Applying Machine Learning Techniques to ASP Solving

Marco Maratea¹, Luca Pulina², and Francesco Ricca³

¹ DIBRIS, Università degli Studi di Genova
Viale F.Causà 15, 16145 Genova, Italy
marco@dist.unige.it

² POLCOMING, Università degli Studi di Sassari
Viale Mancini 5, 07100 Sassari, Italy
lpulina@uniss.it

³ Dipartimento di Matematica, Università della Calabria
Via P. Bucci, 87030 Rende, Italy
ricca@mat.unical.it

Abstract

Having in mind the task of improving the solving methods for Answer Set Programming (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, given the features of the instances in a training set and the solvers performance on these instances, we apply a classification method to inductively learn algorithm selection strategies to be applied to a test set. We report the results of an experiment considering solvers and training and test sets of instances taken from the ones submitted to the “System Track” of the 3rd 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 a higher number of instances compared with any solver that entered the 3rd ASP competition.

1998 ACM Subject Classification D.1.6 Logic Programming, I.2.4 Knowledge Representation Formalisms and Methods, I.2.6 Learning

Keywords and phrases Answer Set Programming, Automated Algorithm Selection, Multi-Engine solvers

Digital Object Identifier 10.4230/LIPIcs.xxx.yyy.p

1 Introduction

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 Answer Set Programming (ASP) [13, 27, 30, 26, 14, 3], 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, e.g., [36, 22, 24, 10, 28, 21, 36], 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 [34]. Related approaches, following a per-instance selection, have been exploited for solving propositional satisfiability (SAT),
Applying Machine Learning Techniques to ASP Solving

e.g., [40], and Quantified SAT (QSAT), e.g., [32] problems. In ASP, an approach for selecting the “best” CLASP internal configuration is followed in [9], while another approach that imposes learned heuristics ordering to SMODELS is [2].

In this paper we pursue this direction, and design 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 benchmark submitted to the “System Track” of the 3rd ASP Competition [5] 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, starting from the features of the instances in a training set, and the solvers performance on these instances, we apply the “Nearest-neighbor” classification method to inductively learn general algorithm selection strategies to be applied to a test set. We perform an analyses that consider as test set the instances evaluated to the 3rd ASP competition.

Our experiments show that it is possible to obtain a very robust performance, by solving a higher number of instances than all the solvers that entered the 3rd ASP competition and DLV [22].

The paper is structured as follow. Section 2 contains preliminaries about ASP and classification methods. 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 and 7 end the paper with discussion about related work and conclusions, respectively.

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) [13, 27, 30, 26, 14, 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 [26].

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 [14, 3, 12] for complementary introductory material on ASP, and to [4] for obtaining the full specification of ASP-Core.

2.1.0.1 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 \( \neg p \), where \( p \) is an atom. A (disjunctive) rule \( r \) is of the form:

\[ a_1 \lor \ldots \lor a_n : \neg b_1, \ldots, \neg b_k, b_{k+1}, \ldots, b_m. \]

where \( a_1, \ldots, a_n, b_1, \ldots, b_m \) are atoms. The disjunction \( a_1 \lor \ldots \lor a_n \) is the head of \( r \), while the conjunction \( b_1, \ldots, b_k, \neg b_{k+1}, \ldots, \neg b_m \) is the body of \( r \). We denote by \( H(r) \) the set of atoms
occuring 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 \( \neg \) sign is omitted); if \( |H(r)| = 0 \) (i.e., \( n = 0 \)) it is called a 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., \( \lor \)-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.

2.1.0.2 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 \( r \), \( \text{Ground}(r) \) 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 \( \text{Ground}(\mathcal{P}) = \bigcup_{r \in \mathcal{P}} \text{Ground}(r) \).

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 \( \neg A \) is true w.r.t. \( I \) if \( A \) is false w.r.t. \( I \); otherwise \( \neg A \) 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 [14] 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 \( \text{Ground}(\mathcal{P})^I \).

As an example consider the program \( \mathcal{P} = \{ \ a \lor b :- c \ , \ b :- \neg a \ , \ a \lor c :- \neg b \ , \ k :- a \ , \ k :- b \ \} \) and \( I = \{ b , k \} \). The reduct \( \mathcal{P}^I \) is \( \{ a \lor b :- c \ , \ b : k :- a , \ k :- b \} \). \( I \) is an answer set of \( \mathcal{P}^I \), and for this reason it is also an answer set of \( \mathcal{P} \).

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. 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.

More in detail, in multinomial classification we are given a set of patterns, i.e., input vectors \( X = \{ x_1 , \ldots , x_k \} \) with \( x_i \in \mathbb{R}^n \), 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
4 Applying Machine Learning Techniques to ASP Solving

<table>
<thead>
<tr>
<th>Problem</th>
<th>Class</th>
<th>#Instances</th>
</tr>
</thead>
<tbody>
<tr>
<td>DisjunctiveScheduling</td>
<td>NP</td>
<td>10</td>
</tr>
<tr>
<td>GraphColouring</td>
<td>NP</td>
<td>60</td>
</tr>
<tr>
<td>HanoiTower</td>
<td>NP</td>
<td>59</td>
</tr>
<tr>
<td>KnightTour</td>
<td>NP</td>
<td>10</td>
</tr>
<tr>
<td>MazeGeneration</td>
<td>NP</td>
<td>50</td>
</tr>
<tr>
<td>Labyrinth</td>
<td>NP</td>
<td>261</td>
</tr>
<tr>
<td>MultiContextSystemQuerying</td>
<td>NP</td>
<td>73</td>
</tr>
<tr>
<td>Numberlink</td>
<td>NP</td>
<td>150</td>
</tr>
<tr>
<td>PackingProblem</td>
<td>NP</td>
<td>50</td>
</tr>
<tr>
<td>SokobanDecision</td>
<td>NP</td>
<td>50</td>
</tr>
<tr>
<td>Solitaire</td>
<td>NP</td>
<td>25</td>
</tr>
<tr>
<td>WeightAssignmentTree</td>
<td>NP</td>
<td>62</td>
</tr>
<tr>
<td>MinimalDiagnosis</td>
<td>Beyond NP</td>
<td>551</td>
</tr>
<tr>
<td>StrategicCompanies</td>
<td>Beyond NP</td>
<td>51</td>
</tr>
</tbody>
</table>

| Table 1 Problems and instances considered, coming from the NP and Beyond NP classes of the 3rd ASP competition.

unknown function $f : \mathbb{R}^n \rightarrow \{1, \ldots, m\}$ applied to the patterns, i.e., $f(x_i) = y_i$ 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 $x^* \in X$ we should ensure that $c(x^*)$ is equals to $f(x^*)$. 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 benchmarks considered for the experiments belong to the suite of the 3rd 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 ASP-Core. That suite includes planning domains, temporal and spatial scheduling problems, combinatorial puzzles, graph problems, and a number of application domains i.e., database, information extraction and molecular biology field. More in 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 [4]. Note that this is a superset of the instances actually selected for running (and, thus

---

1 An exhaustive description of the benchmark problems can be found in [4].
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) [11] 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 in our experiments that entered the System Track of the 3rd ASP Competition [4] with the addition of DLV [22] (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 [10], CLASP-D [7], CLASPFOLIO [9], IDP [39], CMODELS [24], SUP [25], SMODELS [36], and several solvers from both the LP2SAT [20] and LP2DIFF [21] 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; CLASP-D 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 [28]; SMODELS is one of the first robust native ASP solvers that have been made available to the community; DLV [22] 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 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 [8] for actually computing the answer sets; the LP2DIFF family translates programs in difference logic over integers [37] and exploit Z3 [6] as underlying solver (again, G, L and LG indicate different translation strategies). 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 [29], 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 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 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.
Applying Machine Learning Techniques to ASP Solving

<table>
<thead>
<tr>
<th>Solver</th>
<th>Solved</th>
<th>Unique</th>
<th>Solver</th>
<th>Solved</th>
<th>Unique</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLASP</td>
<td>445</td>
<td>26</td>
<td>LP2DIFFGZ3</td>
<td>307</td>
<td>–</td>
</tr>
<tr>
<td>CMODELS</td>
<td>333</td>
<td>6</td>
<td>LP2SAT2GMINISAT</td>
<td>328</td>
<td>–</td>
</tr>
<tr>
<td>DLV</td>
<td>241</td>
<td>37</td>
<td>LP2SAT2LGMINISAT</td>
<td>322</td>
<td>–</td>
</tr>
<tr>
<td>IDP</td>
<td>419</td>
<td>15</td>
<td>LP2SAT2LGMINISAT</td>
<td>324</td>
<td>–</td>
</tr>
<tr>
<td>LP2DIFFGZ3</td>
<td>254</td>
<td>–</td>
<td>LP2SAT2MINISAT</td>
<td>336</td>
<td>–</td>
</tr>
<tr>
<td>LP2DIFFLGZ3</td>
<td>242</td>
<td>–</td>
<td>SMODELS</td>
<td>134</td>
<td>–</td>
</tr>
<tr>
<td>LP2DIFFLZ3</td>
<td>248</td>
<td>–</td>
<td>SUP</td>
<td>311</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 2 Results of a pool of ASP solvers on the NP instances of the 3rd 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 seconds, and, finally, in column “Unique” we report the total amount of instances solved uniquely by the corresponding solver.

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., [32]) 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 [31]. The problem size features are: number of rules \( r \), number of atoms \( a \), ratios \( r/a \), \((r/a)^2\) and \((r/a)^3\) 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 instances out of a total of 1462 in less than 600s. 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 solver (SOTA), i.e., considering a problem instance, the oracle that always fares the best among available solvers. In order to do that, 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 CLASP-D and CLASPFOLIO. Concerning the results of CLASP-D, we report that –considering the NP class– its performance 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 ones related to 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 CLASP-D, CMODELS, and DLV. Considering that both CMODELS and DLV are involved in the previous selection, the

\[2\] The exceptions are 10 and 27 instances of DisjunctiveScheduling and PackingProblem, respectively.
Table 3 Results of the various solvers on the grounded instances evaluated at the 3rd ASP competition. ME-ASP (NN) has been trained on the TS1 training set.

<table>
<thead>
<tr>
<th>Solver</th>
<th>NP</th>
<th>Beyond NP</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>#Solved</td>
<td>Time</td>
<td>#Solved</td>
</tr>
<tr>
<td>CLASP</td>
<td>60</td>
<td>5132.45</td>
<td>–</td>
</tr>
<tr>
<td>CLASP D</td>
<td>–</td>
<td>–</td>
<td>13</td>
</tr>
<tr>
<td>CMODELS</td>
<td>56</td>
<td>5092.43</td>
<td>9</td>
</tr>
<tr>
<td>DLV</td>
<td>37</td>
<td>1682.76</td>
<td>15</td>
</tr>
<tr>
<td>IDP</td>
<td>61</td>
<td>5010.79</td>
<td>–</td>
</tr>
<tr>
<td>ME-ASP (NN)</td>
<td>66</td>
<td>4854.78</td>
<td>15</td>
</tr>
<tr>
<td>CLASPFOLIO</td>
<td>62</td>
<td>4824.06</td>
<td>–</td>
</tr>
<tr>
<td>SOTA</td>
<td>71</td>
<td>5403.54</td>
<td>15</td>
</tr>
</tbody>
</table>

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

pool of engines used in ME-ASP will be composed of 5 solvers, namely CLASP, CLASP D, CMODELS, DLV, and IDP.

4.3 Classification algorithms and training

The classification method employed in our analysis is Nearest-neighbor (NN), already considered in [32] in QBF solving: 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.

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.

As result of the considerations above, we design a training set--TS1 in the following--composed of the 320 instances solved uniquely--without taking into account the instances involved in the 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.

Our next experiment is devoted to training the classifier, and to assessing its 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 \(\hat{f}\). 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., \(\hat{c}\) fails to predict \(f(x^*)\) when \(x^* \neq \hat{x}\). 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 \(i \in \{1, \ldots, 10\}\) 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.

We finally report the accuracy results related to the experiment described above for our classification method: 92.81%.
5 Performance analysis

In this section we present the results of the analysis we have performed. We consider the training sets T51 introduced in Section 4, composed of uniquely solved instances, and as test set the successfully grounded instances evaluated at the 3rd ASP Competition (a total of 88 instances): the goal of this analysis is to test the efficiency of our approach on all the evaluated instances when the model is trained on the whole space of the uniquely solved instances.

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, Beyond NP classes, and on both classes, 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). About the last column, numbers are reported only for ME-ASP and the engines that have been selected on both classes in Section 4.2 (note that CLASP always performs worse than CLASP on NP instances, and CLASPFOIOL can only handle NP instances).

We report the results obtained by running: ME-ASP with the NN classification method introduced in Section 4.3, denoted with ME-ASP(NN) the component engines employed by ME-ASP on each class as explained in Section 4.2, CLASPFOIOL and SOTA, which is the ideal 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 timings include both the time spent for extracting the features from the ground instances, and the time spent by the classifier.

Results are shown in Table 3. We can see that, on problems of the NP class, ME-ASP(NN) solves the highest number of instances, 5 more than IDP, 6 more than CLASP and 4 more than CLASPFOIOL, that we remind the fastest solver in the NP class that entered the System Track of the competition. On the Beyond NP problems, instead, ME-ASP(NN) and DLV solve 15 instances (DLV having best mean CPU time), followed by CLASP and CMODELS, which solve 13 and 9 instances, respectively.

It is interesting to report the overall result of CLASP, i.e., the overall winner of the System Track of the competition on both NP and Beyond NP classes: it solves a total of 62 instances (i.e., 52 NP instances and 13 Beyond NP instances), thus a total of 19 instances less than ME-ASP(NN).

Summarizing, ME-ASP(NN) is the solver that solves the highest number of instances in comparison with (i) its engines, (ii) CLASPFOIOL, i.e., the fastest solver in the NP class that entered the System Track of the competition, and (iii) CLASP, i.e., the overall winner of the System Track of the competition. It is further 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.

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 [34] defined the algorithm selection problem as the problem of finding an effective, or good, or best algorithm, based on an abstract model of the problem at hand. Along this line, several works have been done to tackle combinatorial problems efficiently. [16, 23] described 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. Most related papers to our work are [40, 32] for solving SAT and QSAT problems. Both [40] and [32] rely on a per-instance analysis, like the one we have performed in this paper: in [32], 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. The approach in [40] 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 approach. The algorithm portfolio approach is employed also in, e.g., [16] on Constraint Satisfaction and MIP, [35] on QSAT and [15] on planning problems. The advantage of the algorithm portfolio over a multi-engine is that it is possible, by combining algorithms, to reach, in each instance, better performance than the best engine, while this is the bound for a multi-engine solver. On the other hand, an algorithm portfolio needs internal changes in the code of the engines, while the multi-engine treats the engines as black-box, thus no internal modification, even minor, is requested, resulting in higher modularity for this approach: when a new engine is added, there is just the need to update the model. It has to be noticed that both [32] and [40] reached very good results, e.g., AQME, the multi-engine solver implementing the approach in [32] had top performance at the 2007 QBF competition.\footnote{http://www.qbflib.org/qbfeval.}

Other approaches work by designing methods for automatically tuning and configuring the solver parameters: this approach is followed in, e.g., [19, 18] for solving SAT and MIP problems, and [38] for planning problems. An overview can be found in [17]. In ASP, the approach implemented in CLASPFOLIO [9] 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. In CLASPFOLIO, features are extracted by means of the CLASPRE tool. Thus, like the algorithms 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. An alternative approach is followed in the DORS framework of [2], 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.

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 a multinomial classification method to learning algorithm selection strategies; (iii) implemented these techniques in our multi-engine solver ME-ASP, which is available for download at \url{http://www.mat.unical.it/ricca/me-asp}. The performance of ME-ASP was assessed on an experiment, which was conceived for checking efficiency of our approach, involving training and test sets of instances taken from the ones submitted to the System Track of the 3rd 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 would like to thank Marcello Balduccini for useful discussion on his solver DORS.
References


 Applying Machine Learning Techniques to ASP Solving