes, number of Head-Cycle Free (HCF) and non-HCF components, degree of support for non-HCF components.

For the features implying distributions, e.g., ratio of positive and negative atoms in each body, atoms occurrences in horn rules, and head sizes, five numbers are considered: minimum, 25% percentile, median, 75% percentile and maximum. The five numbers are considered given that we can not *a priori* consider the distributions to be Gaussians; thus, mean and variance are not that informative.

The set of features reported above seems to be adequate for describing an ASP program.[2] On the other hand, we have to consider that the time spent computing the features will be integral part of our solving process: the risk is to spend too much time in calculating the features of a program. This component of the solving process could result in a significant overhead in the solving time in the case of instances that are easily solved by (some of) the engines, or can even cause a time out on programs otherwise solved by (some of) the engines within the time limit.

Given these considerations, our final choice is to consider syntactic features that are cheap-to-compute, i.e., computable in linear time in the size of the input, also given that in previous work (e.g., Pulina and Tacchella 2007) syntactic features have been profitably used for characterizing (inherently) ground instances. To this end, we implemented a tool able to compute the above-reported set of features and conducted some preliminary experiments on all the benchmarks we were able to ground with GrinGo in less than 600 s: 1,425 instances out of a total of 1,462, of which 823 out of 860 NP instances.[3] On the one hand, the results confirmed the need for avoiding the computation of "expensive" features (e.g., SCCs): indeed, in this setting we could compute the whole set of features only for 665 *NP* instances within 600 s; and, on the other hand, the results helped us in selecting a set of "cheap" features that are sufficient for obtaining a robust and efficient multi-engine system. In particular, the features that we selected are a subset of the ones reported above:

- **Problem size features**. Number of rules $r$, number of atoms $a$, ratios $r/a$, $(r/a)^2$, $(r/a)^3$ and ratios reciprocal $a/r$, $(a/r)^2$ and $(a/r)^3$;
- **Balance features**. Fraction of unary, binary and ternary rules;
- **"Proximity to horn" features**. Fraction of horn rules;
- **ASP peculiar features**. Number of true and disjunctive facts, fraction of normal rules and constraints $c$.

This final choice of features, together with some of their combinations (e.g., $c/r$), amounts for a total of 52 features. Our tool for extracting features from ground programs can then compute all these features (in less than 600 s) for 1,371 programs out of 1,462. The distribution of the CPU times for extracting features is characterized by the following five numbers: 0.24, 1.74, 2.40, 4.37, and 541.92

[2] Observations concerning existing proposals are reported in Section 6.
[3] The exceptions are 10 and 27 instances of DisjunctiveScheduling and PackingProblem, respectively.

https://doi.org/10.1017/S1471068413000094 Published online by Cambridge University Press

Table 2. *Results of a pool of ASP solvers on the* NP *instances of the Third ASP Competition. The table is organized as follows: the column "Solver" reports the solver name, column "Solved" reports the total amount of instances solved with a time limit of 600 seconds, and, finally, in column "Unique" we report the total amount of uniquely solved instances by the corresponding solver*

| Solver | Solved | Unique | Solver | Solved | Unique |
|---|---|---|---|---|---|
| CLASP | 445 | 26 | LP2DIFFZ3 | 307 | – |
| CMODELS | 333 | 6 | LP2SAT2GMINISAT | 328 | – |
| DLV | 241 | 37 | LP2SAT2LGMINISAT | 322 | – |
| IDP | 419 | 15 | LP2SAT2LMINISAT | 324 | – |
| LP2DIFFGZ3 | 254 | – | LP2SAT2MINISAT | 336 | – |
| LP2DIFFLGZ3 | 242 | – | SMODELS | 134 | – |
| LP2DIFFLZ3 | 248 | – | SUP | 311 | 1 |

s. It has to be noticed that high CPU times correspond to extracting features for ground programs whose size is in the order of gigabytes. Our set of chosen features is relevant, as will be shown in Section 5.

### 4.2 Solvers selection

The target of our selection is to collect a pool of solvers that is representative of the state-of-the-art solver (SOTA), i.e., considering a problem instance, the oracle that always fares the best among available solvers. Note that, in our settings, the various engines available employ (often substantially) different evaluation strategies, and (it is likely that) different engines behave better in different domains; or, in other words, the engines' performance is "orthogonal". As a consequence, one can find that there are solvers that solve a significant number of instances uniquely (i.e., instances solved by only one solver), which have a characteristic performance and are a fundamental component of the SOTA. Thus, a pragmatic and reasonable choice, given that we want to solve as much instances as possible, is to consider a solver only if it solves a reasonable amount of instances uniquely, since this solver cannot be, in a sense, subsumed performance wise by another behaving similarly.

In order to select the engines, we ran preliminary experiments, and we report the results (regarding the *NP* class) in Table 2. Looking at the table, first we notice that we do not report results related to both CLASPD and CLASPFOLIO. Concerning the results of CLASPD, we report that – considering the *NP* class – its performance, in terms of solved instances, is subsumed by the performance of CLASP. Considering the performance of CLASPFOLIO, we exclude such system from this analysis because we consider it as a yardstick system, i.e., we will compare its performance against the performance of ME-ASP.

Looking at Table 2, we can see that only 4 solvers out of 16 are able to solve a noticeable number of instances *uniquely*, namely CLASP, CMODELS, DLV, and IDP.[4]

[4] The picture of uniquely solved instances does not change even considering the entire family of LP2SAT (resp. LP2DIFF) as a single engine that has the best performance among its variants.

https://doi.org/10.1017/S1471068413000094 Published online by Cambridge University Press

Concerning *Beyond NP* instances, we report that only three solvers are able to cope with such class of problems, namely CLASPD, CMODELS, and DLV. Considering that both CMODELS and DLV are involved in the previous selection, that CLASPD has a performance that does not overlap with the other two in *Beyond NP* instances, the pool of engines used in ME-ASP will be composed of five solvers, namely CLASP, CLASPD, CMODELS, DLV, and IDP.

The experiments reported in Section 5 confirmed that this engine selection policy is effective in practice considering the ASP state of the art. Nonetheless, it is easy to see that in scenarios where the performance of most part of the available solvers is very similar on a common pool of instances, i.e., their performance is not "orthogonal", choosing a solver for the only reason it solves a reasonable amount of instances uniquely may not be an effective policy. Indeed, the straightforward application of that policy to "overlapping" engines could result in discarding the best ones, since it is likely that several of them can solve the same instances. An effective possible extension of the selection policy presented above to deal with overlapping engines is to remove *dominated* solvers, i.e., a solver $s$ dominates a solver $s'$ if the set of instances solved by $s$ is a superset of the instances solved by $s'$. Ties are broken choosing the solver that spends the smaller amount of CPU time. If the resulting pool of engines is still not reasonably distinguishable, i.e., there are not enough uniquely solved instances by each engine of the pool, then one may compute such pool, say $E$, as follows: starting from the empty set ($E = \emptyset$), and trying iteratively to add engine candidates to $E$ from the one that solves more instances, and faster, to the less efficient. At each iteration, an engine $e$ is added to $E$ if both the set of uniquely solved instances by the engines in $E \cup \{e\}$ is larger than in $E$ and the resulting set $E \cup \{e\}$ is reasonably distinguishable.

We have applied the above extended policy, which is to be considered as a pragmatic strategy more than a general solution, obtaining good results in a specific experiment with overlapping engines; more details will be found in Section 5.4.

### 4.3 Classification algorithms and training

In the following, we briefly review the classifiers that we use in our empirical analysis. Considering the wide range of multinomial classifiers described in the scientific literature, we test a subset of algorithms, some of them considered in Pulina and Tacchella (2007). Particularly, we can limit our choice to the classifiers able to deal with numerical attributes (the features) and multinomial class labels (the engines). Furthermore, in order to make our approach as general as possible, our desiderata is to choose classifiers that allow us to avoid "stringent" assumptions on the features distributions, e.g., hypotheses of normality or independence among the features. At the end, we also prefer classifiers that do not require complex parameter tuning, e.g., procedures that are more elaborated than standard parameters grid search. The selected classifiers are listed as follows.

- **Aggregation pheromone density based pattern classification** (APC). It is a pattern classification algorithm modeled on the ants colony behavior and distributed adaptive organization in nature. Each data pattern is considered as an ant,

https://doi.org/10.1017/S1471068413000094 Published online by Cambridge University Press

and the training patterns (ants) form several groups or colonies depending on the number of classes present in the dataset. A new test pattern (ant) will move along the direction where average aggregation pheromone density (at the location of the new ant) formed due to each colony of ants is higher and, hence, eventually it will join that colony. We refer the reader to Halder *et al.* (2009) for further detail.

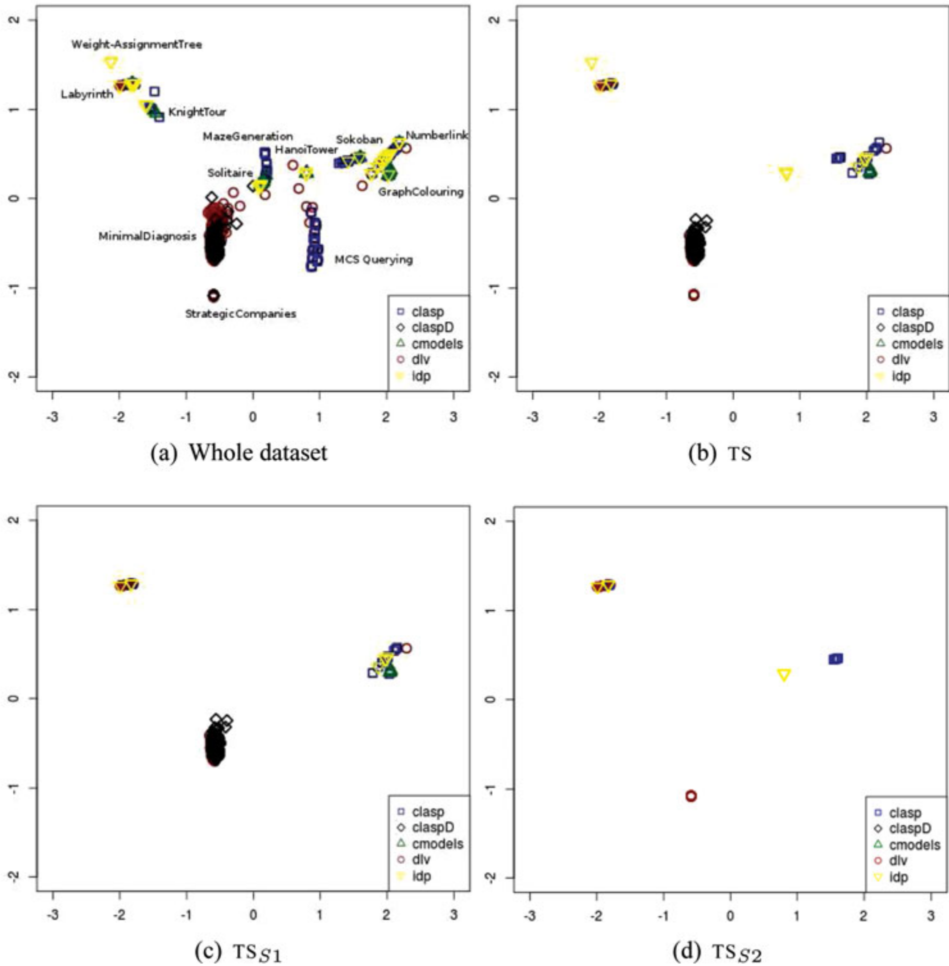
- **Decision rules** (FURIA). A classifier providing a set of rules that generally takes the form of a Horn clause wherein the class labels is implied by a conjunction of some attributes; we use FURIA (Hühn and Hüllermeier 2009) to induce decision rules.
- **Decision trees** (J48). A classifier arranged in a tree structure and used to discover decision rules. Each inner node contains a test on some attributes and each leaf node contains a label; we use J48, an optimized implementation of C4.5 (Quinlan 1993).
- **Multinomial logistic regression** (MLR). A classifier providing a hyperplane of the hypersurfaces that separate the class labels in the feature space; we use the inducer described in Le Cessie and Van Houwelingen (1992).
- **Nearest-neighbor** (NN). It is a classifier yielding the label of the training instance which is closer to the given test instance, whereby closeness is evaluated using, e.g., Euclidean distance (Aha *et al.* 1991).
- **Support vector machine** (SVM). It is a supervised learning algorithm used for both classification and regression tasks. Roughly speaking, the basic training principle of SVMs is finding an optimal linear hyperplane such that the expected classification error for (unseen) test patterns is minimized. We refer the reader to Cortes and Vapnik (1995) for further detail.

The rationale of our choice is twofold. On the one hand, the selected classifiers are “orthogonal”, i.e., they build on different inductive biases in the computation of their classification hypotheses, since their classification algorithms are based on very different approaches. On the other hand, building ME-ASP on top of different classifiers allows to draw conclusions about both the robustness of our approach and the proper design of our testing set. Indeed, as shown in Section 5, performance is positive for each classification method.

As mentioned in Section 2.2, in order to train the classifiers, we have to select a pool of instances for training purpose, called the training set. Concerning such selection, our aim is twofold. On the one hand, we want to compose a training set in order to get a robust model; while, on the other hand, we want to test the generalization performance of ME-ASP also on instances belonging to benchmarks not “covered” by the training set.

As result of the considerations above, we designed three training sets. The first one – TS in the following – is composed of the 320 instances uniquely solved by the pool of engines selected in Section 4.2, i.e. such that only one engine, among the ones selected, solves each instance (without taking into account the instances involved in the competition). The rationale of this choice is to try to “mask” noisy information during model training to obtain a robust model. The remaining training sets are

https://doi.org/10.1017/S1471068413000094 Published online by Cambridge University Press




(a) Whole dataset (b) TS

(c) $\text{TS}_{S1}$ (d) $\text{TS}_{S2}$

Fig. 1. (Colour online) Training set coverage: two-dimensional space projection of (a) the whole dataset, (b) TS, (c) $\text{TS}_{S1}$, and (d) $\text{TS}_{S2}$.

subsets of TS, and they are composed of instances uniquely solved considering only the ones belonging to the problems listed in the following:

- $\text{TS}_{S1}$: 297 instances uniquely solved considering:
  `GraphColouring`, `Numberlink`, `Labyrinth`, `MinimalDiagnosis`.
- $\text{TS}_{S2}$: 59 instances uniquely solved considering:
  `SokobanDecision`, `HanoiTower`, `Labyrinth`, `StrategicCompanies`.

Note that both $\text{TS}_{S1}$ and $\text{TS}_{S2}$ contain one distinct *Beyond NP* problem to ensure a minimum coverage of this class of problems. The rationale of these additional training sets is thus to test our method on “unseen” problems, i.e., on instances coming from domains that were not used for training: a “good” machine learning method should generalize (to some degrees) and obtain good results also in such setting. In this view, both training sets are composed of instances coming from a limited number, i.e., 4 out of 14, of problems. Moreover, $\text{TS}_{S2}$ is also composed of

https://doi.org/10.1017/S1471068413000094 Published online by Cambridge University Press

a very limited number of instances. Such setting will further challenge ME-ASP to understand what is the point in which we can have degradation in performance: we will see that while it is true that $\text{TS}_{S2}$ is a challenging situation in which performance decreases, even in this setting ME-ASP has reasonable performance and performs better than its engines and rival systems.

In order to give an idea of the coverage of our training sets and outline differences among them, we depict in Figure 1 the coverage of: the whole available dataset (Fig. 1a), TS (Fig. 1b), and its subsets $\text{TS}_{S1}$ (Fig. 1c) and $\text{TS}_{S2}$ (Fig. 1d). In particular, the plots report a two-dimensional projection obtained by means of a principal components analysis (PCA), and considering only the first two principal components (PC). The $x$-axis and the $y$-axis in the plots are the first and the second PCs, respectively. Each point in the plots is labeled by the best solver on the related instance. In Figure 1(a) we add a label denoting the benchmark name of the depicted instances, in order to give a hint about the "location" of each benchmark. From the picture it is clear that $\text{TS}_{S1}$ covers less space than TS, which in turn covers a subset of the whole set of instances. Clearly, $\text{TS}_{S2}$, which is the smallest set of instances, has a very limited coverage (see Fig. 1d).

Considering the classification algorithms listed above,[5] we trained the classifiers and assessed their accuracy. Referring to the notation introduced in Section 2.2, even assuming that a training set is sufficient to learn $f$, it is still the case that different sets may yield a different $f$. The problem is that the resulting trained classifier may underfit the unknown pattern – i.e., its prediction is wrong – or overfit – i.e., be very accurate only when the input pattern is in the training set. Both underfitting and overfitting lead to poor *generalization* performance, i.e., $c$ fails to predict $f(\underline{x}^*)$ when $\underline{x}^* \neq \underline{x}$. However, statistical techniques can provide reasonable estimates of the generalization error. In order to test the generalization performance, we use a technique known as *stratified 10-times 10-fold cross validation* to estimate the generalization in terms of *accuracy*, i.e., the total amount of correct predictions with respect to the total amount of patterns. Given a training set $(X, Y)$, we partition $X$ in subsets $X_i$ with $i \in \{1, \dots 10\}$ such that $X = \bigcup_{i=1}^{10} X_i$ and $X_i \cap X_j = \emptyset$ whenever $i \neq j$; we then train $c_{(i)}$ on the patterns $X_{(i)} = X \setminus X_i$ and corresponding labels $Y_{(i)}$. We repeat the process 10 times, to yield 10 different $c$ and we obtain the global accuracy estimate.

We report an accuracy greater than 92% for each classification algorithm trained on TS, while concerning the remaining training sets, just for the sake of completeness we report an average 85% as accuracy result. The main reason for this result is that the training sets different from TS are composed of a smaller number of instances with respect to TS; thus, the classification algorithms are not able to generalize with the same accuracy. This result is not surprising, also considering the plots in Figure 1 and, as we will see in the experimental section, this will influence the performance of ME-ASP.

[5] For all algorithms but APC, we use the the tool RAPIDMINER (Mierswa *et al.* 2006).

https://doi.org/10.1017/S1471068413000094 Published online by Cambridge University Press

## 5 Performance analysis

In this section, we present the results of the analysis we have performed. We consider different combinations of training and test sets, where the training sets are the ones introduced in Section 4, and the test set ranges over the Third ASP Competition ground instances. In particular, the first (resp. second) experiment has TS as training set, and the successfully grounded instances evaluated (resp. submitted) to the Third ASP Competition as test set: the goal of this analysis is to test the *efficiency* of our approach on all the evaluated (resp. submitted) instances when the model is trained on the whole space of the uniquely solved instances. The third experiment considers $\text{TS}_{S1}$ and $\text{TS}_{S2}$ as training sets, and all the successfully grounded instances submitted to the competition as test set: in this case, given that the models are not trained on all the space of the uniquely solved instances, but on a portion, and that the test set contains "unseen" problems (i.e., belonging to domains that were left unknown during training), the goal is to test, in particular, the *robustness* of our approach. We devoted one subsection to each of these experiments, where we compare ME-ASP to its component engines. In detail, for each experiment the results are reported in a table structured as follows: the first column reports the name of the solver and (when needed) its inductive model in a subcolumn, where the considered inductive models are denoted by $\text{MOD}_{\text{TS}}$, $\text{MOD}_{S1}$, and $\text{MOD}_{S2}$, corresponding to the test sets TS, $\text{TS}_{S1}$, and $\text{TS}_{S2}$ introduced before, respectively; the second and third columns report the result of each solver on *NP* and *Beyond NP* classes, respectively, in terms of the number of solved instances within the time limit and sum of their solving times (a subcolumn is devoted to each of these numbers, which are "–" if the related solver was not among the selected engines). We report the results obtained by running ME-ASP with the six classification methods introduced in Section 4.3, and their related inductive models. In particular, ME-ASP (C) indicates ME-ASP employing the classification method $\text{C} \in \{$APC, FURIA, J48, MLR, NN, SVM $\}$. We also report the component engines employed by ME-ASP on each class as explained in Section 4.2, and as reference SOTA, which is the ideal multi-engine solver (considering the engines employed).

An additional subsection summarizes results and compares ME-ASP with state-of-the-art solvers that won the Third ASP Competition.

We remind the reader that the compared engines were run on all the 1,425 instances grounded in less than 600 s, whereas the instances on which ME-ASP was run are limited to the ones for which we were able to compute all features (i.e., 1,371 instances), and the timings for multi-engine systems include both the time spent for extracting the features from the ground instances and the time spent by the classifier.

### 5.1 *Efficiency on instances evaluated at the competition*

In the first experiment, we consider TS introduced in Section 4 as training set, and as test set all the instances evaluated at the Third ASP Competition (a total of 88 instances). Results are shown in Table 3. We can see that on problems of the

https://doi.org/10.1017/S1471068413000094 Published online by Cambridge University Press

Table 3. *Results of the various solvers on the grounded instances evaluated at the Third ASP Competition.* ME-ASP *has been trained on the* TS *training set*

| Solver | Ind. model | *NP* No. solved | *NP* Time (s) | *Beyond NP* No. solved | *Beyond NP* Time (s) |
|---|---|---|---|---|---|
| CLASP | | 60 | 5,132.45 | – | – |
| CLASPD | | – | – | 13 | 2,344.00 |
| CMODELS | | 56 | 5,092.43 | 9 | 2,079.79 |
| DLV | | 37 | 1,682.76 | 15 | 1,359.71 |
| IDP | | 61 | 5,010.79 | – | – |
| ME-ASP (APC) | $\text{MOD}_{\text{TS}}$ | 63 | 5,531.68 | 15 | 3,286.28 |
| ME-ASP (FURIA) | $\text{MOD}_{\text{TS}}$ | 63 | 5,244.73 | 15 | 3,187.73 |
| ME-ASP (J48) | $\text{MOD}_{\text{TS}}$ | 68 | 5,873.25 | 15 | 3,187.73 |
| ME-ASP (MLR) | $\text{MOD}_{\text{TS}}$ | 65 | 5,738.79 | 15 | 3,187.57 |
| ME-ASP (NN) | $\text{MOD}_{\text{TS}}$ | 66 | 4,854.78 | 15 | 3,187.31 |
| ME-ASP (SVM) | $\text{MOD}_{\text{TS}}$ | 60 | 4,830.70 | 15 | 2,308.60 |
| SOTA | | 71 | 5,403.54 | 15 | 1,221.01 |

*NP* class, ME-ASP (J48) solves the highest number of instances, seven more than IDP and eight more than CLASP. Note also that ME-ASP (SVM) (our worst performing version) is basically on par with CLASP (with 60 solved instances) and is very close to IDP (with 61 solved instances). Nonetheless, five out of six classification methods lead ME-ASP to have better performance than each of its engines. On the *Beyond NP* problems, instead, all versions of ME-ASP and DLV solve 15 instances (DLV having best mean CPU time), followed by CLASPD and CMODELS, which solve 13 and 9 instances, respectively. Among the ME-ASP versions, ME-ASP (J48) is, in sum, the solver that solves the highest number of instances: here it is very interesting to note that its performance is very close to the SOTA solver (solving only three instances less) which, we remind, has the ideal performance that we could expect in these instances with these engines.

### 5.2 *Efficiency on instances submitted to the competition*

In the second experiment, we consider the TS training set (as for the previous experiment), and the test set is composed of all successfully grounded instances submitted to the Third ASP Competition. The results are now shown in Table 4. Note here that in both *NP* and *Beyond NP* classes, all ME-ASP versions solve more instances (or in shorter time in one case) than the component engines: in particular, in the *NP* class, ME-ASP (APC) solves the highest number of instances, 52 more than CLASP, which is the best engine in this class, while in the *Beyond NP* class ME-ASP (MLR) solves 519 instances and three ME-ASP versions solve 518 instances, i.e., 86 and 85 more instances than CLASPD, respectively, which is the engine that solves more instances in the *Beyond NP* class. Also, in this case ME-ASP (SVM) solves less instances than other ME-ASP versions; nonetheless, ME-ASP (SVM) can solve as much

https://doi.org/10.1017/S1471068413000094 Published online by Cambridge University Press

Table 4. *Results of the various solvers on the grounded instances submitted to the 3rd ASP Competition.* ME-ASP *has been trained on the* TS *training set*

| Solver | | *NP* | | *Beyond NP* | |
|---|---|---|---|---|---|
| | Ind. model | No. solved | Time (s) | No. solved | Time (s) |
| CLASP | | 445 | 47,096.14 | – | – |
| CLASPD | | – | – | 433 | 52,029.74 |
| CMODELS | | 333 | 40,357.30 | 270 | 38,654.29 |
| DLV | | 241 | 21,678.46 | 364 | 9,150.47 |
| IDP | | 419 | 37,582.47 | – | – |
| ME-ASP (APC) | $\text{MOD}_{\text{TS}}$ | 497 | 55,334.15 | 516 | 60,537.67 |
| ME-ASP (FURIA) | $\text{MOD}_{\text{TS}}$ | 480 | 48,563.26 | 518 | 60,009.23 |
| ME-ASP (J48) | $\text{MOD}_{\text{TS}}$ | 490 | 49,564.19 | 510 | 59,922.86 |
| ME-ASP (MLR) | $\text{MOD}_{\text{TS}}$ | 489 | 49,569.77 | 519 | 58,287.31 |
| ME-ASP (NN) | $\text{MOD}_{\text{TS}}$ | 490 | 46,780.31 | 518 | 55,043.39 |
| ME-ASP (SVM) | $\text{MOD}_{\text{TS}}$ | 445 | 40,917.70 | 518 | 52,553.84 |
| SOTA | | 516 | 39,857.76 | 520 | 24,300.82 |

*NP* instances as CLASP, and is effective on *Beyond NP*, where it is one of the versions that can solve 518 instances.

As far as the comparison with the SOTA solver is concerned, the best ME-ASP version, i.e., ME-ASP (APC) solves, in sum, only 23 out of 1,036 instances less than the SOTA solver, mostly from the *NP* class.

In order to give a different look at the magnitude of improvements of our approach in this experiment, whose test set we remind is a super-set of the one in Section 5.1, in Figure 2 we present the results of ME-ASP (APC), its engines, CLASPFOLIO and SOTA on *NP* instances in a cumulative way as customary in, e.g., Max-SAT and ASP competitions. The $x$-axis reports a CPU time, while the $y$-axis indicates the number of instances solved within a certain CPU time.

Results clearly show that ME-ASP (APC) performs better, in terms of the total number of instances solved, than its engines CLASP, CLASPD, and CLASPFOLIO; also, ME-ASP (APC) it is very close to the SOTA. Looking more in detail at the figure, we can note that along the $x$-axis the distance of ME-ASP (APC) with respect to the SOTA decreases: this is due, for a small portion of instances (given that we have seen that these two steps are efficient), to the time spent to compute features and on classification, and to the fact that we may not always predict the best engine to run. The convergence of ME-ASP (APC) toward SOTA confirms that even if we may sometimes miss to predict the best engine, most of the time we predict an engine that allows us to solve the instance within the time limit.

### *5.3 Robustness on instances submitted to the competition*

In this experiment, we use the two smaller training sets $\text{TS}_{S1}$ and $\text{TS}_{S2}$ introduced in Section 4, while the same test set as that of the previous experiment. The rationale of this last experiment is to test the robustness of our approach on "unseen" problems,

https://doi.org/10.1017/S1471068413000094 Published online by Cambridge University Press

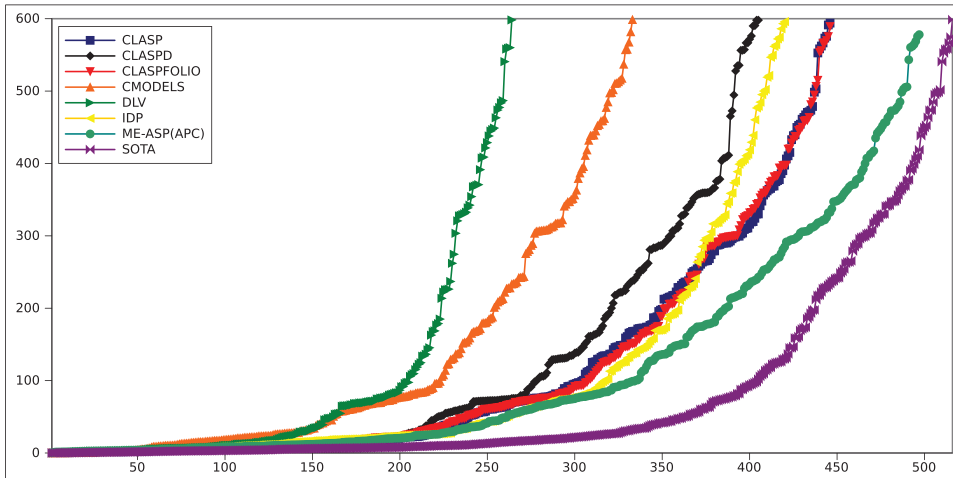



Fig. 2. (Colour online) Results of CLASPFOLIO, ME-ASP engines, ME-ASP (APC) (trained on TS) and SOTA on the *NP* instances submitted to the competition.

i.e., in a situation where the test set does not contain any instance from some problems. Note that $\text{TS}_{S1}$ contains 297 uniquely solved instances, covering 4 domains out of 14; and $\text{TS}_{S2}$ is very small, since it contains only 59 instances belonging to 4 domains. We can thus expect this experiment to be particularly challenging for our multi-engine approach. Results are presented in Table 5, from which it is clear that ME-ASP (APC) trained on $\text{TS}_{S1}$ performs better than the other alternatives and solves 46 instances more than CLASP in the *NP* class, and 11 instances more than CLASPD in the *Beyond NP* class (CLASP and CLASPD being the best engines in *NP* and *Beyond NP* classes, respectively). As expected, if we compare the results with those obtained with the larger training set TS, we note a general performance degradation. In particular, the performance now is less close to the SOTA solver, which solves in total 40 more instances than the best ME-ASP version trained on $\text{TS}_{S1}$, with additional unsolved instances coming mainly from the *Beyond NP* class in this case. This can be explained considering that $\text{TS}_{S1}$ does not contain instances from the Strategic Companies problem and, thus, it is not always able to select DLV on these instances where DLV is often a better choice than CLASPD. However, ME-ASP can also solve in this case far more instances than all the engines, demonstrating a robust performance.

These findings are confirmed when the very small test set $\text{TS}_{S2}$ is considered. In this very challenging setting, there are still ME-ASP versions that can solve more instances than the component engines.

### *5.4 Discussion and comparison to the state of the art*

Summing up the three experiments, it is clear that ME-ASP has a very robust and efficient performance: it often can solve (many) more instances than its engines, even considering the single *NP* and *Beyond NP* classes.

https://doi.org/10.1017/S1471068413000094 Published online by Cambridge University Press

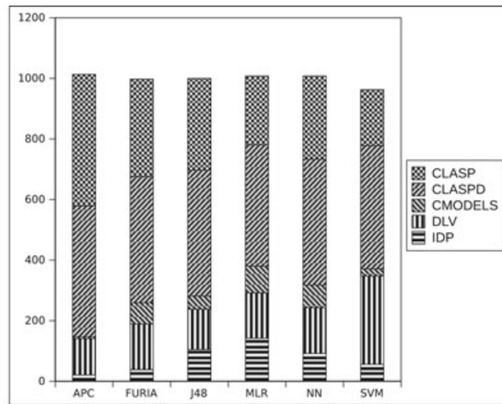
Table 5. *Results of the various solvers on the grounded instances submitted to the Third ASP Competition.* ME-ASP *has been trained on training sets* $TS_{S1}$ *and* $TS_{S2}$

| Solver | Ind. model | *NP* No. solved | *NP* Time (s) | *Beyond NP* No. solved | *Beyond NP* Time (s) |
|---|---|---|---|---|---|
| CLASP | | 445 | 47,096.14 | – | – |
| CLASPD | | – | – | 433 | 52,029.74 |
| CMODELS | | 333 | 40,357.30 | 270 | 38,654.29 |
| DLV | | 241 | 21,678.46 | 364 | 9,150.47 |
| IDP | | 419 | 37,582.47 | – | – |
| ME-ASP (APC) | $MOD_{S1}$ | 491 | 54,126.87 | 505 | 56,250.96 |
| ME-ASP (FURIA) | $MOD_{S1}$ | 479 | 49,226.42 | 507 | 55,777.67 |
| ME-ASP (J48) | $MOD_{S1}$ | 477 | 46,746.65 | 507 | 55,777.67 |
| ME-ASP (MLR) | $MOD_{S1}$ | 471 | 48,404.11 | 507 | 52,499.83 |
| ME-ASP (NN) | $MOD_{S1}$ | 476 | 47,627.06 | 507 | 49,418.67 |
| ME-ASP (SVM) | $MOD_{S1}$ | 459 | 38,686.16 | 507 | 51,462.13 |
| ME-ASP (APC) | $MOD_{S2}$ | 445 | 48,290.97 | 433 | 53,268.62 |
| ME-ASP (FURIA) | $MOD_{S2}$ | 414 | 37,902.37 | 363 | 10,542.85 |
| ME-ASP (J48) | $MOD_{S2}$ | 487 | 51,187.66 | 431 | 57,393.61 |
| ME-ASP (MLR) | $MOD_{S2}$ | 460 | 42,385.66 | 363 | 10,542.01 |
| ME-ASP (NN) | $MOD_{S2}$ | 487 | 48,889.21 | 363 | 10,547.81 |
| ME-ASP (SVM) | $MOD_{S2}$ | 319 | 32,162.37 | 364 | 10,543.00 |
| SOTA | | 516 | 39,857.76 | 520 | 24,300.82 |

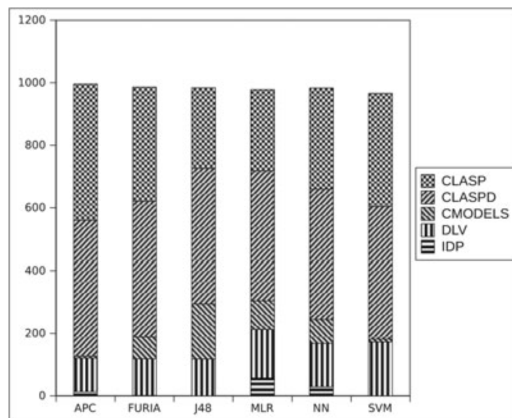
We also report that all versions of ME-ASP have reasonable performance, so – from a machine learning point of view – we can conclude that, on the one hand, the set of cheap-to-compute features that we selected is representative (i.e., they allow to both analyze a significant number of instances and drive the selection of an appropriate engine) independently from the classification method employed. On the other hand, the robustness of our inductive models lets us conclude that we made an appropriate design of our training set TS.

Additional observations can be drawn by looking at Figure 3, where three plots are depicted, one for each inductive model, showing the number of calls to the internal engines for each variant of ME-ASP. In particular, by looking at Figure 3(a), we can conclude that the selection of the engines was also fair. Indeed, all of them were employed in a significant number of cases and, as one would expect, the engines that solved a larger number of instances in the Third ASP Competition (i.e., CLASP and CLASPD) are called more often. Nonetheless, the ability of exploiting all solvers from the pool made a difference in performance, e.g., looking at Figure 3(a) one can note that our best version ME-ASP (APC) exploits all engines, and it is very close to the ideal performance of SOTA. It is worth noting that the ME-ASP versions that select DLV more often (note that DLV solves uniquely a high number of StrategicCompanies instances) performed better on *Beyond NP*. Note also that Figure 3 allows us to explain the performance of ME-ASP (SVM), which often differs from the other methods; indeed, this version often prefers DLV over the other engines also on *NP* instances. Despite the fact that choosing DLV is often decisive

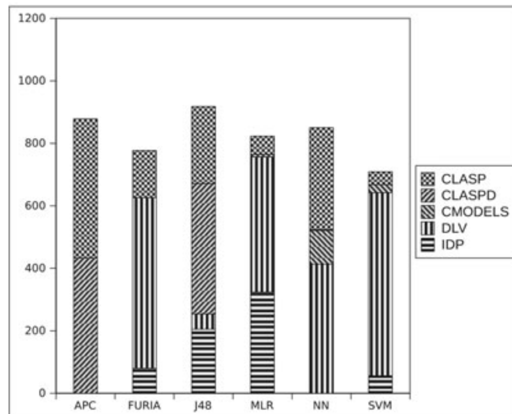
https://doi.org/10.1017/S1471068413000094 Published online by Cambridge University Press




(a) Inductive model $\text{MOD}_{\text{TS}}$




(b) Inductive model $\text{MOD}_{S1}$




(c) Inductive model $\text{MOD}_{S2}$

Fig. 3. Number of calls to the component engines of the various versions of ME-ASP on the instances *submitted* to the Third ASP Competition.

https://doi.org/10.1017/S1471068413000094 Published online by Cambridge University Press

Table 6. *Comparison to the state of the art.* ME-ASP *trained on training set* TS

| Solver | | Evaluated | | | Submitted | | |
|---|---|---|---|---|---|---|---|
| | Ind. model | *NP* | *Beyond NP* | Total | *NP* | *Beyond NP* | Total |
| CLASPD | – | 52 | 13 | 65 | 402 | 433 | 835 |
| CLASPFOLIO | Competition | 62 | – | – | 431 | – | – |
| ME-ASP (APC) | $\text{MOD}_{\text{TS}}$ | 63 | 15 | 78 | 497 | 516 | **1,013** |
| ME-ASP (FURIA) | $\text{MOD}_{\text{TS}}$ | 63 | 15 | 78 | 480 | 518 | 998 |
| ME-ASP (J48) | $\text{MOD}_{\text{TS}}$ | 68 | 15 | **83** | 490 | 510 | 1,000 |
| ME-ASP (MLR) | $\text{MOD}_{\text{TS}}$ | 65 | 15 | 80 | 489 | 519 | 1,008 |
| ME-ASP (NN) | $\text{MOD}_{\text{TS}}$ | 66 | 15 | 81 | 490 | 518 | 1,008 |
| ME-ASP (SVM) | $\text{MOD}_{\text{TS}}$ | 60 | 15 | 75 | 445 | 518 | 963 |

on *Beyond NP*, it is not always a good choice on *NP* as well. As a consequence, ME-ASP (SVM) is always very fast on *Beyond NP* but does not show overall the same performance of ME-ASP equipped with other methods.

Figure 3(a) also gives some additional insight concerning the differences among our inductive models. In particular, the ME-ASP versions trained with $\text{TS}_{S2}$ (containing only StrategicCompanies in *Beyond NP*) prefer more often DLV (see Fig. 3c); thus, the performance is good on this class but deteriorates a bit on *NP*. Concerning $\text{TS}_{S1}$ (see Fig. 3b), we note that IDP is less exploited than in the other cases, even by ME-ASP (MLR), which is the alternative that chooses IDP more often: this is probably due to the minor coverage of this training set on *NP*. Overall, as we would expect, the number of calls for ME-ASP trained with TS is more balanced among the various engines than for ME-ASP trained with the smaller training sets.

We have seen that ME-ASP almost always can solve more instances than its component engines. One might wonder how it compares with the state-of-the-art ASP implementations. Table 6 summarizes the performance of CLASPD and CLASPFOLIO (the overall winner, and the fastest solver in the *NP* class that entered the System Track of the competition, respectively), in terms of the number of solved instances on both instance sets, i.e., evaluated and submitted, and of the various versions of ME-ASP exploiting our inductive model of choice, obtained from the test set TS.

We observe that all ME-ASP versions outperform yardstick state-of-the-art solvers considering all submitted instances.[6]

Concerning the comparison on the instances *evaluated* at the Third ASP Competition, we note that all ME-ASP versions outperform the winner of the System Track of the competition CLASPD that could solve 65 instances, whereas ME-ASP (J48) (i.e., the best solver in this class) solves 83 instances (and is very close to the ideal SOTA solver). Even ME-ASP (SVM) (i.e., the worst performing version of our system) could solve 10 instances more than CLASPD; moreover, also ME-ASP (APC) is very effective here, solving 78 instances.

[6] Recall that CLASPFOLIO can deal with *NP* instances only.

https://doi.org/10.1017/S1471068413000094 Published online by Cambridge University Press

Concerning the comparison on the larger set of instances *submitted* to the Third ASP Competition, the picture is similar. All ME-ASP versions outperform CLASPD, which solves 835 instances where the worst performing version of our system, ME-ASP (SVM), solves 963 instances, and the best version overall ME-ASP (APC) solvers 1,013 instances, i.e., 178 instances more than the winner of the Third ASP System Competition. We remind that this holds even considering the most challenging settings when ME-ASP is trained with $TS_{S1}$ and $TS_{S2}$ (see Table 5).

If we limit our attention to the instances belonging to the *NP* class, the yardstick for comparing ME-ASP with the state of the art is clearly CLASPFOLIO. Indeed, CLASPFOLIO was the solver that could solve more *NP* instances at the Third ASP Competition, and also CLASPFOLIO is the state-of-the-art portfolio system for ASP, selecting from a pool of different CLASP configurations.

The picture that comes out from Table 6 shows that all versions of ME-ASP could solve more instances than CLASPFOLIO, especially considering the instances *submitted* to the competition. In particular, ME-ASP (APC) solves 497 *NP* instances, while CLASPFOLIO solves 431. Concerning the comparison on the instances *evaluated* at the Third ASP Competition, we note that CLASPFOLIO could solve 62 instances and performs similarly to, e.g., ME-ASP (SVM) (with 60 instances) and ME-ASP (FURIA) (with 63 instances); our best performing version (i.e., ME-ASP (J48)) could solve 68 instances, i.e., six instances more than CLASPFOLIO (i.e., about 10% more).

Up to now, we have compared the raw performance of ME-ASP with out-of-the-box alternatives. A more precise picture of the comparison between the two machine learning based approaches (ME-ASP and CLASPFOLIO) can be obtained by performing some additional analysis.

First of all, note that the above comparison was made considering as reference the CLASPFOLIO version (trained by the Potassco team) that entered the Third ASP Competition. One might wonder what is the performance of CLASPFOLIO when trained on our training set TS. As will be discussed in detail in Section 6, CLASPFOLIO exploits a different method for algorithm selection; thus, this datum is reported here only for the sake of completeness. We have trained CLASPFOLIO on TS with the help of the Potassco team.[7] As a result, the performance of CLASPFOLIO trained on TS is analogous to the one obtained by the CLASPFOLIO trained for the competition (i.e., it solves 59 instances from the *evaluated* set, and 433 of the *submitted* set).

On the other hand, one might want to analyze what would be the result of applying the approach to algorithm selection implemented in ME-ASP to the setting of CLASPFOLIO. As pointed out in Section 6, the multi-engine approach that we have followed in ME-ASP is very flexible, and we could easily develop an *ad hoc* version of our system, which we called ME-CLASP, that is based on the same "algorithms" portfolio of CLASPFOLIO. In practice, we considered as a separate engine each of the 25 CLASP versions employed in CLASPFOLIO, and we applied the same steps as described in Section 4 to build ME-CLASP. Concerning the selection of the engines, as one might expect, many engines are overlapping and the number of uniquely

[7] Following the suggestion of the Potassco team, we have run CLASPFOLIO (ver. 1.0.1 – August 19, 2011), since the feature extraction tool CLASPRE has been recently updated and integrated in CLASPFOLIO.

https://doi.org/10.1017/S1471068413000094 Published online by Cambridge University Press

solved instances considering all the available engines was very low (we get only ten uniquely solved instances). Thus, we applied the extended engine selection policy and we selected five engines, we trained ME-CLASP on TS, and selected a classification algorithm, in this case NN. (We also tried other settings with different combinations, both more and less engines, still obtaining similar overall results.)

The goal of this final experiment is to confirm the prediction power of our approach. The resulting picture is that ME-CLASP (NN) solves 458 *NP* instances, where the ideal limit that one can reach considering all the 25 heuristics in the portfolio is 484. This is substantially more than CLASPFOLIO, solving 431 instances. Nonetheless, ME-ASP (NN) (that solves 490) outperforms ME-CLASP (NN).

All in all, one can conclude that the approach introduced in this paper, combining cheap-to-compute features and multinomial classification, also works well when applied to a portfolio of heuristics. On the other hand, as one might expect, the possibility to select among several different engines featuring (often radically different) evaluation strategies with non-overlapping performance, gives additional advantages with respect to a single-engine portfolio. Indeed, even in the presence of an ideal prediction strategy, a portfolio approach based on variants of the same algorithm cannot achieve the same performance of an ideal multi-engine approach. This is clear observing that the SOTA solver on *NP* can solve 516 instances, whereas the ideal performance for both ME-CLASP and CLASPFOLIO tops at 484 instances. The comparison of ME-CLASP and ME-ASP seems to confirm that ME-ASP can exploit this ideal advantage also in practice.

## 6 Related work

Starting from the consideration that, on empirically hard problems, there is rarely a "global" best algorithm, while it is often the case that different algorithms perform well on different problem instances, Rice (1976) defined the algorithm selection problem as the problem of finding an effective algorithm based on an abstract model of the problem at hand. Along this line, several works have been done to tackle combinatorial problems efficiently. In Gomes and Selman (2001) and Leyton-Brown *et al.* (2003), the concept of "algorithm portfolio" as a general method for combining existing algorithms into new ones that are unequivocally preferable to any of the component algorithms is described. Most related papers to our work are Xu *et al.* (2008) and Pulina and Tacchella (2007) for solving SAT and QSAT problems. Both Xu *et al.* (2008) and Pulina and Tacchella (2007) rely on a per-instance analysis, such as the one we have performed in this paper: in Pulina and Tacchella (2007), which is the work closest to our, the goal is to design a multi-engine solver, i.e. a tool that can choose among its engines the one which is more likely to yield optimal results. Pulina and Tacchella (2009) extends Pulina and Tacchella (2007) by introducing a self-adaptation of the learned selection policies when the approach fails to give a good prediction. The approach by Xu *et al.* (2008) has also the ability to compute features on-line, e.g., by running a solver for an allotted amount of time and looking "internally" to solver statistics, with the option of changing the solver on-line: this is a per-instance algorithm portfolio

https://doi.org/10.1017/S1471068413000094 Published online by Cambridge University Press

approach. The related solver, SATZILLA, can also combine portfolio and multi-engine approaches. The algorithm portfolio approach is employed also in Gomes and Selman (2001) on Constraint Satisfaction and MIP, Samulowitz and Memisevic (2007) on QSAT and Gerevini *et al.* (2009) on planning problems. If we consider "pure" approaches, the advantage of the algorithm portfolio over a multi-engine is that it is possible, by combining algorithms, to reach a performance that is better than that of the best engine, which is an upper bound for a multi-engine solver instead. On the other hand, a multi-engine treats the engines as a black-box, and this is a fundamental assumption to have a flexible and modular system: to add a new engine, one just needs to update the inductive model. Other approaches, an overview of which can be found in Hoos (2012), work by designing methods for automatically tuning and configuring the solver parameters: e.g., Hutter *et al.* (2009, 2010) for solving SAT and MIP problems, and Vallati *et al.* (2011) for planning problems.

About the other approaches in ASP, the one implemented in CLASPFOLIO (Gebser *et al.* 2011) mixes characteristics of the algorithm portfolio approach with others more similar to this second trend: it works by selecting the most promising CLASP internal configuration on the basis of both "static" and "dynamic" features of the input program, the latter obtained by running CLASP for a given amount of time. Thus, like the algorithm portfolio approaches, it can compute both static and dynamic features, while trying to automatically configure the "best" CLASP configuration on the basis of the computed features.

The work presented here is in a different ballpark with respect to CLASPFOLIO for a number of motivations. First, from a machine learning point of view, the inductive models of ME-ASP are based on *classification* algorithms, while the inductive models of CLASPFOLIO are mainly based on *regression* techniques, as in SATZILLA, with the exception of a "preliminary" stage, in which a classifier is invoked in order to predict the satisfiability result of the input instance. Regression-based techniques usually need many training instances to have a good prediction while, as shown in our paper, this is not required for our method that is based on classification. To highlight consideration of the prediction power, in Section 5.4 we have applied our approach to CLASPFOLIO, showing that relying on classification instead of regression in CLASPFOLIO can lead to better results. Second, as mentioned before, in our approach we consider the engines as a black-box: ME-ASP architecture is designed to be independent from the engines internals. ME-ASP, being a multi-engine solver, has thus higher modularity/flexibility with respect to CLASPFOLIO: adding a new solver to ME-ASP is immediate, while this is problematic in CLASPFOLIO, and likely would boil down to implement the new strategy in CLASP. Third, as a consequence of the previous point, we use only static features: dynamic features, as in the case of CLASPFOLIO, usually are both strongly related to a given engine and possibly costly to compute, and we avoided such kind of features. For instance, one of the CLASPFOLIO dynamic feature is related to the number of "learnt constraints", which could be a significant feature for CLASP but not for other systems, e.g., DLV that does not adopt learning and is based on look-ahead. Lastly, as described in Section 4.1, we use only cheap-to-compute features, while CLASPFOLIO relies one some quite "costly"

https://doi.org/10.1017/S1471068413000094 Published online by Cambridge University Press

features, e.g., number of SCCs and loops. This was confirmed on some preliminary experiments: it turned out that the CLASPFOLIO feature extractor could compute, in 600 s, all its features for 573 out of 823 *NP* ground instances.

An alternative approach in ASP is followed in the DORS framework of Balduccini (2011), where in the off-line learning phase, carried out on representative programs from a given domain, a heuristic ordering is selected to be then used in SMODELS when solving other programs from the same domain. The target of this work seems to be real-world problem domains where instances have similar structures, and heuristic ordering learned in some (possibly small) instances in the domain can help to improve the performance on other (possibly big) instances. According to its author,[8] the solving method behind DORS can be considered "complementary" more than alternative with respect to the one of ME-ASP, i.e., they could in principle be combined. An idea can be the following: while computing features, one can (in parallel) run one or more engines in order to learn a (possibly partial) heuristic ordering. Then, in the solving phase, engines can take advantage from the learned heuristic (but, of course, assuming minimal changes in the engines). This would come up to having two "sources" of knowledge: the "most promising" engine, learned with the multi-engine approach, and the learned heuristic ordering.

Finally, we remark that this work is an extended and revised version of Maratea *et al.* (2012a), the main improvements include:

(i) the adoption of six classification methods (instead of the only one, i.e., NN, employed in Maratea *et al.* 2012a);
(ii) a more detailed analysis of the dataset and the test sets;
(iii) a wider experimental analysis, including (iiia) more systems, i.e., different versions of ME-ASP and CLASPFOLIO, and (iiib) more investigations on training and test sets, and
(iv) an improved related work, in particular with respect to the comparison with CLASPFOLIO.

## 7 Conclusion

In this paper, we have applied machine learning techniques to ASP, solving with the goal of developing a fast and robust multi-engine ASP solver. To this end, we have (i) specified a number of cheap-to-compute syntactic features that allow for accurate classification of ground ASP programs; (ii) applied six multinomial classification methods to learning algorithm selection strategies; and (iii) implemented these techniques in our multi-engine solver ME-ASP, which is available for download at

http://www.mat.unical.it/ricca/me-asp.

The performance of ME-ASP was assessed on three experiments, which were conceived for checking efficiency and robustness of our approach, involving different training and test sets of instances taken from those submitted to the System Track of

[8] Personal communications with Marcello Balduccini.

https://doi.org/10.1017/S1471068413000094 Published online by Cambridge University Press

the Third ASP Competition. Our analysis shows that our multi-engine solver ME-ASP is very robust and efficient, and outperforms both its component engines and state-of-the-art solvers.

## Acknowledgements

The authors thank Marcello Balduccini for useful discussions (by email and in person) about the solving algorithm underlying his system DORS, and all the members of the CLASPFOLIO team, in particular Torsten Schaub and Thomas Marius Schneider, for clarifications and the valuable support to train CLASPFOLIO in the most proper way.

## References

Aha, D., Kibler, D. and Albert, M. 1991. Instance-based learning algorithms. *Machine learning 6,* 1, 37–66.

Balduccini, M. 2011. Learning and using domain-specific heuristics in ASP solvers. *AI Communications – The European Journal on Artificial Intelligence 24,* 2, 147–164.

Balduccini, M. and Lierler, Y. 2012. Practical and methodological aspects of the use of cutting-edge ASP tools. In *Proc. of the 14th International Symposium on Practical Aspects of Declarative Languages (PADL 2012)*, C. V. Russo and N.-F. Zhou, Eds. LNCS, vol. 7149. Springer, Philadelphia, PA, 78–92.

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

Brooks, D. R., Erdem, E., Erdogan, S. T., Minett, J. W. and Ringe, D. 2007. Inferring phylogenetic trees using answer set programming. *Journal of Automated Reasoning 39,* 4, 471–511.

Calimeri, F., Ianni, G. and Ricca, F. 2011. The third answer set programming system competition. Accessed July 2011 [online]. URL: https://www.mat.unical.it/aspcomp2011/.

Calimeri, F., Ianni, G.and Ricca, F. 2012. The third open answer set programming competition. *Theory and Practice of Logic Programming*, doi.10.1017/S1471068412000105.

Calimeri, F., Ianni, G., Ricca, F., Alviano, M., Bria, A., Catalano, G., Cozza, S., Faber, W., Febbraro, O., Leone, N., Manna, M., Martello, A., Panetta, C., Perri, S., Reale, K., Santoro, M. C., Sirianni, M., Terracina, G. and Veltri, P. 2011. The Third Answer Set Programming Competition: Preliminary report of the system competition track. In *Proc. of LPNMR11*. LNCS, Springer, Vancouver, Canada, 388–403.

Clark, K. L. 1978. Negation as failure. In *Logic and Data Bases*, H. Gallaire and J. Minker, Eds. Plenum Press, New York, 293–322.

Cortes, C. and Vapnik, V. 1995. Support-vector networks. *Machine learning 20,* 3, 273–297.

de Moura, L. M. and Bjørner, N. 2008. Z3: An efficient SMT solver. In *Proc. of the 14th International Conference on Tools and Algorithms for Construction and Analysis of Systems, TACAS 2008*, Budapest, Hungary, 337–340.

Drescher, C., Gebser, M., Grote, T., Kaufmann, B., König, A., Ostrowski, M. and Schaub, T. 2008. Conflict-driven disjunctive answer set solving. In *Proc. of the 11th International Conference on Principles of Knowledge Representation and Reasoning (KR 2008)*, G. Brewka and J. Lang, Eds. AAAI Press, Sydney, Australia, 422–432.

Eén, N. and Sörensson, N. 2003. An extensible SAT-solver. In *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003*. LNCS, Springer, Berlin, 502–518.

https://doi.org/10.1017/S1471068413000094 Published online by Cambridge University Press

Eiter, T., Gottlob, G. and Mannila, H. 1997. Disjunctive datalog. *ACM Transactions on Database Systems 22,* 3(Sept.), 364–418.

Friedrich, G. and Ivanchenko, V. 2008. *Diagnosis from first principles for workflow executions*. Technical repport, Alpen Adria University, Applied Informatics, Klagenfurt, Austria. URL: http://proserver3-iwas.uni-klu.ac.at/download_area/Technical-Reports/technical_report_2008_02.pdf.

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 *Proc. of the 11th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR)*, J. P. Delgrande and W. Faber, Eds. LNCS, vol. 6645. Springer, Vancouver, Canada, 352–357.

Gebser, M., Kaufmann, B., Neumann, A. and Schaub, T. 2007. Conflict-driven answer set solving. In *Proc. of the 12th International Joint Conference on Artificial Intelligence (IJCAI-07)*. Morgan Kaufmann Publishers, Hyderabad, India, 386–392.

Gebser, M., Schaub, T. and Thiele, S. 2007. GrinGo : A new grounder for answer set programming. In *Logic Programming and Nonmonotonic Reasoning, 9th International Conference, LPNMR 2007*. LNCS, vol. 4483. Springer, Tempe, AZ, 266–271.

Gebser, M., Schaub, T., Thiele, S. and Veber, P. 2011. Detecting inconsistencies in large biological networks with answer set programming. *Theory and Practice of Logic Programming 11,* 2–3, 323–360.

Gelfond, M. and Leone, N. 2002. Logic programming and knowledge representation – the A-Prolog perspective . *Artificial Intelligence 138,* 1–2, 3–38.

Gelfond, M. and Lifschitz, V. 1988. The stable model semantics for logic programming. In *Logic Programming: Proceedings Fifth Intl Conference and Symposium*. MIT Press, Cambridge, MA, 1070–1080.

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

Gerevini, A., Saetti, A. and Vallati, M. 2009. An automatically configurable portfolio-based planner with macro-actions: Pbp. In *Proc. of the 19th International Conference on Automated Planning and Scheduling*, A. Gerevini, A. E. Howe, A. Cesta and I. Refanidis, Eds. AAAI, Thessaloniki, Greece.

Giunchiglia, E., Lierler, Y. and Maratea, M. 2006. Answer set programming based on propositional satisfiability. *Journal of Automated Reasoning 36,* 4, 345–377.

Gomes, C. P. and Selman, B. 2001. Algorithm portfolios. *Artificial Intelligence 126,* 1–2, 43–62.

Halder, A., Ghosh, A. and Ghosh, S. 2009. Aggregation pheromone density based pattern classification. *Fundamenta Informaticae 92,* 4, 345–362.

Hoos, H. H. 2012. Programming by optimization. *Communications of the ACM 55,* 2, 70–80.

Hühn, J. and Hüllermeier, E. 2009. Furia: An algorithm for unordered fuzzy rule induction. *Data Mining and Knowledge Discovery 19,* 3, 293–319.

Hutter, F., Hoos, H. H. and Leyton-Brown, K. 2010. Automated configuration of mixed integer programming solvers. In *Proc. of the 7th International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, A. Lodi, M. Milano, and P. Toth, Eds. LNCS, vol. 6140. Springer, Bologna, Italy, 186–202.

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.

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

https://doi.org/10.1017/S1471068413000094 Published online by Cambridge University Press

Janhunen, T., Niemelä, I. and Sevalnev, M. 2009. Computing stable models via reductions to difference logic. In *Proc. of the 10th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR)*. LNCS. Springer, Postdam, Germany, 142–154.

Le Cessie, S. and Van Houwelingen, J.C. 1992. Ridge estimators in logistic regression. *Applied Statistics 41*, 1, 191–201.

Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S. and Scarcello, F. 2006. The DLV system for knowledge representation and reasoning. *ACM Transactions on Computational Logic 7,* 3 (July), 499–562.

Leyton-Brown, K., Nudelman, E., Andrew, G., Mcfadden, J. and Shoham, Y. 2003. A portfolio approach to algorithm selection. In *Proc. of the 18th International Joint Conference on Artificial Intelligence, IJCAI-03*, Acapulco, Mexico.

Lierler, Y. 2005. Disjunctive answer set programming via satisfiability. In *Logic Programming and Nonmonotonic Reasoning – 8th International Conference, LPNMR'05,* Diamante, Italy, September 2005, Proceedings, C. Baral, G. Greco, N. Leone, and G. Terracina, Eds. LNCS, vol. 3662. Springer Verlag, Cosenza, Italy, 447–451.

Lierler, Y. 2008. Abstract answer set solvers. In *Logic Programming, 24th International Conference (ICLP 2008)*. LNCS, vol. 5366. Springer, Udine, Italy, 377–391.

Lifschitz, V. 1999. Answer set planning. In *Proc. of the 16th International Conference on Logic Programming (ICLP'99)*, D. D. Schreye, Ed. MIT Press, Las Cruces, NM, 23–37.

Maratea, M., Pulina, L. and Ricca, F. 2012a. Applying machine learning techniques to ASP solving. In *Technical Communications of the 28th International Conference on Logic Programming (ICLP 2012)*. LIPIcs, vol. 17. Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, Germany, 37–48.

Maratea, M., Pulina, L. and Ricca, F. 2012b. The multi-engine ASP solver ME-ASP. In *Proc. of Logics in Artificial Intelligence, JELIA 2012*. LNCS, vol. 7519. Springer, Toulouse, France, 484–487.

Marczak, W. R., Huang, S. S., Bravenboer, M., Sherr, M., Loo, B. T. and Aref, M. 2010. Secureblox: Customizable secure distributed data processing. In *SIGMOD Conference*. ACM, Indianapolis, 723–734.

Marek, V. W. and Truszczyński, M. 1998. Stable models and an alternative logic programming paradigm. CoRR cs.LO/9809032.

Mariën, M., Wittocx, J., Denecker, M. and Bruynooghe, M. 2008. Sat(id): Satisfiability of propositional logic extended with inductive definitions. In *Proc. of the 11th International Conference on Theory and Applications of Satisfiability Testing, SAT 2008*. LNCS. Springer, Guangzhou, China, 211–224.

Masoni, A., Carpinelli, M., Fenu, G., Bosin, A., Mura, D., Porceddu, I. and Zanetti, G. 2009. Cybersar: A lambda grid computing infrastructure for advanced applications. In *Nuclear Science Symposium Conference Record (NSS/MIC), 2009 IEEE*. IEEE, Orlando, FL, 481–483.

Mierswa, I., Wurst, M., Klinkenberg, R., Scholz, M. and Euler, T. 2006. Yale: Rapid prototyping for complex data mining tasks. In *Proc. of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, Philadelphia, PA, 935–940.

Niemelä, I. 1998. Logic programs with stable Model Semantics as a Constraint Programming Paradigm. In *Proc. of the Workshop on Computational Aspects of Nonmonotonic Reasoning*, I. Niemelä and T. Schaub, Eds. Trento, Italy, 72–79.

Nogueira, M., Balduccini, M., Gelfond, M., Watson, R. and Barry, M. 2001. An A-Prolog decision support system for the space shuttle. In *Practical Aspects of Declarative Languages, Third International Symposium (PADL 2001)*, I. Ramakrishnan, Ed. LNCS, vol. 1990. Springer, Las Vegas, NV, 169–183.

https://doi.org/10.1017/S1471068413000094 Published online by Cambridge University Press

NUDELMAN, E., LEYTON-BROWN, K., HOOS, H. H., DEVKAR, A. AND SHOHAM, Y. 2004. Understanding random SAT: Beyond the clauses-to-variables ratio. In *Proc. of the 10th International Conference on Principles and Practice of Constraint Programming (CP)*, M. Wallace, Ed. LNCS. Springer, Toronto, Canada, 438–452.

PULINA, L. AND TACCHELLA, A. 2007. A multi-engine solver for quantified boolean formulas. In *Proc. of the 13th International Conference on Principles and Practice of Constraint Programming (CP)*, C. Bessiere, Ed. LNCS. Springer, Providence, RI, 574–589.

PULINA, L. AND TACCHELLA, A. 2009. A self-adaptive multi-engine solver for quantified boolean formulas. *Constraints 14,* 1, 80–116.

QUINLAN, J. 1993. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, San Francisco, CA.

RICCA, F., DIMASI, A., GRASSO, G., IELPA, S.M., IIRITANO, S., MANNA, M. AND LEONE, N. 2010. A logic-based system for e-Tourism. *Fundamenta Informaticae 105,* 2010, 35–55

RICCA, F., GALLUCCI, L., SCHINDLAUER, R., DELL'ARMI, T., GRASSO, G. AND LEONE, N. 2009. OntoDLV: An ASP-based system for enterprise ontologies. *Journal of Logic and Computation 19*, 4, 643–670.

RICCA, F., GRASSO, G., ALVIANO, M., MANNA, M., LIO, V., IIRITANO, S. AND LEONE, N. 2012. Team-building with answer set programming in the Gioia-Tauro Seaport. *Theory and Practice of Logic Programming 12*, 3, 361–381.

RICE, J. R. 1976. The algorithm selection problem. *Advances in Computers 15*, 65–118.

RULLO, P., POLICICCHIO, V. L., CUMBO, C. AND IIRITANO, S. 2009. Olex: Effective rule learning for text categorization. *IEEE Transactions on Knowledge and Data Engineering 21,* 8, 1118–1132.

SAMULOWITZ, H. AND MEMISEVIC, R. 2007. Learning to solve QBF. In *Proceedings of the 22th AAAI Conference on Artificial Intelligence*. AAAI Press, Vancouver, Canada, 255–260.

SIMONS, P., NIEMELÄ, I. AND SOININEN, T. 2002. Extending and implementing the stable model semantics. *Artificial Intelligence 138*, 181–234.

SMARAGDAKIS, Y., BRAVENBOER, M. AND LHOTÁK, O. 2011. Pick your contexts well: Understanding object-sensitivity. In *Proc. of the 38th Symposium on Principles of Programming Languages, POPL 2011*. ACM, Austin, TX, 17–30.

SMT-LIB-WEB. 2011. The Satisfiability Modulo Theories Library. Accessed July 2011 [online]. URL: http://www.smtlib.org/.

VALLATI, M., FAWCETT, C., GEREVINI, A., HOOS, H. AND SAETTI, A. 2011. Generating fast domain-specific planners by automatically configuring a generic parameterised planner. In *Working notes of 21st International Conference on Automated Planning and Scheduling (ICAPS-11) Workshop on Planning and Learning*. Freiburg, Germany.

WITTOCX, J., MARIËN, M. AND DENECKER, M. 2008. The IDP system: A model expansion system for an extension of classical logic. In *Logic and Search, Computation of Structures from Declarative Descriptions (LaSh 2008)*. Leuven, Belgium, 153–165.

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.

https://doi.org/10.1017/S1471068413000094 Published online by Cambridge University Press