Advances in Multi-Engine ASP Solving

Marco Maratea[1], Luca Pulina[2], and Francesco Ricca[3]

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

Abstract. Algorithm selection techniques are known to improve the performance of systems for several knowledge representation and reasoning frameworks. This holds also in the case of Answer Set Programming (ASP), which is a rule-based programming paradigm with roots in logic programming and non-monotonic reasoning. Indeed, the multi-engine approach to ASP solving implemented in ME-ASP was particularly effective on the instances of the third ASP competition. In this paper we report about the advances we made on ME-ASP in order to deal with the new standard language ASPCore 2.0, which substantially extends the previous version of the standard language. An experimental analysis conducted on the Fifth ASP Competition benchmarks and solvers confirms the effectiveness of our approach also in comparison to rival systems.

1 Introduction

Algorithm selection [36] techniques are known to improve the performance of solvers for several knowledge representation and reasoning frameworks [24, 31, 34, 35, 37, 38, 40]. It is well-established in the scientific literature that the usage of these techniques is very useful to deal with empirically hard problems, in which there is rarely an overall best algorithm, while it is often the case that different algorithms perform well on different domains. In order to take advantage of this behavior, these systems are able to select automatically the "best" algorithm/solver on the basis of the characteristics of the instance in input (called *features*). Algorithm selection techniques proved to be particularly effective [5, 25, 26, 31, 38] in the case of solvers for Answer Set Programming (ASP) [22, 23], which is declarative programming paradigm based on logic programming and non-monotonic reasoning.

The application of algorithm selection techniques to ASP solving was ignited by the release of the portfolio solver CLASPFOLIO ver. 1 [20]. This solver ported to ASP the SATZILLA [40] approach for SAT. Indeed, the main selection technique of CLASPFOLIO was based on *regression*, which tries to estimate solving time to choose the "best" configuration/heuristic of the ASP solver CLASP.

Later, in [30, 29], it was introduced the first multi-engine solver for ASP, called ME-ASP [31]. ME-ASP ports to ASP an approach applied before to QBF [35]. In particular, ME-ASP exploits inductive techniques based on *classification*, i.e., membership to a class, to choose, on a per instance basis, a solver among a selection of black-box heterogeneous ASP solvers that participated to the third ASP Competition [13] and DLV [28], being able to combine the strengths of its component engines. Other proposals [25, 38] employ parameters tuning and/or design a solvers schedule. CLASPFOLIO ver. 2 [26] is a framework that includes and can combine several techniques implemented in other ASP solvers based on algorithm selection techniques.

Among all approaches, the one implemented in ME-ASP seems to be very effective in ASP, given it outperforms the other solvers on a broad set of benchmarks encoded in the standard language ASPCore 1.0 [12]. The input language of ME-ASP was, however, limited to ASPCore 1.0, which is the very basic language of the System track of the third ASP Competition [12]. The next editions of this event [2, 10] were based on a substantially extended language, called ASPCore 2.0 [9], supporting more expressive language features such as aggregates [14], weak constraints [8] and choice rules [39]. Supporting these additional constructs require a substantial update of the system, including the design of proper (syntactic) features for classification and an update of the engines, and of the consequent inductive model.

In this paper we report about the advances we made on ME-ASP in order to deal with the new standard language ASPCore 2.0. An experimental analysis, conducted on all domains of the fifth ASP Competition and considering the solvers that entered the Single Processor category of the competition, confirms the effectiveness of our approach. In particular, our results show that

- the new features allow to properly classify benchmarks encoded in ASPCore 2.0;
- ME-ASP performs better than its component engines, and is able to outperform alternative solutions at the state of the art, implemented in CLASPFOLIO ver. 2.2, on the benchmarks of the fifth ASP Competition [10].

The paper is structured as follows. Section 2 introduces needed preliminaries on ASP and classification. Section 3 reviews the key ingredients of a multi-engine approach and explains the choices made in the new version of ME-ASP. Section 4 then presents the results, and the paper ends in Section 5 with some conclusions.

2 Preliminaries

In this section we recall some preliminary notions concerning Answer Set Programming and machine learning techniques for algorithm selection.

2.1 Answer Set Programming

In this section we recall Answer Set Programming syntax and semantics. We refer in particular to the syntax and semantics of the ASPCore 2.0 [9] standard

specification that has been employed in ASP competitions [2, 10] from 2013. More detailed descriptions and a more formal account of ASP can be found in [8, 17, 21, 23], whereas a nice introduction to ASP can be found in [6]. Hereafter, we assume the reader is familiar with logic programming conventions.

Syntax. The syntax of ASP is similar to the one of Prolog. Variables are strings starting with uppercase letter and constants are non-negative integers or strings starting with lowercase letters. A *term* is either a variable or a constant. A *standard atom* is an expression $p(t_1, \ldots, t_n)$, where p is a *predicate* of arity n and $t_1, \ldots, t_n$ are terms. An atom $p(t_1, \ldots, t_n)$ is ground if $t_1, \ldots, t_n$ are constants. A *ground set* is a set of pairs of the form $\langle consts : conj \rangle$, where $consts$ is a list of constants and $conj$ is a conjunction of ground standard atoms. A *symbolic set* is a set specified syntactically as $\{Terms_1 : Conj_1; \cdots; Terms_t : Conj_t\}$, where $t > 0$, and for all $i \in [1, t]$, each $Terms_i$ is a list of terms such that $|Terms_i| = k > 0$, and each $Conj_i$ is a conjunction of standard atoms. A *set term* is either a symbolic set or a ground set. Intuitively, a set term $\{X : a(X, c), p(X); Y : b(Y, m)\}$ stands for the union of two sets: The first one contains the X-values making the conjunction $a(X, c), p(X)$ true, and the second one contains the Y-values making the conjunction $b(Y, m)$ true. An *aggregate function* is of the form $f(S)$, where S is a set term, and f is an *aggregate function symbol*. Basically, aggregate functions map multisets of constants to a constant. The most common functions implemented in ASP systems are: `#min` and `#max` (undefined for the empty set) computing minimum and maximum, respectively; `#count` counting the number of terms; and `#sum` the computes the sum of integers. An *aggregate atom* is of the form $f(S) \prec T$, where $f(S)$ is an aggregate function, $\prec \in \{<, \leq, >, \geq\}$ is a comparison operator, and T is a term called guard. An aggregate atom $f(S) \prec T$ is ground if T is a constant and S is a ground set. A *rule* r has the following form:

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

where $a_1, \ldots, a_n$ are standard atoms, $b_1, \ldots, b_k$ are atoms, $b_{k+1}, \ldots, b_m$ are standard atoms, and $n, k, m \geq 0$. A literal is either a standard atom a or its negation `not` a. The disjunction $a_1 \vee \ldots \vee a_n$ is the *head* of r, while the conjunction $b_1, \ldots, b_k, \texttt{not } b_{k+1}, \ldots, \texttt{not } b_m$ is its *body*. Rules with empty body are called *facts*. Rules with empty head are called *constraints*. A variable that appears uniquely in set terms of a rule r is said to be *local* in r, otherwise it is a *global* variable of r. An ASP program is a set of *safe* rules. A rule r is *safe* if both the following conditions hold: *(i)* for each global variable X of r there is a positive standard atom ℓ in the body of r such that X appears in ℓ; *(ii)* local variables of r appearing in a symbolic set $\{Terms : Conj\}$ also appear in $Conj$.

The ASPCore 2.0 language used in competitions also includes choice rules, weak constraints and queries. Choice rules [39] are of the form:

$$\{a : l_1, \ldots, l_k\} \geq u \text{ :- } b_1, \ldots, b_k, \texttt{not } b_{k+1}, \ldots, \texttt{not } b_m.$$

where a is a an atom and $l_1, \ldots, l_k$ are literals for $k \geq 0$, u is a term, and the body is defined as for standard rules. Intuitively, a choice rule means that, if the body of the rule is *true*, an arbitrary subset of atoms of at least u elements in the head must be chosen as *true*. According to the standard specification [9] we interpret choice rules as a syntactic shortcut for a disjunctive program simulating this behavior.[4]

A *weak constraint* [8] is of the form:

$$:\sim b_1, \ldots, b_k, \texttt{not } b_{k+1}, \ldots, \texttt{not } b_m.[w@l, t_1, \cdots, t_k]$$

where w and l are the weight and level of ω, and $t_1, \cdots, t_k$ are distinguishing terms. (Intuitively, $[w@l, \overline{t}]$ is read "as weight w at level l for substitution $\overline{t}$", for more details on distinguishing terms see [9]). An ASP program with weak constraints is $\Pi = \langle P, W \rangle$, where P is a program and W is a set of weak constraints. A standard atom, a literal, a rule, a program or a weak constraint is *ground* if no variables appear in it.

A *query* on an ASP program is of the form $q?$, where q is a positive ground atom.

Semantics. Let P be an ASP program. The *Herbrand universe* U_P and the *Herbrand base* B_P of P are defined as usual (see e.g.,[6]). The ground instantiation G_P of P is the set of all the ground instances of rules of P that can be obtained by substituting variables with constants from U_P.

An *interpretation* I for P is a subset I of B_P. A ground literal ℓ (resp., $\texttt{not } \ell$) is true w.r.t. I if $\ell \in I$ (resp., $\ell \notin I$), and false (resp., true) otherwise. An aggregate atom is true w.r.t. I if the evaluation of its aggregate function (i.e., the result of the application of f on the multiset S) with respect to I satisfies the guard; otherwise, it is false.

A ground rule r is *satisfied* by I if at least one atom in the head is true w.r.t. I whenever all conjuncts of the body of r are true w.r.t. I.

A model is an interpretation that satisfies all the rules of a program. Given a ground program G_P and an interpretation I, the *reduct* [16] of G_P w.r.t. I is the subset G_P^I of G_P obtained by deleting from G_P the rules in which a body literal is false w.r.t. I. An interpretation I for P is an *answer set* (or stable model [23]) for P if I is a minimal model (under subset inclusion) of G_P^I (i.e., I is a minimal model for G_P^I) [16].

Given a program with weak constraints $\Pi = \langle P, W \rangle$, the semantics of Π extends from the basic case defined above. Thus, let $G_\Pi = \langle G_P, G_W \rangle$ be the instantiation of Π; a constraint $\Omega \in G_W$ is violated by an interpretation I if all the literals in Ω are true w.r.t. I. An *optimum answer set* O for Π is an answer set of G_P that minimizes the sum of the weights of the violated weak constraints in G_W as a prioritized way.

[4] Roughly, choice rules can be seen as a shortcut for $a \mid na \leftarrow b_1, \ldots, b_n, e_1, \ldots, e_m.$, $\leftarrow b_1, \ldots, b_n, \texttt{not } \#count\{a : a, e_1, \ldots, e_m\} \geq k.$ where na is an fresh auxiliary atom that is projected out of the answer.

The semantics of queries is given in terms of cautious reasoning. Given a program P and a query $q?$, the query is true if q is true in all answer sets of P, and is false otherwise.

2.2 Classification for Algorithm Selection

In this work we rely on a per-instance selection algorithm that chooses the best (or a good) algorithm among a pool of available. The selection in our case is of an ASP solver and is made using a set of *features*, i.e., numeric values that represent particular characteristics of a given instance, of a ground ASP program.

In order to make such a selection in an automatic way, we model the problem using *multinomial classification* algorithms, i.e., machine learning techniques that allow automatic classification of a set of instances, given some instance features. In more detail, in multinomial classification we are given a set of patterns, i.e., input vectors $X = \{\underline{x}_1, \ldots \underline{x}_k\}$ with $\underline{x}_i \in \mathbb{R}^n$, and a corresponding set of labels, i.e., output values $Y \in \{1, \ldots, m\}$, where Y is composed of values representing the m classes of the multinomial classification problem. In our modeling, the m classes are m ASP solvers. We think 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 Multi-Engine Answer Sets Computation

The key idea at the basis of the application of automated algorithm selection algorithms can be summarized as follows: There is rarely a best solver to solve a given combinatorial problem, while it is often the case that different solvers perform well on different instances. Thus, a method that is able to select a good algorithm among a pool of available ones can perform much better than a static choice. In our framework a number of features of the input are measured, and *multinomial classification* algorithms are used to learn a selection strategy. More in details, the design of a multi-engine ASP solver involves the following steps:

1. Design of cheap-to-compute (syntactic) features that are significant for classifying the instances.
2. Fair design of training and test sets.
3. Selection of solvers that are representative of the state of the art.
4. Induction of a robust selection strategy by applying a classification algorithm.

In this section we report the choices we made in the design of the new version ME-ASP, by instantiating the ingredients we outlined above.

3.1 Features

The design of features is a crucial step of the development: indeed, features must be able to characterize the instances, but also should be cheap to compute, in the sense that they can be extracted very efficiently. Indeed, the overhead introduced by feature computation must be negligible.

The features of ground programs we selected for characterizing our instances are a super-set of those employed in the earlier version of ME-ASP for dealing with ASPCore 1.0. The new set includes features for taking into account the new language constructs of ASPCore 2.0, e.g., number of choice rules, aggregates and weak constraints.

The new features of ME-ASP are divided into four groups (such a categorization is borrowed from [33]):

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

This final choice of features, together with some of their combinations (e.g., c/r), amounts to a total of 58 features.

3.2 Dataset

In order to train the classifiers, we have to select a pool of instances for training purpose, called the training set. The training set must be broad enough to get a robust model; on the other hand, for reporting a fair analysis, we test the system on instances belonging to benchmarks not "covered" by the training set.

The benchmarks considered for the experiments correspond to the suite of the fifth ASP Competition – see [10] for details about the last event. This is a large and heterogeneous suite of hard benchmarks encoded in ASPCore 2.0, which was already employed for evaluating the performance of state-of-the-art ASP solvers. That suite includes planning domains, temporal and spatial scheduling problems, combinatorial puzzles, graph problems, and a number of application domains, i.e., databases, information extraction and molecular biology field.[5]

The considered pool of benchmarks is composed of 26 domains which are based on both complexity issues and language constructs of ASPCore 2.0. Starting from a total amount of 8572 instances – with *instance* we refer to the complete input program (i.e., encoding+facts) –, we pragmatically randomly split the amount of instances in each domain, using 50% of the total amount for training purpose, and the remaining ones for testing. All the instances were subject to feature selection after grounding them by using GRINGO (ver. 4) [18].

[5] An exhaustive description of the benchmark problems can be found in [11].

3.3 Solvers Selection

The selection of solvers has the goal of collecting a pool of engines that are both representative of the state-of-the-art solver (SOTA) and that have "orthogonal" performance (i.e., cover as much as possible of the set of solved instances, with minimal overlap on solved instances).

In order to find the set of training set labels, we have run ASP solvers that entered the Single Processor category of the fifth ASP Competition. In detail, we have run: CLASP [15], several solvers based on translation[6], i.e., LP2SAT3+GLUCOSE, LP2SAT3+LINGELING [27], LP2BV2+BOOLECTOR [32], LP2GRAPH [19], LP2MAXSAT+CLASP and LP2NORMAL2+CLASP [7], and some incarnations of the WASP solver [3, 4] (ver. 1, ver. 1.5 and ver. 2, called WASP-1, WASP-1.5 and WASP-2, respectively). In the following, we give more details for each solver. CLASP is a native ASP solver relying on conflict-driven nogood learning, and in this edition includes also the capabilities of CLASPD, an extension of CLASP that is able to deal with disjunctive logic programs. The LP2SAT3* family employs a translation strategy to SAT and resorts to the SAT solvers GLUCOSE and LINGELING for computing the answer sets. The translation strategy mentioned includes the normalization of aggregates as well as the encoding of level mappings for non-tight ground programs: LP2BV2+BOOLECTOR and LP2GRAPH are variants that express the latter in terms of bit-vector logic or acyclicity checking, respectively, supported by a back-end SMT solver. LP2MAXSAT+CLASP competes by translating to a Max-SAT problem and solving with CLASP. LP2NORMAL2+CLASP normalizes aggregates and uses CLASP.

WASP is a native ASP solver based on conflict-driven learning, extended with techniques specifically designed for solving disjunctive logic programs. Unlike WASP-1, which uses a prototype version of DLV [28] for grounding. WASP-2 relies on GRINGO and adds techniques for program simplification and further deterministic inferences. WASP-1.5, instead, combines the two solvers by switching between them depending on whether a logic program is non-HCF or subject to a query.

In order to choose the engines of ME-ASP, we computed the total amount of training instances solved by the state-of-the-art solver (SOTA) i.e., given an instance, the oracle that always fares the best among all the solvers. Looking at the results of the Fifth ASP Competition, we can see that only four solver can deal with the whole set of instances, namely CLASP, LP2NORMAL2+CLASP, WASP1, and WASP1.5. Starting from these results, we look for the minimum number of solvers such that the total amount of instances solved by the pool is the closest to the SOTA solver on the training instances. The result of this procedure allow us to choose as ME-ASP engines three solvers, namely CLASP, LP2NORMAL2+CLASP, and WASP1. Thus, each pattern of the training set is labeled with the solver having the best CPU time on the given instance.

[6] We have not considered LP2MIP2 given that we did not receive the license of CPLEX on time.

Solver	Solved	Time
ME-ASP	2378	70144.99
CLASP	2253	63385.74
LP2NORMAL2+CLASP	2198	94560.98
CLASPFOLIO	1841	75044.14
WASP1.5	1532	52478.95
WASP2	1407	46939.06
LP2MAXSAT+CLASP	1387	82500.12
LP2GRAPH	1344	72633.53
LP2SAT3+LINGELING	1334	90644.33
WASP1	1313	87193.62
LP2SAT3+GLUCOSE	1305	73893.54
LP2BV2+BOOLECTOR	1011	57498.48

Table 1. Results of the evaluated solvers. The first column contains the solver names, and it is followed by two columns, reporting the number of solved instances within the time limit (column "Solved"), and the sum of their CPU times in seconds ("Time").

3.4 Classification Algorithm and Training

Concerning the choice of a multinomial classifier, we considered a classifier able to deal with numerical features and multinomial class labels (the solvers). According to the study reported in the original paper on multi-engine ASP solving [29, 31], we selected *k-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]. The NN implementation used in ME-ASP is built on top built of the ANN library (www.cs.umd.edu/ mount/ANN). In order to test the generalization performance, we use a technique known as *stratified 10-times 10-fold cross validation* to estimate the generalization in terms of *accuracy*, i.e., the total amount of correct predictions with respect to the total amount of patterns. Given a training set (X, Y), we partition X in subsets X_i with $i \in \{1, \dots 10\}$ such that $X = \bigcup_{i=1}^{10} X_i$ and $X_i \cap X_j = \emptyset$ whenever $i \neq j$; we then train $c_{(i)}$ on the patterns $X_{(i)} = X \setminus X_i$ and corresponding labels $Y_{(i)}$. We repeat the process 10 times, to yield 10 different c and we obtain the global accuracy estimate. To tune the parameter k of NN, we repeated the process described above for $k \in [1, 10], k \in \mathbb{N}$. As a result of cross-validation and parameter tuning, we choose $k = 1$, for which we obtained an accuracy greater than 87%.

4 Experiments

We assessed the performance of ME-ASP on the test set, that as described in the previous section contains the half of the instances of the fifth ASP competition that were not comprised in the training set used for generating the inductive

model of ME-ASP. All the experiments run on a cluster of Intel Xeon E31245 PCs at 3.30 GHz equipped with 64 bit Ubuntu 12.04, granting 600 seconds of CPU time and 2GB of memory to each solver.

In Table 1 we report the performance of ME-ASP compared to the one obtained by the solvers described in Section 3.3. We involved in the analysis also CLASPFOLIO ver. 2.2 [7] for a direct comparison with approaches of algorithm selection.

As a general comment we note that ME-ASP is the solver that solves in absolute terms more instances than any other alternative considered in our analysis, which represents the state of the art in ASP solving.

In detail, comparing ME-ASP with its engines, we can see that it solves 125 instances more than CLASP, 155 instances more than LP2NORMAL2+CLASP, and 1065 instances more than WASP1. This outlines that ME-ASP is consistently better than the component engines, thus confirming that our algorithm selection strategy is effective. This proves empirically that the features of ME-ASP are able to characterize the input programs, and also that the inductive model learned during training is effective in suggesting a good solver for solving a given instance. Indeed, ME-ASP run the best solver 80% of the times, while it makes a suboptimal choice 17% of the times, i.e., it does not predict the best engine, but it runs a solver able to deal the input instance within the time limit. In fact, the SOTA solver composed of the engines of ME-ASP is able to solve 2462 instances, so ME-ASP is able to reach – in terms of solved instances – about 97% of the SOTA solver performance. Regarding the average CPU time per instance, we report that ME-ASP CPU time is about 5% more than the average CPU time per instance of the SOTA solver (29.50 and 28.05, respectively). Finally, we report that feature computation is basically negligible thanks to the selection of cheap-to-compute features, and remains, in average, within the 0.6% of computation time, i.e., 0.19 seconds in average.

Concerning the comparison with CLASPFOLIO, which is the only other system in this comparison that is based on algorithm selection, and represents the state of the art in algorithm selections for ASP, we note that CLASPFOLIO solves 537 instances less than ME-ASP.

Summing up, ME-ASP outperforms any solved that entered the fifth ASP competition, as well as alternative solvers based on algorithm selection, performing very efficiently on this benchmark set.

5 Conclusion

In this paper we presented an extension of the multi-engine ASP solving technique presented in [31] to deal with the with the broader set of language features included in the standard language ASPCore 2.0. We implemented an extended

[7] CLASPFOLIO has been run with its default setting, and with CLASP ver. 3 as a back-end solver. This improved version has been provided by Marius Lindauer, who is thanked.

version of the ME-ASP solver, which is now able to process powerful constructs such as choice rules, aggregates, and weak constraints.

An experimental analysis conducted on the fifth ASP Competition benchmarks and solvers shows that the new version of ME-ASP is very efficient, indeed it outperforms state-of-the-art systems in terms of number of solved instances.

References

1. Aha, D., Kibler, D., Albert, M.: Instance-based learning algorithms. Machine learning 6(1), 37–66 (1991)
2. Alviano, M., Calimeri, F., Charwat, G., Dao-Tran, M., Dodaro, C., Ianni, G., Krennwallner, T., Kronegger, M., Oetsch, J., Pfandler, A., Pührer, J., Redl, C., Ricca, F., Schneider, P., Schwengerer, M., Spendier, L.K., Wallner, J.P., Xiao, G.: The fourth answer set programming competition: Preliminary report. In: Cabalar, P., Son, T.C. (eds.) Logic Programming and Nonmonotonic Reasoning, 12th International Conference, LPNMR 2013, Corunna, Spain, September 15-19, 2013. Proceedings. Lecture Notes in Computer Science, vol. 8148, pp. 42–53. Springer (2013)
3. Alviano, M., Dodaro, C., Faber, W., Leone, N., Ricca, F.: WASP: A native ASP solver based on constraint learning. In: Cabalar, P., Son, T. (eds.) Proceedings of the Twelfth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2013). Lecture Notes in Computer Science, vol. 8148, pp. 54–66. Springer-Verlag (2013)
4. Alviano, M., Dodaro, C., Ricca, F.: Preliminary Report on WASP 2.0. In: Konieczny, S., Tompits, H. (eds.) Proceedings of the 15th International Workshop on Non-Monotonic Reasoning (NMR 2014). pp. 1–5. Vienna, Austria (2014)
5. Balduccini, M.: Learning and using domain-specific heuristics in ASP solvers. AI Communications – The European Journal on Artificial Intelligence 24(2), 147–164 (2011)
6. Baral, C.: Knowledge Representation, Reasoning and Declarative Problem Solving. Cambridge University Press, Tempe, Arizona (2003)
7. Bomanson, J., Janhunen, T.: Normalizing cardinality rules using merging and sorting constructions. In: Cabalar, P., Son, T. (eds.) Proceedings of the Twelfth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2013). Lecture Notes in Computer Science, vol. 8148, pp. 187–199. Springer-Verlag (2013)
8. Buccafurri, F., Leone, N., Rullo, P.: Enhancing Disjunctive Datalog by Constraints. IEEE Transactions on Knowledge and Data Engineering 12(5), 845–860 (2000)
9. Calimeri, F., Faber, W., Gebser, M., Ianni, G., Kaminski, R., Krennwallner, T., Leone, N., Ricca, F., Schaub, T.: Asp-core-2 input language format (since 2013), `https://www.mat.unical.it/aspcomp2013/ASPStandardization`
10. Calimeri, F., Gebser, M., Maratea, M., Ricca, F.: The design of the fifth answer set programming competition. ICLP 2014 Technical Communications - CoRR abs/1405.3710 (2014), `http://arxiv.org/abs/1405.3710`
11. Calimeri, F., Gebser, M., Maratea, M., Ricca, F.: The fifth answer set programming system competition (since 2014), `https://www.mat.unical.it/aspcomp2014/`
12. Calimeri, F., Ianni, G., Ricca, F.: The third open answer set programming competition. TPLP 14(1), 117–135 (2014), `http://dx.doi.org/10.1017/S1471068412000105`

13. Calimeri, F., Ianni, G., Ricca, F., Alviano, M., Bria, A., Catalano, G., Cozza, S., Faber, W., Febbraro, O., Leone, N., Manna, M., Martello, A., Panetta, C., Perri, S., Reale, K., Santoro, M.C., Sirianni, M., Terracina, G., Veltri, P.: The Third Answer Set Programming Competition: Preliminary Report of the System Competition Track. In: Proc. of LPNMR11. pp. 388–403. LNCS Springer, Vancouver, Canada (2011)
14. Dell'Armi, T., Faber, W., Ielpa, G., Leone, N., Pfeifer, G.: Aggregate Functions in Disjunctive Logic Programming: Semantics, Complexity, and Implementation in DLV. In: Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI) 2003. pp. 847–852. Morgan Kaufmann Publishers, Acapulco, Mexico (Aug 2003)
15. Drescher, C., Gebser, M., Grote, T., Kaufmann, B., König, A., Ostrowski, M., Schaub, T.: Conflict-Driven Disjunctive Answer Set Solving. In: Brewka, G., Lang, J. (eds.) Proceedings of the Eleventh International Conference on Principles of Knowledge Representation and Reasoning (KR 2008). pp. 422–432. AAAI Press, Sydney, Australia (2008)
16. Faber, W., Leone, N., Pfeifer, G.: Recursive aggregates in disjunctive logic programs: Semantics and complexity. In: Alferes, J.J., Leite, J. (eds.) Proceedings of the 9th European Conference on Artificial Intelligence (JELIA 2004). Lecture Notes in AI (LNAI), vol. 3229, pp. 200–212. Springer Verlag (Sep 2004)
17. Faber, W., Leone, N., Pfeifer, G.: Semantics and complexity of recursive aggregates in answer set programming. Artificial Intelligence 175(1), 278–298 (2011), special Issue: John McCarthy's Legacy
18. Gebser, M., Kaminski, R., Kaufmann, B., Schaub, T.: Clingo = ASP + control: Preliminary report. In: Theory and Practice of Logic Programming – Online-Supplement: Proc. of 30th International Conference on Logic Programming (ICLP 2014). pp. 1–9. Cambridge University Press (2014)
19. Gebser, M., Janhunen, T., Rintanen, J.: Answer set programming as sat modulo acyclicity. In: Schaub, T., Friedrich, G., O'Sullivan, B. (eds.) Proceedings of the Twenty-first European Conference on Artificial Intelligence (ECAI 2014). Frontiers in Artificial Intelligence and Applications, vol. 263, pp. 351–356. IOS Press (2014)
20. Gebser, M., Kaminski, R., Kaufmann, B., Schaub, T., Schneider, M.T., Ziller, S.: A portfolio solver for answer set programming: Preliminary report. In: Delgrande, J.P., Faber, W. (eds.) Proc. of the 11th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR). LNCS, vol. 6645, pp. 352–357. Springer, Vancouver, Canada (2011)
21. Gelfond, M., Leone, N.: Logic Programming and Knowledge Representation – the A-Prolog perspective . Artificial Intelligence 138(1–2), 3–38 (2002)
22. Gelfond, M., Lifschitz, V.: The Stable Model Semantics for Logic Programming. In: Logic Programming: Proceedings Fifth Intl Conference and Symposium. pp. 1070–1080. MIT Press, Cambridge, Mass. (1988)
23. Gelfond, M., Lifschitz, V.: Classical Negation in Logic Programs and Disjunctive Databases. New Generation Computing 9, 365–385 (1991)
24. Gomes, C.P., Selman, B.: Algorithm portfolios. Artificial Intelligence 126(1-2), 43–62 (2001)
25. Hoos, H., Kaminski, R., Schaub, T., Schneider, M.T.: ASPeed: Asp-based solver scheduling. In: Technical Communications of the 28th International Conference on Logic Programming (ICLP 2012). LIPIcs, vol. 17, pp. 176–187. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2012)
26. Hoos, H., Lindauer, M.T., Schaub, T.: claspfolio 2: Advances in algorithm selection for answer set programming. TPLP 14(4-5), 569–585 (2014)

27. Janhunen, T.: Some (in)translatability results for normal logic programs and propositional theories. Journal of Applied Non-Classical Logics 16, 35–86 (2006)
28. Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The DLV System for Knowledge Representation and Reasoning. ACM Transactions on Computational Logic 7(3), 499–562 (Jul 2006)
29. Maratea, M., Pulina, L., Ricca, F.: Applying machine learning techniques to ASP solving. In: Technical Communications of the 28th International Conference on Logic Programming (ICLP 2012). LIPIcs, vol. 17, pp. 37–48. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2012)
30. Maratea, M., Pulina, L., Ricca, F.: The multi-engine ASP solver ME-ASP. In: Proceedings of Logics in Artificial Intelligence, JELIA 2012. LNCS, vol. 7519, pp. 484–487. Springer (2012)
31. Maratea, M., Pulina, L., Ricca, F.: A multi-engine approach to answer-set programming. TPLP 14(6), 841–868 (2014), http://dx.doi.org/10.1017/S1471068413000094
32. Nguyen, M., Janhunen, T., Niemelä, I.: Translating answer-set programs into bit-vector logic. In: Tompits, H., Abreu, S., Oetsch, J., Pührer, J., Seipel, D., Umeda, M., Wolf, A. (eds.) Proceedings of the 19th International Conference on Applications of Declarative Programming and Knowledge Management (INAP 2011) and 25th Workshop on Logic Programming (WLP 2011). Revised Selected Papers. Lecture Notes in Computer Science, vol. 7773, pp. 105–116. Springer (2011)
33. Nudelman, E., Leyton-Brown, K., Hoos, H.H., Devkar, A., Shoham, Y.: Understanding random SAT: Beyond the clauses-to-variables ratio. In: Wallace, M. (ed.) Proc. of the 10th International Conference on Principles and Practice of Constraint Programming (CP). pp. 438–452. Lecture Notes in Computer Science, Springer, Toronto, Canada (2004)
34. O'Mahony, E., Hebrard, E., Holland, A., Nugent, C., O'Sullivan, B.: Using case-based reasoning in an algorithm portfolio for constraint solving. In: Proc. of the 19th Irish Conference on Artificial Intelligence and Cognitive Science (2008)
35. Pulina, L., Tacchella, A.: A self-adaptive multi-engine solver for quantified boolean formulas. Constraints 14(1), 80–116 (2009)
36. Rice, J.R.: The algorithm selection problem. Advances in Computers 15, 65–118 (1976)
37. Samulowitz, H., Memisevic, R.: Learning to solve QBF. In: Proc. of the 22th AAAI Conference on Artificial Intelligence. pp. 255–260. AAAI Press, Vancouver, Canada (2007)
38. Silverthorn, B., Lierler, Y., Schneider, M.: Surviving solver sensitivity: An asp practitioner's guide. In: Technical Communications of the 28th International Conference on Logic Programming (ICLP 2012). LIPIcs, vol. 17, pp. 164–175. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2012)
39. Simons, P.: Extending and Implementing the Stable Model Semantics. Ph.D. thesis, Helsinki University of Technology, Finland (2000)
40. Xu, L., Hutter, F., Hoos, H.H., Leyton-Brown, K.: SATzilla: Portfolio-based algorithm selection for SAT. JAIR 32, 565–606 (2008)