Beyond NP: Quantifying over Answer Sets

GIOVANNI AMENDOLA[1]

FRANCESCO RICCA[1]

MIREK TRUSZCZYNSKI[2]

[1]*University of Calabria, Rende, Italy*
(*e-mail:* {amendola,ricca}@mat.unical.it)
[2]*University of Kentucky, KY, USA*
(*e-mail:* mirek@cs.uky.edu)

submitted 1 January 2003; revised 1 January 2003; accepted 1 January 2003

Abstract

Answer Set Programming (ASP) is a logic programming paradigm featuring a purely declarative language with comparatively high modeling capabilities. Indeed, ASP can model problems in NP in a compact and elegant way. However, modeling problems beyond NP with ASP is known to be complicated, on the one hand, and limited to problems in Σ_2^P on the other. Inspired by the way Quantified Boolean Formulas extend SAT formulas to model problems beyond NP, we propose an extension of ASP that introduces quantifiers over *stable models* of programs. We name the new language ASP with Quantifiers (ASP(Q)). In the paper we identify computational properties of ASP(Q); we highlight its modeling capabilities by reporting natural encodings of several complex problems with applications in artificial intelligence and number theory; and we compare ASP(Q) with related languages. Arguably, ASP(Q) allows one to model problems in the Polynomial Hierarchy in a direct way, providing an elegant expansion of ASP beyond the class NP.

KEYWORDS: ASP, Quantified Logics, Polynomial Hierarchy

1 Introduction

Answer Set Programming (ASP) (Brewka et al. 2011) is a logic programming paradigm for modeling and solving search and optimization problems. It is supported by a purely declarative formalism of logic programs with the semantics of stable models (Gelfond and Lifschitz 1991) (also known as *answer sets* (Lifschitz 2002)), and by several systems able to compute them (Gebser et al. 2018). ASP was primarily aimed at problems whose decision versions are in the class NP. Indeed, ASP can model problems in NP in a compact and elegant way by means of an intuitive and easy to follow methodology known as generate-define-test (Lifschitz 2002) (also known as guess and check (Eiter et al. 2000)). Furthermore, implementations such as *clasp* (Gebser et al. 2015), and *wasp* (Alviano et al. 2015; Alviano et al. 2019) have been shown to be effective in solving problems of practical interest (Gebser et al. 2017) on industrial-grade instances (Dodaro et al. 2016; Gebser et al. 2018; Erdem et al. 2016).

Modeling problems beyond the class NP with ASP is possible to some extent. Namely, when disjunctions are allowed in the heads of rules, every decision problem in the class Σ_2^P can be modeled in a uniform way by a finite program (Dantsin et al. 2001). However, modeling problems beyond NP with ASP is complicated and the generate-define-test approach is no longer sufficient

in general. Additional techniques such as *saturation* (Eiter and Gottlob 1995) are needed but they are difficult to use, and may introduce constraints that have no direct relation to constraints of the problem being modeled. As stated explicitly in (Gebser et al. 2011) "unlike the ease of common ASP modeling, [...] these techniques are rather involved and hardly usable by ASP laymen."

The primary goal of our work is to address the shortcomings of ASP in modeling problems beyond NP. Building on the way Quantified Boolean formulas (QBFs) extend SAT formulas to model problems from PSPACE, we propose a generalization of ASP that introduces quantifiers over stable models of programs. We name the new language *ASP with Quantifiers* (ASP(Q)) and refer to programs in that language as *quantified programs*.

In the paper we formally introduce the language ASP(Q) and its semantics. We identify computational properties of ASP(Q). In particular, we show that every problem in the Polynomial Hierarchy can be uniformly modeled by a quantified program. Moreover, we show that no loss of expressivity results if we restrict programs defining quantifiers to be normal. An important consequence of that observation is that when using ASP(Q) to model problems, one can resort to the generate-define-test approach to specify these "quantifying" programs. This typically simplifies modeling and verifying correctness. We illustrate these claims by presenting natural encodings of several complex problems with applications in artificial intelligence and mathematics.

In the last part of the paper, we compare ASP(Q) with alternative approaches for modeling problems beyond NP. Earlier efforts in this direction include: the *stable-unstable* formalism (Bogaerts et al. 2016), various program transformations (Eiter and Polleres 2006; Redl 2017; Faber and Woltran 2011), applications of meta-programming (Redl 2017; Gebser et al. 2011) and more.[1] In particular, we deepen the comparison with disjunctive programs and the stable-unstable formalism, indicating key differences and their implications by means of additional modeling examples. We also extensively compare ASP(Q) with the language of QBFs, which served as a direct inspiration for our work. A single sentence summary of our work is: ASP(Q) allows one to model problems in the Polynomial Hierarchy in a direct way, providing an elegant expansion of ASP beyond the class NP.

2 Formal Framework

We start by recalling syntax and semantics of *Answer Set Programming* (ASP). We then introduce syntax and semantics of *ASP with Quantifiers* (ASP(Q)).

2.1 Answer Set Programming

Let $\mathscr{R}$ be a set of predicates, C a set of constants, and V a set of variables. A *term* is a constant or a variable. An atom a of arity $n \in \mathbb{N}$ is of the form $p(t_1,...,t_n)$, where p is a predicate from $\mathscr{R}$ and $t_1,...,t_n$ are terms. A *disjunctive rule* r is of the form

$$a_1 \vee \ldots \vee a_l \leftarrow b_1,\ldots,b_m,\ not\ c_1,\ldots,\ not\ c_n, \tag{1}$$

where all a_i, b_j, and c_k are atoms; $l,m,n \geq 0$ and $l+m+n > 0$; *not* represents *negation-as-failure*, also known as *default negation*. The set $H(r) = \{a_1,...,a_l\}$ is the *head* of r; the sets $B^+(r) = \{b_1,...,b_m\}$ and $B^-(r) = \{c_1,\ldots,c_n\}$ are the sets of the *positive body* and the *negative body* atoms of r, respectively. A rule r is *safe* if each of its variables occurs in some positive body atom. We

[1] For example, weak constraints allow to model decision problems that are Δ_3^P-complete (Buccafurri et al. 2000).

restrict attention to programs built of safe rules only. A rule r is a *fact*, if $B^+(r) \cup B^-(r) = \emptyset$ (we then omit $\leftarrow$ from the notation); a *constraint*, if $H(r) = \emptyset$; *normal*, if $|H(r)| \leq 1$; and *positive*, if $B^-(r) = \emptyset$. A *(disjunctive logic) program* P is a finite set of disjunctive rules. P is called *normal* [resp. *positive*] if each $r \in P$ is normal [resp. positive]. We define $At(P) = \bigcup_{r \in P} At(r)$, that $At(P)$ is the set of all atoms occurring in the program P. A program P is *stratified* if there is a level mapping $\|.\|_s$ of P such that for every rule r of P: (i) For any predicate p occurring in $B^+(r)$, and for any p' occurring in $H(r)$, $\|p\|_s \leq \|p'\|_s$, and (ii) For any predicate p occurring in $B^-(r)$, and for any p' occurring in $H(r)$, $\|p\|_s < \|p'\|_s$.

The *Herbrand universe* of P, denoted by U_P, is the set of all constants appearing in P, except that when no constants appear in P, we take $U_P = \{a\}$, where a is an arbitrary constant. The *Herbrand base* of P, denoted as B_P, is the set of all ground atoms that can be obtained from the predicate symbols appearing in P and the constants of U_P. Given a rule r occurring in a program P, a *ground instance* of r is a rule obtained from r by replacing every variable X in r by $\sigma(X)$, where σ is a substitution mapping the variables occurring in r to constants in U_P. The *ground instantiation* of P, denoted by $ground(P)$, is the set of all the ground instances of the rules occurring in P. Any set $I \subseteq B_P$ is an *interpretation*; it is a *model* of a program P (denoted $I \models P$) if for each rule $r \in ground(P)$, we have $I \cap H(r) \neq \emptyset$ whenever $B^+(r) \subseteq I$ and $B^-(r) \cap I = \emptyset$ (in such case, I is a model of r, denoted $I \models r$). A model M of P is *minimal* if no model $M' \subset M$ of P exists. We denote by $MM(P)$ the set of all minimal models of P. For a program P without constraints we write P^I for the well-known *Gelfond-Lifschitz reduct* (Gelfond and Lifschitz 1991) with respect to interpretation I, that is, the set of rules $H(r) \leftarrow B^+(r)$, obtained from rules $r \in ground(P)$ such that $B^-(r) \cap I = \emptyset$. An answer set (or stable model) of a program P without constraints is an interpretation I such that $I \in MM(P^I)$. For the general case, we write $P_\leftarrow$ for the set of constraints of a disjunctive logic program P. We denote by $AS(P)$ the set of all *answer sets (or stable models)* of such programs P, that is, the set of all answer sets of $P \setminus P_\leftarrow$ that are models for $P_\leftarrow$.

We say that a program P is *coherent*, if it has at least one answer set (that is, $AS(P) \neq \emptyset$), otherwise, P is *incoherent*.

2.2 Answer Set Programming with Quantifiers

An *ASP with Quantifiers* (ASP(Q)) program Π is an expression of the form:

$$\Box_1 P_1 \; \Box_2 P_2 \; \cdots \; \Box_n P_n : C, \tag{2}$$

where, for each $i = 1, \ldots, n$, $\Box_i \in \{\exists^{st}, \forall^{st}\}$, P_i is an ASP program, and C is a stratified normal ASP program.[2] Symbols $\exists^{st}$ and $\forall^{st}$ are named *existential* and *universal answer set quantifiers*, respectively. An ASP(Q) program Π of the form (2) is *existential* (*universal*, respectively) if $\Box_1 = \exists^{st}$ ($= \forall^{st}$, respectively). If for each $i = 1, \ldots, n$ the ASP program P_i is normal, then Π is called a *normal* ASP(Q) program. Given a logic program P and an intepretation I over B_P, and an ASP(Q) program Π the form (2), we denote by $fix_P(I)$ the set of facts and constraints $\{a \mid a \in I\} \cup \{\leftarrow a \mid a \in B_P \setminus I\}$, and by $\Pi_{P,I}$ the ASP(Q) program of the form (2), where P_1 is replaced by $P_1 \cup fix_P(I)$, that is, $\Pi_{P,I} = \Box_1 (P_1 \cup fix_P(I)) \cdots \Box_n P_n : C$. We now define *coherence* of ASP(Q) programs by induction on the number of quantifiers in the program.

[2] This condition is sufficient to model compactly constraints by exploiting the modeling advantages of inductive definitions. C is contemplated in the definition of ASP(Q) just because it makes more natural the modeling of problems.

- $\exists^{st} P : C$ 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.

For instance, an ASP(Q) program $\Pi = \exists^{st} P_1 \forall^{st} P_2 \cdots \exists^{st} P_{n-1} \forall^{st} P_n : C$ is coherent if there exists an answer set M_1 of P'_1 such that for each answer set M_2 of P'_2 there is an answer set M_3 of $P'_3, \ldots,$ there is an answer set M_{n-1} of P'_{n-1} such that for each answer set M_n of P'_n, there is an answer set of $C \cup fix_{P'_n}(M_n)$, where $P'_1 = P_1$, and $P'_i = P_i \cup fix_{P'_{i-1}}(M_{i-1})$, if $i \geq 2$.

For an ASP(Q) program Π of the form (2) such that $\Box_1 = \exists^{st}$, we say that $M \in AS(P_1)$ is a *quantified answer set* of Π, whenever $(\Box_2 P_2 \cdots \Box_n P_n : C)_{P_1,M}$ is coherent, in case of $n > 1$, and whenever $C \cup fix_{P_1}(M)$ is coherent, in case of $n = 1$. We denote by $QAS(\Pi)$ the set of all quantified answer sets of Π. Finally, note that the definition of quantified answer set can be naturally extended to programs with strong negation, choice rules, aggregates and other extensions (Gebser and Schaub 2016). Thus, in the examples we resort also to these extensions that are part of the ASPCore standard input language (Gebser et al. 2018).

Example 1
Consider the ASP(Q) program $\Pi = \exists^{st} P_1 \forall^{st} P_2 : C$, where $P_1 = \{a(1) \vee a(2)\}$, $P_2 = \{b(1) \vee b(2) \leftarrow a(1);\ b(2) \leftarrow a(2)\}$, and $C = \{\leftarrow b(1),\ not\ b(2)\}$. The program P_1 has two answer sets $\{a(1)\}$ and $\{a(2)\}$. Hence, to establish the coherence of Π, we have to check if at least one of $\{a(1)\}$ and $\{a(2)\}$ is a quantified answer set of Π. Considering $\{a(1)\}$, we have $fix_{P_1}(\{a(1)\}) = \{a(1); \leftarrow a(2)\}$. Under the notation used above, $P'_2 = P_2 \cup fix_{P_1}(\{a(1)\})$. Thus, $AS(P_2 \cup fix_{P_1}(\{a(1)\})) = \{\{a(1), b(1)\}, \{a(1), b(2)\}\}$. For $M = \{a(1), b(1)\}$ we have $fix_{P'_2}(M) = \{a(1); b(1); \leftarrow a(2); \leftarrow b(2)\}$, and it is clear that the program $C \cup fix_{P'_2}(M)$ is not coherent. Therefore, $\{a(1)\}$ is not a quantified answer set of Π. On the other hand, a similar analysis for the other answer set of P_1, $\{a(2)\}$, shows that it is a quantified answer set of Π.

ASP(Q) is a straightforward generalization of ASP in a sense made formal in the following theorem.[3]

Theorem 1
Let P be an ASP program, and let Π be the ASP(Q) program of the form (2), where $n = 1$, $\Box_1 = \exists^{st}$, $P_1 = P$, and $C = \emptyset$. Then, $AS(P) = QAS(\Pi)$.

3 Complexity issues

We now study the computational properties of the ASP(Q) language. As it is customary in the literature we focus on the ground case, that is we assume that no variable occurs in programs.

Because it is possible to alternate universal and existential answer set quantifiers, it is clear that ASP(Q) can model probelms beyond NP. In particular, each problem in PSPACE can be modeled by using an ASP(Q) program. Formally, we define the COHERENCE problem as follows: Given an ASP(Q) program Π as input, decide whether Π is coherent.

[3] The proof of this result and of some other theorems are given in the appendix available as supplemental materials published with the paper.

Theorem 2
The COHERENCE problem is PSPACE-complete, even under the restriction to normal ASP(Q) programs.

As for QBFs, there is a direct correspondence between the number of alternating quantifiers and the level of the Polynomial Hierarchy (PH) for which we have competeness of the coherence problem.

Theorem 3
The COHERENCE problem is (i) Σ_n^P-complete for normal existential ASP(Q) programs with n quantifiers in the prefix; and (ii) Π_n^P-complete for normal universal ASP(Q) programs with n quantifiers in the prefix.

We note that, for classes of disjunctive programs that can be translated in polynomial time to normal ones, such as Head-Cycle Free (HCF) (Ben-Eliyahu and Dechter 1996), the correspondence between quantifier alternations and the level of the Polynomial Hierarchy is preserved.

We also note that the theorem concerns, in each of the two cases, the corresponding class of *all* ASP(Q) programs with n quantifiers. In particular, the membership part is proved for that class. The proof of hardness explicitly usues special programs in that class, the ones in which quantifiers *alternate*.

4 Modeling in ASP(Q)

In this section, we focus on the modeling capabilities of our language. Thus, we study some well-known problems that are computationally beyond NP, and show how to solve them in ASP(Q).

4.1 Minmax Clique

Minmax problems play a key role in various fields of research, including game theory, combinatorial optimization and computational complexity (Cao et al. 1995). A minimax problem can be formulated as $min_{x\in X} max_{y\in Y} f(x,y)$, where $f(x,y)$ is a function defined on the product set of X and Y. Here, we focus on the so-called *Minmax Clique* problem (Ko 1995), but our approach can be easily adapted to model other minmax problems.

Let $G = \langle N, E\rangle$ be a graph, I and J two finite sets of indices, and $(A_{i,j})_{i\in I, j\in J}$ a partition of N. We write J^I for the set of all total functions from I to J. For every total function $f: I \to J$ we denote by G_f the subgraph of G induced by $\bigcup_{i\in I} A_{i,f(i)}$. We define the MINMAX CLIQUE problem as follows: Given a graph G, sets of indices I and J, a partition $(A_{i,j})_{i\in I, j\in J}$ (all as above), and an integer k, decide whether

$$\min_{f\in J^I} \max\{|Q| : Q \text{ is a clique of } G_f\} \geq k.$$

It is known that this problem is Π_2^p-complete (Ko 1995).

Consider the following ASP(Q) program $\Pi = \forall^{st} P_1 \exists^{st} P_2 : C$. The ASP program P_1 is given by:

$$P_1 = \left\{ \begin{array}{rcll} edge(a,b) & & & \forall (a,b) \in E \\ node(a) & & & \forall a \in N \\ v(i,j,a) & & & \forall i \in I,\ j \in J,\ a \in A_{i,j} \\ setI(X) & \leftarrow & v(X,\_,\_) & \\ setJ(X) & \leftarrow & v(\_,X,\_) & \\ 1\{f(X,Y) : setJ(Y)\}1 & \leftarrow & setI(X) & \end{array} \right\}$$

Informally, the role of P_1 is to specify the input graph, the sets I and J of indices, a partition $(A_{i,j})$, and the search space of all total functions from I to J. Specifically, the first two sets of facts encode the graph by using two predicates: a binary one named *edge*, collecting all edges of the graph; and a unary one named *node* collecting all nodes of the graph. Then, the third set of facts encodes the partition $(A_{i,j})$ by using a ternary predicate v. Projections applied to v (rules four and five) define elements of the sets I and J, respectively. Finally, the last rule defines the space of all total functions f from I to J. The ASP program P_2 is defined as follows:

$$P_2 = \left\{ \begin{array}{rcl} inInduced(Z) & \leftarrow & v(X,Y,Z),\ f(X,Y) \\ edgeP(X,Y) & \leftarrow & edge(X,Y),\ inInduced(X),\ inInduced(Y) \\ \{inClique(X) : inInduced(X)\} & & \\ & \leftarrow & inClique(X),\ inClique(Y),\ not\ edgeP(X,Y) \end{array} \right\}$$

Its role is to define the subgraph G_f of G determined by a total function f, and to select a clique in this subgraph. In particular, the first rule defines the set of nodes of the subgraph G_f (whenever a node Z belongs to the set $A_{X,Y}$, and the function f maps X to Y, then Z is a node of G_f). The second rule ensures that whenever there is an edge from X to Y, and both X and Y are nodes of G_f, then the edge (X,Y) is an edge of G_f (G_f is the *induced* subgraph). The third rule allows to select nodes of the partition as candidates for a clique. The final constraint requires that it is not possible that two nodes X and Y are in a clique and there is no edge in the subgraph G_f from X to Y. Finally, the program C is defined as follows.

$$C = \left\{ \ \leftarrow \quad \#\text{count}\{X : inClique(X)\} < k \ \right\}$$

The constraint forces the number of nodes in a clique to be greater or equal to k.

Intuitively, we check if for each answer set of P_1, that is for each total function f from I to J, there exists an answer set of P_2, that is a clique in the subgraph of G induced by f, such that its cardinality is not less than k. If so, a quantified answer set of Π exists.

Theorem 4
Let $\mathscr{I} = \langle G, (A_{i,j})_{i\in I, j\in J}, k\rangle$ be an instance of the MINMAX CLIQUE problem. Then,

$$\min_{f\in J^I} \max\{|Q| : Q \text{ is a clique of } G_f\} \geq k$$

if and only if the ASP(Q) program Π, defined as above, has a quantified answer set.

4.2 Pebbling Number

Graph pebbling is a well-known mathematical game (Hurlbert 1999). It was first suggested as a tool for solving a particular problem in number theory (Chung 1989). The game consists of a graph with pebbles placed on (some of) its nodes. The goal is to place a pebble on a *target* node by performing a sequence of *pebbling* moves. More formally, let $G = \langle N, E\rangle$ be a directed graph whose nodes may contain pebbles. A *pebbling move* along an edge $(a,b) \in E$ requires that node a contains at least two pebbles; the move removes two pebbles from a and adds one pebble to b. The *pebbling number*, denoted by $\pi(G)$, is the smallest number of pebbles such that for every assignment of k pebbles to nodes of G and for every node $w \in N$ (the target), some sequence (possibly empty) of pebbling moves results in a pebble on w. The PEBBLING NUMBER problem asks whether $\pi(G)$ is less than or equal to k. This problem is Π_2^p-complete, and it remains so also when the target node is part of the input (Milans and Clark 2006). (For the latter version,we redefine $\pi(G)$ accordingly.)

To capture the definition of the PEBBLING NUMBER problem we construct an ASP(Q) program $\Pi = \forall^{st} P_1 \exists^{st} P_2 : C$. Its program P_1 is defined as follows:

$$P_1 = \left\{ \begin{array}{rcl} edge(a,b) & & \forall (a,b) \in E \\ node(a) & & \forall a \in N \\ pebble(i) & & \forall i = 0,1,\ldots,k \\ 1\{onNode(X,N) : pebble(N)\}1 & \leftarrow & node(X) \\ & \leftarrow & \#\mathrm{sum}\{N,X : onNode(X,N)\} \neq k \\ 1\{target(X) : node(X)\}1 & & \end{array} \right\}$$

The first two sets of facts encode the input graph, and the third one the set of integers that can serve as the number of pebbles a node can have. The first rule of the program (line 4) selects, for each node X, the number N of pebbles on X. The second rule (line 5) ensures the total number of pebbles on all nodes of G is k. The last rule selects exactly one node as the target allowing any node to be selected. Thus, answer sets of P_1 capture all possible "input configurations" for G, each configuration defined by a distribution of k pebbles among nodes of G and the target node.

The ASP program P_2 in Π is defined as follows:

$$P_2 = \left\{ \begin{array}{rcl} step(i) & & \forall i = 0,1,\ldots,k-1 \\ 1\{endstep(S) : step(S)\}1 & & \\ onNode(X,N,0) & \leftarrow & onNode(X,N) \\ 1\{move(X,Y,S) : edge(X,Y)\}1 & \leftarrow & step(S), endstep(T), 1 \leq S,\ S \leq T \\ & \leftarrow & move(X,Y,S),\ onNode(X,N,S),\ N < 2 \\ affected(X,S) & \leftarrow & move(X,Y,S) \\ affected(Y,S) & \leftarrow & move(X,Y,S) \\ onNode(X,N-2,S) & \leftarrow & onNode(X,N,S-1), move(X,Y,S) \\ onNode(Y,M+1,S) & \leftarrow & onNode(Y,M,S-1), move(X,Y,S) \\ onNode(X,N,S) & \leftarrow & onNode(X,N,S-1), not\ affected(X,S) \end{array} \right\}$$

The first set of facts (line 1) encodes all integers i that can serve as the number of pebbling moves. Since each pebbling move removes one pebble, any successful sequence of pebbling moves has length at most $k-1$. Consequently, we may (and do) restrict these integers to $0,1,\ldots,k-1$. The first rule of P_2 (line 2) selects a single integer to represent the number of pebbling moves. The second rule of P_2 (the next line) defines the initial state of the graph (before any pebbling moves). It is given by the initial distribution of pebbles obtained from an answer set of the program P_1 (we overload the notation here; the predicate *onNode* defining the initial configuration in P_1 is binary, while the predicate *onNode* defined in P_2 is ternary; it has an additional argument to represent the step). The third rule selects an edge for the pebbling move step $S = 1,2,\ldots,T$, where T is the end step (defined via *endstep*). The constraint that follows imposes the pebbling move precondition: there must be at least two pebbles on the node where the pebbling move originates. The next two rules define the two nodes affected by the move. The last three rules define the state of the graph after the pebbling move in step S (applied to the graph after $S-1$ pebbling moves). The first two of these three rules describe how the number of pebbles change on the nodes that are involved in the move. The last rule is the inertia rule that keeps the number of pebbles unchanged on all nodes unaffected by the move. Informally, answer sets of P_2 correspond to all valid sequences of pebbling moves that do not eliminate all pebbles and start in the initial state of the graph, together with the corresponding sequence of states of the graph.

Finally, the program C in Π is defined as follows.

$$C = \left\{ \begin{array}{lcl} ok(W) & \leftarrow & onNode(W,N,S),\ target(W), endstep(T)\ N > 0 \\ & \leftarrow & target(W),\ not\ ok(W) \end{array} \right\}$$

First rule defines $ok(W)$ to hold whenever W is a target node and there is a pebble on it after the last pebbling move T. The constraint ensures no answer set if $ok(W)$ has not been inferred.

Intuitively then, Π is coherent precisely when for each assignment of k pebbles to nodes of a given graph and for every choice of a target node (that is, for every answer set M_1 of P_1) there is a sequence of pebbling moves of length at most $k-1$ (that is, there is an answer set M_2 for $P_2 \cup fix_{P_1}(M_1) = P_1'$) such that the target node has a pebble on it (that is, $C \cup fix_{P_1'}(M_2)$ has an answer set).

Theorem 5
Let $\mathscr{I} = \langle G,k \rangle$ be an instance of the Pebbling Number Problem. Then, $\pi(G) \leq k$ if and only if the ASP(Q) program Π, defined as above, is coherent.

4.3 Vapnik-Chervonenkis Dimension

The *Vapnik-Chervonenkis dimension* (VC dimension) is a fundamental concept in machine learning theory (Vapnik and Chervonenkis 2015). The VC dimension is a measure of the capacity of a space of functions that can be learned by a statistical classification algorithm (Blumer et al. 1989). In particular, it is the cardinality of the largest set of points that the algorithm can shatter. In statistical learning theory, the VC dimension can predict probabilistic upper bounds on the test error of a classification model (Vapnik 1998). Further applications include finite automata, complexity theory, computability theory, and computational geometry.

Here, we focus on the so-called *discrete* VC dimension problem, where the considered universe is finite. The problem concerns families of subsets that are represented by Boolean circuits. However, we assume that the representation is given by a logic program capturing the corresponding formula. Specifically, we assume that a program $P_{\mathscr{C}}$ representing a family $\mathscr{C}$ of subsets of U contains a unary predicate $true$, and that extensions of the predicate $true$ in answer sets of $P_{\mathscr{C}}$ are precisely the elements of $\mathscr{C}$. Constructing a program $P_{\mathscr{C}}$ from a Boolean circuit representing $\mathscr{C}$ is a matter of routine and can be accomplished in linear time. Let k be an integer, U a finite set, and $\mathscr{C} = \{S_1, \ldots, S_n\} \subseteq 2^U$ a collection of subsets of U represented by a program $P_{\mathscr{C}}$. The VC DIMENSION problem asks whether there is a subset X of U of size at least k, such that for each subset S of X, there exists S_i such that $S = S_i \cap X$. The VC dimension of $\mathscr{C}$ is defined as maximum size of such a set X and is denoted by $VC(\mathscr{C})$. Hence, the VC DIMENSION problem asks whether $VC(\mathscr{C}) \geq k$. It is known that this problem (assuming a circuit or a program representation of $\mathscr{C}$) is Σ_3^p-complete (Schaefer 1999). We will show that the problem can be described by an ASP(Q) program $\Pi = \exists^{st} P_1 \forall^{st} P_2 \exists^{st} P_3 : C$. The ASP program P_1 is defined as follows:

$$P_1 = \left\{ \begin{array}{rl} inU(x) & \forall x \in U \\ k\{inX(X) : inU(X)\} & \end{array} \right\}$$

The set of facts in line 1 encodes the elements of the set U, while the choice rule in line 2 selects a subset X of U with at least k elements. It is clear that answer sets of P_1 are all subsets of U with at least k elements.

The ASP program P_2 consists of a single choice rule:

$$P_2 = \left\{ \ \{inS(X) : inX(X)\} \ \right\}$$

Thus, answer sets of P_2 are subsets of a set X (determined by a selected answer set of P_1).

For P_3 we simply take $P_{\mathscr{C}}$. Wlog, we may assume that $P_{\mathscr{C}}$ shares no vocabulary elements with P_1 and P_2. Thus, for every possible "input" from P_1 and P_2, answer sets of P_3', that is, P_3 extended with the input from P_1 and P_2, determine elements of $\mathscr{C}$ via extensions of the predicate *true*.

Finally, the program C is defined as follows (understanding *true* as defined above):

$$C = \left\{ \begin{array}{lcl} inIntersection(X) & \leftarrow & true(X),\ inX(X) \\ & \leftarrow & inIntersection(X),\ not\ inS(X) \\ & \leftarrow & not\ inIntersection(X),\ inS(X) \end{array} \right\}$$

The first rule collects into predicate *inIntersection*, the intersection of the selected set S_i from $\mathscr{C}$ (represented by an answer set of P_3' by means of the predicate *true*) and X, a subset of U selected via an answer set of P_1. The two constraints force this intersection to coincide with the subset S of X (an answer set of P_2 extended with a selected answer set of P_1 as input representing X).

Intuitively, the program Π is coherent when there exists an answer set M_1 of P_1 (that is, a subset X of U of size at least k) such that for each answer set M_2 of $P_2' = P_2 \cup fix_{P_1}(M_1)$ (that is, for each subset S of X), there exists an answer set M_3 of $P_3' = P_3 \cup fix_{P_2'}(M_2)$ (that is, an element S_i of $\mathscr{C}$), such that $C \cup fix_{P_3'}(M_3)$ is coherent (that is, $S_i \cap X$ is equal to S).

Theorem 6
Let $\mathscr{I} = \langle U, \mathscr{C}, k \rangle$ be an instance of the VC dimension problem. Then, $VC(\mathscr{C}) \geq k$ if and only if the ASP(Q) program Π defined as above has a quantified answer set.

5 Related Work and Discussion

We now compare ASP(Q) with related work discussing pros and cons of the various approaches.

ASP(Q) vs QBF. We first compare our proposal with Quantified Boolean Formulas (QBF) (Biere et al. 2009). QBF is a natural extension of propositional formulas with quantifiers $\exists$ (existential) and $\forall$ (universal) operating on propositional variables. QFB was motivated by questions arising from computational complexity (Stockmeyer and Meyer 1973). The problem of checking the satisfiability of a propositional formula (SAT) is the canonical problem for the complexity class NP. The addition of quantifiers increases the complexity of satisfiability problem (QSAT) to PSPACE (Stockmeyer 1976), and prefixes of k alternating quantifiers yield problems that are complete for each complexity class of the Polynomial Hierarchy. For this reason the satisfiability problem of QBF formulas with prefixes of alternating k quantifiers (k-*QSAT* becomes the canonical problem for the k-th level of the Polynomial Hierarchy). More precisely, k-*QSAT* restricted to prefixes of length k starting with an existential (resp. universal) quantifier is complete for Σ_k^P (resp. Π_k^P). ASP(Q) and QBF share the same motivation and intuition, indeed ASP(Q) extends ASP with quantifiers (as QBF extends SAT) to increase the modeling capabilities of the language beyond NP. As studied in Section 3, propositional ASP(Q) and QBF have similar computational properties. In particular, the coherence problem for both is PSPACE-complete and an even tighter correspondence holds between propositional normal ASP(Q) and QSAT. Nonetheless, there are important differences among the two languages, some inherited form the relation between SAT and ASP, and other concerning the semantics of quantifiers.

First, ASP(Q) supports variables, which gives a modeling advantage, and supports rapid prototyping, program optimization and maintenance of problem solution. Indeed, variables allow one to encode uniform compact representation of a problem over varying instances, while in QBF (as

in SAT) each instance of a problem needs to be encoded in a specific formula by means of an encoding procedure. Second, even if in general QBF and ASP(Q) can solve the same computational problems, ASP(Q) inherits from ASP the possibility of encoding *inductive definitions* (Denecker and Vennekens 2014), which are useful in modeling properties such as reachability in graphs (inductive definitions require larger instances in SAT and QBF that slow down modeling and solving). Next, ASP supports modeling extensions such as aggregates, choice rules, strong negation, and disjunction in rule heads that significantly simplify encodings used in SAT (Brewka et al. 2011). We have made extensive use of inductive definitions and aggregates in our examples in Section 4. Finally, we note that in QBF quantifiers range over variable assignments, whereas in ASP(Q) they quantify over the answer sets of each subprogram. This is yet another difference and a reason that ASP(Q) cannot be seen as a straightforward porting of the ideas behind QBF.

ASP(Q) vs ASP. One of the distinguishing features of ASP is the capability of modeling problems in Σ_2^P. This is possible because of the additional expressive power provided by disjunctive rules. Modeling in Σ_2^P problems with ASP is rather natural if one can use only *positive* rules. For example, let us consider the *strategic companies* problem (Cadoli et al. 1997). In that problem, one has to compute a set of companies that cover the production of a set of goods also controlling other companies. A set of companies S is said to be strategic if it: (i) covers the productions of all goods; (ii) is subset-minimal; and, (iii) every company c controlled by at most three strategic companies is also strategic. In the setting in which each product is produced by at most two companies the problem is Σ_2^P-complete and can be modeled as follows (Leone et al. 2006):

$$\begin{aligned} strat(Y) \vee strat(X) &\leftarrow prod\_by(P,X,Y) \\ strat(W) &\leftarrow contr\_by(W,X,Y,Z), strat(X), strat(Y), strat(Z) \end{aligned}$$

The first rule models condition (i), the second rule models condition (iii), and the minimality of answer sets ensures (ii). It is clear that this encoding of the problem can be directly translated to a single-quantifier disjunctive ASP(Q).

When problem constraints to be modeled involve negation, ASP modeling becomes less intuitive. In particular one has to resort to an encoding technique called *saturation* (Eiter and Gottlob 1995). It allows one to simulate a co-NP check in the program reduct. Saturation is at the basis of the celebrated encoding of 2-QBF by Eiter and Gottlob (1995) used to prove the complexity of checking existence of answer sets in presence of disjunction in rule heads. Given a 2-QBF formula $\Phi = \exists X \forall Y G$, where $G = D_1 \vee \ldots \vee D_h$ is a DNF, and $D_i = L_{i,1} \wedge \ldots \wedge L_{i,k_i}$ and $L_{i,j}$ are literals over $X \cup Y$, we encode Φ in an ASP program as follows. First introduce a fresh atom *sat* modeling satisfiability, and a fresh atom nz for every atom $z \in X \cup Y$; and set $\sigma(z) = z$ and $\sigma(\neg z) = nz$ for every $z \in X \cup Y$. Then write the program $P_\Phi = \{z \vee nz | \forall z \in X \cup Y\} \cup \{y \leftarrow sat | \forall y \in Y\} \cup \{ny \leftarrow sat | \forall y \in Y\} \cup \{sat \leftarrow \sigma(L_{i,1}), \ldots, \sigma(L_{i,k_i}) | i = 1, \ldots, m\} \cup \{sat \leftarrow not\ sat\}$.

Here the atoms corresponding to universally quantified variables Y are "saturated" (i.e., they are forced to be true in any answer set), and since the last rule is always removed while computing the reduct, *sat* must be derived for all assignments of truth values to Y to have an answer set. This trick ensures that Φ is satisfiable if and only if P_Φ has an answers set. Again, one could reformulate the program above into a *disjunctive* program with a single quantifier. However, using saturation in modeling is considered difficult. ASP(Q) offers an alternative and more intuitive approach, It uses *normal* quantified programs with *two* quantifiers that also capture Σ_2^P (see Theorem 3). Indeed, let us consider a normal quantified program $\Pi_\Phi = \exists^{st} P_1 \forall^{st} P_2 : C$ where

$$P_1 = \{\{x_1, \ldots, x_n\}\}, \quad P_2 = \{\{y_1, \ldots, y_m\}\},$$

$$C = \{sat \leftarrow \sigma(L_{i,1}), \ldots, \sigma(L_{i,k_i}) \mid \forall i = 1, \ldots, m\} \cup \{\leftarrow not\ sat\}.$$

Here, a satisfiability of an existential 2-QBF is encoded directly. Indeed P_1 guesses an assignment to X s.t. for all assignments to Y generated by P_2, *sat* must be derived by satisfying at least one conjunct in φ, i.e., Π_Φ is satisfiable iff Φ is. This discussion suggests that ASP(Q) improves on ASP modeling capabilities. It keeps the advantages of ASP in modeling concisely Σ_2^P problems with positive programs, as for strategic companies, but also allows us to model other problems without resorting to difficult to use encoding techniques.

ASP(Q) vs Stable-Unstable. To handle problems beyond NP, Bogaerts et al. (2016) proposed an extension of ASP inspired by an internal working principle of ASP solvers (Gebser et al. 2018). Usually, in ASP solvers designed for problems in Σ_2^P one procedure generates model candidates and another one, acting as an oracle, tests minimality of the candidates produced by the first procedure. It does so by verifying that a certain subprogram (in some cases, a SAT formula) has no stable models (is not satisfiable). Following this principle, Bogaerts et al. (2016) introduced *combined logic programs*, in which two normal logic programs play a role analogous to the one of the two procedures of ASP solvers mentioned above. A combined logic program is a pair $\Pi = (P_g, P_t)$ of normal logic programs. Its semantics is given by parameterized stable models (Oikarinen and Janhunen 2006; Denecker et al. 2012); a *stable-unstable model* of a combined program Π is a parameterized stable model of P_g, say I, such that no parameterized stable model of P_t exists that coincides with I in the intersection of the signatures of the two programs.

Comparing ASP(Q) programs with combined programs, we first note that combined programs involve the concept of parameters. In applications, the parameters of the generator program are used to represent problem instances (are "extensional"). This use of parameters is quite natural to ASP programmers and does not pose a conceptual difficulty. It is also used implicitly in ASP(Q) (stable models from each quantifier are passed on as "input" parameters to the next one).[4] However, the stable-unstable approach applies the notion of a parameterized stable model also in the checking phase using "negation," that is, referring to non-existence of a certain parameterized stable model. This, arguably, makes the formalism much less direct than ASP(Q). It is especially clear when we move beyond the second level of the PH and the non-existence conditions become nested (incidentally, the stable-unstable paper contains no examples of modeling such problems).

If we factor out the issue of parameters, and limit ourselves to problems in Σ_2^P, combined programs and ASP(Q) are closely related. Indeed, in ASP(Q) one has direct means to model "testing" conditions of the form "for all stable models (answer sets) of some program, a certain property holds." In contrast, combined programs provide direct means to model "testing" conditions of the form "there exists *no* stable model of some program such that a certain property holds." Switching between ASP(Q) and combined programs amounts then to simulating conditions of one form with conditions of the other and *vice versa* (effectively, negating constraints in a program). Such simulations are easy to design with the use of a small number of auxiliary variables (often one such new variable suffices). Consequently, both formalisms are on par for modeling problems that are complete for Σ_2^P. However, for problem in Π_2^P, the difference between ASP(Q) and combined programs becomes evident. As an example, let us consider a 2-QBF formula $\Psi = \forall X \exists Y \psi$, where ψ is a 3-CNF formula. This problem can be naturally represented in ASP(Q) by using the encoding employed in the proof of Theorem 2. However once we try to

[4] We could also distinguish extensional predicates to specify "parameters," that is, input instances, That would allow us to keep instance specification separate from the program. We decided not to do so here to simplify our presentation.

encode it using a combined logic program (for well-known complexity reasons) we have either to adopt an exponential encoding, something analogous to quantifier expansion in QBF, or we have to use an additional nesting of programs (i.e., we are have to push the entire computation in the oracle). In both cases, the modeling would not result in a solution as natural and direct as the one provided by ASP(Q). The reason is that combined programs (as well as their generalizations beyond the second level) represent existential statements. Hence, they model *complements* of Π_2^P problems and not the problems themselves. In contrast, ASP(Q) can be used for such problems in a direct way providing representations closely following original problem descriptions (our examples illustrate this).

A related aspect concerns modeling itself, the process of mapping natural language specifications to formal expressions, which surfaces when one considers problems that require more than one quantifier alternation. It is important to note that combined logic programs were extended to deal with problems from any level of the PH in (Bogaerts et al. 2016) by resorting to a recursive definition. This definition forces the programmer to think in terms of "nested oracles", instead of translating problem description directly into a formal expression. Whereas for problems at the second level of the polynomial hierarchy it roughly corresponds to searching for a counterexample, for problems at higher levels, the recursion and the negation (needed because of the absence of direct means to represent universal statements), makes it harder to maintain the connection between problem description and oracles forming nested combined programs. In contrast, the interface between natural language problem description and ASP(Q) programs is transparent (in the same way as it is for QBF), as it is explicitly supported by the quantifiers, which may be existential or universal, as needed. In particular, the difficulty of modeling problems in Π_2^P, noted above, appears in the general setting of problems in Π_k^P, for $k \geq 2$: the stable-unstable formalism is not designed to directly express universal statements that characterize problems in Π_k^P.

The discussion above compares at an intuitive informal level the modeling features of the two formalism. It also suggests how the two are formally related. In the statement specifying the relation, the *depth* of the basic combined program is defined as 2. Each next level of nesting increments the depth by 1.

Theorem 7
(i) There is a polynomial-time reduction that assigns to every propositional nested combined program Π of depth n, a normal existential ASP(Q) program Π_q with $n \geq 2$ quantifiers such that answer sets of Π and Π_q, correspond to each other.
(ii) There is a polynomial-time reduction that assigns to every propositional normal existential ASP(Q) program Π with $n \geq 2$ quantifiers in the prefix, a propositional nested combined program Π_c of depth n such that answer sets of Π and Π_c correspond to each other.

Thus, at the level of expressive power, combined programs of depth n and existential $ASP(Q)$ programs with n quantifiers are formally equivalent, even if from the modeling point of view, as we argued, ASP(Q) programs seem to have an advantage. However, unless the polynomial hierarchy collapses, no reduction from universal $ASP(Q)$ programs with n quantifiers to combined nested programs of depth n is possible. The following proposition specifies this property for the particular case of the validity of 2-QBFs, which we discussed above.

Proposition 1
Unless the polynomial hierarchy collapses, there exists no polynomial reduction that encodes formulas $\Psi = \forall X \exists Y \psi$, where ψ is a 3-CNF formula, as a combined program $P = (P_1, P_2)$, where P_1 and P_2 are normal logic programs, such that Ψ is valid iff P admits stable unstable models.

A trivial consequence of Theorem 2 is that this limitation is absent from ASP(Q).

Finally, we note that combined programs under stable-unstable semantics have been implemented in a proof of concept prototype (Bogaerts et al. 2016) that can only handle problems at the second level of the polynomial hierarchy. A similar prototype implementation for ASP(Q) (programs with at most two quantifiers) is possible, too. However, devising efficient implementations for either formalism in their full generality remains a non-trivial open research problem.

Further related work. The problem of modeling in a natural way Σ_2^P problems with ASP was also addressed by Eiter and Polleres (2006). They model problems combining "guess" program P_{solve} and "check" program P_{check}, which are transformed into a single disjunctive ASP program such that its answer sets encode the solutions of the original problem by means of a polynomial-time transformation. The programs P_{solve} and P_{check} must be HCF and propositional, thus limiting this approach to the modeling capabilities of propositional ASP. An idea analogous to that developed by Eiter and Polleres (2006) was also proposed by Redl (2017). Redl's proposal appears to be conceptually simpler than the earlier one because of the use of conditional literals but suffers from the same limitations. A general technique to reuse existing ASP systems to evaluate problems of higher complexity (such as various forms of qualitative preferences among answer sets) was proposed by Gebser et al. (2011). The idea there was to use a meta program encoding the saturation technique which, in this way, became transparent to the user. As in the approach by Eiter and Polleres (2006), the resulting program is a plain ASP program which can be evaluated by a standard ASP system. Thus, the approach of Gebser et al. (2011) cannot be used to model problems beyond the second level of the polynomial hierarchy. Another solution that allows for reasoning within a program over the answer sets of another program, and thus encode reasoning tasks beyond NP, is provided by manifold programs (Faber and Woltran 2011; Faber and Woltran 2009). In manifold programs the calling and the called program are encoded into a single program using weak constrains. The answer sets of the called program are thus represented within each answer set of the calling program. Also this approach is limited to the second level of the polynomial hierarchy, and might generate large specifications.

HEX-programs are an extension of ASP with external sources such as description logic ontologies and Web resources (Eiter et al. 2008). In HEX-programs external atoms can exchange information from the logic program to eternal theories in terms of predicate extensions and constants. Redl (2017) studied a way to avoid saturation for modeling Σ_2^P problems with HEX-programs. In particular, the author proposes the modeling technique of query answering over subprograms. While encoding a problem on the second level of the polynomial hierarchy, one has to provide two components. A first program P_{guess} modeling the NP part, and a second one P_{check} modeling the co-NP check. The first program, P_{guess}, is a HEX program that can query on the answer sets of the normal ordinary ASP program P_{check} using specific external atoms. This modeling approach avoids saturation without introducing quantifiers, but this nice modeling behavior is limited to Σ_2^P problems. Indeed, the focus of query answering over subprograms is on overcoming saturation and not on reaching high expressibility (Redl 2017). A recent proposal of an extension of propositional ASP to model planning problems was described in (Romero et al. 2017; Amendola 2018). The main difference with ASP(Q) is on the nature of quantifiers allowed in the two specifications. Indeed, the proposal of (Romero et al. 2017), mimicking 2QBF, allows quantifiers over propositional atoms, whereas in ASP(Q) quantifiers are over answer sets.

As a final mention, we observe that the idea of extending the base language with quantifiers

has been applied also in the neighboring area of Constraint Satisfaction Problems (CSP) (Rossi et al. 2006), obtaining Quantified CSP (QCSP) (Bordeaux and Monfroy 2002).

6 Conclusions

In this paper we approached the modeling of problems beyond NP with ASP programs. Inspired by the way QBFs extend SAT formulas, we have introduced ASP(Q), which extends ASP via quantifiers over stable models of programs. We have studied the computational properties of the language, provided a number of examples to demonstrate its modeling capabilities, and compared alternative approaches to the same problem. The analysis provided in the paper suggests that ASP(Q) is able to model uniformly problems in the Polynomial Hierarchy in the same compact and elegant way as ASP models problems in NP.

The definition of ASP(Q) allows for disjunctive programs, thus all the features of the basic language are retained. However, by limiting to normal (or HCF) programs (extended with aggregates and other useful modeling constructs) in ASP(Q), one can take advantage of the classic generate-define-test modular programming methodology and other modeling techniques developed for these best understood classes of programs to model any problem in the Polynomial Hierarchy. Indeed, the presence of quantifiers allows one to model complex properties in a direct way, without the need of recasting them in terms of checking the minimality of a model, e.g., using saturation. The examples provided in the paper, indeed, employ normal programs, and the solutions follow directly from the definition in natural language of the problem at hand.

The key task for the future is to implement ASP(Q). In this respect many possible solutions are possible, from encoding ASP(Q) in QBF and resorting to QBF solvers, to evolving ASP solvers to handle quantifiers over stable models.

Acknowledgements

The work of the third author has been partially supported by the NSF grant IIS-1707371. This work has been partially supported by MIUR under PRIN 2017 project n. 2017*M*9*C*25*L*_001 (CUP *H*24*I*17000080001).

References

ALVIANO, M., AMENDOLA, G., DODARO, C., LEONE, N., MARATEA, M., AND RICCA, F. 2019. Evaluation of disjunctive programs in WASP. In *LPNMR*. Lecture Notes in Computer Science, vol. 11481. Springer, 241–255.

ALVIANO, M., DODARO, C., LEONE, N., AND RICCA, F. 2015. Advances in WASP. In *LPNMR*. LNCS, vol. 9345. Springer, 40–54.

AMENDOLA, G. 2018. Towards quantified answer set programming. In *RCRA@FLoC*. CEUR Workshop Proceedings, vol. 2271. CEUR-WS.org.

BEN-ELIYAHU, R. AND DECHTER, R. 1996. On computing minimal models. *Ann. Math. Artif. Intell. 18,* 1, 3–27.

BIERE, A., HEULE, M., VAN MAAREN, H., AND WALSH, T., Eds. 2009. *Handbook of Satisfiability*. Frontiers in Artificial Intelligence and Applications, vol. 185. IOS Press.

BLUMER, A., EHRENFEUCHT, A., HAUSSLER, D., AND WARMUTH, M. K. 1989. Learnability and the Vapnik-Chervonenkis dimension. *J. ACM 36,* 4, 929–965.

BOGAERTS, B., JANHUNEN, T., AND TASHARROFI, S. 2016. Stable-unstable semantics: Beyond NP with normal logic programs. *TPLP 16,* 5-6, 570–586.

BORDEAUX, L. AND MONFROY, E. 2002. Beyond NP: arc-consistency for quantified constraints. In *CP*. LNCS, vol. 2470. Springer, 371–386.

BREWKA, G., EITER, T., AND TRUSZCZYNSKI, M. 2011. Answer set programming at a glance. *Commun. ACM 54,* 12, 92–103.

BUCCAFURRI, F., LEONE, N., AND RULLO, P. 2000. Enhancing disjunctive datalog by constraints. *IEEE Trans. Knowl. Data Eng. 12,* 5, 845–860.

CADOLI, M., EITER, T., AND GOTTLOB, G. 1997. Default logic as a query language. *IEEE Trans. Knowl. Data Eng. 9,* 3, 448–463.

CAO, F., DU, D.-Z., GAO, B., WAN, P.-J., AND PARDALOS, P. M. 1995. *Minimax Problems in Combinatorial Optimization*. Springer US, Boston, MA, 269–292.

CHUNG, F. R. 1989. Pebbling in hypercubes. *SIAM J. Discret. Math. 2,* 4 (Nov.), 467–472.

DANTSIN, E., EITER, T., GOTTLOB, G., AND VORONKOV, A. 2001. Complexity and expressive power of logic programming. *ACM Comput. Surv. 33,* 3, 374–425.

DENECKER, M., LIERLER, Y., TRUSZCZYNSKI, M., AND VENNEKENS, J. 2012. A Tarskian informal semantics for answer set programming. In *ICLP-TC*. LIPIcs, vol. 17. 277–289.

DENECKER, M. AND VENNEKENS, J. 2014. The well-founded semantics is the principle of inductive definition, revisited. In *KR*. AAAI Press.

DODARO, C., GASTEIGER, P., LEONE, N., MUSITSCH, B., RICCA, F., AND SCHEKOTIHIN, K. 2016. Combining answer set programming and domain heuristics for solving hard industrial problems (application paper). *TPLP 16,* 5-6, 653–669.

EITER, T., FABER, W., LEONE, N., AND PFEIFER, G. 2000. Declarative problem-solving using the dlv system. In *Logic-based Artificial Intelligence*. 79–103.

EITER, T. AND GOTTLOB, G. 1995. On the computational cost of disjunctive logic programming: Propositional case. *Ann. Math. Artif. Intell. 15,* 3-4, 289–323.

EITER, T., IANNI, G., LUKASIEWICZ, T., SCHINDLAUER, R., AND TOMPITS, H. 2008. Combining answer set programming with description logics for the semantic web. *Artif. Intell. 172,* 12-13, 1495–1539.

EITER, T. AND POLLERES, A. 2006. Towards automated integration of guess and check programs in answer set programming: a meta-interpreter and applications. *TPLP 6,* 1-2, 23–60.

ERDEM, E., GELFOND, M., AND LEONE, N. 2016. Applications of answer set programming. *AI Magazine 37,* 3, 53–68.

FABER, W. AND WOLTRAN, S. 2009. A framework for programming with module consequences. In *SEA*. CEUR Workshop Proceedings, vol. 546. CEUR-WS.org, 34–48.

FABER, W. AND WOLTRAN, S. 2011. Manifold answer-set programs and their applications. In *Logic Programming, Knowledge Representation, and Nonmonotonic Reasoning*. LNCS, vol. 6565. 44–63.

GEBSER, M., KAMINSKI, R., KAUFMANN, B., ROMERO, J., AND SCHAUB, T. 2015. Progress in clasp series 3. In *LPNMR*. LNCS, vol. 9345. Springer, 368–383.

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*. ijcai.org, 5450–5456.

GEBSER, M., MARATEA, M., AND RICCA, F. 2017. The sixth answer set programming competition. *J. Artif. Intell. Res. 60*, 41–95.

GEBSER, M., OBERMEIER, P., SCHAUB, T., RATSCH-HEITMANN, M., AND RUNGE, M. 2018. Routing driverless transport vehicles in car assembly with answer set programming. *TPLP 18,* 3-4, 520–534.

GEBSER, M. AND SCHAUB, T. 2016. Modeling and language extensions. *AI Magazine 37,* 3, 33–44.

GELFOND, M. AND LIFSCHITZ, V. 1991. Classical negation in logic programs and disjunctive databases. *New Generation Comput. 9,* 3/4, 365–386.

HURLBERT, G. 1999. A Survey of Graph Pebbling. *Congr. Num. 139,* math.CO/0406024, 41–64.

KO, KER-IAND LIN, C.-L. 1995. *On the Complexity of Min-Max Optimization Problems and their Approximation*. Springer US, Boston, MA, 219–239.

LEONE, N., PFEIFER, G., FABER, W., EITER, T., GOTTLOB, G., PERRI, S., AND SCARCELLO, F. 2006. The DLV system for knowledge representation and reasoning. *ACM Trans. Comput. Log. 7,* 3, 499–562.

LIFSCHITZ, V. 2002. Answer set programming and plan generation. *Artif. Intell. 138,* 1-2, 39–54.

MILANS, K. AND CLARK, B. 2006. The complexity of graph pebbling. *SIAM J. Discret. Math. 20,* 3 (Mar.), 769–798.

OIKARINEN, E. AND JANHUNEN, T. 2006. Modular equivalence for normal logic programs. In *ECAI*. Frontiers in Artificial Intelligence and Applications, vol. 141. IOS Press, 412–416.

REDL, C. 2017. Explaining inconsistency in answer set programs and extensions. In *LPNMR*. LNCS, vol. 10377. Springer, 176–190.

ROMERO, J., SCHAUB, T., AND SON, T. C. 2017. Generalized answer set planning with incomplete information. *CEUR Workshop Proceedings 1868*.

ROSSI, F., VAN BEEK, P., AND WALSH, T. 2006. Introduction. In *Handbook of Constraint Programming*. Foundations of Artificial Intelligence, vol. 2. Elsevier, 3–12.

SCHAEFER, M. 1999. Deciding the Vapnik-Chervonenkis dimension in Σ_3^p-complete. *J. Comput. Syst. Sci. 58,* 1, 177–182.

STOCKMEYER, L. J. 1976. The polynomial-time hierarchy. *Theor. Comput. Sci. 3,* 1, 1–22.

STOCKMEYER, L. J. AND MEYER, A. R. 1973. Word problems requiring exponential time: Preliminary report. In *STOC*. ACM, 1–9.

VAPNIK, V. 1998. *Statistical learning theory*. Wiley.

VAPNIK, V. N. AND CHERVONENKIS, A. Y. 2015. *On the Uniform Convergence of Relative Frequencies of Events to Their Probabilities*. Springer International Publishing, Cham, 11–30.