2-ASP(Q) Solving Based on CEGAR

Andrea Cuteri, Giuseppe Mazzotta, Francesco Ricca,

University of Calabria, Rende, Italy
andrea.cuteri@unical.it, giuseppe.mazzotta@unical.it, francesco.ricca@unical.it

Abstract

The ASP(Q) language extends Answer Set Programming (ASP) with Quantifiers that operate over answer sets. Thus, ASP(Q) facilitates a more natural encoding of problems whose complexity exceeds NP within the ASP framework. In this paper we focus on ASP(Q) programs with two quantifiers, i.e., 2-ASP(Q) programs, which can be used to model problems in the second level of the Polynomial Hierarchy. In particular, we propose an approach for evaluating 2-ASP(Q) programs that is inspired by Counterexample Guided Abstraction Refinement (CEGAR). Unlike existing state-of-the-art ASP(Q) solvers, which are typically based on QBF solvers, our new approach leverages ASP solvers, and suffers no overhead due to the effects of translating ASP(Q) in QBF. Experimental results demonstrate that our technique consistently outperforms state-of-the-art ASP(Q) solvers, across benchmark problems located at the second level of the polynomial hierarchy.

Introduction

Answer Set Programming (ASP) (Brewka, Eiter, and Truszczynski 2011; Gelfond and Lifschitz 1991; Brewka, Eiter, and Truszczynski 2016) is a well-recognized symbolic AI formalism based on the stable model semantics. ASP allows for compact declarative modeling and solving of problems in NP (Gebser et al. 2018). In theory, ASP can also express problems beyond the class NP (Eiter and Gottlob 1995b). However, the encoding of a problem located at the second level of the Polynomial Hierarchy (PH) is often neither elegant nor intuitive, and typically requires the use of advanced techniques, such as saturation (Eiter and Gottlob 1995b), which complicate modeling *"beyond the capabilities of the common ASP laymen"* (Gebser, Kaminski, and Schaub 2011). To overcome this expressive limitation of ASP, several extensions have been proposed (Bogaerts, Janhunen, and Tasharrofi 2016; Amendola, Ricca, and Truszczynski 2019; Fandinno et al. 2021). Among these, the Answer Set Programming with Quantifiers (ASP(Q)) extends ASP with quantifiers over answer sets (Amendola, Ricca, and Truszczynski 2019). This allows to modularly formulate problems in the whole PH via the typical Guess & Check paradigm used in ASP. Moreover, available implementations of ASP(Q) enabled the development of interesting applications in many fields (Faber, Morak, and Chrpa 2022; Faber and Morak 2022; Bellusci, Mazzotta, and Ricca 2022; Azzolini et al. 2025).

State-of-the-art ASP(Q) solvers are based on QBF solvers (Amendola et al. 2022; Faber, Mazzotta, and Ricca 2023). These systems translate ASP(Q) programs into QBF and then exploit efficient QBF solvers for solving the encoded formula. Even though the encoding of ASP(Q) programs in QBF represents a viable solution, it might be that obtained QBF formulas are unnecessarily large and hard to solve for QBF solvers (Faber, Mazzotta, and Ricca 2023). This naturally raises the question of whether a direct implementation, taking inspiration from techniques used in QBF solvers, can overcome this limitation.

Some of the most efficient QBF solvers adopt advanced solving techniques like Counterexample Guided Abstraction Refinement (CEGAR) (Clarke et al. 2003b). The CEGAR methodology was born for symbolic model checking (Clarke et al. 2003a), but was adapted with success to QBF solving (Janota and Marques-Silva 2011; Janota et al. 2016).

The main goal of this work is to investigate if CEGAR can be adapted to ASP(Q) solving with the intent of providing an alternative evaluation technique for the ASP(Q) language. We positively answer this question by providing an evaluation technique for 2-ASP(Q) which is inspired by the work of Janota and Marques-Silva. Moreover, we present an implementation of such a technique in the CASPER system. Our system leverages the CLINGO ASP solver (Gebser et al. 2016) and does not rely on any QBF encoding. Empirical results show that our technique consistently outperforms both state-of-the-art ASP(Q) solvers and ASP solvers that rely on disjunctive encodings.

Answer Set Programming with Quantifiers

We recall some preliminary notions about ASP (Brewka, Eiter, and Truszczynski 2011; Gelfond and Lifschitz 1991) and ASP(Q) (Amendola et al. 2022). In particular, we focus on a significant ASP fragment (i.e., normal programs) which is sufficient to model NP-complete problems (Truszczynski 2011). Nonetheless, the proposed approach can be extended to wider ASP fragments (Calimeri et al. 2020), which would only require a longer and more involved presentation.

ASP Language. A *term* is either a *constant* (i.e., integer numbers or strings starting with a lowercase letter) or a *variable* (i.e., strings starting with an uppercase letter). An *atom* is of the form $p(\vec{t})$ where $\vec{t} = t_1, \ldots, t_n$, p is a predicate of arity $n \geq 0$, and $t_1, \ldots, t_n$ are terms. A *literal* l is either an atom a or its negation $\sim a$, where $\sim$ represents *negation as failure*. A literal of the form a (resp. $\sim a$) is said to be *positive* (resp. *negative*). For a set of atoms A, $\overline{A}$ is the set of literals $\{\sim a \mid a \in A\}$. A *rule* is an expression of the form $h \leftarrow l_1, \ldots, l_n$, where h is an atom referred to as *head* which can be also omitted, and $l_1, \ldots, l_n$, with $n \geq 0$, is a conjunction of literals referred to as *body*. Given a rule r, $H(r)$ denotes the set of atoms appearing in the head of r, while $B(r)$ denotes the set of literals in the body of r. A rule r is a *constraint* if $H(r) = \emptyset$; it is a *fact* if $B(r) = \emptyset$. A rule r is *safe* if variables appearing in r appear also in some positive literal in $B(r)$. A *program* is a set of safe rules. For a program P, $\mathcal{H}(P)$ denotes the set of atoms appearing in the head of rules in P. An ASP expression (i.e., atom, rules, etc.) is said to be *ground* if it does not contain any variable.

The *Herbrand Universe* of a program P, denoted by U_P, is the set of constants appearing in P. The *Herbrand Base* B_P of P is the set of ground atoms that can be constructed using constants in U_P and predicate in P. Given a rule $r \in P$, an *instantiation* of r is the rule obtained by substituting each variable in r with a constant in U_P. For a program P, we denote by $ground(P)$ program made of possible instantiations of rules in P. The *dependency graph* of P is a directed labeled graph, G_P, whose nodes are atoms in B_P and there is a positive (resp. negative) edge from u to v if there exists $r \in ground(P)$ such that $v \in H(r)$ and $u \in B(r)$ (resp. $\sim u \in B(r)$). P is *stratified* if G_P has no loops involving negative edges.

An *interpretation* $I \subseteq B_P$ is a set of atoms. A literal $l = a$ (resp. $l = \sim a$) is true w.r.t. I if $a \in I$ (resp. $a \notin I$); otherwise l is false w.r.t. I. A conjunction of literals, $l_1, \ldots, l_n$, is true w.r.t. I if for each $i \in \{1, \ldots, n\}$, l_i is true w.r.t. I; otherwise it is false. A rule $r \in ground(P)$ is satisfied w.r.t. I if the the head of r is true whenever the body is true. I is a model of P if each rule in P is satisfied w.r.t. I. The reduct (Gelfond and Lifschitz 1991), denoted by P^I, is the program obtained from $ground(P)$ by removing (i) rules $r \in ground(P)$ such that some negative literal l in $B(r)$ if false w.r.t. I, and (ii) each negative literal $l \in B(r)$ from the body of remaining rules. I is an *answer set* of P if I is a $\subset$-minimal model of P^I. We denote by $AS(P)$ the set of answer sets of P. P is *incoherent* if $AS(P) = \emptyset$; otherwise P is *coherent*. Let $M \in AS(P)$ be an answer set of P and A be a set of atoms, then $M|_A$ denotes the projection of M over atoms in A (i.e., $M \cap A$).

Example 1 *Let us consider the program P and its reduct P^M w.r.t. $M = \{b, c, d\}$:*

$$P = \left\{ \begin{array}{l} a \leftarrow \sim b \\ b \leftarrow \sim a \\ c \leftarrow d \\ d \leftarrow c \\ d \leftarrow a \end{array} \right\} \qquad P^M = \left\{ \begin{array}{l} b \leftarrow \\ c \leftarrow d \\ d \leftarrow c \\ d \leftarrow a \end{array} \right\}$$

Here $\{b\} \subseteq M$ is a model of P^M and so M is not an answer set of P. Conversely, $M' = \{a, c, d\}$ is a model of P and a minimal model of $P^{M'}$, thus M' is an answer set of P.

ASP(Q) Language. A 2-ASP(Q) program Π is an expression of the form:

$$\Box_1 P_1 \Box_2 P_2 : C \tag{1}$$

where $\Box_1, \Box_2 \in \{\exists^{st}, \forall^{st}\}$ with $\Box_1 \neq \Box_2$, P_1 and P_2 are programs, and C is a stratified program with constraints. If $\Box_1 = \exists^{st}$ then Π is said to be *existential*; otherwise Π is *universal*. A 2-ASP(Q) program of the form (1) satisfies the *stratified definition assumption* if $\mathcal{H}(P_2) \cap \mathcal{H}(P_1) = \emptyset$ and $\mathcal{H}(C) \cap (\mathcal{H}(P_1) \cup \mathcal{H}(P_2)) = \emptyset$. W.l.o.g., in what follows, we assume that 2-ASP(Q) programs satisfy the stratified definition assumption (Mazzotta, Ricca, and Truszczynski 2024).

Given a program P and an interpretation I, $fix_P(I)$ denotes the set of facts and constraints of the form $\{a \mid a \in I\} \cup \{\leftarrow a \mid a \in B_P \setminus I\}$.

Coherence of 2-ASP(Q) programs is defined as follows:

- $\exists^{st} P_1 \forall^{st} P_2 : C$ is coherent, if there exists $M_1 \in AS(P_1)$ such that, for each $M_2 \in AS(P_2')$, with $P_2' = P_2 \cup fix_{P_1}(M_1)$, $C \cup fix_{P_2'}(M_2)$ is coherent.
  In this case M_1 is a *quantified answer set*.
- $\forall^{st} P_1 \exists^{st} P_2 : C$ is coherent, if for each $M_1 \in AS(P_1)$, there exists $M_2 \in AS(P_2')$, with $P_2' = P_2 \cup fix_{P_1}(M_1)$ such that $C \cup fix_{P_2'}(M_2)$ is coherent;

Example 2 *Let Π be a 2-ASP(Q) of the form $\exists P_1 \forall P_2 : C$*

$$P_1 = \left\{ \begin{array}{lll} a & \leftarrow & \sim na. \\ na & \leftarrow & \sim a. \\ b & \leftarrow & \sim nb. \\ nb & \leftarrow & \sim b. \end{array} \right\} \quad P_2 = \left\{ \begin{array}{lll} c & \leftarrow & \sim nc. \\ nc & \leftarrow & \sim c. \\ & \leftarrow & a,\ \sim nc. \end{array} \right\}$$

$$C = \{ \quad \leftarrow \quad nc,\ nb. \ \}$$

Here, $M_1 = \{na, nb\}$ is an answer set of P_1. To verify that M_1 is quantified answer set we need to check that each answer set of $P_2' = P_2 \cup fix_{P_1}(M_1^1)$ satisfied C. Let $M_2 = \{na, nb, nc\} \in AS(P_2')$ then the program $C \cup fix_{P_2'}(M_2)$ is incoherent. Thus M_1 is not a quantified answer set. On the other hand, if we consider $M_1' = \{a, b\} \in AS(P_1)$ then we have that $P_2' = P_2 \cup fix_{P_1}(M_1')$ has two answer sets $M_2' = \{a, b, c\}$ and $M_2'' = \{a, b, nc\}$. Since neither $C \cup fix_{P_2'}(M_2')$ nor $C \cup fix_{P_2'}(M_2'')$ is incoherent then we can conclude that M_1' is a quantified answer set of Π.

CEGAR for QBF

In this section, we describe a game-centric view to QBF semantics and the CEGAR-based technique proposed by Janota and Marques-Silva (2011) for 2-QBF.

QBF. Boolean variables are strings like x_1, x_2 etc. A boolean formula is constructed by combining boolean variables with logical connectives $(\wedge, \vee, \neg)$ having their usual semantics. Let X be a set of variables, then a *truth assignment* τ is a function mapping variables in X to the boolean constants $\top$ (true) or $\bot$ (false). Given a formula ϕ over variables in X and a truth assignment τ, $\phi[\tau]$ denotes the formula obtained from ϕ by replacing variables with the truth value they are mapped to. The formula ϕ is: (i) *satisfiable*

Algorithm 1: CEGAR for 2-QBF

Input: $QX\overline{Q}Y\phi$
Output: v if there exists a winning move v for QX, $NULL$ otherwise

1: $\omega \leftarrow \emptyset$
2: **while** $true$ **do**
3: $\quad\alpha \leftarrow (Q = \exists)? \bigwedge_{\mu\in\omega} \phi[\mu] \ : \ \bigvee_{\mu\in\omega} \phi[\mu]$
4: $\quad\tau \leftarrow (Q = \exists) \ ? \ SAT(\alpha) : SAT(\neg\alpha)$
5: $\quad$**if** $\tau = NULL$ **then**
6: $\quad\quad$**return** $NULL$
7: $\quad$**end if**
8: $\quad\mu \leftarrow (Q = \exists \ ? \ SAT(\neg\phi[\tau]) : \ SAT(\phi[\tau]))$
9: $\quad$**if** $\mu = NULL$ **then**
10: $\quad\quad$**return** τ
11: $\quad$**end if**
12: $\quad\omega \leftarrow \omega \cup \mu$
13: **end while**

if there exists a truth assignment τ such that $\phi[\tau] = \top$ (i.e., the formula evaluates to true); (ii) *valid* if for every truth assignment τ, $\phi[\tau] = \top$; (iii) *unsatisfiable* if there exists no truth assignment τ such that $\phi[\tau] = \top$.

A 2-QBF formula is an expression of the form $QX_1\overline{Q}X_2\phi$ where $Q, \overline{Q} \in \{\exists, \forall\}$ with $Q \neq \overline{Q}$, X_1, X_2 are sets of variables, and ϕ is a boolean formula over variables in $X_1 \cup X_2$. A 2-QBF formula of the form $\exists X_1 \forall X_2 \phi$ is *satisfiable* if there exists a truth assignment τ over variables in X_1 such that $\phi[\tau]$ is valid. A 2-QBF formula of the form $\forall X_1 \exists X_2 \phi$ is *satisfiable* if for every truth assignment τ over variables in X_1, $\phi[\tau]$ is satisfiable.

Game-centric view - QBF The 2-QBF semantics can be formulated as a game between the existential and universal players (Janota and Marques-Silva 2011; Papadimitriou 1994). In this game, the existential player *wins* if the QBF evaluates to true, while the universal player *wins* if the QBF evaluates to false. Given a 2-QBF of the form $QX_1\overline{Q}X_2\phi$, a *move* for player Q is a truth assignment τ of variables in X_1. Similarly, a move for $\overline{Q}$ is a truth assignment μ of variables in X_2. Given a move τ of player Q, a *countermove* to Q is a move μ for $\overline{Q}$ such that $\phi[\tau][\mu]$ is false if $Q = \exists$, or $\phi[\tau][\mu]$ is true if $Q = \forall$. If there exists a move τ for Q such that the opponent (i.e, $\overline{Q}$) does not hold a countermove then Q wins.

CEGAR for 2-QBF The CEGAR-based approach for 2-QBF developed by Janota and Marques-Silva (2011), reported in Algorithm 1, is based on this game-centric view of 2-QBF semantics. Intuitively, the approach iteratively searches for a *winning move* of Q. In particular, for each move of Q, it checks if there exists a countermove for $\overline{Q}$. Whenever a countermove μ for $\overline{Q}$ is found, it is used to refine the original formula in such a way that the next moves computed for Q do not have μ as countermove.

In more detail, Algorithm 1 starts by initializing the set of countermoves ω to the empty set (line 1) and then enters the CEGAR pipeline (line 2). At each iteration of the pipeline, the refined abstraction is obtained as the conjunction (resp. the disjunction) of the refinements of ϕ w.r.t. each countermove μ, that is $\phi[\mu]$ (line 3). Then, the refined abstraction is used to compute a new move τ for Q that does not admit any of the previously discovered countermoves (line 4). Given a move τ for Q, the algorithm searches for a possible countermove for $\overline{Q}$. To this end, the formula $\neg\phi[\tau]$ is solved if $Q = \exists$, otherwise the algorithm solves $\phi[\tau]$ (line 8). If the solved formula is unsatisfiable then there are no countermoves for $\overline{Q}$ and thus the algorithm returns τ as winning move (line 10). Otherwise, the computed countermove μ is added to the set ω (line 12) and the pipeline continues by searching the next move for Q. If there are no further moves available for Q then the algorithm terminates returning $NULL$ (line 6) since no winning move exists for Q.

CEGAR for 2-ASP(Q)

We are now ready to give a game-centric view of the 2-ASP(Q) semantics which allows to design a CEGAR-based approach for deciding the coherence of 2-ASP(Q) programs.

Game-centric view - ASP(Q) Let Π be a 2-ASP(Q) program of the form $\Box_1 P_1 \Box_2 P_2 \ : \ C$. An answer set M_1 of P_1 is a *move* for $\Box_1$. Let M_2 be an answer set of $P_2' = P_2 \cup \mathit{fix}_{P_1}(M_1)$ then $CM = M_2|_{\mathcal{H}(P_2)}$ is a *candidate countermove* to M_1 for $\Box_2$. A candidate countermove CM is a *countermove* if either $\mathit{fix}_{P_2'}(M_2) \cup C$ is incoherent and $\Box_2 = \forall^{st}$; or $C \cup \mathit{fix}_{P_2'}(M_2)$ is coherent and $\Box_2 = \exists^{st}$. A *winning move* for $\Box_1$ is a move M_1 of $\Box_1$ such that there does not exist a countermove for $\Box_2$ to M_1. $\Box_1$ *wins* if there exists a winning move for it.

Thus, $\Box_1 = \exists^{st}$ wins if there exists an answer set $M_1 \in AS(P_1)$ such that there does not exist an $M_2 \in AS(P_2')$ where $P_2' = P_2 \cup \mathit{fix}_{P_1}(M_1)$ and $C \cup \mathit{fix}_{P_2'}(M_2)$ is incoherent. On the other hand, $\Box_1 = \forall^{st}$ wins if there exists $M_1 \in AS(P_1)$ such that there does not exist $M_2 \in AS(P_2')$ where $P_2' = P_2 \cup \mathit{fix}_{P_1}(M_1)$ and $C \cup \mathit{fix}_{P_2'}(M_2)$ is coherent. Hence, we can characterize the coherence of 2-ASP(Q) programs by the existence of a winning move.

Proposition 1 *Let* Π *be a 2-ASP(Q) program of the form* $\Box_1 P_1 \Box_2 P_2 : C$ *then there exists a winning move for* $\Box_1$ *iff (i)* Π *is incoherent and* $\Box_1 = \forall^{st}$ *or (ii)* Π *is coherent and* $\Box_1 = \exists^{st}$.

From Proposition 1, we can introduce a CEGAR-based approach for deciding the coherence of 2-ASP(Q) programs.

CEGAR for ASP(Q). We now introduce the transformations that are necessary for mimicking the CEGAR approach by: (i) computing countermoves for $\Box_2$ to moves of $\Box_1$, and (ii) computing moves of $\Box_1$ which avoid known countermoves of $\Box_2$ (abstraction refinement).

Unlike the 2-QBF setting, where there is a single formula with variables quantified either existentially or universally, in 2-ASP(Q) there are three distinct ASP programs (i.e., P_1, P_2, and C) that play different roles in the ASP(Q) game. Thus, we introduce the transformations that, by properly combining such programs, simulate the ASP(Q) game.

The first transformation is aimed at modeling the incoherence of stratified programs with constraints (specifically the program C). Recall that a stratified program with constraints P has at most one answer set (Dantsin et al. 2001).

In particular, such an answer set does not exist (i.e., P is incoherent) if some constraint is violated. Thus, to detect the incoherence of P we need to capture constraints violation.

Definition 1 (Complement of stratified programs) *Let P be a stratified ASP program with constraints, then the complement of P, denoted by $\neg P$, is:*

$$\neg P = \begin{cases} H(r) \leftarrow B(r). & \forall r \in P : H(r) \neq \emptyset \\ unsat_P \leftarrow B(r). & \forall r \in P : H(r) = \emptyset \\ \leftarrow \sim unsat_P. \end{cases}$$

where $unsat_P$ is a fresh atom not appearing in P.

Intuitively, $\neg P$ captures constraints violations by means of the fresh atom $unsat_P$ and enforces that at least one constraint is violated by means of the constraint $\leftarrow \sim unsat_P$. This leads to the following property.

Proposition 2 *Let P be a stratified program with constraints then P is incoherent if and only if $\neg P$ is coherent.*

Proposition 2 allows us to detect the incoherence of the program C of a 2-ASP(Q) program as the existence of an answer set of $\neg C$. This property is instrumental for defining the two main phases of the proposed approach: abstraction refinement and countermove search, each formalized through ad-hoc transformations.

Definition 2 (Countermove program) *Let Π be a 2-ASP(Q) program of the form $\square_1 P_1 \square_2 P_2 : C$, then:*

$$ctr(\Pi) = \begin{cases} P_2 \cup \neg C & if\ \square_2 = \forall^{st} \\ P_2 \cup C & if\ \square_2 = \exists^{st} \end{cases}$$

Essentially, $ctr(\Pi)$ combines P_2 and C according to the notion of countermove for $\square_2$. Thus, given a move M_1 for $\square_1$, countermoves to M_1 for $\square_2$ can be computed from answer sets of $ctr(\Pi) \cup fix_{P_1}(M_1)$.

Example 3 *Let Π be the same program from Example 2. The countermove program $ctr(\Pi)$ is the following program:*

$$ctr(\Pi) = \left\{ \begin{array}{lll} c & \leftarrow & \sim nc. \\ nc & \leftarrow & \sim c. \\ & \leftarrow & a,\ \sim nc. \\ unsat_C & \leftarrow & nc,\ nb. \\ & \leftarrow & \sim unsat_C. \end{array} \right\}$$

From Example 1, $M_1 = \{na, nb\} \in AS(P_1)$ and $M_2 = \{na, nb, nc\} \in AS(P_2 \cup fix_{P_1}(M_1))$ violates C. Thus $M_2|_{\mathcal{H}(P_2)} = \{nc\}$ is a countermove to M_1. Analogously, $ctr(\Pi) \cup fix_{P_1}(M_1)$ admits the answer set $MC = \{na, nb, nc, unsat_C\}$ and $MC|_{\mathcal{H}(P_2)} = \{nc\} = M_2|_{\mathcal{H}(P_2)}$ is a countermove to M_1.

Proposition 3 *Let Π be a 2-ASP(Q) program of the form $\square_1 P_1 \square_2 P_2 : C$, and M_1 be a move for $\square_1$ (i.e., $M_1 \in AS(P_1)$), then there exists an answer set M_2 of $ctr(\Pi) \cup fix_{P_1}(M_1)$ iff $M_2|_{\mathcal{H}(P_2)}$ is a countermove to M_1.*

Let M_1 be a move for $\square_1$. If $\square_2 = \exists^{st}$ then $ctr(\Pi) = P_2 \cup C$. Thus, an answer set of $ctr(\Pi) \cup fix_{P_1}(M_1)$ corresponds to an answer set M_2 of $P_2 \cup fix_{P_1}(M_1)$ that satisfies C. Instead, if $\square_2 = \forall^{st}$ then $ctr(\Pi) = P_2 \cup \neg C$. Thus, an answer set of $ctr(\Pi) \cup fix_{P_1}(M_1)$ corresponds to an answer set M_2 of $P_2 \cup fix_{P_1}(M_1)$ that satisfies $\neg C$, and so, from Proposition 2, M_2 violates C. As a result, countermoves can be obtained from answer sets of $ctr(\Pi) \cup fix_{P_1}(M_1)$.

We now shift the attention to the second phase of the CEGAR pipeline which is *abstraction refinement*.

Before diving into the details of abstraction refinement for 2-ASP(Q) let us recall how it works in 2-QBF.

Let $\Phi = \exists X \forall Y \phi$, τ_1 be an assignment for variables in X, and μ_1 be an assignment for variables in Y such that $\phi[\tau_1][\mu_1]$ evaluates to false. Here, μ_1 is a countermove for each assignment τ_2 of variables in X such that $\phi[\tau_2][\mu_1]$ evaluates to false. Thus, to avoid considering such τ_2 as possible moves of $\exists$, the abstraction refinement adds the formula $\phi[\mu_1]$ to the refined abstraction. In this way, if $\phi[\mu_1]$ is satisfiable, then there exists a truth assignment τ of variables in X such that $\phi[\tau][\mu_1]$ evaluates to true. Thus, μ_1 is not a countermove to τ and τ can be a winning move. Conversely, if $\phi[\mu_1]$ is unsatisfiable then no winning moves for $\exists$ exist.

However, such a construction cannot be used in the ASP(Q) setting, mainly because in ASP(Q) countermoves are answer sets instead of arbitrary interpretations. The following example clarifies this issue.

Let Π be a 2-ASP(Q) of the form (1) and M_1, M_1' be two different answer sets of P_1. The answer sets of P_2 given M_1 (i.e., $P_2 \cup fix_{P_1}(M_1)$) can be, in the worst case, disjoint from answer sets of P_2 given M_1' (i.e., $P_2 \cup fix_{P_1}(M_1')$). Thus, it may happen that a countermove M to M_1 is not even a candidate countermove to M_1'. In general, let $M_2 \in AS(P_2 \cup fix_{P_1}(M_1))$ be such that $M = M_2|_{\mathcal{H}(P_2)}$ is a countermove to M_1. Then M is not a countermove to M_1' if one of the following holds:

1. there exists no $M_2' \in AS(P_2')$ such that $M = M_2'|_{\mathcal{H}(P_2)}$;
2. there exists $M_2' \in AS(P_2')$ such that $M = M_2'|_{\mathcal{H}(P_2)}$ and $C \cup fix_{P_2'}(M_2')$ is coherent if $\square_2 = \forall^{st}$; or $C \cup fix_{P_2'}(M_2')$ is incoherent.

where $P_2' = P_2 \cup fix_{P_1}(M_1)$.

Thus, the refinement process in ASP(Q) consists of extending the program P_1 (i.e., adding ad-hoc rules) in such a way that answer sets of the refined program correspond to answer sets of P_1 for which M is not a countermove. The following transformations define the rules used to refine P_1 according to a given countermove.

Definition 3 (Controlled program) *Let P be an ASP program and a be a fresh atom not appearing in P then $or(P, a) = \{H(r) \leftarrow B(r), \sim a \mid r \in P\}$.*

Intuitively, the controlled program for an ASP program P allows to decide whether rules in P should be ignored or not depending on the atom a. If a is true then all rules of P are trivially satisfied, thus they are "ignored". Otherwise, since a is false, $\sim a$ can be removed from each rule of $or(P, a)$ obtaining back rules of P. Along with $or(\cdot, \cdot)$, we need another transformation to map predicates over a fresh signature for each countermove. Let L be a set of literals, then $\sigma_\epsilon^\lambda(L, P)$ denotes the program obtained from P by replacing: (i) each literal $p(\vec{t}) \in L$ with $p_\epsilon^\lambda(\vec{t})$; and (ii) each literal $\sim p(\vec{t}) \in L$ with $\sim p_\epsilon^\lambda(\vec{t})$. We are now ready to define the refinement program by leveraging $or(\cdot, \cdot)$ and $\sigma_\epsilon^\lambda(\cdot, \cdot)$ transformations.

Definition 4 (Refinement Program) *Let Π be a 2-ASP(Q) program of the form (1), $M_1 \in AS(P_1)$ be a move for $\square_1$, and $M_2 \in AS(P_2 \cup fix_{P_1}(M_1))$ be such that $M = M_2|_{\mathcal{H}(P_2)}$ is a countermove to M_1 for $\square_2$. The refined program, denoted by Π^M, is the following program:*

$$\Pi^M = \begin{cases} \sigma^+_M(\mathcal{H}(P_2), \sigma^-_M(\overline{\mathcal{H}(P_2)}, P_2)) & \\ \sigma^-_M(\mathcal{H}(P_2), \{a \leftarrow | \ a \in M\}) & \\ fail_M \leftarrow p^-_M(\vec{t}), \sim p^+_M(\vec{t}) & \forall p(\vec{t}) \in \mathcal{H}(P_2) \\ fail_M \leftarrow p^+_M(\vec{t}), \sim p^-_M(\vec{t}) & \forall p(\vec{t}) \in \mathcal{H}(P_2) \\ \sigma^+_M(B, or(C, fail_M)) & if \ \square_2 = \forall^{st} \\ \sigma^+_M(B, or(\neg C, fail_M)) & if \ \square_2 = \exists^{st} \end{cases}$$

where $B = \mathcal{H}(P_2) \cup \overline{\mathcal{H}(P_2)} \cup \mathcal{H}(C) \cup \overline{\mathcal{H}(C)}$,

The idea behind the refinement program is to use a set of rules to (i) simulate the reduct of the program P_2 w.r.t. M; and (ii) verify the (in)coherence of C iff M is a candidate countermove. The following example better motivates the main intuition behind refinement program.

Example 4 *Let Π be the program from Example 2, then $\mathcal{H}(P_1) = \{a, na, b, nb\}$, $\mathcal{H}(P_2) = \{c, nc\}$ and $\mathcal{H}(C) = \emptyset$.*

In this case, $M_1 = \{na, nb\} \in AS(P_1)$ is a move for $\exists^{st}$ and $M_2 = \{na, nb, nc\} \in AS(P_2 \cup fix_{P_1}(M_1))$ is such that $M = M_2|_{\mathcal{H}(P_2)} = \{nc\}$ is a countermove to M_1 for $\forall^{st}$. Thus, Π^M is made of the following set of rules:

$$\Pi^M = \left\{ \begin{array}{lll} c^+_M & \leftarrow & \sim nc^-_M. \\ nc^+_M & \leftarrow & \sim c^-_M. \\ & \leftarrow & a, \sim nc^-_M. \\ nc^-_M & \leftarrow & \\ fail_M & \leftarrow & c^-_M, \sim c^+_M. \\ fail_M & \leftarrow & c^+_M, \sim c^-_M. \\ fail_M & \leftarrow & nc^-_M, \sim nc^+_M. \\ fail_M & \leftarrow & nc^+_M, \sim nc^-_M. \\ & \leftarrow & nc^+_M, \ \sim nb^+_M, \sim fail_M. \end{array} \right\}$$

In this construction, the following rules simulate the reduct of P_2 w.r.t. M by keeping literals over atoms in $\mathcal{H}(P_1)$ unchanged:

$$\begin{array}{lll} nc^+_M & \leftarrow & \sim c^-_M. \\ c^+_M & \leftarrow & \sim nc^-_M. \\ & \leftarrow & a, \sim nc^-_M. \\ nc^-_M & \leftarrow & \end{array}$$

In particular, c^-_M and nc^-_M are fixed w.r.t. M. More precisely, nc^-_M is fixed as true by the fact $nc^-_M \leftarrow$ and c^-_M is fixed as false since it does not appear in any rule head. Moreover, c^-_M and nc^-_M also appear in rules negative body, and thus they are used to obtain the reduct (Gelfond and Lifschitz 1991). On the other hand, c^+_M and nc^+_M appear only in rule heads and (possibly) in the positive body to compute a model of the reduct.

Thus, if it is possible to derive c^+_M as false and nc^+_M as true then M is a model of the reduct. This condition is verified by means of the following rules contained in Π^M:

$$\begin{array}{lll} fail_M & \leftarrow & c^-_M, \sim c^+_M. \\ fail_M & \leftarrow & nc^-_M, \sim nc^+_M. \\ fail_M & \leftarrow & c^+_M, \sim c^-_M. \\ fail_M & \leftarrow & nc^+_M, \sim nc^-_M. \end{array}$$

More precisely, $fail_M$ is derived as false iff M is a model of the reduct. Thus, we can leverage $fail_M$ to control the activation of rules in C. If $fail_M$ is derived as false then the controlled program of C, included in Π^M, activates the rules of C. Conversely, if $fail_M$ is derived as true then the controlled program of C disables the rules of C. □

Solving ASP(Q) via CEGAR. Building upon the definitions of *countermove program* (i.e., Definition 2) and *refinement program* (i.e., Definition 4) we can define a CEGAR-based procedure for deciding the coherence of 2-ASP(Q) programs (see Algorithm 2).

Given a 2-ASP(Q) program, Algorithm 2 returns a winning move for $\square_1$ if any, otherwise returns $NULL$. As first step, Algorithm 2 computes a move for $\square_1$ by solving P_1 (line 2). If P_1 is incoherent, then no move exists for $\square_1$ and so there is no winning move. Thus, the algorithm stops by returning $NULL$ (line 15). Conversely, if P_1 is coherent, then $solve(P_1)$ returns an answer set M_1 of P_1, which is a move for $\square_1$. At this point, the algorithm enters the CEGAR pipeline which iteratively alternates countermove search and abstraction refinement. Thus, first of all the algorithm computes a countermove to M_1 by solving the program $ctr(\Pi) \cup fix_{P_1}(M1)$ (line 4). If $ctr(\Pi) \cup fix_{P_1}(M1)$ is incoherent, then no countermove to M_1 exists and thus the algorithm terminates, returning M_1 as a winning move (line 6). Otherwise, an answer set M_2 of $ctr(\Pi) \cup fix_{P_1}(M1)$ is obtained. At this point, the countermove M to M_1 is computed by projecting M_2 over the atoms in $\mathcal{H}(P_2)$ (line 8) and M is stored in the set of all countermoves CMs. Now, since M_1 cannot be a winning move, the algorithm enters the second stage of the CEGAR pipeline which is abstraction refinement. To this end, the refined program Ref is constructed by adding to P_1 the refinement program for each countermove in CMs. By solving the refined program Ref (line 12), the algorithm searches for a new move for $\square_1$. Thus, if Ref is incoherent, $\square_1$ does not have any other move and so the algorithm terminates by returning $NULL$ (line 15). Otherwise, a new move for $\square_1$ is obtained by projecting the answer set computed by $solve(Ref)$ over atoms of P_1 (line 13) and the loop repeats until either no move can be found for $\square_1$, or no countermove exists for $\square_2$. A complete solving example is given in the supplementary material.

Implementation and Experiments

The proposed approach has been implemented into a new ASP(Q) solver named CASPER. In this section, we discuss implementation details and an empirical evaluation conducted to assess the impact of CASPER in ASP(Q) solving.

Implementation

The *casper* system is a Python implementation of the proposed approach for solving 2-ASP(Q), which leverages the CLINGO Python API to compute answer sets of the refined and countermove programs. More in detail, CASPER takes as input a $2-$ASP(Q) program Π and starts the search for a winning move. At this stage, the solver calls CLINGO as an oracle to compute a move for the first player. Then, for each move M_1, as reported in Algorithm 2, CASPER uses

Algorithm 2: CEGAR for 2-ASP(Q)

Input: A 2-ASP(Q) program Π of the form $\square_1 P_1 \square_2 P_2 : C$
Output: M_1 if there exists a winning move for $\square_1$, $NULL$ otherwise

1: $CMs = \emptyset$
2: M_1 = solve(P_1)
3: **while** $M_1 \neq \bot$ **do**
4: $\quad M_2 = solve(ctr(\Pi) \cup fix_{P_1}(M_1))$.
5: $\quad$ **if** $M_2 = \bot$ **then**
6: $\quad\quad$ **return** M_1
7: $\quad$ **else**
8: $\quad\quad M = M_2|_{\mathcal{H}(P_2)}$
9: $\quad\quad CMs = CMs \cup \{M\}$
10: $\quad\quad Ref = P_1 \cup \bigcup_{M \in CMs} \Pi^M$
11: $\quad$ **end if**
12: $\quad Next$ = solve(Ref)
13: $\quad M_1 = Next|_{\mathcal{H}(P_1)}$
14: **end while**
15: **return** $NULL$

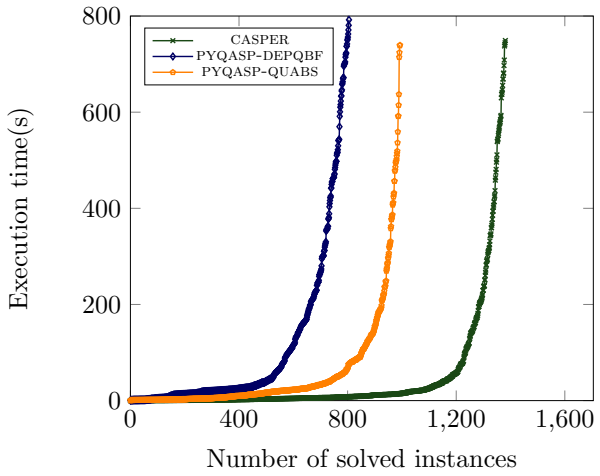

Figure 1: Overall execution time

another CLINGO call to compute a countermove for the second player, by solving $ctr(\Pi)$ with M_1 under assumptions. Each found countermove is then used by CASPER to compute refinement (sub)programs which are cached in memory for efficiently computing the refined program Ref.

As a result, CASPER computes quantified answer sets for existential 2-ASP(Q), or returns *SAT* or *UNSAT* for universal 2-ASP(Q) programs. Note that CASPER can also enumerate quantified answer sets of existential 2-ASP(Q) programs.

Experiments

Now we present an experimental evaluation for assessing the impact of the proposed approach. Benchmarks and source code are available at https://osf.io/wcy5n

Benchmarks. In this evaluation we included seven benchmarks. More precisely, we considered benchmarks used in previous assessments of ASP(Q) systems (Amendola et al. 2022; Faber, Mazzotta, and Ricca 2023): *Argumentation Coherence* (AC) (http://argumentationcompetition.org/2019); *Minmax Clique* (Cao et al. 1995) (MMC); *Paracoherent ASP* (PAR) (Amendola et al. 2021); and *Point of No Return* (Janhunen 2022) (PONR). Then, we built two benchmarks for Clique Coloring (CC), and Thue Number (THUE) problems (Schaefer and Umans 2002). For these problems, we generated instances using the Erdős-Renyi model (Erdős and Rényi 1959), from Python NetworkX library (Hagberg, Schult, and Swart 2008), which is a well-known model for real-world networks. Finally, we considered the Propositional Abduction Problem (Eiter and Gottlob 1995a; Saikko, Wallner, and Järvisalo 2016), including instances with formulae up to 9851 clauses and 3284 variables.

Compared Methods. We compared the CASPER system with the best performing versions of the PYQASP solver selected in accordance with the results reported by Faber, Mazzotta, and Ricca (2023). More precisely, we considered only two of the three backends supported by PYQASP, since during experiments we noticed that the version using the RAREQS QBF solver (by Janota http://sat.inesc-id.pt/~mikolas/sw/areqs) provided incorrect results. In what follows, we denote by PYQASP-QUABS the version of PYQASP using QUABS(by Tentrup - https://github.com/ltentrup/quabs) as backend solver, and by PYQASP-DEPQBF the version of PYQASP using DEPQBF (by Lonsing - https://lonsing.github.io/depqbf) as backend solver.

Experimental Setup. All experiments were executed on Intel(R) Xeon(R) CPUs E5-4610 v2 @ 2.30GHz running Debian Linux 12, with memory and CPU time (i.e., user+system) limited to 8GB and 800s. Each system was limited to run on a single core. Source code and benchmark suite are provided in the supplementary material.

Results. Obtained results are reported in the cactus plot in Figure 1 and in Table 1. Recall that in a cactus plot instances are sorted by solving time. A point (i, j) in the plot indicates that a system solved the i-th instance within j seconds. Overall, it is possible to observe that CASPER is the best performing system, solving 388 instances more than the best version of *pyqasp* (i.e., PYQASP-QUABS) and requiring a smaller execution time. However, we can observe different trends by considering each benchmark separately.

Table 1 reports, for each benchmark and cumulatively, the number of solved instances (Sol.), total execution time (Sum t.), and PAR-2 score for each solver. Recall that, the PAR-2 is defined as the sum of all execution times for solved instances and 2 times the timeout for unsolved ones. Thus, low PAR-2 scores indicate good performance.

More precisely, the only benchmark in which CASPER exhibited overhead is PAR. This is due to the fact that instances of PAR have a huge number of moves and countermoves for the two players (i.e., millions of answer set for each player). In this setting, even if countermoves computation is very fast, CASPER was not able to aggressively prune the search space. Moreover, this overhead can be also observed in *pyqasp-quabs* which solved fewer instances than CASPER. On the other hand, *pyqasp-depqbf* is efficient here because the QBF solver DEPQBF is associated to a pre-processor

Bench.	#inst	CASPER			PYQASP-QUABS			PYQASP-DEPQBF		
		Sol.	Sum t.(s)	PAR-2(s)	Sol.	Sum t.(s)	PAR-2(s)	Sol.	Sum t.(s)	PAR-2(s)
ABD	294	**294**	**3718.14**	3718.14	**294**	16796.68	16796.68	98	28987.47	28987.47
AC	326	**267**	16092.74	63292.74	99	13675.55	195275.55	131	11804.05	167804.05
CC	165	**93**	6321.65	63921.65	34	4652.32	109452.32	10	3181.91	127181.91
MMC	225	**225**	**862.42**	862.42	**225**	1099.25	1099.25	**225**	15077.46	15077.46
PAR	441	306	26162.43	134162.43	248	7390.73	161790.73	**340**	26969.09	107769.09
PONR	94	**94**	177.79	177.79	63	2314.63	27114.63	0	0	75200.00
THUE	160	**102**	7242.76	53642.76	30	6993.96	110993.96	1	441.94	127641.94
OVERALL	1705	**1381**	60577.93	319777.93	993	52923.12	622523.12	805	86461.92	806461.92

Table 1: Overall system comparison

(i.e., BLOQQER by Biere et al. - http://fmv.jku.at/bloqqer) that is very effective on these instances. For ABD and MMC, PYQASP and CASPER solved all the instances. However, CASPER introduced a significant improvement in solving time w.r.t. PYQASP. Finally, we observe that CASPER outperformed PYQASP on the remaining benchmarks (AC, CC, PONR, and THUE). In particular, CASPER was able to solve, in every benchmark, more than 30 additional instances w.r.t. each version of PYQASP. On these benchmarks, PYQASP generated hard to evaluate QBF formulas, whereas CASPER quickly converged to a winning move (roughly within 800 countermoves) thanks to its effective refinement procedure.

Finally, we carried out an additional experiment in which we compared CASPER with CLINGO (Gebser et al. 2016) on benchmarks for which a suitable ASP encoding is available in the literature; these are ABD (Saikko, Wallner, and Järvisalo 2016) and PAR (Amendola et al. 2021). For the remaining ones we are not aware of suitable ASP encodings. However, it is worth noting that CASPER is built on top of CLINGO. Consequently, it can accept the same input encodings as CLINGO, interpreted as ASP(Q) programs with a single existential quantifier. Thus, even if a suitable ASP encoding were available for the remaining benchmarks, CASPER and CLINGO would exhibit the same performance. Nonetheless, obtained results, reported in appendix due to space constraints, show that CASPER is more efficient than CLINGO solving 200 more instances overall. This improvement is due to the fact that, in these benchmarks, the ASP(Q) formulation enables a more efficient solving.

Related Work

State-of-the-art ASP(Q) systems, such as PYQASP (Faber, Mazzotta, and Ricca 2023) and QASP (Amendola et al. 2022), are based on a translation of ASP(Q) programs (possibly with more than 2 quantifiers) in QBF by leveraging lp2* tools (Janhunen 2004, 2018). Even though PYQASP adopts effective techniques for obtaining compact QBFs, in some cases, produced QBFs are hard to evaluate. As a result, this translation easily becomes a bottleneck in ASP(Q) solving. In contrast, CASPER avoids such a transformation for 2-ASP(Q) programs by implementing a CEGAR-based approach, inspired by 2QBF (Janota and Marques-Silva 2011). Note that, analogously to QBF (Janota and Marques-Silva 2011; Janota et al. 2016), the CASPER approach can be extend to ASP(Q) program with an arbitrary number of quantifiers. However, since in the ASP(Q) game we have different ASP programs that are part of the game instead of a unique formula and so the approach proposed by Janota et al. (2016) cannot be used for arbitrary ASP(Q) programs.

Among related formalisms with implementations, we mention stable-unstable semantics (Bogaerts, Janhunen, and Tasharrofi 2016) and quantified answer set semantics (Fandinno et al. 2021).

The stable-unstable semantics, like 2-ASP(Q), can express problems at the second level of the PH. It was first implemented as a prototype (Bogaerts, Janhunen, and Tasharrofi 2016), and later through a rewriting to ASP (Janhunen 2022). However, the latter requires users to define module interfaces and manually coordinate the toolchain whereas CASPER is available as a Python module for direct use. An empirical comparison of PYQASP and the stable-unstable implementation on the same benchmarks is available in (Faber, Mazzotta, and Ricca 2023).

Quantified answer set semantics (Fandinno et al. 2021) has also been implemented via a translation to QBF, which differs from the one used by PYQASP due to a different semantics of quantifiers. However, a translation from quantified answer set semantics to ASP(Q) and back was proposed by Fandinno et al. (2021) but never implemented. Thus, CASPER is not directly usable under this semantics.

For further comparison of ASP(Q) with alternative formalisms, we refer the reader to (Amendola, Ricca, and Truszczynski 2019; Fandinno et al. 2021).

Conclusion

State-of-the-art ASP(Q) solvers rely on translating programs into QBF, which might lead to hard-to-evaluate formulae. In this work, we propose a novel 2-ASP(Q) solving technique inspired by the CEGAR methodology, which has proven successful in QBF solving. To this end, we provided a game-based characterization of ASP(Q) semantics, and introduced ad-hoc rewriting strategies for adapting CEGAR to the ASP(Q) setting. This approach paves the way to a new class of ASP(Q) solvers, that do not require translation to QBF. The impact of the proposed technique is demonstrated by the results of an extensive experimental evaluation, which showed that our implementation, CASPER, consistently outperforms state-of-the-art ASP(Q) solvers. As future work, we aim to generalize the approach to handle ASP(Q) programs with an arbitrary number of quantifiers.

Acknowledgments

This work was supported by the Italian Ministry of Industrial Development (MISE) under project EI-TWIN n. F/310168/05/X56 CUP B29J24000680005; and by the Italian Ministry of Research (MUR) under projects: PNRR FAIR - Spoke 9 - WP 9.1 CUP H23C22000860006, and Tech4You CUP H23C22000370006.

References

Amendola, G.; Cuteri, B.; Ricca, F.; and Truszczynski, M. 2022. Solving Problems in the Polynomial Hierarchy with ASP(Q). In *LPNMR*, volume 13416 of *Lecture Notes in Computer Science*, 373–386. Springer.

Amendola, G.; Dodaro, C.; Faber, W.; and Ricca, F. 2021. Paracoherent answer set computation. *Artif. Intell.*, 299: 103519.

Amendola, G.; Ricca, F.; and Truszczynski, M. 2019. Beyond NP: Quantifying over Answer Sets. *Theory Pract. Log. Program.*, 19(5-6): 705–721.

Azzolini, D.; Mazzotta, G.; Ricca, F.; and Riguzzi, F. 2025. Most Probable Explanation in Probabilistic Answer Set Programming. In *Proceedings of the Thirty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2025, Montreal, Canada, August 16-22, 2025*, 9049–9057. ijcai.org.

Bellusci, P.; Mazzotta, G.; and Ricca, F. 2022. Modelling the Outlier Detection Problem in ASP(Q). In *PADL*, volume 13165 of *Lecture Notes in Computer Science*, 15–23. Springer.

Bogaerts, B.; Janhunen, T.; and Tasharrofi, S. 2016. Stable-unstable semantics: Beyond NP with normal logic programs. *TPLP*, 16(5-6): 570–586.

Brewka, G.; Eiter, T.; and Truszczynski, M. 2011. Answer set programming at a glance. *Commun. ACM*, 54(12): 92–103.

Brewka, G.; Eiter, T.; and Truszczynski, M. 2016. Answer Set Programming: An Introduction to the Special Issue. *AI Mag.*, 37(3): 5–6.

Calimeri, F.; Faber, W.; Gebser, M.; Ianni, G.; Kaminski, R.; Krennwallner, T.; Leone, N.; Maratea, M.; Ricca, F.; and Schaub, T. 2020. ASP-Core-2 Input Language Format. *Theory Pract. Log. Program.*, 20(2): 294–309.

Cao, F.; Du, D.-Z.; Gao, B.; Wan, P.-J.; and Pardalos, P. M. 1995. *Minimax Problems in Combinatorial Optimization*, 269–292. Boston, MA.

Clarke, E. M.; Fehnker, A.; Han, Z.; Krogh, B. H.; Ouaknine, J.; Stursberg, O.; and Theobald, M. 2003a. Abstraction and Counterexample-Guided Refinement in Model Checking of Hybrid Systems. *Int. J. Found. Comput. Sci.*, 14(4): 583–604.

Clarke, E. M.; Grumberg, O.; Jha, S.; Lu, Y.; and Veith, H. 2003b. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5): 752–794.

Dantsin, E.; Eiter, T.; Gottlob, G.; and Voronkov, A. 2001. Complexity and expressive power of logic programming. *ACM Computing Surveys*, 33(3): 374–425.

Eiter, T.; and Gottlob, G. 1995a. The Complexity of Logic-Based Abduction. *J. ACM*, 42(1): 3–42.

Eiter, T.; and Gottlob, G. 1995b. On the Computational Cost of Disjunctive Logic Programming: Propositional Case. *Ann. Math. Artif. Intell.*, 15(3-4): 289–323.

Erdős, P.; and Rényi, A. 1959. On random graphs I. *Publicationes Mathematicae Debrecen*, 6(290-297): 18.

Faber, W.; Mazzotta, G.; and Ricca, F. 2023. An Efficient Solver for ASP(Q). *Theory Pract. Log. Program.*, 23(4): 948–964.

Faber, W.; and Morak, M. 2022. Evaluating Epistemic Logic Programs via Answer Set Programming with Quantifiers. In *HYDRA/RCRA@LPNMR*, volume 3281 of *CEUR Workshop Proceedings*, 78–89. CEUR-WS.org.

Faber, W.; Morak, M.; and Chrpa, L. 2022. Determining Action Reversibility in STRIPS Using Answer Set Programming with Quantifiers. In *PADL*, volume 13165 of *Lecture Notes in Computer Science*, 42–56. Springer.

Fandinno, J.; Laferrière, F.; Romero, J.; Schaub, T.; and Son, T. C. 2021. Planning with Incomplete Information in Quantified Answer Set Programming. *Theory Pract. Log. Program.*, 21(5): 663–679.

Gebser, M.; Kaminski, R.; Kaufmann, B.; Ostrowski, M.; Schaub, T.; and Wanko, P. 2016. Theory Solving Made Easy with Clingo 5. In *ICLP (Technical Communications)*, volume 52 of *OASICS*, 2:1–2:15. Schloss Dagstuhl.

Gebser, M.; Kaminski, R.; and Schaub, T. 2011. Complex optimization in answer set programming. *TPLP*, 11(4-5): 821–839.

Gebser, M.; Leone, N.; Maratea, M.; Perri, S.; Ricca, F.; and Schaub, T. 2018. Evaluation Techniques and Systems for Answer Set Programming: a Survey. In *IJCAI*, 5450–5456. ijcai.org.

Gelfond, M.; and Lifschitz, V. 1991. Classical Negation in Logic Programs and Disjunctive Databases. *New Gener. Comput.*, 9(3/4): 365–386.

Hagberg, A. A.; Schult, D. A.; and Swart, P. J. 2008. Exploring Network Structure, Dynamics, and Function using NetworkX. In Varoquaux, G.; Vaught, T.; and Millman, J., eds., *Proceedings of the 7th Python in Science Conference*, 11–15. Pasadena, CA USA.

Janhunen, T. 2004. Representing Normal Programs with Clauses. In de Mántaras, R. L.; and Saitta, L., eds., *Proceedings of ECAI'2004.*, 358–362. IOS Press.

Janhunen, T. 2018. Cross-Translating Answer Set Programs Using the ASPTOOLS Collection. *Künstliche Intell.*, 32(2-3): 183–184.

Janhunen, T. 2022. Implementing Stable-Unstable Semantics with ASPTOOLS and Clingo. In *PADL*, volume 13165 of *Lecture Notes in Computer Science*, 135–153. Springer.

Janota, M.; Klieber, W.; Marques-Silva, J.; and Clarke, E. M. 2016. Solving QBF with counterexample guided refinement. *Artif. Intell.*, 234: 1–25.

Janota, M.; and Marques-Silva, J. 2011. Abstraction-Based Algorithm for 2QBF. In *SAT*, volume 6695 of *Lecture Notes in Computer Science*, 230–244. Springer.

Mazzotta, G.; Ricca, F.; and Truszczynski, M. 2024. Quantifying over Optimum Answer Sets. *Theory Pract. Log. Program.*, 24(4): 716–736.

Papadimitriou, C. H. 1994. *Computational complexity*. Addison-Wesley.

Saikko, P.; Wallner, J. P.; and Järvisalo, M. 2016. Implicit Hitting Set Algorithms for Reasoning Beyond NP. In *KR*, 104–113. AAAI Press.

Schaefer, M.; and Umans, C. 2002. Completeness in the polynomial-time hierarchy: A compendium. *SIGACT news*, 33(3): 32–49.

Truszczynski, M. 2011. Trichotomy and dichotomy results on the complexity of reasoning with disjunctive logic programs. *Theory Pract. Log. Program.*, 11(6): 881–904.