MULTI-ENGINE ASP SOLVING WITH POLICY ADAPTATION

Marco Maratea[1], *Luca Pulina and Francesco Ricca*[2]

October 8, 2012

University of Sassari

Computer Vision Laboratory

Technical Report No. CVL -2012-004

UNIVERSITY
of
SASSARI

MULTI-ENGINE ASP SOLVING WITH POLICY ADAPTATION

Marco Maratea[1], *Luca Pulina and Francesco Ricca*[2]

University of Sassari
Computer Vision Laboratory
Porto Conte Ricerche - Loc. Tramariglio
07041 Alghero (SS)
`www.uniss.it`

October 8, 2012

Technical Report No. CVL -2012-004

Abstract

The recent application of Machine Learning techniques to the Answer Set Programming (ASP) field proved to be effective. In particular, the multi-engine ASP solver ME-ASP is efficient: it is able to solve more instances than any other ASP system that participated to the 3rd ASP Competition on the "System Track" benchmarks. In the ME-ASP approach, classification methods inductively learn off-line algorithm selection policies starting from both a set of *features* of instances in a *training* set, and the solvers performance on such instances. In this paper we present an improvement to the multi-engine framework of ME-ASP, in which we add the capability of updating the learned policies when the original approach fails to give good predictions. An experimental analysis, conducted on training and test sets of ground instances obtained from the ones submitted to the "System Track" of the 3rd ASP Competition, shows that the policy adaptation improves the performance of ME-ASP when applied to test sets containing domains of instances that were not considered for training.

1 Introduction

Answer Set Programming [3, 9, 14, 15, 30, 33] (ASP) 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 system to find such solutions [26]. The language of ASP is very expressive: it can be used to solve all problems in the second level of the polynomial hierarchy [9]. Moreover, on the side of *computation* a number of efficient ASP systems is available, e.g., [11, 21, 22, 24, 31, 40]; nonetheless, there is room for improvement. For instance, the recent application of Machine Learning techniques to ASP solving has contributed to push forward the state of the art. Machine-Learning-based approaches to ASP solving range from algorithm portfolios [10], to learning heuristics orders [2], to multi-engine solvers [27, 28]. In particular, the latter approach is very promising; indeed, the multi-engine ASP system ME-ASP [29] was able to solve (see [27, 28]) more ground instances than any other ASP solver that participated to the 3rd ASP Competition [5] on the "System Track" benchmarks. In the multi-engine approach, classification methods inductively learn engine selection policies, starting from a set of *features* of instances in a *training* set, and the solvers performance on such instances. Basically, Machine Learning techniques are applied for inductively choosing, among a set of available ones, the "best" ASP solver on a per-instance basis. In [28, 27] ME-ASP engines were selected among the ones that entered the 3rd ASP Competition, plus DLV [22], and the learned algorithm selection policies are decided during training and are never updated. This is similar to what was

[1]`marco@dist.unige.it`, DIST, Università degli Studi di Genova, Viale F.Causa 15, 16145 Genova, Italy
[2]`ricca@mat.unical.it`, Dipartimento di Matematica, Università della Calabria, Via P. Bucci, 87030 Rende, Italy

done in QSAT solving [36], and also portfolio-based ASP solvers do not modify their policies on-line; whereas there are already approaches in QSAT and ASP [18, 37] that exploit methods for updating the policies. A consequence of using selection policies that are learned off-line in the ME-ASP framework is that the multi-engine solver might do suboptimal predictions when the instances in input belong to "unseen" domains (i.e., belong to family of instances that were not known or were not considered in the training phase). Despite ME-ASP already showed (see [27, 28]) good performance also on unseen domains, there is room for improvement in this specific setting for the above considerations.

In this paper we cope with this issue. In particular, we introduce a *retraining* procedure in the framework for multi-engine ASP solving, which has the ability of updating the learned policies when the original approach fails to give good predictions. In particular, our proposal relies on a method that classifies instances according to their "similarity" (defined by some measures computed on the features), and a policy that (*i*) first tries to solve the program by granting to all engines a fixed amount of time, starting from the predicted engine; then, (*iia*) in case all these runs fail, grants all the remaining time to the predicted engine; or (*iib*) updates the policy if the program was solved by an engine different from the one predicted at the beginning. A similar approach was already employed in [37], where it lead to positive results on QSAT instances. We have implemented these ideas in the multi-engine system ME-ASP [29], obtaining the enhanced system ME-ASP^A, i.e., ME-ASP with policy Adaptation. Moreover, we conducted an experimental analysis, considering training and test sets of ground instances taken from the ones submitted to the "System Track" of the 3rd ASP Competition [5]. The analysis focuses on settings that are challenging for ME-ASP, i.e., whose test sets contain a large number of domains of instances that were not considered for training. The results show that the ability of ME-ASP^A to adapt the algorithm selection policy on-line can significantly improve the performance of the multi-engine system on these settings. To sum up, the main contributions of this paper are:

1. The extension of the framework for multi-engine ASP solving with policy adaptation.
2. The implementation of an extended version of the multi-engine solver ME-ASP, called ME-ASP^A, that features on-line adaptation of the engine selection policy.
3. An experimental analysis involving the enhanced system ME-ASP^A, performed on the computationally-hard benchmarks of the 3rd ASP Competition, that highlights the advantages of the new solutions when the test set contains a large number of domains that were not used for training.

The paper is structured as follows. Section 2 introduces basic concepts about ASP and classification methods. Section 3 then describes our benchmark setting in terms of dataset, solvers and hardware employed. Section 4 shows the architecture of the multi-engine ASP solver ME-ASP, and the choices made for its basic components. Section 5 shows the performance analysis, while Section 6 and 7 end the paper with discussion about the related work and conclusions.

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 [5], which is the language specification that was originally introduced in the 3rd ASP Competition [5], as well as the one supported by our system and 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. *Terms* are variables and constants. An *atom* is of the form $p(t_1, ..., t_n)$, where p is a *predicate* and $t_1, ..., t_n$ are terms, and n is the arity of p. 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 *head* of r is the disjunction $a_1 \vee \ldots \vee a_n$, while the conjunction $b_1, \ldots, b_k, \texttt{not}\ b_{k+1}, \ldots, \texttt{not}\ b_m$ is the *body* of r. The set of atoms occurring in the head of r is denoted by $H(r)$, and $B(r)$ denotes the set of body literals. A rule s.t. $n = 1$ is called a *normal rule*; and if $k = m = 0$ (i.e., the body is empty) it is called a *fact* (and the :− sign is omitted); if $n = 0$ (i.e., empty head) 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., $\vee$-free) program is called *positive* (resp., *normal*). A term, an atom, a literal, a rule, or a program is *ground* (or *propositional*) 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 $Ground(\mathcal{P})^I$.

As an example consider the program $\mathcal{P} = \{\ a \vee b \,{:}{-}\, c.,\ b \,{:}{-}\, \texttt{not}\ a, \texttt{not}\ c.,\ a \vee c \,{:}{-}\, \texttt{not}\ b.,$ $k \,{:}{-}\, a.,\ k \,{:}{-}\, b.\ \}$ and $I = \{b, k\}$. The reduct $\mathcal{P}^I$ is $\{a \vee b \,{:}{-}\, c.,\ b.\ k \,{:}{-}\, a.,\ k \,{:}{-}\, b.\}$. 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

The results of the recent ASP competitions show that ASP solvers are not that robust, i.e., able to perform well across different problem domains. Considering hard combinatorial problems, this is a not surprising behavior: every heuristic algorithm will find problem instances that are exceptionally hard to solve, while the same instances can easily be solved by another algorithm, or by using a different heuristic. In this work, we model the problem using *multinomial classification* algorithms, i.e., Machine Learning techniques that allow automatic classification of a set of instances, given some sets of numeric values representing syntactic characteristics of the instances, i.e., the features. Leveraging on such kind of per-instance selection algorithm, it is possible to select in an automatic way the best algorithm among a pool of them –in our case, tools to solve ASP instances. 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 data concerning the hardware platform, benchmarks and ASP solvers employed, as well as and the execution settings for reproducibility of experiments.

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
PartnerUnitsPolynomial	*NP*	65
SokobanDecision	*NP*	50
Solitaire	*NP*	25
StableMarriage	*NP*	56
WeightAssignmentTree	*NP*	62
MinimalDiagnosis	*Beyond NP*	551
StrategicCompanies	*Beyond NP*	51
Total		1583

Table 1: Problems and instances.

3.1 Executables and Hardware Settings

We have run all the ASP solvers 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. We have run: CLASP [11], CLASPD [7], CLASPFOLIO [10], IDP [43], CMODELS [24], SUP [25], SMODELS [40], and several solvers from both the LP2SAT [20] and LP2DIFF [21] families, namely: LP2GMINISAT, LP2LMINISAT, LP2LGMINISAT, LP2MINISAT, LP2DIFFGZ3, LP2DIFFLGZ3, LP2DIFFLZ3, and LP2DIFFZ3. 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 option of CLASP; IDP is a finite model generator for extended first-order logic theories, which is based on *MiniSatID* [31]; 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 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 on MINISAT [8] for actually computing the answer sets. Finally, the LP2DIFF family translates programs in difference logic over integers [41] and exploit *Z3* [6] as underlying solver (again, G, L and LG indicate different translation strategies). DLV was run with default setting, while the 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 a cluster of Intel Xeon E31245 PCs at 3.30 GHz equipped with 64 bit Lubuntu 12.04. Unless otherwise specified, the resources granted to the solvers are 3600s of CPU time and 4GB of memory. Time measurements were carried out using the `time` command shipped with Lubuntu.

3.2 Dataset

The benchmarks used in this paper belong to a large and heterogeneous suite of benchmarks encoded in ASP-Core that has been submitted to the 3rd ASP Competition [5]. Such benchmarks are related to a wide range of combinatorial problems, including, e.g., planning, temporal and spatial scheduling problems, combinatorial puzzles, and graph problems, related to a number of application domains, e.g., database, information extraction and molecular biology field.

The set of benchmarks considered in this work is reported in Table 1 where they are classified according to the problem they solve, the corresponding complexity class, and the total amount of instances available. We considered only computationally-hard benchmarks, corresponding to all problems belonging to the categories *NP* and *Beyond NP* of the competition, together with *StableMarriage* and *PartnerUnitsPolynomial*, which are problems that can be solved in polynomial time but featured in the competition a natural declarative encoding making usage of disjunction and, thus, can not be solved by the grounder. Both problems are classified in this paper as *NP* for simplicity, even if they are not hard for this complexity class; however, note that no solver is able to detect from the provided encoding that the corresponding problem instance could be evaluated by employing a different (i.e., cheaper) strategy from the one employed for evaluating "true" *NP* problems. For the domains listed in Table 1 we employ a superset of the instances actually *evaluated* to System Track of the competition. In particular, we considered all the 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]. ME-ASP is a solver for propositional programs, thus to obtain a net measure of its performance we have first grounded all the mentioned instances by using GrinGo (v.3.0.3) [12]. Thus, we actually considered the 1540 instances (out of a total of 1583 instances) that we were able to ground with GrinGo in less than 3600s, of which 938 are NP instances. (The exceptions are 43 instances of the `PackingProblem` domain.) In the following, with *instance* we refer to a ground ASP program, which was obtained by running GrinGo on the corresponding ASP program made of non-ground encoding+facts that is available from the competition web site [4].

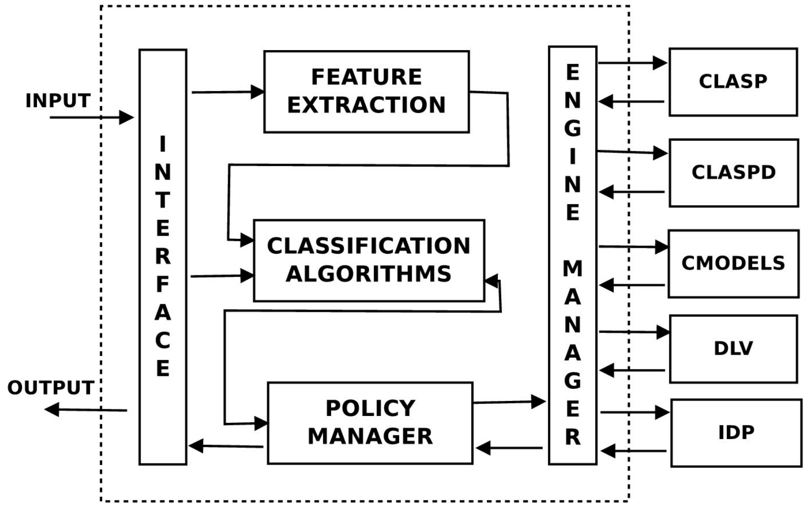

Figure 1: The architecture of ME-ASP^A. The dotted box denotes the whole system and, inside it, each solid box represents its modules. Arrows denote functional connections between modules.

4 Design and Architecture of ME-ASP with Policy Adaptation

In this section we present the architecture of ME-ASP^A describing the modules composing a multiengine ASP solver with policy adaptation; moreover, at the same time, we describe the key design and implementation choices behind the development our system. The general architecture of a multiengine ASP solver with policy adaptation is depicted in Figure 1. In the following we describe, in a separate paragraph, the functionality of each module and the way it was designed and implemented.

`INTERFACE.` It manages both the input received by the user and the output of the whole system. It also dispatches the input data to the remaining modules, as denoted by the outgoing arrows. In particular, `INTERFACE` collects (*i*) the ground ASP program in ASP-Core format [5], (*ii*) the classifier and (*iii*) its inductive model.

`FEATURE EXTRACTION.` This module extracts a number of *syntactic* features from the input program, which will be used for its classification. For each ground program, we consider only "cheap-to-compute" features, i.e., computable in linear time in the size of the program. The set of features employed by ME-ASP (see [27] for more details) can be divided into four groups (such a categorization is borrowed from [34]):

- **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 and number of occurrences in a horn rule for each atom;
- **ASP specific features**: number of true and disjunctive facts, fraction of normal rules and constraints c;

plus some of their combinations (e.g., c/r), for a total number of 52 computed features.

The main motivations that lead to the choice of easy to compute features are the following: first, we have to consider that the time spent computing the features will be integral part of our solving process: thus, in case "costly" features are considered, the risk is to spend too much time in calculating the features of a program. Second, the selected features are sufficient to obtain a robust classification process, as it is witnessed by our experiments. In order to corroborate the first point, we run preliminary experiments with CLASPFOLIO, a portfolio ASP solver that relies on some "costly" features, e.g., number of Strongly Connected Components and loops: it turned out that CLASPFOLIO feature extractor could not compute all its features for a significant number of programs in a reasonable CPU time.[1] The same choice was also done in related works (see e.g., [36]) where syntactic features have been profitably used.

We then implemented a feature extractor tool that is able to compute the above-reported set of features. Concerning its performance we report that it can compute all the features (in less than 3600s) for 1500 programs out of the 1540 available in our benchmarks. The distribution of the CPU times (in seconds) for extracting features is characterized by the following five numbers: 0.002, 2.961, 4.720, 10.596, 3211.041. We remark that, the highest CPU times correspond processing instances for ground programs whose size is in the order of tens of GB.

`ENGINES.` The final set of engines of ME-ASP^A, as depicted in Figure 1 is composed by five state-of-the-art ASP solvers, namely CLASP [11] and its disjunctive version CLASPD [7], CMODELS [24], DLV [22], and IDP [31]; nonetheless, the architecture of ME-ASP^A is modular and allows one to easily update the engines set with additional solvers. Finally note that engines are used as "black-boxes", i.e., ME-ASP^A interacts with them via system calls.

The selection of these engines arise from the idea to collect a pool of solvers that is representative of the state-of-the-art solver (SOTA), i.e., the oracle that always fares the best among the available solvers. In order to do that, we have done some preliminary experiment (details reported in [27]). This is done by selecting the solvers that are able to solve a noticeable amount of instances *uniquely*. This analysis revealed that, concerning *NP* problems, only 4 solvers out of the sixteen mentioned in Section 3 are able to give an effective contribution, 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, namely CLASPD, CMODELS, and DLV. Considering that both CMODELS and DLV are involved in the previous selection, the pool of engines used in ME-ASP^A is composed by the 5 solvers mentioned above.

[1]In a preliminary experiment, see [27], with different hardware setting, CPU time limited to 600s, and a memory limit of 2GB, it turned out that CLASPFOLIO feature extractor could compute all its features for 573 out of 823 *NP* instances belonging to the ASP Competition instance set.

`ENGINE MANAGER AND POLICY MANAGER.` The interaction with the engines is handled by `ENGINE MANAGER`. It receives from `POLICY MANAGER` data about the engine to fire. At the end of the engine computation, `ENGINE MANAGER` returns to `POLICY MANAGER` the result. This is the module that contains the retraining procedure. It also works as a coordinator of ME-ASP^A modules, and, finally, it provides the final result to `INTERFACE`.

Given the engines available, we need to define what is the policy to be used when the classifier fails to give a good prediction. A general behavior of heuristic-based solvers that deal with hard combinatorial problems is that systems have a point at which the time taken to find solutions starts to increase dramatically. After this point, called Peter Principle Point (PPP) – see, e.g., [42], a linear increase in the computational resources would not lead to the solution of a noticeable amount of additional problems. It is also the case for the considered ASP solvers, as we will detail in the next section.

In the light of the above considerations, the method we employ exploits the PPP, granting to the predicted solver a small (w.r.t. the global time limit) amount of CPU time t, and, in the case of failure, run in sequence all the remaining engines for the same time t. If all solvers fail to solve the instance, the remaining time to the timeout is all granted to the predicted engine. If the successful engine is different to the predicted one, a new pattern, labeled with the fired engine, is added to the training set, and the classification algorithm is retrained; thus, the selection policy is updated.

The rationale about this choice is that we mostly trust the prediction coming from the learned model, but we also give a chance to the other engines. This method has been introduced and implemented in [37] in QSAT solving, where it is called TPE (Trust the Predicted Engine). The value of t will be empirically determined considering the distribution of the solving times of the various engines; we will discuss this point in the next section.

`CLASSIFICATION ALGORITHMS AND TRAINING.` This module receives as input the classifier and its inductive model (from `INTERFACE`) and a vector of features (from `FEATURE EXTRACTION`). It returns to `POLICY MANAGER` the name of the predicted engine.

The classifier considered in this paper is *Nearest-neighbor*, NN in the following. NN 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, and training instances are stored in order to have fast look-up, see, e.g., [1]. NN is built on top of the RAPIDMINER library [32].

The reasons for choosing this classification method are manifold, and summarized in the following: first, it is the classification method employed in the original paper on multi-engine ASP solving [28]; second, we already had some experience in coupling this classifier with policy adaptation methods, given that one of the authors dealt with this issue in the multi-engine QSAT solver AQME; finally, in [35] it has been shown that – in the case of QSAT problems and solvers – NN coupled with the TPE policy is the best choice w.r.t. some issues as the growing-up of the training set due to the retraining process if compared with other classification algorithms, e.g., decision trees, decision rules, and sub-symbolic learning algorithms.

To give a hint of the performance of NN, in comparison with other classification methods in a multi-engine framework for ASP solving, we direct the reader to [27],

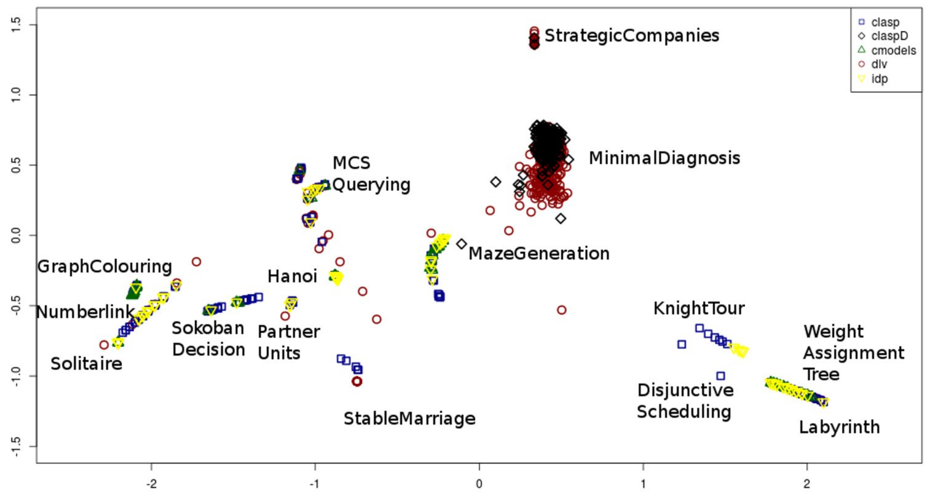

Figure 2: Dataset coverage: two-dimensional space projection of the whole dataset.

where it is possible to find such analysis within ME-ASP. The analysis shows that NN has good performance w.r.t. other classifiers; moreover, Section 5 will confirm its good behavior also in ME-ASP^A.

As mentioned in Section 2.2, in order to train our classifier, we have to select a pool of instances for training purpose, called the training set. In order to give an idea about what is the distribution of all our available instances, we depict in Figure 2 the coverage of whole available dataset. In particular, the plot reports 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 2 we add a label denoting the problem name of the depicted instances, in order to give an idea about the "location" of each benchmark.

As a result of the considerations above, we designed our reference training set (TS in the following) that is composed of the 316 instances solved uniquely – without taking into account the instances evaluated in the competition – by the pool of engines previously selected. The rationale of this choice is to try to "mask" noisy information during model training to obtain a robust model. We can think about this training set as being the one designed by a domain expert that perfectly knows what are the "good" instances to be considered for each domain for training.

In order to test the effectiveness of the policy adaptation in ME-ASP, the idea is to design further, challenging training sets having the following desiderata: a very limited number of instances, coming from only one problem, and such that each engine solves at least one instance uniquely, or a large amount of selected engines are SOTA solvers on the instances of such problems. The rationale of employing these additional training sets is to test our solution on very challenging and corner cases. On these settings we expect (i) ME-ASP not to perform that well, and (ii) the retraining solution of ME-ASP^A to be useful. In Section 5 we will see what are the problems on which building these training sets, and the intuition about the results will be confirmed.

We then trained the classifier. 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 w.r.t. 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.

5 Experimental Analysis

In the experimental results reported in [27], we have shown how such performance of ME-ASP – in terms of total amount of solved instances – can be obtained through an appropriate design of the training set, that in this work correspond to TS. In [27] we also showed that changes in the training set could lead to a degradation impact on ME-ASP performance when it deals with "unseen" problems, i.e., in a situation where instances in the test set have a feature configuration that is completely unknown to the model. In practice, when a reasoning task is modeled and encoded into ASP, it is very difficult to establish in advance how the syntactic structure of the computed instances will fit to a trained model. Our main goal in the design of ME-ASP^A is thus to improve the performance in situations that are challenging without retraining.

This section is devoted to test the benefits of an on-line policy adaptation in the case of the test set is composed of "unseen" instances – i.e., belonging to domains that were left unknown during training. This is a challenging experiment for ME-ASP, because the models are not trained on all the space of the uniquely solved instances; here, we made the settings even harder by considering a single domain in the training sets.

The first step is thus devoted to define training sets that are challenging for ME-ASP, having the characteristics mentioned in the previous section. The second step in order to use ME-ASP^A is to tune the self-adaptive component doing some considerations about runtime performance of its engines. Finally, we run the engines, ME-ASP and ME-ASP^A on the chosen benchmarks. We devoted one subsection to each of these steps.

5.1 Challenging training sets for ME-ASP

We remind that the first test set is TS, as defined in the previous section. We can think about this training set as being the one designed by a domain expert, that knows all available domains. We expect TS to lead to robust performance, and not be that sensitive to policy adaptation, given that no unseen problems is in it.

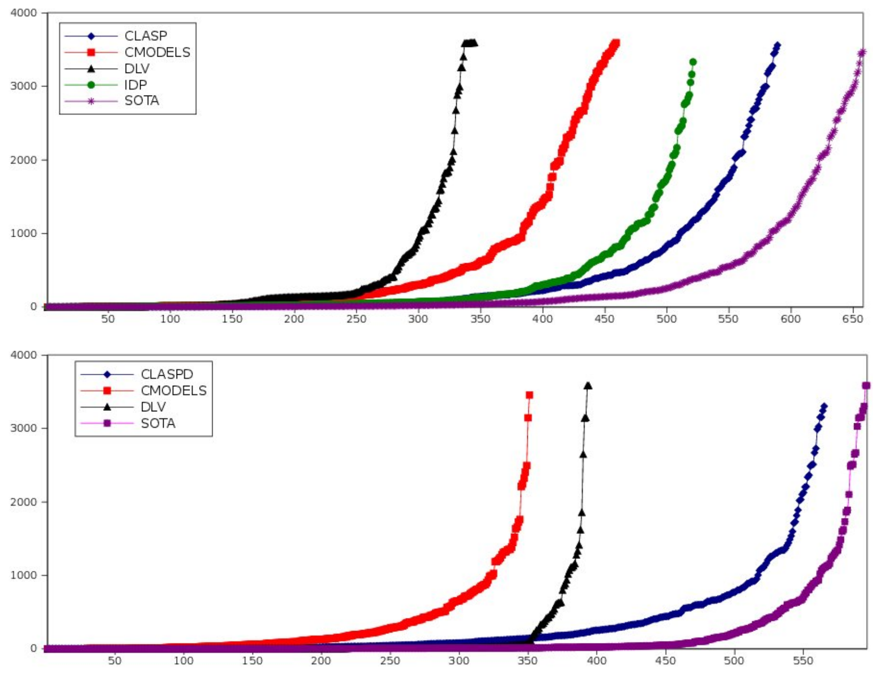

Figure 3: Runtimes of the engines of ME-ASP on *NP* (top) and *Beyond NP* (bottom) instances.

About the other sets, concerning *NP* problems, in our experimental setting we found two problems close to the desiderata mentioned, namely `Labyrinth` and `Numberlink`. In the case of `Labyrinth`, we compute a training set – TS_l in the following – composed of 35 uniquely solved instances, of which 21 solved by CLASP, 4 by CMODELS, 7 by DLV, and 3 by IDP. Concerning the problem `Numberlink` – TS_n in the following – , it is composed of 22 uniquely solved instances, of which 19 are solved by CLASP, 1 and 2 are solved by DLV and IDP, respectively.

Considering *Beyond NP* problems, the picture is even more challenging given that we have only two problems, namely `MinimalDiagnosis` and `StrategicCompanies`. More, in our setting we report that for each problem we have only one solver that is able to solve instances uniquely. In the first case, it is CLASPD– able to solve 197 instances –, while in the latter is DLV, that solves uniquely 32 `StrategicCompanies` instances. Accordingly, we compute two different training sets, namely TS_m and TS_s, related to the `MinimalDiagnosis` and `StrategicCompanies` problems, respectively.

5.2 Tuning the self-adaptive component

In Figure 3, we plot the performance – in terms of CPU time of solved instances – for each engine of ME-ASP^A on the instances submitted to the 3rd ASP Competition, both for *NP* (top-most plot) and *Beyond NP* (the plot in the bottom) classes. Looking at the plot, the x axis is labeled by the total amount of solved instances, while in the y axis it is reported the CPU time (in seconds). Each dot in the plot represents the performance of the related solver on a given instance. Concerning the top-most plot,

Solver		#Solved	Time
	Ind. Model		
CLASP		588	232556.53
CMODELS		459	245029.81
DLV		344	130804.50
IDP		521	152632.95
ME-ASP	TS	638	270848.09
ME-ASP	TS_l	589	230017.95
ME-ASP^A	TS_l	613	337359.31
ME-ASP	TS_n	514	103169.88
ME-ASP^A	TS_n	608	289735.77
SOTA		658	222244.38

Table 2: Results of ME-ASP, ME-ASP^A, and their engines on the *NP* instances submitted to the 3rd ASP Competition.

CLASP performance is represented by blue diamonds, CMODELS performance is denoted by red squares, DLV by yellow triangles, IDP performance is denoted by using green circles, and, finally, SOTA solver performance is depicted by purple stars. Considering the plot in the bottom, we represented by blue diamonds CLASPD instead of CLASP.

Looking at Figure 3 – the top-most plot –, we can see that after 60s all solvers were able to solve a noticeable amount of instances with respect to the total amount: CLASP is able to solve 49% of its solved set of instances (287 instances out of 588), CMODELS solves 41% (190 out of 459), DLV 46% (158 out of 344), and, finally, IDP solves 54% of its set of solved instances (281 out of 521). Shifting the time cap to 600s, we can see that all solvers are able to solve at least 75% of their set of solved instances: CLASP solves 81% of its set (477 instances out of 588), CMODELS 76% (348 out of 459), DLV 83% (285 out of 344), and IDP 84% (438 instances out of 521). These results let us conclude that 600s could be a good setting in order to tune the time t granted to each engine for the adaptive policy and try to exploit such mechanism in an effective way. Analogous conclusions can be drawn in the case of *Beyond NP* instances.

5.3 Analysis

In order to test the effectiveness of the policy adaptation on ME-ASP we investigate the performance of ME-ASP^A considering the challenging training sets previously described, namely TS_l, TS_n, TS_m, and TS_s.

We remind the reader that the compared engines were run on all the 1540 instances (938 *NP*, and 602 *Beyond NP*) grounded in less than 3600s and 4GB of memory, whereas the instances on which ME-ASP^A was run are limited to the ones for which we were able to compute all features (i.e., 1500 instances – 898 *NP*, and 602 *Beyond NP*), 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.

In Tables 2 and 3 we report the experiments concerning *NP* and *Beyond NP* in-

Solver		#Solved	Time
	Ind. Model		
CLASPD		565	171659.25
CMODELS		351	104161.58
DLV		394	46293.13
ME-ASP	TS	595	192538.72
ME-ASP	TS_m	565	171659.25
ME-ASP^A	TS_m	569	173719.26
ME-ASP	TS_s	394	46293.14
ME-ASP^A	TS_s	563	163952.77
SOTA		596	115226.80

Table 3: Results of ME-ASP, ME-ASP^A, and their engines on the *Beyond NP* instances submitted to the 3rd ASP Competition.

stances, respectively. The tables are structured as follows. The first column reports the name of the solver and (when needed) the related training set on which the inductive model is computed in a subcolumn; the second and third columns report the result of each solver in terms of total amount of solved instances (column "#Solved") within the time limit and sum of their solving times (column "Time").

Looking at Table 2, we can see that ME-ASP solves more instances than the component engines, i.e., it solves 50 instances more than CLASP, which is the best engine in this class, and 117 more than IDP, which is the second best. Looking now at Table 3, about *Beyond NP* instances, ME-ASP solves 595 instances, i.e., 30 more instances than CLASPD which is the best engine. Summing up, it is interesting to note that the ME-ASP 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. In sum, ME-ASP solves only 21 out of 1254 instances less than the SOTA solver, mostly from the *NP* class. We also report that the results of ME-ASP^A with TS are very similar to ME-ASP. This confirms the intuition mentioned before, and for this reason we do not show the related numbers in the tables.

Concerning the analysis of the self-adaptive component, considering training set TS_l, we can see that ME-ASP is able to perform only slightly better than its engines, solving one more instance than CLASP, and its performance are not close to the one reported for ME-ASP. Comparing its performance with ME-ASP^A, we can see that the retraining procedure allows to solve 24 instances more, i.e. 25 more than CLASP. The price that has to be paid for this increasing of performance on the total amount of solved instances is the increasing in solving time: about 50% of the CPU time spent by ME-ASP^A is reported as a consequence of the policy adaptation. In the whole solving process, the retraining procedure has been called 157 times.

Looking now at performance with the training set TS_n, we can see that ME-ASP is not able to do better than its engines. It solves 514 instances, and it is far from the performance reported for both CLASP and IDP. On the other hand ME-ASP^A solves

Solver	Ind. Model	#CLASP	#CLASPD	#CMODELS	#DLV	#IDP	#SOTA	#SB
ME-ASP	TS_l	586	–	0	3	0	303	166
ME-ASP^A	TS_l	283	–	33	110	187	365	133
ME-ASP	TS_n	510	–	0	0	4	253	145
ME-ASP^A	TS_n	370	–	18	102	118	378	129
ME-ASP	TS_m	–	565	0	0	–	210	325
ME-ASP^A	TS_m	–	427	0	142	–	334	223
ME-ASP	TS_s	–	0	0	394	–	386	8
ME-ASP^A	TS_s	–	428	0	135	–	323	229

Table 4: Total amount of successfully calls to the predicted component engines of the various versions of ME-ASP^A on the instances submitted to the 3rd ASP Competition.

20 instances more than its best engine and a total of 94 instances more than ME-ASP. Also in this case, the CPU time devoted to retrain is not negligible – about 41% of the total amount. In the case of TS_n, the retraining procedure has been called 113 times.

Finally, concerning the performance of ME-ASP related to training sets composed of *Beyond NP* instances, namely TS_m, and TS_s, not surprisingly – in terms of total amount of solved instances – its performance is equal to CLASPD and DLV, respectively. As mentioned above, this is a challenging setting given there are two problems only, and the starting training sets contain only one label. Concerning the usage of the self-adaptive component, ME-ASP^A is able to solve 4 instance more than CLASPD in the first case, while, considering the last test set, it solves 2 instances less than CLASPD, but the improvements w.r.t. ME-ASP is impressive, solving 169 instances more.

We report in Table 4 some detailed results about the behavior of the various versions of ME-ASP^A on the instances submitted to the 3rd ASP Competition, among the others the number of times the component engines were called as predicted solver and solved the instance. The table is composed of 9 columns. In the first column it is reported the considered solver, followed by a column that denotes the related inductive model. It is followed by five columns that report the total amount of calls to the related engine, e.g., "#CLASP" denotes the total amount of calls to CLASP. Last two columns, namely "#SOTA" and "#SB", denote the total amount of calls to the SOTA solver, and to the second best – considering the CPU time –, respectively.

Looking at Table 4, we can draw some conclusions about the improvements of performance due to the policy adaptation of ME-ASP^A w.r.t. ME-ASP. Concerning TS_l, we can see that ME-ASP calls substantially CLASP – with the noticeable exception of 3 times, in which DLV is called –, and ME-ASP predicts the SOTA solver 303 times only. Considering now ME-ASP^A on the same training set, we can see that the distribution of the calls between the engines is substantially different. The self-adaptive component improves the engine selection, and it leads ME-ASP^A to predict the SOTA solver 62 times more than ME-ASP. We report a similar picture for TS_n, for which ME-ASP^A predicts the SOTA solver 125 times more than ME-ASP. This explains the fact that ME-ASP^A solves 94 instances more than ME-ASP, as reported in Table 2. Concerning *Beyond NP*

Solver	Ind. Model	#Solved	Time
CLASPD		543	263705.94
CLASPFOLIO		569	237697.56
ME-ASP	TS	638	270848.09
ME-ASP^A	TS$_l$	613	337359.31
ME-ASP^A	TS$_n$	608	289735.77

Table 5: Performance of solvers on the *NP* grounded instances submitted to the System Track of the 3rd ASP Competition.

instances, we confirm the same picture for TS$_m$, while it seems to be different for TS$_s$. This is explained by the fact that, concerning *Beyond NP* instances, the CPU times reported for CLASPD and DLV are similar.

To conclude our analysis, in order to compare ME-ASP^A them with other state-of-the-art ASP solvers, in Table 5 we report the results of the solvers on *NP* submitted to the System Track of the 3rd ASP Competition. The analysis includes CLASPFOLIO, i.e., the winner of the track at the competition on *NP* instances, and CLASPD, the winner of the whole System Track of the 3rd ASP Competition. Looking at the table, we can confirm the general picture described above: we can see that, on *NP* instances, ME-ASP^A solves more instances than CLASPFOLIO and CLASPD, and it is very close to ME-ASP.

6 Related Work

We first remark that this paper is an extended and revised version of [27]. The main improvements are: (*i*) the introduction of a retraining technique, to update the algorithm selection policy in a multi-engine framework, when it fails to give good prediction; (*ii*) the implementation of this technique in the ME-ASP multi-engine solver; (*iii*) an experimental analysis involving the enhanced system ME-ASP^A, that proved its effectiveness on (*iv*) domains on instances that are a superset of the ones in [27], where the test set is composed of unseen instances. In addition, also the related work comparison is extended in this paper, which discusses also the papers that deal with policies adaptation in hard combinatorial problems.

Three main directions have been recently followed for exploiting Machine Learning techniques in ASP solving, possibly including methods for policy adaptation.
Portfolio-based. This approach is implemented in CLASPFOLIO [10], the winner of the 3rd ASP Competition on *NP* domains. It builds on ideas originally presented in [17, 23], where it is described the concept of "algorithm portfolio" as a general method for combining existing algorithms into new ones that are preferable to any of the component algorithms. CLASPFOLIO works by selecting the most promising CLASP internal configuration on the basis of both "static" and "dynamic" features of the input program, the latter being on-line features obtained by running CLASP for a given amount of time.

The ideas underlying CLASPFOLIO are highly related with the papers [44, 38, 17, 16], where a portfolio-based approach is applied for solving SAT, QSAT, CSP and planning problems, respectively. A recent paper in ASP that builds on CLASPFOLIO is [39], where the authors present a framework where portfolio and automatic algorithm configuration approaches are combined. An automatic algorithm configuration approach, whose idea is to design methods for automatically tuning and configuring the solver parameters, see, e.g., [19], is (partly) already employed in CLASPFOLIO for choosing the "best" CLASP configuration on the basis of the computed features. [18] is another recent paper that deals with the issue of policy adaptation: the ASPEED solver presented automatically selects a scheduling of solvers for minimizing some metrics, the number of overall timeouts being the most important. It relies on a policy for ordering solvers based on similar ideas to the one we use in this paper, originally employed in [37]. It is not based on computed features, and solves the problem as a multi-criteria optimization problem provided as ASP encoding. In [18], the scheduled solvers are various CLASP configurations.

Multi-engine. This is the approach followed in [28, 27], and it is at the basis of this work. Another application of multi-engine techniques to problem solving is [36], where it is designed, implemented and experimental evaluated a multi-engine QSAT solver. [37] extends [36] by introducing a self-adaptation of the learned selection policies when the approach fails to give a good prediction. The present work imports in multi-engine ASP solving a method for policy adaptation which is similar to one employed in [37], that allows a fixed amount of time to all the engines and, if all fail, grants all the remaining time (assuming there is time left) to the predicted solver. The NN classifier, used in our paper, was among the classifiers evaluated in [37], and has other features we already discussed in Section 4, e.g., it is the classification method employed in [28], which we remind is the first work published on multi-engine ASP solving. SATZILLA [44], i.e., a popular SAT solver that won several prizes in SAT competitions, can act as a multi-engine solver. The differences w.r.t. our work is that SATZILLA can also compute dynamic features, but do not provide any policy adaption. As a general comment, the advantage of the algorithm portfolio over a multi-engine is that it is possible, by combining algorithms, to reach a performance than is better than the one of the best engine on each instance, which is instead the upper 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 a black-box. No internal modification of engines, even minor, is requested in a multi-engine system, which thus has higher modularity. Focusing on the difference between ME-ASP^A and CLASPFOLIO on the topic of this paper, CLASPFOLIO does not implement retraining but grants all the time to the chosen configuration of CLASP.

Balduccini. This is an alternative approach, employed in ASP, followed by the DORS framework of [2]. In this framework, 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 the second run of the underlying engine (SMODELS in [2]) 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 "small" instances in the domain can help to improve the performance on other "big" instances. This approach, differently from the majority of the ones presented above, does not need features computation; on the other hand, differently from the multi-engine approach, it needs to internally modify the engine. Given its nature, it does not employ policy adaptation. According to Marcello Balduccini, i.e., DORS author the solving method behind DORS can be considered "complementary" more than alternative w.r.t. the one of ME-ASP, i.e., they could in principle be combined. For this reason, DORS is not included in the analysis.

7 Conclusion

In this paper we have presented an improvement to the multi-engine ASP computation framework that deals with updating the learned algorithm selection policy when it fails to give good predictions. In fact, the policy considered in the multi-engine solver ME-ASP is decided off-line and never updated, thus this framework can not "react" to imprecise predictions. In this respect, we have implemented a retraining procedure in ME-ASP, resulting in the enhanced system ME-ASP^A. The experimental analysis, conducted on benchmarks from the 3rd ASP Competition, shows that the retraining procedure is useful on training sets that are challenging for ME-ASP. ME-ASP^A is available for download at `http://www.mat.unical.it/ricca/me-aspA`.

Acknowledgments. The authors would like to 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 clarifying discussions.

References

[1] D.W. Aha, D. Kibler, and M.K. Albert. Instance-based learning algorithms. *Machine learning*, 6(1):37–66, 1991.

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

[3] Chitta Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press, Tempe, Arizona, 2003.

[4] Francesco Calimeri, Giovambattista Ianni, and Francesco Ricca. The third answer set programming system competition, since 2011. `https://www.mat.unical.it/aspcomp2011/`.

[5] Francesco Calimeri, Giovambattista Ianni, Francesco Ricca, Mario Alviano, Annamaria Bria, Gelsomina Catalano, Susanna Cozza, Wolfgang Faber, Onofrio Febbraro, Nicola Leone, Marco Manna, Alessandra Martello, Claudio Panetta, Simona Perri, Kristian Reale, Maria Carmela Santoro, Marco Sirianni, Giorgio Terracina, and Pierfrancesco Veltri. The Third Answer Set Programming Competition: Preliminary Report of the System Competition Track. In *Proc. of LPNMR11.*, pages 388–403, Vancouver, Canada, 2011. LNCS Springer.

[6] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In *TACAS*, pages 337–340, 2008.

[7] Christian Drescher, Martin Gebser, Torsten Grote, Benjamin Kaufmann, Arne König, Max Ostrowski, and Torsten Schaub. Conflict-Driven Disjunctive Answer Set Solving. In Gerhard Brewka and Jérôme Lang, editors, *Proceedings of the Eleventh International Conference on Principles of Knowledge Representation and Reasoning (KR 2008)*, pages 422–432, Sydney, Australia, 2008. AAAI Press.

[8] Niklas Eén and Niklas Sörensson. An Extensible SAT-solver. In *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003.*, pages 502–518. LNCS Springer, 2003.

[9] Thomas Eiter, Georg Gottlob, and Heikki Mannila. Disjunctive Datalog. *ACM Transactions on Database Systems*, 22(3):364–418, September 1997.

[10] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, Torsten Schaub, Marius Thomas Schneider, and Stefan Ziller. A portfolio solver for answer set programming: Preliminary report. In James P. Delgrande and Wolfgang Faber, editors, *Proc. of the 11th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR)*, volume 6645 of *LNCS*, pages 352–357, Vancouver, Canada, 2011. Springer.

[11] Martin Gebser, Benjamin Kaufmann, André Neumann, and Torsten Schaub. Conflict-driven answer set solving. In *Twentieth International Joint Conference on Artificial Intelligence (IJCAI-07)*, pages 386–392, Hyderabad, India, January 2007. Morgan Kaufmann Publishers.

[12] Martin Gebser, Torsten Schaub, and Sven Thiele. GrinGo : A New Grounder for Answer Set Programming. In *Logic Programming and Nonmonotonic Reasoning, 9th International Conference, LPNMR 2007*, volume 4483 of *Lecture Notes in Computer Science*, pages 266–271, Tempe, Arizona, 2007. Springer.

[13] Michael Gelfond and Nicola Leone. Logic Programming and Knowledge Representation – the A-Prolog perspective . *Artificial Intelligence*, 138(1–2):3–38, 2002.

[14] Michael Gelfond and Vladimir Lifschitz. The Stable Model Semantics for Logic Programming. In *Logic Programming: Proceedings Fifth Intl Conference and Symposium*, pages 1070–1080, Cambridge, Mass., 1988. MIT Press.

[15] Michael Gelfond and Vladimir Lifschitz. Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Computing*, 9:365–385, 1991.

[16] Alfonso Gerevini, Alessandro Saetti, and Mauro Vallati. An automatically configurable portfolio-based planner with macro-actions: Pbp. In Alfonso Gerevini, Adele E. Howe, Amedeo Cesta, and Ioannis Refanidis, editors, *Proc. of the 19th International Conference on Automated Planning and Scheduling*, Thessaloniki, Greece, 2009. AAAI.

[17] Carla P. Gomes and Bart Selman. Algorithm portfolios. *Artificial Intelligence*, 126(1-2):43–62, 2001.

[18] Holger Hoos, Roland Kaminski, Torsten Schaub, and Marius Thomas Schneider. ASPeed: Asp-based solver scheduling. volume 17 of *LIPIcs*, pages 176–187. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2012.

[19] Frank Hutter, Holger H. Hoos, Kevin Leyton-Brown, and Thomas Stützle. ParamILS: An automatic algorithm configuration framework. *Journal of Artificial Intelligence Research*, 36:267–306, 2009.

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

[21] Tomi Janhunen, Ilkka 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, pages 142–154, Postdam, Germany, 2009. Springer.

[22] Nicola Leone, Gerald Pfeifer, Wolfgang Faber, Thomas Eiter, Georg Gottlob, Simona Perri, and Francesco Scarcello. The DLV System for Knowledge Representation and Reasoning. *ACM Transactions on Computational Logic*, 7(3):499–562, July 2006.

[23] K. Leyton-Brown, E. Nudelman, G. Andrew, J. Mcfadden, and Y. Shoham. A portfolio approach to algorithm selection. In *In IJCAI-03*, 2003.

[24] Yuliya Lierler. Disjunctive Answer Set Programming via Satisfiability. In Chitta Baral, Gianluigi Greco, Nicola Leone, and Giorgio Terracina, editors, *Logic Programming and Nonmonotonic Reasoning — 8th International Conference, LPNMR'05, Diamante, Italy, September 2005, Proceedings*, volume 3662 of *Lecture Notes in Computer Science*, pages 447–451. Springer Verlag, September 2005.

[25] Yuliya Lierler. Abstract Answer Set Solvers. In *Logic Programming, 24th International Conference (ICLP 2008)*, volume 5366 of *Lecture Notes in Computer Science*, pages 377–391. Springer, 2008.

[26] Vladimir Lifschitz. Answer Set Planning. In Danny De Schreye, editor, *Proceedings of the 16th International Conference on Logic Programming (ICLP'99)*, pages 23–37, Las Cruces, New Mexico, USA, November 1999. The MIT Press.

[27] Marco Maratea, Luca Pulina, and Francesco Ricca. Applying machine learning techniques to asp solving. Number CVL 2012/003, page 21. University of Sassari Tech. Rep., March 2012.

[28] Marco Maratea, Luca Pulina, and Francesco Ricca. Applying Machine Learning Techniques to ASP Solving. In *Technical Communications of the 28th International Conference on Logic Programming (ICLP 2012)*, volume 17 of *LIPIcs*, pages 37–48. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2012.

[29] Marco Maratea, Luca Pulina, and Francesco Ricca. The Multi-Engine ASP Solver ME-ASP. In *To appear in the Proceedings of the 13th European Conference on Logics in Artificial Intelligence (JELIA 2012)*, Lecture Notes in Computer Science. Springer, 2012.

[30] V. Wiktor Marek and Mirosław Truszczyński. Stable models and an alternative logic programming paradigm. *CoRR*, cs.LO/9809032, 1998.

[31] Maarten Mariën, Johan Wittocx, Marc Denecker, and Maurice 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, pages 211–224, Guangzhou, China, 2008. Springer.

[32] I. Mierswa, M. Wurst, R. Klinkenberg, M. Scholz, and T. Euler. Yale: Rapid prototyping for complex data mining tasks. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 935–940. ACM, 2006.

[33] Ilkka Niemelä. Logic Programs with Stable Model Semantics as a Constraint Programming Paradigm. In Ilkka Niemelä and Torsten Schaub, editors, *Proceedings of the Workshop on Computational Aspects of Nonmonotonic Reasoning*, pages 72–79, Trento, Italy, May/June 1998.

[34] Eugene Nudelman, Kevin Leyton-Brown, Holger H. Hoos, Alex Devkar, and Yoav Shoham. Understanding random SAT: Beyond the clauses-to-variables ratio. In Mark Wallace, editor, *Proc. of the 10th International Conference on Principles and Practice of Constraint Programming (CP)*, Lecture Notes in Computer Science, pages 438–452, Toronto, Canada, 2004. Springer.

[35] L. Pulina and A. Tacchella. Time to learn or time to forget? Strengths and weaknesses of a self-adaptive approach to reasoning in Quantified Boolean Formulas. In *International Conference on Principles and Practice of Constraint Programming (Doctoral Programme)*, 2008.

[36] Luca Pulina and Armando Tacchella. A multi-engine solver for quantified boolean formulas. In Christian Bessiere, editor, *Proc. of the 13th International Conference on Principles and Practice of Constraint Programming (CP)*, Lecture Notes in Computer Science, pages 574–589, Providence, Rhode Island, 2007. Springer.

[37] Luca Pulina and Armando Tacchella. A self-adaptive multi-engine solver for quantified boolean formulas. *Constraints*, 14(1):80–116, 2009.

[38] Horst Samulowitz and Roland Memisevic. Learning to solve QBF. In *Proc. of the 22th AAAI Conference on Artificial Intelligence*, pages 255–260, Vancouver, Canada, 2007. AAAI Press.

[39] Bryan Silverthorn, Yuliya Lierler, and Marius Schneider. Surviving solver sensitivity: An asp practitioner's guide. In *Technical Communications of the 28th International Conference on Logic Programming (ICLP 2012)*, volume 17 of *LIPIcs*, pages 164–175. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2012.

[40] Patrik Simons, Ilkka Niemelä, and Timo Soininen. Extending and Implementing the Stable Model Semantics. *Artificial Intelligence*, 138:181–234, June 2002.

[41] smt-lib-web. The Satisfiability Modulo Theories Library, 2011. `http://www.smtlib.org/`.

[42] G. Sutcliffe. The cade-23 automated theorem proving system competition–casc-23. *AI Communications*, 25(1):49–63, 2012.

[43] Johan Wittocx, Maarten Mariën, and Marc 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)*, pages 153–165, Leuven, Belgium, November 2008.

[44] Lin Xu, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. SATzilla: Portfolio-based algorithm selection for SAT. *JAIR*, 32:565–606, 2008.