ME-ASP: A Multi-Engine Solver for Answer Set Programming

Marco Maratea, Luca Pulina, and Francesco Ricca

[1]DIBRIS, Univ. degli Studi di Genova, Viale F.Causa 15, 16145 Genova, Italy
[2]POLCOMING, Univ. degli Studi di Sassari, Viale Mancini 5, 07100 Sassari, Italy
[3]Dipartimento di Matematica, Univ. della Calabria, Via P. Bucci, 87030 Rende, Italy
marco@dist.unige.it, lpulina@uniss.it, ricca@mat.unical.it

Abstract. In this paper we present ME-ASP, a new multi-engine solver for Answer Set Programming (ASP). ME-ASP relies on machine learning techniques for inductively determining its algorithm selection strategy for choosing the "most promising" ASP solver among the ones available. We describe the architecture of ME-ASP and the classification methods it supports. An experimental analysis, performed on benchmarks from the 3rd ASP competition, shows how ME-ASP performs with the various methods, and outlines that ME-ASP can have very robust performance.

1 Introduction

In order to improve 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 Answer Set Programming (ASP) [14, 24, 27, 23, 15, 3], the followed directions are (i) extending existing state-of-the-art techniques implemented in ASP solvers, or (ii) designing 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 ASP systems, e.g., [33, 20, 21, 11, 25, 19, 33], 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* [32]. Related approaches, following a per-instance selection, have been exploited for solving propositional satisfiability (SAT), e.g., [36], and Quantified SAT (QSAT), e.g., [29] problems. In ASP, an approach for selecting the "best" CLASP internal configuration is followed in [10], while another approach that imposes learned heuristics ordering to SMODELS is shown in [2].

In this paper we pursue the alternative direction, by presenting ME-ASP, a new multi-engine solver for Answer Set Programming (ASP). 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 problems submitted to the "System Track" of the 3rd ASP Competition [4] 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 solver performances on these instances, inductively learn general algorithm selection strategies to be applied to a *test* set. We consider five well-known multinomial classification methods, some of them considered in [29]. We perform a number of analyses considering different training and test sets taken from the grounded instances submitted to the System Track of the 3rd ASP competition. Our analysis shows that ME-ASP has very robust performance, and can solve significantly more instances than all the solvers that entered the 3rd ASP competition, DLV and CLASPFOLIO, the latter being the implementation of the approach in [10].

The paper is structured as follow. Section 2 contains preliminaries about ASP and Machine Learning. Section 3 then describes our benchmarks 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 shows the performance analysis, while Section 6 ends the paper with conclusions.

2 Preliminaries

In this section we recall some preliminary notions concerning answer set programming and machine learning techniques for algorithm selection.

2.1 Answer Set Programming

Answer Set Programming (ASP) [14, 24, 27, 23, 15, 3] is a declarative programming formalism 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 those solutions [23].

In the following, we recall both the syntax and semantics of ASP. The presented constructs are included in ASP-Core [5], which is the language specification that was originally introduced in the 3rd ASP Competition [5] 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 [15, 3, 13] for complementary introductory material on ASP, and to [4] for obtaining the full specification of ASP-Core.

Syntax. A variable or a constant is a *term*. An *atom* is $p(t_1, ..., t_n)$, where p is a *predicate* of arity n and $t_1, ..., t_n$ 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:

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

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 an *integrity constraint*. A rule r is *safe* if each variable appearing in r appears also in some positive body literal of r.

An *ASP program* $\mathcal{P}$ 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 $\mathcal{P}$, the *Herbrand Universe* $U_\mathcal{P}$ is the set of all constants appearing in $\mathcal{P}$, and the *Herbrand Base* $B_\mathcal{P}$ is the set of all possible ground atoms which can be constructed from the predicates appearing in $\mathcal{P}$ with the constants of $U_\mathcal{P}$. 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 $U_\mathcal{P}$. Similarly, given a program $\mathcal{P}$, the *ground instantiation* of $\mathcal{P}$ is $Ground(\mathcal{P}) = \bigcup_{r \in \mathcal{P}} Ground(r)$. An *interpretation* for a program $\mathcal{P}$ is a subset I of $B_\mathcal{P}$. 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 `not` A is true w.r.t. I if A is false w.r.t. I; otherwise `not` A is false w.r.t. I.

The answer sets of a program $\mathcal{P}$ 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* [15] 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 $\mathcal{P}$ is an interpretation I of $\mathcal{P}$ such that I is an answer set of $Ground(\mathcal{P})^I$.

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. Among the approaches for solving this problem, 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 algorithm among a pool of them – in our case, tools to solve ASP instances. 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 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 of the labels as generated by some unknown function $f : \mathbb{R}^n \to \{1, \ldots, m\}$ 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 some information concerning 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 benchmark problems considered for the experiments belong to the benchmark suite of the third ASP Competition [5]. This is a large and heterogeneous suite of hard benchmarks, which was already employed for evaluating the performance of state-of-the-art ASP solvers, which are encoded in a common fragment of ASP called ASP-Core. That suite includes planning domains, temporal and spatial scheduling problems, combinatory puzzles, graph problems, and a number of applicative domains taken from the database, information extraction and molecular biology field.In more detail, we have employed the encodings used in the system track of the competition, and all the instances made available from the contributors of the problem submission stage of the competition. Note that this is a superset of the instances actually selected for running the competition itself. These benchmarks, along with their descriptions, are available from the competition website [4].

The techniques presented in this paper are conceived for dealing with propositional programs, thus we have grounded all the mentioned problem instances by using GRINGO (v.3.0.3) [12] to obtain a setup very close to the one of the competition.We considered only computationally-hard problems, that is 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 in our experiments that entered the system track of the last ASP Competition [4] with the addition of DLV [20] (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 [11], CLASPD [8], CLASPFOLIO [10], IDP [35], CMODELS [21], SUP [22], SMODELS [33], and several solvers from both the LP2SAT [18] and LP2DIFF [19] families, namely: LP2GMINISAT, LP2LMINISAT, LP2LGMINISAT, LP2MINISAT, LP2DIFFGZ3, LP2DIFFLGZ3, LP2DIFFLZ3, and LP2DIFFZ3.

Table 1. Benchmark problems and instances.

Problem	Class	#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
StrategicCcompanies	*Beyond NP*	51

In more 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* [25]; SMODELS is one of the first robust native ASP solvers that have been made available to the community; DLV [20] is one of the first systems able to cope with disjunctive programs; CMODELS that exploits an SAT solver as a search engine for enumerating models, and also verifying model minimality 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 on MINISAT [9] for actually computing the answer sets; the LP2DIFF family translates programs in difference logic over integers [34] and exploit *Z3* [7] as underlying solver (again, G, L and LG indicate different translation strategies).

Concerning the hardware employed and the execution settings, all the experiments were carried out on CyberSAR [26], 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 600s 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 based on multinomial classification (see Section 2.2) 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 obtain the best possible performance in any considered instance); and (iii) selection of

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

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

We consider syntactic features that are cheap-to-compute, i.e., computable in linear time in the size of the input, given that in previous work (e.g., [29]) syntactic features have been profitably used for characterizing (inherently) ground instances. The features that we compute for each ground program are divided into four groups: problems size, balance, "proximity to horn" and ASP-based peculiar features. This categorization is borrowed from [28]. The problem size features are: 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 $(a/r)^3$. The balance features are: fraction of unary, binary and ternary rules. The "proximity to horn" features are: fraction of horn rules and number of occurrences in a horn rule for each atom. We have added a number of ASP peculiar features, namely: number of true and disjunctive facts, fraction of normal rules and constraints c. Also some combinations, e.g., c/r, are considered for a total of 52 features.

We were able to ground with GRINGO 1425 programs out of a total of 1462 in less than 600s.[1] Our system for extracting features from ground programs can then compute all features (in less than 600s) for 1371 programs: to have an idea of its performance, it can compute all features of a ground program of approximately 20MB in about 4s.

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 (SOTA) solver, i.e., considering a problem instance, the oracle that always fares the best among available solvers. In order to do that – concerning *NP* instances –, we

[1] The exceptions are 10 and 27 instances of the DisjunctiveScheduling and PackingProblem domains, respectively.

ran preliminary experiments, and we report the results 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 NP benchmarks – its performance is subsumed by the performance of CLASP. Considering the performance of CLASPFOLIO, we exclude such system from our analysis because we consider it as a "rival" 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 amount of instances *uniquely*, namely CLASP, CMODELS, DLV, and IDP. Concerning *Beyond NP* instances, we report that only three solvers are able to cope with such class of problems, name CLASPD, CMODELS, and DLV. Considering that both CMODELS and DLV are involved in the previous selection, the pool of engines used in ME-ASP will be composed of 5 solvers, namely CLASP, CLASPD, CMODELS, DLV, and IDP.

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 built on different inductive biases in the computation of their classification hypotheses:

- **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, and the training patterns (ants) form several groups or colonies depending on the number of classes present in the data set. 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 direct the reader to [16] for further details.
- **Decision rules** (FURIA): A classifier providing a set of "if-then-elsif" constructs, wherein the "if" part contains a test on some attributes and the "then" part contains a label; we use FURIA [17] to induce decision rules.
- **Decision trees** (J48): A classifier arranged in a tree structure, wherein 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 [31], to induce decision trees.
- **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 some proximity measure, e.g., Euclidean distance; we use the method described in [1] to store the training instances for fast look-up.
- **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 address the reader to [6] for further details.

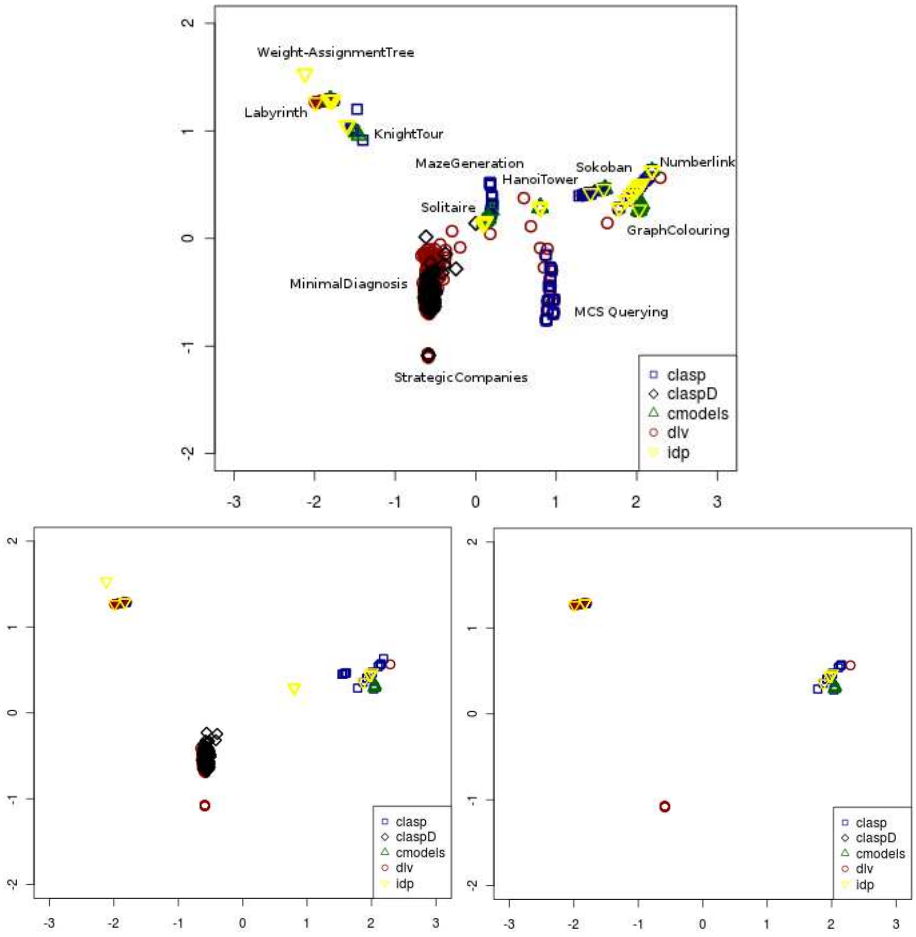

Fig. 1. Two-dimensional space projection of the whole dataset (top), TS1, and TS2 (bottom-left and bottom-right, respectively).

Table 3. Accuracy of the trained models of ME-ASP using cross-validation. The table is structured as follows. In the first column ("**Classifier**"), we report the classifier, and it is followed by a group of columns ("**Accuracy**"). The group is composed of two columns, reporting the accuracy – in percentage – related to MOD1 and MOD2 (columns "MOD1" and "MOD2", respectively).

Classifier	**Accuracy**	
	MOD1	MOD2
APC	96.58%	89.83%
FURIA	94.09%	83.39%
J48	93.12%	79.46%
NN	92.81%	80.71%
SVM	94.38%	82.32%

As mentioned in Section 2.2, in order to train the classifier, we have to select a pool of instances for training purpose, i.e., the training set. Concerning such selection, our aim is twofold. On the one hand, we want to compose a training set in order to train a robust model, while, on the other hand, we want to test the generalization performance of ME-ASP also on instances comprised in benchmarks not comprised in the training set.

As result of the considerations above, we compute two training sets. The first one – TS1 in the following – is composed of the 320 instances solved uniquely – without taking into account the instances involved in the 3rd ASP Competition – by the pool of engines selected in Section 4.2. The rational of this choice is to try to "mask" noisy information during model training. The second one – TS2 in the following – is a subset of TS1, and it is composed of the 77 instances uniquely solved considering only the benchmarks `GraphColouring`, `Labyrinth`, `Numberlink`, and `StrategicCompanies`. The rationale of this choice is to draw some considerations about the trained models considering unknown parts of the instances space.

In order to depict both the differences of TS1 and TS2 and the coverage of our training set with respect to the whole available dataset, in Figure 1 we considered each instance as a point in the multidimensional feature space. In the plots, we consider 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 the top-most plot, we add a label denoting the benchmark name of the depicted instances, in order to give a hint about the "location" of each benchmark.

Considering the classification algorithms listed above, our next experiment is devoted to training the classifiers, and to assessing 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 tech-

nique 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 $i \in \{1, \ldots 10\}$ 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.

In Table 3 we report the accuracy results related to the experiment described above. Looking at the table, we denote as MOD1 and MOD2 the inductive models computed training the classifiers on TS1 and TS2, respectively. Notice that, in this stage, we also explore for each algorithm its parameter space, in order to tune it. Looking at Table 3, we report a 90% greater accuracy for each classification algorithm trained on TS1. Concerning MOD2, we report a lower accuracy with respect to MOD1. The main motivation for this result is that TS2 is composed of a smaller number of instances with respect to TS1, so 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 Performance analysis

In this section we present the results of the analysis we have performed. We consider three different combinations of training and test sets, where the training sets are the TS1 and TS2 sets introduced in Section 4, composed of uniquely solved instances, and the test set ranges over the 3rd ASP competition ground instances. In particular, the first (resp. second) experiment has TS1 as training set, and as test set the successfully grounded instances evaluated (resp. submitted) to the 3rd ASP Competition: 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 TS2 as a training set, composed of uniquely solved instances of some domains, and all the successfully grounded instances submitted to the competition as test set: in this case, given that the model is not trained on all the space of the uniquely solved instances, but on a portion, and that the test set contains "unseen" instances, the goal is to test, in particular, the *robustness* of our approach.

We devoted one subsection to each of our experiments. For each experiment the results are reported in a table structured as follows: the first column reports the name of a solver, the second, third and fourth columns report the results of each solver on *NP*, and *Beyond NP*, respectively, in terms of the number of solved instances within the time limit and sum of their solving times (a sub-column is devoted to each of these numbers). We report the results obtained by running: ME-ASP with the five classification methods introduced in Section 4.3, in particular ME-ASP(X) indicates ME-ASP employing the classification method X$\in$ {APC, FURIA, J48, NN, SVM }, the component engines employed by ME-ASP on each class as explained in Section 4.2, and as reference CLASPFOLIO and SOTA, the latter being the ideal (State-Of-The-Art) multi-engine solver (considering the engines employed).

We remind the reader that, for ME-ASP, the number of instances on which ME-ASP is run is further limited to the ones for which we were able to compute all features, and its

Table 4. Results of the various solvers on the grounded instances evaluated at the 3rd ASP competition. ME-ASP has been trained on the TS1 training set.

Solver	*NP*		*Beyond NP*		Total	
	#Solved	Time	#Solved	Time	#Solved	Time
CLASP	60	5132.45	–	–	–	–
CLASPD	–	–	13	2344.00	–	–
CMODELS	56	5092.43	9	2079.79	65	7172.22
DLV	37	1682.76	15	1359.71	52	3042.47
IDP	61	5010.79	–	–	–	–
ME-ASP (APC)	63	5531.68	15	3286.28	78	8817.96
ME-ASP (FURIA)	63	5244.73	15	3187.73	78	8432.46
ME-ASP (J48)	68	5873.25	15	3187.73	83	9060.98
ME-ASP (NN)	66	4854.78	15	3187.31	81	8042.09
ME-ASP (SVM)	60	4830.70	15	2308.60	75	7139.30
CLASPFOLIO	62	4824.06	–	–	–	–
SOTA	71	5403.54	15	1221.01	86	6624.55

timings include both the time spent for extracting the features from the ground instances, and the time spent by the classifier.

5.1 Efficiency of ME-ASP on instances evaluated at the Competition

In the first experiment, we consider TS1 introduced in Section 4 as training set, and as test set all the instances evaluated at the 3rd ASP Competition (a total of 88 instances). Results are shown in Table 4. We can see that, on programs of the *NP* class, ME-ASP(FURIA) solves the highest number of instances, 6 more than CLASPFOLIO and, moreover, 4 out of 5 classification methods lead ME-ASP to have better performance than each of its engines, and of CLASPFOLIO. On the *Beyond NP* programs, 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. Summarizing, ME-ASP(FURIA) is 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 which, we remind, has the ideal performance that we could expect in these instances with these engines.

5.2 Efficiency of ME-ASP on instances submitted to the Competition

In the second experiment we consider the same training set as for the previous experiment, while the test set is composed of all successfully grounded instances submitted to the 3rd ASP competition.

The results are now shown in Table 5. It is immediately noticeable here that in both *NP* and *Beyond NP* instances, all ME-ASP versions solve more instances (or in shorter time in one case) than their engines and CLASPFOLIO: in particular, in the *NP* instances, ME-ASP(APC) solves the highest number of instances, 52 more than CLASP, which is

Table 5. Results of the various solvers on the grounded instances submitted to the 3rd ASP competition. ME-ASP has been trained on the TS1 training set.

Solver	*NP*		*Beyond NP*		Total	
	#Solved	Time	#Solved	Time	#Solved	Time
CLASP	445	47096.14	–	–	–	–
CLASPD	–	–	433	52029.74	–	–
CMODELS	333	40357.30	270	38654.29	603	79011.59
DLV	241	21678.46	364	9150.47	605	30828.93
IDP	419	37582.47	–	–	–	–
ME-ASP (APC)	497	55334.15	516	60537.67	1013	115871.82
ME-ASP (FURIA)	480	48563.26	518	60009.23	998	108572.49
ME-ASP (J48)	490	49564.19	510	59922.86	1000	109487.05
ME-ASP (NN)	490	46780.31	518	55043.39	1008	101823.70
ME-ASP (SVM)	445	40917.70	518	52553.84	963	93471.54
CLASPFOLIO	431	41874.53	–	–	–	–
SOTA	516	39857.76	520	24300.82	1036	64158.58

the best engine in this class, and 66 more than CLASPFOLIO, while in the *Beyond NP* instances three ME-ASP versions solve 518 instances, i.e., 85 more instances than CLASPD which is the engine that solves more instances. As far as the comparison with the SOTA solver is concerned, the best ME-ASP version solves only 25, out of 1036, instances less than the SOTA solve, mostly from the *NP* class.

5.3 Robustness of ME-ASP on instances submitted to the Competition

In this experiment, we use the TS2 training set as introduced in Section 4, and the same test set as that of previous experiment. The rationale of this last experiment is to test our approach on "unseen" instances, i.e., in a situation where the test set contains instances that come from program domains whose instances have not been used to train the model. We can thus expect this experiment to be particularly challenging for our multi-engine approach. Results are presented in Table 6. By looking at the results, it is clear that ME-ASP(APC) performs much better that the other alternatives, and solves 46 instances more than CLASP in the *NP* class, 60 instances more then CLASPFOLIO in the same class, and 11 more instances than CLASPD in the *Beyond NP* class, CLASP and CLASPD being the best engines in the two classes. However, even if with a multi-engine approach we can solve also in this case far more instances than all the engines and rival solvers, we report that in this case the performance of our best configuration are not that close to the SOTA solver, which solves in total 101 more instances, the majority coming from the *Beyond NP* class in this case.

5.4 Discussion

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

Table 6. Results of the various solvers on the grounded instances submitted to the 3rd ASP competition. ME-ASP has been trained on the TS2 training set.

Solver	*NP*		*Beyond NP*		Total	
	#Solved	Time	#Solved	Time	#Solved	Time
CLASP	445	47096.14	–	–	–	–
CLASPD	–	–	433	52029.74	–	–
CMODELS	333	40357.30	270	38654.29	603	79011.59
DLV	241	21678.46	364	9150.47	605	30828.93
IDP	419	37582.47	–	–	–	–
ME-ASP (APC)	491	53875.41	444	57555.34	935	111430.75
ME-ASP (FURIA)	450	50495.50	365	10483.81	815	61429.31
ME-ASP (J48)	450	53272.70	366	10486.43	816	63759.13
ME-ASP (NN)	484	52191.49	364	10550.01	848	62741.50
ME-ASP (SVM)	383	36786.04	364	10543.00	747	47329.04
CLASPFOLIO	431	41874.53	–	–	–	–
SOTA	516	39857.76	520	24300.82	1036	64158.58

Further, we also run CLASPD on *NP* instances: it solves, as expected, less instances than CLASP, i.e., 52 instances in the first experiment, and 402 in the second and third experiments. Considering the column "Total", all ME-ASP versions solve more instances than CLASPD on each experiment, but for ME-ASP(SVM) in the last. Moreover, it is interesting to note that even considering the performance, in terms of solved instances, of CLASP on *NP* benchmarks and CLASPD on *Beyond NP* benchmarks together, there is always at least one version of ME-ASP that solves more instances in each of the three experiments. We also report that all versions of ME-ASP return reasonable performance, so – from a machine learning point of view – we can conclude that, on one hand, we computed a representative pool of features, and, on the other hand, the robustness of our inductive models let us conclude that we made a good selection of the instances used for classifier training purpose.

A final consideration is about experiment 3: we have seen that this is the only experiment where the difference in performance between ME-ASP and SOTA is significant. One option to try to reduce the gap is to introduce adaptations of the learned selection policies when the approach fails to give a good prediction: in, e.g., [30], this proved to be effective on QSAT problems.

6 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 system. 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 five multinomial classification methods to learning algorithm selection strategies; (iii) implemented these techniques in our multi-engine solver ME-ASP, which is available for download at `http://www.mat.unical.it/ricca/downloads/measp20120323.zip` . 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 the ones submitted to the System Track of the 3rd ASP competition. Our analysis shows that ME-ASP is very robust and efficient, and outperforms both its component engines and rival solvers.

References

1. D.W. Aha, D. Kibler, and M.K. Albert. Instance-based learning algorithms. *Machine learning*, 6(1):37–66, 1991.
2. M. Balduccini. Learning and using domain-specific heuristics in ASP solvers. *AI Communications*, 24(2):147–164, 2011.
3. C. Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving*. CUP, 2003.
4. F. Calimeri, G. Ianni, and F. Ricca. The third answer set programming system competition, since 2011. `https://www.mat.unical.it/aspcomp2011/`.
5. F. Calimeri, G. Ianni, F. Ricca, M. Alviano, A. Bria, G. Catalano, S. Cozza, W. Faber, O. Febbraro, N. Leone, M. Manna, A. Martello, C. Panetta, S. Perri, K. Reale, M. C. Santoro, M. Sirianni, G. Terracina, and P. Veltri. The Third Answer Set Programming Competition: Preliminary Report of the System Competition Track. In *Proc. of LPNMR11*, LNCS, pp. 388–403, Vancouver, Canada, 2011.
6. C. Cortes and V. Vapnik. Support-vector networks. *Machine learning*, 20(3):273–297, 1995.
7. Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In *TACAS*, pp. 337–340, 2008.
8. C. Drescher, M. Gebser, T. Schaub. Conflict-Driven Disjunctive Answer Set Solving. In *KR 2008*, pp. 422–432, AAAI Press. Sydney, Australia, 2008.
9. N. Eén and N. Sörensson. An Extensible SAT-solver. In *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003.*, pp. 502–518. LNCS 2003.
10. M. Gebser, R. Kaminski, B. Kaufmann, T. Schaub, M. T. Schneider, and S. Ziller. A portfolio solver for answer set programming: Preliminary report. In *Proc. of the 11th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR)*, LNCS 6645, pp. 352–357, Vancouver, Canada, 2011.
11. M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub. Conflict-driven answer set solving. In *IJCAI 2007*, pp. 386–392, Hyderabad, India, 2007.
12. M. Gebser, T. Schaub, and S. Thiele. GrinGo : A New Grounder for Answer Set Programming. In *Logic Programming and Nonmonotonic Reasoning, 9th International Conference, LPNMR 2007, 15-17, 2007, Proceedings*, LNCS 4483, pp. 266–271, Tempe, Arizona, 2007.
13. M. Gelfond and N. Leone. Logic Programming and Knowledge Representation – the A-Prolog perspective . *AI*, 138(1–2):3–38, 2002.
14. M. Gelfond and V. Lifschitz. The Stable Model Semantics for Logic Programming. In *ICLP/SLP 1988*, pp. 1070–1080, Cambridge, Mass., 1988. MIT Press.
15. M. Gelfond and V. Lifschitz. Classical Negation in Logic Programs and Disjunctive Databases. *NGC*, 9:365–385, 1991.
16. A. Halder, A. Ghosh, and S. Ghosh. Aggregation pheromone density based pattern classification. *FI*, 92(4):345–362, 2009.
17. J. Hühn and E. Hüllermeier. Furia: an algorithm for unordered fuzzy rule induction. *Data Mining and Knowledge Discovery*, 19(3):293–319, 2009.
18. T. Janhunen. Some (in)translatability results for normal logic programs and propositional theories. *Journal of Applied Non-Classical Logics*, 16:35–86, 2006.

19. T. Janhunen, I. Niemelä, and Mark Sevalnev. Computing stable models via reductions to difference logic. In *Proceedings of the 10th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR)*, LNCS, pp. 142–154, Postdam, Germany, 2009.
20. N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, and F. Scarcello. The DLV System for Knowledge Representation and Reasoning. *ACM TOCL*, 7(3):499–562, 2006.
21. Y. Lierler. Disjunctive Answer Set Programming via Satisfiability. In *LPNMR'05*, LNCS 3662, pp. 447–451. 2005.
22. Y. Lierler. Abstract Answer Set Solvers. In *Logic Programming, 24th International Conference (ICLP 2008)*, LNCS 5366, pp. 377–391. 2008.
23. V. Lifschitz. Answer Set Planning. In *ICLP'99*, pp. 23–37.
24. V. W. Marek and M. Truszczyński. Stable models and an alternative logic programming paradigm. *CoRR*, cs.LO/9809032, 1998.
25. M. Mariën, J. Wittocx, M. Denecker, and M. Bruynooghe. 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)*, LNCS, pp. 211–224, Guangzhou, China, 2008.
26. A. Masoni, M. Carpinelli, G. Fenu, A. Bosin, D. Mura, I. Porceddu, and G. Zanetti. Cybersar: A lambda grid computing infrastructure for advanced applications. In *Nuclear Science Symposium Conference Record (NSS/MIC), 2009 IEEE*, pp. 481–483. IEEE, 2009.
27. I. Niemelä. Logic Programs with Stable Model Semantics as a Constraint Programming Paradigm. In *Proceedings of the Workshop on Computational Aspects of Nonmonotonic Reasoning*, pp. 72–79, Trento, Italy, 1998.
28. E. Nudelman, K. Leyton-Brown, H. H. Hoos, A. Devkar, and Y. Shoham. Understanding random SAT: Beyond the clauses-to-variables ratio. In *Proc. of the 10th International Conference on Principles and Practice of Constraint Programming (CP)*, pp. 438–452, Toronto, Canada, 2004.
29. L. Pulina and A. Tacchella. A multi-engine solver for quantified boolean formulas. In *Proc. of the 13th International Conference on Principles and Practice of Constraint Programming (CP)*, pp. 574–589, Providence, Rhode Island, 2007.
30. L. Pulina and A. Tacchella. A self-adaptive multi-engine solver for quantified boolean formulas. *Constraints*, 14(1):80–116, 2009.
31. J.R. Quinlan. *C4.5: programs for machine learning*. Morgan kaufmann, 1993.
32. J. R. Rice. The algorithm selection problem. *Advances in Computers*, 15:65–118, 1976.
33. P. Simons, I. Niemelä, and T. Soininen. Extending and Implementing the Stable Model Semantics. *AI*, 138:181–234, 2002.
34. smt-lib-web. The Satisfiability Modulo Theories Library, 2011. `http://www.smtlib.org/`.
35. J. Wittocx, M. Mariën, and M. Denecker. 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)*, pp. 153–165, Leuven, Belgium, 2008.
36. Lin Xu, F. Hutter, H. H. Hoos, and K. Leyton-Brown. SATzilla: Portfolio-based algorithm selection for SAT. *JAIR*, 32:565–606, 2008.