

# Enhancing ASP(Q) evaluation

Wolfgang Faber<sup>1</sup>, Giuseppe Mazzotta<sup>2</sup> and Francesco Ricca<sup>2</sup>

<sup>1</sup>Alpen-Adria Universität Klagenfurt, Austria

<sup>2</sup>University of Calabria, Rende, Italy

## Abstract

is an extension of Answer Set Programming (ASP) that enables the declarative and modular modeling of problems within the entire polynomial hierarchy. The first implementation of ASP(Q), known as QASP, utilized a translation to Quantified Boolean Formulae (QBF) to take advantage of the advanced and mature QBF-solving technology. However, the QBF encoding implemented by QASP is highly general, potentially leading to complex formulas that existing QBF solvers struggle to evaluate due to the large number of symbols and sub-clauses.

In the paper titled "An efficient solver for ASP(Q)", that has been presented during the 39th International Conference on Logic Programming (ICLP23), we introduced a novel implementation that builds upon the concepts of QASP and incorporates a more efficient encoding procedure, optimized encodings of ASP(Q) programs in QBF, and a machine learning model for the automatic selection of QBF-solving back-ends.

In this paper, we give an overview of the results we have obtained in the above-mentioned paper and discuss possible future directions.

## Keywords

ASP with Quantifiers, Quantified Boolean Formulas, Well-founded semantics

## 1. Introduction

Answer Set Programming (ASP) [1, 2] is a very well-known logic programming paradigm based on the stable models semantics, offering the capabilities for (i) modeling search and optimization problems in a declarative (and often compact) way and (ii) solving them using efficient systems [3, 4, 5] that can handle real-world problems [6, 7]. Thanks to advanced programming strategies, such as *saturation* [8, 9], ASP can model problems up to  $\Sigma_2^P$  (i.e. the second level of the Polynomial Hierarchy (PH)). However, such techniques are not very intuitive for non-expert users and so it can be difficult for them to model problems of such complexity.


Recently, these shortcomings of ASP have been overcome by the introduction of language extensions that expand the expressivity of ASP [10, 11, 12]. Among these, Answer Set Programming with Quantifiers ASP(Q) extends ASP, allowing for declarative and modular modeling of problems of the entire PH [12]. The language of ASP(Q) expands ASP with quantifiers over answer sets of ASP programs and allows the programmer to use the standard and natural pro-


---

22nd International Conference of the Italian Association for Artificial Intelligence (AIXIA 2023) - Discussion Papers, November 06–09, 2023, Rome, Italy

✉ Wolfgang.Faber@aau.at (W. Faber); giuseppe.mazzotta@unical.it (G. Mazzotta); francesco.ricca@unical.it (F. Ricca)

ORCID 0000-0002-0330-5868 (W. Faber); 0000-0003-0125-0477 (G. Mazzotta); 0000-0001-8218-3178 (F. Ricca)

 © 2022 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

 CEUR Workshop Proceedings (CEUR-WS.org)

gramming methodology, known as generate-define-test [13], to encode also problems beyond NP. The most important aspect that supports the adoption of ASP(Q) both in academia and industry is the availability of efficient systems that evaluate ASP(Q) programs.

The first implementation of ASP(Q) is the solver QASP [14] which features a translation of ASP(Q) to Quantified Boolean Formula (QBF) in order to exploit the well-developed and mature QBF-solving implementations [15]. The encoding strategy adopted in QASP is very general and might produce formulas that are hard to evaluate for existing QBF solvers because of the large number of symbols and sub-clauses. Moreover, from previous assessments [14] it has been observed that the implementation of the translation procedure could –in some specific cases– be so memory-hungry to prevent the production of the QBF formula even when a considerable amount of memory is available. These weaknesses motivated the idea of implementing a more efficient encoding in QBF that can generate more affordable formulas that can be handled by QBF solvers.

As a result, we proposed a new implementation of ASP(Q), the PYQASP system, that builds on the ideas of QASP and exploits syntactic properties of the ASP(Q) program for obtaining ad-hoc and compact QBF formulas. This contribution has been presented during the 39th International Conference on Logic Programming (ICLP23) and it has been published in the Journal: Theory and Practice of Logic Programming [16]. In particular, the new translation in QBF proposed by PYQASP produces more compact encodings by exploiting the well-founded semantics [17] for simplifying ASP(Q) programs and, if it is possible, avoids costly normalization steps that are needed for obtaining a QBF in Conjunctive Normal Form (QCNF). The new system has been empirically validated on ASP(Q) benchmarks proposed in the literature and the obtained results highlight significant performance improvements. In particular, PYQASP outperformed QASP by pushing forward the state of the art in ASP(Q) solving. In the following, we assume the reader familiar with logic programming syntax and both Answer Set and Well-founded semantics and refer the reader to introductory and founding papers for more details [1, 2, 17].

## 2. Answer Set Programming with Quantifiers

An ASP(Q) program  $\Pi$  is an expression of the form [12]:

$$\square_1 P_1 \square_2 P_2 \cdots \square_n P_n : C,$$

where, for each  $i = 1, \dots, n$ ,  $\square_i \in \{\exists^{st}, \forall^{st}\}$ ,  $P_i$  is an ASP program, and  $C$  is a stratified ASP program possibly with constraints.

Given a logic program  $P$ , a total interpretation  $I$  over the Herbrand base  $\mathcal{B}_P$ , and an ASP(Q) program  $\Pi$ , we denote by  $fix_P(I)$  the set of facts and constraints  $\{a \mid a \in I \cap \mathcal{B}_P\} \cup \{\leftarrow a \mid a \in \mathcal{B}_P \setminus I\}$ , and by  $\Pi_{P,I}$  the ASP(Q) program  $\Pi$ , where  $P_1$  is replaced by  $P_1 \cup fix_P(I)$ , i.e.  $\Pi_{P,I} = \square_1(P_1 \cup fix_P(I)) \square_2 P_2 \cdots \square_n P_n : C$ .

The *coherence* of ASP(Q) programs is defined by induction as follows:

- $\exists^{st} P : C$  is coherent, if there exists  $M \in AS(P)$  such that  $C \cup fix_P(M)$  is coherent;
- $\forall^{st} P : C$  is coherent, if for every  $M \in AS(P)$ ,  $C \cup fix_P(M)$  is coherent;

- $\exists^{st} P \Pi$  is coherent, if there exists  $M \in AS(P)$  such that  $\Pi_{P,M}$  is coherent;
- $\forall^{st} P \Pi$  is coherent, if for every  $M \in AS(P)$ ,  $\Pi_{P,M}$  is coherent.

Given a set of propositional atoms  $A$ , we denote by  $ch(A)$  the program  $\{\{a\} | a \in A\}$  made of choice rules over atoms in  $A$ . For two ASP programs  $P$  and  $P'$ ,  $Int(P, P')$  denotes the set of common atoms between  $P$  and  $P'$ . For two programs  $P$  and  $P'$ , the choice interface program  $CH(P, P')$  is defined as  $ch(Int(P, P'))$ . For a propositional formula  $\Phi$ ,  $var(\Phi)$  denotes the variables occurring in  $\Phi$ . For an ASP(Q) program  $\Pi$ , and an integer  $1 \leq i \leq n$ , we define the program  $P_i^{\leq}$  as the union of programs  $P_j$  with  $1 \leq j \leq i$ . Given an ASP(Q) program  $\Pi$ , the intermediate versions  $G_i$  of its subprograms, and the QBF  $\Phi(\Pi)$  encoding  $\Pi$  are:

$$G_i = \begin{cases} P_1 & i = 1 \\ P_i \cup CH(P_{i-1}^{\leq}, P_i) & 1 < i \leq n \\ C \cup CH(P_n^{\leq}, C) & i = n + 1 \end{cases}$$

$$\Phi(\Pi) = \boxplus_1 \cdots \boxplus_{n+1} \left( \bigwedge_{i=1}^{n+1} (\phi_i \leftrightarrow CNF(G_i)) \right) \wedge \phi_c$$

where  $CNF(P)$  is a CNF formula encoding the program  $P$  (such that models of  $CNF(P)$  correspond to  $AS(P)$ );  $\phi_1, \dots, \phi_{n+1}$  are fresh propositional variables;  $\boxplus_i = \exists x_i$  if  $\square_i = \exists^{st}$  or  $i = n + 1$ , and  $\boxplus_i = \forall x_i$  otherwise, where  $x_i = var(\phi_i \leftrightarrow CNF(G_i))$  for  $i = 1, \dots, n + 1$ , and  $\phi_c$  is the formula

$$\phi_c = \phi'_1 \odot_1 (\phi'_2 \odot_2 (\cdots \phi'_n \odot_n (\phi_{n+1}) \cdots))$$

where  $\odot_i = \vee$  if  $\square_i = \forall^{st}$ , and  $\odot_i = \wedge$  otherwise, and  $\phi'_i = \neg \phi_i$  if  $\square_i = \forall^{st}$ , and  $\phi'_i = \phi_i$  otherwise.

**Theorem 1 (Amendola et al. [14]).** *Let  $\Pi$  be a quantified program. Then  $\Phi(\Pi)$  is true iff  $\Pi$  is coherent.*

### 3. Enhancing the Encoding of ASP(Q) in QBF

In this section, we discuss some of the weaknesses of the encoding proposed by Amendola et al. [14] and describe the proposed optimizations, in order to overcome such limitations. First of all, in the encoding proposed in Section 2 the intermediate version of each program  $G_i$  is computed by introducing atoms from previous levels by means of choice rules that introduce even loops through negation. This aspect increases the number of clauses and does not allow the system to further simplify the intermediate grounded programs. Moreover, the standard grounding may produce rules that are trivially satisfied in each answer set of a given program increasing the number of clauses in the resulting translation in SAT. Secondly, it can be noted that the formula  $\Phi(\Pi)$  is not in CNF because of the presence of equivalences for each subprogram and the final formula  $\phi_c$  (which is not in CNF either). While this might be seen as a minor issue, the translation of non-CNF formulas into CNF by means of a Tseytin transformation

can be a time-consuming procedure that increases the length of the formulas and introduces extra symbols that could slow down QBF solvers. A natural question, therefore, is whether it is possible to identify classes of ASP(Q) programs such that the resulting QBF formula is in CNF. In what follows, we are going to investigate the proposed optimizations that will address the aforementioned limitation of the QASP encoding strategy.

**Simplification based on well-founded semantics.** A possible solution to obtain more compact encodings is exploiting the well-founded semantics. In particular, the well-founded model defines literals that are true in every answer set and so this can be exploited to simplify each subprogram and to propagate this ground truth to the following levels. In particular, given a program  $P$  and its well-founded model  $\mathcal{W}$ ,  $P$  can be simplified by removing all those rules with a false body w.r.t.  $\mathcal{W}$  and true literals in  $\mathcal{W}$  from the bodies of the remaining ones. The program obtained by applying such transformation admits a more compact CNF encoding and can be proved to have the same answer sets of the original one. The next step is to propagate ground truth from previous levels to the following ones. This can be achieved by representing positive literals that are true w.r.t. the well-founded model  $\mathcal{W}$  as facts and by omitting negative ones, whereas all the undefined literals are encoded as choice rules.

**Example 3.1.** *Given an ASP(Q) program  $Q$  the intermediate grounding exploiting the well-founded semantics is obtained as follows:*

@exists % P1		%G <sub>1</sub> <sup>WF</sup>
$a \leftarrow a, \text{ not } b$		$c \leftarrow$
$c \leftarrow \text{ not } a$		$\{b\} \leftarrow$
$\{b\} \leftarrow$		
@forall % P2	↦	%G <sub>2</sub> <sup>WF</sup>
$\{d(1..2)\} \leftarrow c$		$\{d(1..2)\} \leftarrow$
$\{d(3..100)\} \leftarrow a$		$c \leftarrow$
@constraint		%G <sub>3</sub> <sup>WF</sup>
% C is empty		% empty program

*The well-founded model  $P1$  is  $\mathcal{W} = \{\text{not } a, c\}$ . It means that  $a$  is false in every answer set of  $P1$  and  $c$  is true in every answer set of  $P1$ . Thus we can restrict the models of  $P2$  to those in which  $a$  is false and  $c$  is true. Thus, the choice interface  $CH'$  contains only the fact  $c \leftarrow$  and we can recursively simplify also  $P2$ .*

As a result, we can observe that the simplified program admits a smaller QBF encoding both in terms of the number of clauses, since potentially fewer rules are encoded, and also in average clause length since each rule is transformed into one or more clauses that have fewer literals. Moreover, by propagating information from the well-founded model of previous levels, answer sets of the following levels are restricted to those that are coherent with previous ones, if any. If no answer sets exist, then the resulting QBF formula can be pruned at the incoherent level. More formal definitions and proofs are available at [16].

**Direct CNF encodings for ASP(Q) programs.** In our work, we identified a class of ASP(Q) programs that admits a direct encoding in QCNF. In such a class of programs, all universal subprograms are said to be trivial. Basically, a trivial subprogram is a program that depends only on ground truth from previous levels and its stable models coincide with the power set of the atoms that the subprogram exposes to the following levels. If a subprogram is trivial then it can be omitted in the final QBF by leaving the quantification of the symbols that are exposed to the following levels. In this way, we can obtain a direct encoding in QCNF. However, triviality conditions are hard to verify but the class of programs made of only choice rules (under certain syntactic restrictions) features such property. In order to expand the class of programs featuring a direct encoding in QCNF we proposed a rewriting strategy that exploits the modularity of the Guess&Check paradigm. Essentially, each universal subprogram is split into two programs, namely Guess and Check. The Check program is pushed in the following levels with an ad-hoc rewriting that preserves the coherence of the entire ASP(Q) program and the Guess program (made only by choice rules) substitutes the original subprogram. By recursively applying such rewriting to all universal programs, we can obtain an equivalent ASP(Q) program that admits a direct encoding in QCNF. More formal definitions and proofs of the proposed optimization can be found in [16].

**Example 3.2.** *Let us consider the ASP(Q) program  $\Pi$  encoding the Q-3DNF Satisfiability Problem:*

```

@exists % P1
{ exists(X, true); exists(X, false) } ← X = 1..3
@forall % P2
{ forall(Y, true); forall(Y, false) } ← Y = 4..5
@constraint
asgn(X, V) ← exists(X, V)
asgn(X, V) ← forall(X, V)
conj(1, true, 2, true, 4, false) ←
conj(1, true, 3, true, 5, false) ←
conj(2, true, 4, true, 5, true) ←
sat ← conj(X1, S1, X2, S2, X3, S3), asgn(X1, S1), asgn(X2, S2), asgn(X3, S3)
← not sat

```

*P1 and P2 expose to the subsequent levels, respectively, the following atoms:*

$$Ext_1 = \{exists(X, V) \mid 1 \leq X \leq 3 \wedge V \in \{true, false\}\}$$

$$Ext_2 = \{forall(X, V) \mid 4 \leq X \leq 5 \wedge V \in \{true, false\}\}$$

*The stable models of P1 (resp. P2) coincide with  $2^{Ext_1}$  (resp.  $2^{Ext_2}$ ) and so  $\Pi$  can be encoded in a formula of the form:  $\exists Ext_1 \forall Ext_2 \text{ CNF}(G_3)$*

## 4. Experiments

In this section, we discuss an experimental analysis conducted to (i) demonstrate empirically the efficacy of the techniques described above, and (ii) compare PYQASP with QASP. We considered different benchmarks that have already been used to assess the performance of ASP(Q)

implementations [14]. The suite contains encodings in ASP(Q) and instances of four problems: Quantified Boolean Formulas (QBF); Argumentation Coherence (AC); Minmax Clique (MMC); Paracoherent ASP (PAR). A detailed description of these benchmarks was provided by Amendola et al. [14]. All the experiments were run on a system with 2.30GHz Intel(R) Xeon(R) Gold 5118 CPU with Ubuntu 20.04.2 LTS (GNU/Linux 5.4.0-137-generic x86\_64). Execution time and memory were limited to 800 seconds (of CPU time, i.e., user+system) and 12 GB, respectively.

In order to assess the impact of the proposed optimizations we run three variants of `pyQASP`, (i) basic encoding without optimization: `PYQASP`; (ii) encoding with well-founded simplification: `PYQASPWF`; and (iii) encoding with well-founded simplification and Guess&Check rewriting: `PYQASPWF+GC`.

These variants were combined with the following three QBF back-end solvers:

- (*RQS*) *RareQS* by Janota – <http://sat.inesc-id.pt/~mikolas/sw/areqs>;
- (*DEPS*) *DepQBF* by Lonsin – <https://lonsing.github.io/depqbf> – equipped with the *blokker* preprocessor by Biere et al. – <http://fmv.jku.at/blokker>;
- (*QBS*) *Quabs* by Tentrup – <https://github.com/ltentrup/quabs>.

All this amounts to running 9 variants of `pyQASP`.

In our naming conventions, the selected back-end is identified by a superscript, and a subscript identifies the enabled optimizations (e.g., `PYQASPDEPS` indicates `pyQASP` with back-end *DEPS*, and `PYQASPDEPSWF+GC` indicates `pyQASP` with *DEPS* back-end and all optimizations enabled).

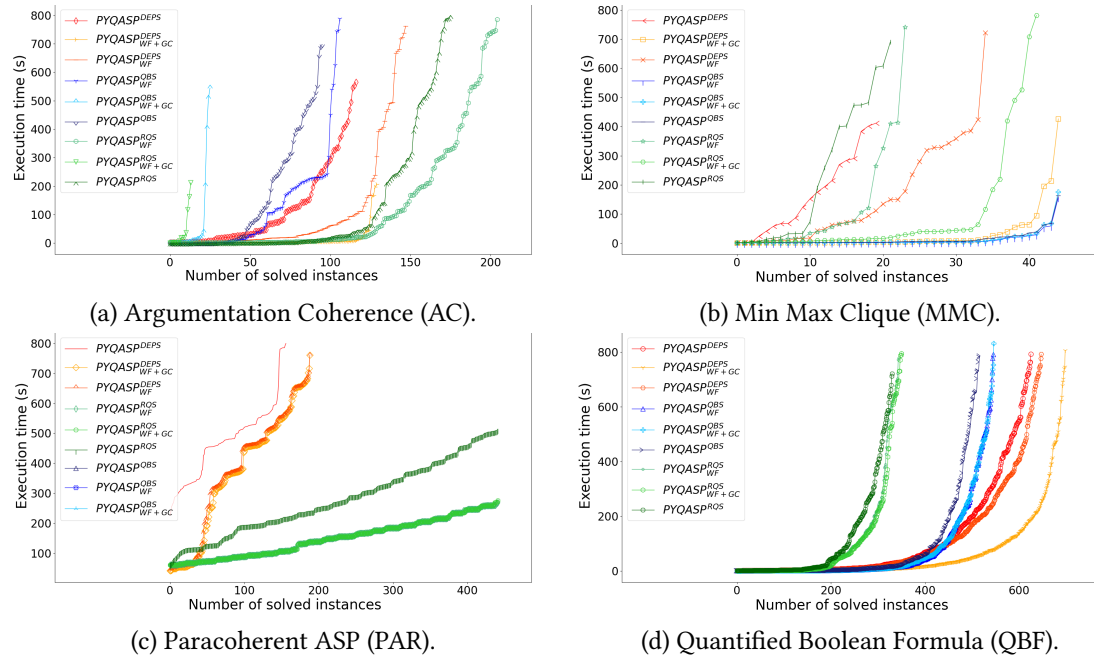
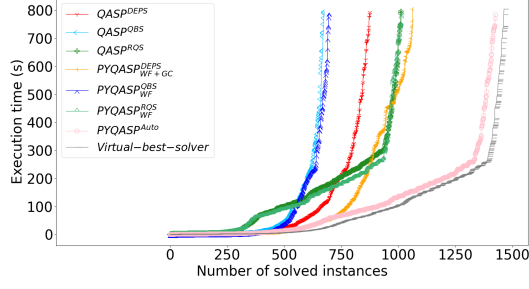


Figure 1: Analysis of proposed optimizations.



**Figure 2:** Comparison with QASP.

Obtained results are summarized in Figure 1, which aggregates the performance of each compared method in four cactus plots, one per considered problem. Recall that, a line in a cactus plot contains a point  $(x, y)$  whenever the corresponding system solves at most  $x$  instances in  $y$  seconds. The well-founded optimization allows to solve more instances and in less time in AC, MMC, and QBF benchmarks, independently of the back-end; whereas, the identification of *guess&check* programs pays off in terms of solved instances in QBF and MMC, again independently of the back-end solver. The two techniques combine their positive effects in MMC, PAR, and QBF. In particular,  $PYQASP_{WF+GC}^{DEPS}$  solves 25 more instances than  $PYQASP^{DEPS}$  in MC, and 73 in QBF; moreover,  $PYQASP_{WF+GC}^{RQS}$  solves 20 more instances than  $PYQASP^{RQS}$  in MC, and 20 in QBF. However, the application of the *guess&check* optimization has a negative effect on AC, since the well-founded operator, applied to the rewritten program, is no longer able to derive some simplifications that instead can be derived from the original program. For this reason,  $PYQASP_{WF}^{RQS}$  is the best option in AC, solving 29 instances more than  $PYQASP^{RQS}$ . All in all, the results summarized in Figure 1 confirm the efficacy of both well-founded optimization and identification of *guess&check* programs.

**Comparison with QASP.** In this comparison, we considered the best variants of PYQASP identified in the previous paragraph with QASP running the same back-end QBF solvers. As before, the selected back-end is identified by a superscript. In addition, we run a version of PYQASP that automatically selects a suitable back-end solver for each instance, denoted by  $PYQASP^{AUTO}$ . This latter was obtained by applying to PYQASP the methodology used in the ME-ASP solver [18] for ASP. In particular, we measured some syntactic program features, the ones of ME-ASP augmented with the number of quantifiers, existential (resp. universal) atoms count, and existential (resp. universal) quantifiers to characterize QASP instances. Then, we used the *random forest* classification algorithm for predicting a suitable back-end solver. As it is customary in the literature, to assess on the field the efficacy of the algorithm selection strategy, we also computed the Virtual Best Solver (VBS). VBS is the *ideal* system one can obtain by always selecting the best solver for each instance. Obtained results are reported in the cactus plot of Figure 2.

First of all, we note that PYQASP is faster and solves more instances than QASP no matter the back-end solver. In particular,  $PYQASP_{WF+GC}^{DEPS}$  solves 186 instances more than  $QASP^{DEPS}$ ,  $PYQASP_{WF}^{RQS}$  solves 4 instances more than  $QASP^{RQS}$ , and  $PYQASP_{WF}^{QBS}$  solves 21 instances more

than  $QASP^{QBS}$ .

Diving into the details, we observed that  $PYQASP$  also uses less memory on average than  $QASP$ . Indeed,  $QASP$  used more than 12GB in some instances of PAR and AC, whereas  $PYQASP$  never exceeded the memory limit in these domains. This is due to a combination of factors. On the one hand,  $PYQASP$  never caches the entire program in main memory; on the other hand, the formulas built by  $PYQASP$  are smaller than the ones of  $QASP$  and this causes the back-end QBF solver to use less memory and be faster during the search.

Finally, as one might expect, the best solving method is  $PYQASP^{AUTO}$ . Comparing  $PYQASP^{AUTO}$  with the VBS there is only a small gap (38 instances overall). In particular, we observe that, in the majority of cases, the selector is able to pick the best method; it sometimes misses a suitable back-end (especially in MMC which is the smallest and least represented domain in the training set). As a result,  $PYQASP^{AUTO}$  is generally effective in combining the strengths of all the back-end solvers. Indeed,  $PYQASP^{AUTO}$  solves 363 instances more than  $PYQASP^{DEPS_{WF+GC}}$  (i.e., the best variant of  $PYQASP$  with fixed back-end) and 414 instances more than  $QASP^{RQS}$  (i.e., the best variant of  $QASP$ ).

## 5. Conclusion

An important aspect that can boost the adoption of ASP(Q) as a practical tool for developing applications is the availability of more efficient implementations. The  $PYQASP$  system for ASP(Q) features both a memory-aware implementation in Python and a new optimized translation of ASP(Q) programs in QBF. In particular,  $PYQASP$  exploits the well-founded operator to simplify ASP(Q) programs and can recognize a (popular) class of ASP(Q) programs that can be encoded directly in CNF, and thus do not require to perform any additional normalization to be handled by QBF solvers. Moreover,  $PYQASP$  is able to select automatically a suitable back-end for the given input program and can deliver steady performance over varying problem instances.  $PYQASP$  outperforms  $QASP$ , the first implementation of ASP(Q), and pushes forward the state of the art in ASP(Q) solving. As future work, we plan to further optimize  $PYQASP$  by providing more efficient encodings in QBFs, and improve the algorithm selection model with extended training and a deeper tuning of parameters.

## Acknowledgments

This work was partially supported by the Italian Ministry of Industrial Development (MISE) under project MAP4ID “Multipurpose Analytics Platform 4 Industrial Data”, N. F/190138/01-03/X44, by MUR under PRIN project PINPOINT Prot. 2020FNEB27, CUP H23C22000280006, and PNRR project PE0000013-FAIR, Spoke 9 - Green-aware AI – WP9.1.

## References

- [1] G. Brewka, T. Eiter, M. Truszczynski, Answer set programming at a glance, *Commun. ACM* 54 (2011) 92–103.



- [2] M. Gelfond, V. Lifschitz, Classical negation in logic programs and disjunctive databases, *New Gener. Comput.* 9 (1991) 365–386.
- [3] C. Dodaro, G. Mazzotta, F. Ricca, Compilation of tight ASP programs, in: *ECAI 2023 - 26th European Conference on Artificial Intelligence*, IOS Press, 2023. URL: <https://doi.org/10.3233/FAIA230316>. doi:10.3233/FAIA230316.
- [4] G. Mazzotta, F. Ricca, C. Dodaro, Compilation of aggregates in ASP systems, in: *Thirty-Sixth AAAI Conference on Artificial Intelligence, AAAI 2022*, AAAI Press, 2022. URL: <https://doi.org/10.1609/aaai.v36i5.20527>. doi:10.1609/AAAI.V36I5.20527.
- [5] M. Gebser, N. Leone, M. Maratea, S. Perri, F. Ricca, T. Schaub, Evaluation techniques and systems for answer set programming: a survey, in: *Proceedings of IJCAI 2018*, [ijcai.org](http://ijcai.org), 2018, pp. 5450–5456. doi:10.24963/ijcai.2018/769.
- [6] E. Erdem, M. Gelfond, N. Leone, Applications of answer set programming, *AI Magazine* 37 (2016) 53–68.
- [7] M. Gebser, M. Maratea, F. Ricca, The sixth answer set programming competition, *J. Artif. Intell. Res.* 60 (2017) 41–95. doi:10.1613/jair.5373.
- [8] T. Eiter, G. Gottlob, On the computational cost of disjunctive logic programming: Propositional case, *Ann. Math. Artif. Intell.* 15 (1995) 289–323.
- [9] E. Dantsin, T. Eiter, G. Gottlob, A. Voronkov, Complexity and expressive power of logic programming, *ACM Comput. Surv.* 33 (2001) 374–425.
- [10] B. Bogaerts, T. Janhunnen, S. Tasharrofi, Stable-unstable semantics: Beyond NP with normal logic programs, *TPLP* 16 (2016) 570–586. doi:10.1017/S1471068416000387.
- [11] J. Fandinno, F. Laferrière, J. Romero, T. Schaub, T. C. Son, Planning with incomplete information in quantified answer set programming, *TPLP* 21 (2021) 663–679. doi:10.1017/S1471068421000259.
- [12] G. Amendola, F. Ricca, M. Truszczynski, Beyond NP: quantifying over answer sets, *TPLP* 19 (2019) 705–721. URL: <https://doi.org/10.1017/S1471068419000140>. doi:10.1017/S1471068419000140.
- [13] V. Lifschitz, Answer set programming and plan generation, *Artif. Intell.* 138 (2002) 39–54.
- [14] G. Amendola, B. Cuteri, F. Ricca, M. Truszczynski, Solving problems in the PH with ASP(Q), in: *Proceedings of LPNMR, volume 13416 of LNCS*, Springer, 2022, pp. 373–386. doi:10.1007/978-3-031-15707-3\_29.
- [15] L. Pulina, M. Seidl, The 2016 and 2017 QBF solvers evaluations (qbfeval’16 and qbfeval’17), *Artif. Intell.* 274 (2019) 224–248.
- [16] W. Faber, G. Mazzotta, F. Ricca, An efficient solver for asp(q), *Theory and Practice of Logic Programming* (2023). doi:10.1017/S1471068423000121.
- [17] A. Van Gelder, K. A. Ross, J. S. Schlipf, The well-founded semantics for general logic programs, *J. ACM* 38 (1991) 620–650.
- [18] M. Maratea, L. Pulina, F. Ricca, A multi-engine approach to answer-set programming, *TPLP* 14 (2014) 841–868. doi:10.1017/S1471068413000094.