Solving and Visualizing ASP(Q) programs in ASP Chef

Mario Alviano[1], Andrea Cuteri[1], Marco Duca[1], Giuseppe Mazzottta[1] and Francesco Ricca[1]

[1]Department of Mathematics and Computer Science, University of Calabria, Rende, Italy

Abstract

The ASP(Q) language extends Answer Set Programming (ASP) with quantifiers over answer sets, enabling the modeling of problems across the entire Polynomial Hierarchy (PH). Despite its expressive power, the practical adoption of ASP(Q) is still limited by the lack of integrated tools supporting modeling, execution, and visualization. In this paper, we address this gap by integrating the state-of-the-art ASP(Q) solvers, namely PYQASP and CASPER, as operations within the ASP Chef platform. This allows users to seamlessly combine ASP and ASP(Q) components within a unified workflow, where ASP(Q) programs can be treated as reusable ingredients in ASP Chef recipes. This integration provides an intuitive and user-friendly environment for designing, composing, and solving hard combinatorial problems beyond NP by means of ASP(Q). To illustrate the practical benefits of our approach, we present two case studies from graph theory and game theory, namely the Clique Coloring (CC) problem and the Strong Nash Equilibria (SNE) problem, covering the entire workflow from modeling to solution visualization.

Keywords

Answer Set Programming, ASP with Quantifiers, ASP Chef,

1. Introduction

Answer Set Programming (ASP)[1, 2] is a well-established Knowledge Representation and Reasoning formalism based on stable model semantics. Over the years, ASP has been successfully applied to a wide range of real-world domains, including planning[3, 4], scheduling [5, 6, 7], and business process management [8, 9], among many others [10, 11]. One of the key strengths of ASP lies in its expressive yet intuitive declarative language, which enables the compact encoding of NP-complete problems through the well-known *Guess-and-Check* paradigm [1, 12]. However, when it comes to modeling problems beyond NP, ASP remains limited to the second level of the Polynomial Hierarchy (PH). In such cases, advanced techniques, such as saturation [13], are required which are often difficult to apply for ASP users [14]. To overcome this limitation, several language extensions have been proposed [15, 16, 17, 18]. Among these approaches, Answer Set Programming with Quantifiers (ASP(Q)) [18] extends ASP by introducing quantifiers over answer sets of ASP programs. More precisely, ASP(Q) allows a problem to be decomposed into several ASP programs, whose answer sets are combined through quantifiers, thus enabling the modeling of problems across the entire PH. Nowadays, ASP(Q) is not only of theoretical interest but it is actually used to solve many interesting problems among different fields such as Probabilistic Reasoning [19, 20], Planning [21], Outlier Detection [22], among others [23].

This practical applicability is made possible thanks to the availability of efficient solvers [24, 25, 26] that can evaluate ASP(Q) programs. State-of-the-art ASP(Q) solvers follow two possible approaches: rewriting into QBF formulae, or Counterexample-Guided Abstraction Refinement (CEGAR) [27]. In particular, CASPER [24] is the first ASP(Q) solver based on CEGAR and can handle 2-ASP(Q) programs (i.e., programs with at most two quantifiers). On the other hand, QASP [26] and PYQASP [25] evaluate ASP(Q) programs with an arbitrary number of quantifiers by translating programs into QBF formulae and then leverage well-mature QBF solving techniques. However, all these systems lack of a user-friendly environment which may hinder the practical adoption of ASP(Q), as it can pose significant challenges for non-expert users.

CILC 2026: The 41st Italian Conference on Computational Logic, June 23–25, 2026, Ferrara, Italy

mario.alviano@unical.it (M. Alviano); andrea.cuteri@unical.it (A. Cuteri); marco.duca@unical.it (M. Duca); giuseppe.mazzotta@unical.it (G. Mazzottta); francesco.ricca@unical.it (F. Ricca)

https://alviano.net/ (M. Alviano)

0000-0002-2052-2063 (M. Alviano); 0009-0000-7629-7347 (A. Cuteri); 0009-0003-8731-866X (M. Duca); 0000-0003-0125-0477 (G. Mazzottta); 0000-0001-8218-3178 (F. Ricca)

To some extent, similar usability challenges were encountered also for the base ASP language. These challenges have been effectively tackled by the introduction of user-friendly environments such as ASP Chef [28], ASPIDE [29], and SeaLion [30]. More specifically, the ASP Chef framework is built around the notion of *ASP recipes*, which are sequences of operations (i.e., solver calls, answer sets visualization, among others) referred to as *ingredients*. By combining different ingredients it is possible to design ad-hoc recipes which implement complex ASP workflows. The modular nature of ASP Chef operations suggests that the design of ASP(Q)-specific operations would enable to extend the benefits of ASP Chef also to the ASP(Q) language.

In this paper, we introduce two new ASP Chef operations, namely *@dumbo/pyqasp* and *@dumbo/casper*, which can be used as ingredients within ASP Chef recipes for solving ASP(Q) programs by means of PYQASP and the CASPER respectively. To showcase how these ingredients can be used in practice, we propose two case studies on interesting problems from graph and game theory, namely the Clique Coloring (CC) problem and the Strong Nash Equilibria (SNE) problem. For each of these use cases we show how to construct a complete ASP Chef recipe that starts from problem modeling, uses ASP(Q) solvers ingredient, and finally visualizes the result (i.e. problem solutions) by means of a dedicated visualization ingredient. Overall, this integration significantly enhances the usability of ASP(Q), providing a practical and user-friendly framework for modeling and solving complex problems beyond NP, fostering its adoption by a broader community.

2. Background

In this section we provide some preliminary notions about ASP, ASP(Q) and ASP Chef.

2.1. Answer Set Programming

Syntax. In ASP *variables* are alphanumeric strings starting with an uppercase letter whereas *constants* are either numbers or alphanumeric strings starting with a lowercase letter. A *term* can be either a constant or a variable. An *atom* is an expression of the form $p(\vec{t})$ where $t = t_1, \ldots, t_n$ is a list of terms and p is a *predicate* of arity n, denoted as p/n. An atom $p(\vec{t})$ is said to be *ground* if all terms in $\vec{t}$ are constants. A *literal* can be either an atom a or its negation $not\ a$, where not denotes *negation as failure*. A literal is said to be *negative* if it is of the form $not\ a$; otherwise it is *positive*. Given a set of literals L, L^+ (resp. L^-) denotes the set of positive (resp. negative) literals appearing in L. A *rule* is an expression of the form

$$h\ :-\ l_1, \ldots, l_n \tag{1}$$

where h is an atom referred to as *head*, and $l_1, \ldots, l_n$ with $n \geq 0$ is a conjunction of literals referred to as *body*. Given a rule r, we denote by $H(r)$ the set of atoms appearing in the head of r, while $B(r)$ denotes the set of literals appearing in the body of r. A rule r is a *constraint* if $H(r) = \emptyset$ or it is a *fact* if $B(r) = \emptyset$. A rule is safe if each variable appears in at least one literal in $B(r)^+$. A *program* P is a set of safe rules. Given a program P, $\mathcal{H}(P)$ denotes the set of atoms appearing in the head of some rules in P. Given an ASP program P, the *Herbrand Universe* of P, denoted as HU_P, is the set of all constants appearing in P; the *Herbrand Base*, denoted as B_P, is the set of all possible ground atoms that can be constructed using constants in HU_P and predicates in P; $ground(P)$ denotes the set of all possible ground rules that can be obtained from P substitutions of variables in P with constants in HU_P. The *dependency graph* of P, denoted by G_P, is a directed labeled graph whose nodes are ground atoms in B_P and there exists a positive (resp. negative) edge $(u, v, +)$ (resp. $(u, v, -)$) if there exists a rule $r \in ground(P)$ such that $H(r) = v$ and $u \in B(r)^+$ (resp. $u \in B(r)^-$). The program P is said to be *stratified* if G_P does not contain any loop involving negative edges. Programs can be extended with $\#show$ directives which are expressions of the form $\#show\ p(\vec{t}) : l_1, \ldots, l_n.$ where p is an optional predicate, $\vec{t}$ is a possibly empty sequence of terms, and $l_1, \ldots, l_n$ is a conjunction of literals.

Semantics. Given a program P, an *interpretation* $I \subseteq B_P$ is a set of atoms. Given an interpretation I, a positive literal a is true w.r.t. I if $a \in I$, false otherwise; a negative literal $not\ a$ is true w.r.t. I if $a \notin I$, false otherwise. A conjunction $conj$ of literals is true if all the literals in the conjunction are true w.r.t. I, false otherwise. An interpretation I is a *model* of P if for each rule $r \in ground(P)$ either $H(r)$ is true w.r.t. I or $B(r)$ is false w.r.t. I. Let M be a model of P, then the program reduct [2] of P w.r.t. M, denoted by P^M, is the program obtained from P by: (i) removing all rules $r \in ground(P)$ having a literal $l \in B(r)^-$ that is false w.r.t. M; (ii) removing all negative literals from the body of the remaining rules. A model M of P is an answer set of P iff no $M' \subset M$ exists such that M' is a model of P^M. We denote by $AS(P)$ the set of all answer sets of P. The program P is said to be *coherent* if $AS(P) \neq \emptyset$, otherwise P is *incoherent*. For programs containing $\#show$ directives, answer sets are projected according to such directives. We refer the reader to ASP-Core-2 standard for further details [31].

2.2. Answer Set Programming with Quantifiers

An ASP(Q) program [32] is an expression of the form:

$$\square_1 P_1 \ldots \square_n P_n : C \tag{2}$$

where $\square_1, \ldots, \square_n$ are quantifiers in $\{\exists^{st}, \forall^{st}\}$, $P_1, \ldots, P_n$ are ASP programs, and C is a stratified program with constraints. An ASP(Q) program Π such that $\square_1 = \exists^{st}$ is said to be *existential*, *universal* otherwise. For short we denote by an n-ASP(Q) an ASP(Q) program made of n quantifiers. We now define the semantics of ASP(Q) programs.

Let P be an ASP program, $M \in AS(P)$ be an answer set of P, and Π be an ASP(Q) program of the form (2); we denote by $\mathit{fix}_P(M)$ the set of facts and constraints of the form $\{a \mid a \in M\}$ and $\{ : -a \mid a \in B_P \setminus M\}$, and by $\Pi_{P,M}$ the ASP(Q) program obtained from Π by replacing P_1 with $P_1 \cup \mathit{fix}_P(M)$. The coherence of ASP(Q) programs is defined inductively as follows:

- $\exists^{st} P : C$ is coherent if there exists $M \in AS(P)$ such that $C \cup \mathit{fix}_P(M)$ is coherent;
- $\forall^{st} P : C$ is coherent if for each $M \in AS(P)$, $C \cup \mathit{fix}_P(M)$ 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 each $M \in AS(P)$, $\Pi_{P,M}$ is coherent;

For an existential ASP(Q) program $\exists^{st} P\ \Pi$, an answer set $M \in AS(P)$ such that $\Pi_{P,M}$ is coherent is called *quantified answer set*. We denote the set of all quantified answer sets of Π by $QAS(\Pi)$. Thus when Π is incoherent we have that $QAS(\Pi)$ is empty. Note that, for universal ASP(Q) programs the notion of quantified answer set is not defined as the universal quantifier requires that the ASP(Q) program has to be coherent for all answer set of the first program.

2.3. ASP Chef

In ASP Chef an *operation* O is a function which receives as input a sequence of interpretations and produces as output a new sequence of interpretations. Additionally, an operation may be customized according to operation-specific parameters and may produce supplementary outputs (e.g., a graph visualization). An *ingredient* is an instantiation of an operation. A *recipe* is a tuple of the form $(encode,\ Ingredients,\ decode)$, where $encode$ and $decode$ are boolean values, while $Ingredients$ is a sequence of ingredients. Thus, given a recipe r of the form $(encode,\ Ingredients,\ decode)$, the input of r is an alphanumeric string s_{in}. If $encode$ is true then s_{in} is encoded using the $Base64$ encoding. The result of the encoding is an atom of $\_\_base64\_\_(\text{“}s\text{”})$, where $s = Base64(s_{in})$ which is given as input to the first ingredient contained in $Ingredients$. At this point each ingredient of the pipeline (i.e., $Ingredients$) takes as input the output of previous one and produces as output a sequence of interpretations that will serve as input to the next ingredient. The output of the last ingredient o corresponds to the final output of the pipeline. If *decode* is true, every occurrence of $\_\_base64\_\_(s)$ in the final output is replaced with (the ASCII string associated with) $Base64^{-1}(s)$.

Mustache Templates. The ASP Chef Framework has been recently enriched by the possibility of using Mustache Templates. Mustache is a logic-less templating system which is used to insert dynamic contents in a text using placeholders represented by $\{\{\ldots\}\}$. Given an ASP program P containing also $\#show$ directive then a *Mustache query* is an expression of the form $\{\{P\}\}$. A *Mustache template* is a text that contains one or more Mustache queries. Given an input interpretation I, the evaluation of a Mustache query w.r.t. I yields a projected answer set of $P \cup \{a \mid a \in I\}$. The application of a Mustache template to I gives a text where each Mustache query is replaced with its evaluation w.r.t. I. A Mustache query in shortcut form is a more compact representation of a Mustache query.

Example 1. *Let us consider an input interpretation* $I = \{size(3)\}$ *and the following Mustache template:*

$$\text{“}The\ size\ is\ \{\{\#show\ (X) : size(X).\}\}\text{”}$$

The application of this template to I *yields the following text:*

$$\text{“}The\ size\ is\ (3)\text{”}$$

The same Mustache template can be obtained by using the following shortcut form for Mustache queries:

$$\{\{= (K) : size(K).\}\}$$

3. ASP Chef as an interactive environment for ASP(Q)

This section presents the integration of the ASP(Q) solvers PYQASP and CASPER as two ASP Chef operation. Specifically, we illustrate how these operations enable the execution and visualization of ASP(Q) programs within ASP Chef recipes.

3.1. Integration of ASP(Q) solvers in ASP Chef

As discussed in the preliminaries, ASP Chef provides a modular interface in which it is possible to construct pipelines by composing operations that take as input some interpretations and give as output new ones thus having a uniform input/output format. This design facilitates the construction of declarative pipelines, where operation results can be easily passed, inspected, and reused across different processing steps. As a result, ASP Chef supports the rapid development of ASP-powered pipelines for practical applications, including prototyping, debugging, and data analysis. Thus, ASP Chef is a natural candidate for supporting also the development of ASP(Q) applications. To this end, we propose the integration of available ASP(Q) solvers, namely PYQASP and CASPER, within ASP Chef. Concretely, we encapsulate them as two operations, *@dumbo/pyqasp* and *@dumbo/casper*, enabling their use within ASP Chef pipelines. This integration allows users to evaluate ASP(Q) programs, inspect their outcomes, and combine the resulting data (i.e. a quantified answer set, if any) with subsequent processing steps, such as visualization. *@dumbo/casper* is an operation for evaluating 2-ASP(Q) program (i.e., ASP(Q) programs made of 2 quantifiers) and accepting in input three parameters, namely $program$, $echo$, $enumerate$, where $program$ is a predicate name, whereas *echo* and *enumerate* are two boolean values. More precisely, this operation allows to call the CASPER system that takes as input the 2-ASP(Q) program Π which is obtained by decoding a fact $program(s_\Pi)$ from the input interpretation. Then, CASPER is executed to compute quantified answer sets of Π. Specifically, if $enumerate$ is true, CASPER is executed to compute all quantified answer sets of Π. Otherwise CASPER computes only one quantified answer set if it exists. The output of CASPER is collected by *@dumbo/casper* which will produce the output interpretation(s) that corresponds to quantified answer set(s) of Π. If *echo* is set to true, then *@dumbo/casper* includes in the interpretation given as output also the atom $progam(s_\Pi)$ which encodes the input program evaluated by CASPER.

The *@dumbo/pyqasp* operation is defined following the same line as for the *@dumbo/casper* operation. In particular, *@dumbo/pyqasp* is defined on the same input parameters of *@dumbo/casper* but it can

take as input an n-ASP(Q) program as the PYQASP solver is able to evaluate ASP(Q) programs with an arbitrary number of quantifiers.

Unlike other ASP Chef operations, which are executed entirely within the browser, both of these operations rely on the ASP Chef CLI (https://github.com/alviano/asp-chef-cli). In particular, PYQASP and CASPER must be installed within the ASP Chef CLI environment, as they are invoked locally when executing recipes that include the *@dumbo/casper* or *@dumbo/pyqasp* ingredients. When local execution terminates, the output produced by the ASP(Q) solver is gathered to be returned to the web application for subsequent processing.

3.2. Developing ASP(Q) Applications in ASP Chef

We now show how the introduced ASP(Q) operation can be used within ASP Chef recipes to develop ASP(Q) applications. In particular, we consider two interesting problems drawn from graph and game theory fields, namely Clique Coloring (CC) [33] and Strong Nash Equilibria (SNE) [34].

Clique Coloring. Given a graph $G = \langle V, E \rangle$ and an integer k, the clique coloring problem asks whether it is possible to assign vertices of G to colors $\{1, \ldots, k\}$ in such a way that, for every maximal clique $C \subseteq V$ of G, there are at least two nodes in C assigned to different colors. As shown by Amendola and Rotondaro (2021), the clique coloring problem can be modeled in ASP(Q) as an existential program of the form $\exists^{st} P_1 \forall^{st} P_2 : C$. Let us proceed step by step in the construction of the ASP(Q) program for the clique coloring problem. First of all, we need an ASP program P_1 for identifying valid colorings of the nodes of the graph. This task can be achieved by the following program.

```
%@exists // ∃st P1
asgn(X,C) :- node(X), color(C), not nasgn(X,C).
nasgn(X,C) :- node(X), color(C), not asgn(X,C).

:- asgn(X,C1), asgn(X,C2), C1!=C2.
colored(X) :- asgn(X,C).
:- node(X), not colored(X).
```

In particular, the above program guesses, via the first two rules, an assignment of colors to nodes of the graph. Whereas, the last three rules ensure that the guessed colors assignment is a valid coloring which means each node is assigned to exactly one color.

Then, to check the second property, we need another ASP program which identifies the maximal cliques of size at least two of G. The following ASP program P_2 serves this purpose.

```
%@forall // ∀st P2
inClique(X) :- node(X), not outClique(X).
outClique(X) :- node(X), not inClique(X).

out(X) :- node(X), inClique(Y), X!=Y, not edge(X,Y).
:- outClique(X), not out(X).
:- inClique(X), out(X).

atLeast2 :- inClique(X), inClique(Y), X<Y.
:- not atLeast2.
```

In particular, the first block of rules guesses a subset of the nodes which may form a clique of G; the second block enforces that guessed nodes form a clique and no other nodes can be added to the guessed clique (i.e., the clique is also maximal); finally the last block is used to discard cliques having less than two nodes.

At this point, we need to perform the final check required by our problem: maximal clique have at least two nodes colored with different colors. This check can be done by means of the following program C:

```
%@constraint // C
sat :- inClique(X), inClique(Y), asgn(X,C1),
asgn(Y,C2), C1!=C2.
:- not sat.
```

Intuitively, the first rule in C derives the atom sat if for a given clique and a given coloring of the graph, there are two nodes in the clique assigned to different colors. Then, the constraint : $-not\ sat$ forces the atom sat to be true. Thus, C is incoherent only if all nodes in the clique are assigned to the same color, coherent otherwise.

Given this ASP(Q) program, we can solve whatever instance of the clique coloring problem by properly encoding the input graph G and the available colors as facts within the program P_1. In particular, for each $v \in V$, $node(v)$ denotes that v is a node of G; for each $(u, v) \in E$, $edge(u, v)$ denotes that there exists an edge between the nodes u and v in G; while for each $i \in \{1, \ldots, k\}$, $color(i)$ denotes that i is a color which can be used to color nodes of G.

Now that we have an ASP(Q) encoding for solving the clique coloring problem, we can construct an ASP Chef recipe to solve CC instances and visualize their solutions. A possible ASP Chef recipe for CC can be obtained by using the following sequence of ingredients:

1. An *Encode* ingredient contains the ASP(Q) encoding of CC;
2. A *@dumbo/casper* (resp. *@dumbo/pyqasp*) compute quantified answer set of the ASP(Q) encoding
3. An *Encode* ingredient contains a Mustache template which generates the configuration for the visualization ingredient;
4. A *@vis.js/Network* visualize the solution.

In more detail, the *recipe* starts with an Encode ingredient encapsulating the 2-ASP(Q) encoding of the problem alongside the facts defining a specific instance, as reported in Figure 1. The result of this

#3. Encode

Height 700 | Predicate __program__ | Language

```
1  %@exists
2  node(a). node(b). node(c). node(d). node(e). node(f).
3  color(red). color(green). color(blue).
4  edge(a,b). edge(a,f). edge(b,c). edge(b,d). edge(c,f). edge(d,e). edge(e,f).
5  edge(b,a). edge(f,a). edge(c,b). edge(d,b). edge(f,c). edge(e,d). edge(f,e).
6
7  asgn(X,C) :- node(X), color(C), not nasgn(X,C).
8  nasgn(X,C) :- node(X), color(C), not asgn(X,C).
9
10 :- asgn(X,C1), asgn(X,C2), C1!=C2.
11 colored(X) :- asgn(X,C).
12 :- node(X), not colored(X).
13
14 %@forall
15 inClique(X) :- node(X), not outClique(X).
16 outClique(X) :- node(X), not inClique(X).
17
18 out(X) :- node(X), inClique(Y), X!=Y, not edge(X,Y).
19 :- outClique(X), not out(X).
20 :- inClique(X), out(X).
21
22 atLeast2 :- inClique(X), inClique(Y), X<Y.
23 :- not atLeast2.
24
25 %@constraint
26 sat :- inClique(X), inClique(Y), asgn(X,C1),
27 asgn(Y,C2), C1!=C2.
28 :- not sat.
29
30
31
```

Figure 1: Encode ingredient containing the 2-ASP(Q) encoding of the cc problem

#3. Encode

Height 420 | Predicate __vis__ | Language JSON

```
{
  data: {
    nodes: [
      {{= show({{f'{{ id: "${X}", label: "${X}", color: {{ background: "${C}" }} }}'}}, X) : asgn(X,C) }}
    ],
    edges: [
      {{= {{f'{{ from: "${X}", to: "${Y}" }}'}} : edge(X,Y) }}
    ]
  },
  options: {
    nodes: { shape: "circle", size: 16, borderWidth: 2, shadow: true, font: { color: "white" } },
    edges: { width: 2, shadow: true, arrows: { to: { enabled: false } }, smooth: { enabled: false } },
    physics: {
      stabilization: false,
      barnesHut: { gravitationalConstant: -750, springLength: 60, springConstant: 0.08 }
    }
  }
}
```

Figure 2: Template used to populate a Relaxed JSON configuration for visualizing quantified answer sets of the cc problem

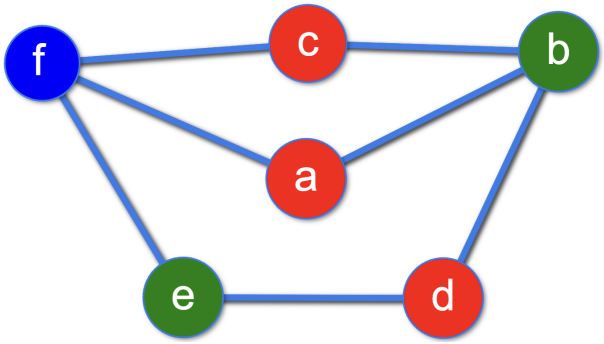

Figure 3: Visualization of a cc quantified answer set

ingredient is the ASP(Q) program $\Pi = \exists^{st} P_1 \cup Facts\ \forall^{st} P_2 : C$, that is encoded using the $Base64$ into a string s_Π which is used to construct the fact $\_\_program\_\_(s_\Pi)$.

The produced fact is given to the following ingredient by encapsulating it into an interpretation $I = \{\_\_program\_\_(s_\Pi)\}$. This interpretation is then processed by the @dumbo/casper (or alternatively *@dumbo/pyqasp*) ingredient, parameterized to read the `__program__` predicate. This ingredient takes the fact $\_\_program\_\_(s_\Pi)$, decodes from $Base64$ the string s_Π and obtains back the program Π. Then, the ingredient dispatches Π to the CASPER solver, installed in the ASP Chef CLI, which computes a quantified answer set M if it exists. In our example such a quantified answer set exists and so CASPER returns it to *@dumbo/casper*. Consequently, *@dumbo/casper* gives M as its output interpretation that will be taken as input by another *Encode* ingredient. Here we recall that M corresponds to a valid coloring of a graph G where each maximal clique of G has at least two nodes colored with different colors. Thus, to visualize a graph and a coloring of its nodes we have to use *@vis.js/Network* operation. This operation requires a configuration described by a JSON document that can be obtained from M by using a specific Mustache template for the *@vis.js/Network* configuration.

This Mustache template is reported in the *Encode* ingredient from Figure 2. More in detail, the template relies on the shortcut form of Mustache queries alongside f-strings to enable the direct interpolation of variables into the JSON structure. A series of inline ASP queries are computed over the quantified answer set M to populate the `nodes` and `edges` arrays of the JSON configuration. The former is updated by atoms of the form $asgn(X, C)$ that are projected into JSON objects representing nodes. Each object uses X as the node label and C as the background color assigned to the node. The latter (i.e., $edges$ array) is similarly populated by mapping $edge(X, Y)$ atoms into JSON objects that represent the graph's undirected links. The remaining part of the template contains static `options` (i.e., do not

depend on M) and allows to customize the layout physics and visual styling of the network.

At this point, the *Encode* ingredient encode the Mustache template in $Base64$, thus obtaining a string representation s which is then used as term of the fact $\_\_vis\_\_(s)$. This fact is added to the output interpretation and it is given to the next ingredient of the recipe that is *@vis.js/Network*. This operation first decodes the atom $\_\_vis\_\_(s)$ obtaining back the Mustache template and then applies the template on the input interpretation. As a result, it obtains the specific JSON configuration that is given to the Vis.js framework (https://visjs.org/) to map the input interpretation M to a dynamic and automatically organized network views yielding, in our example, a colored graph for the CC problem as shown in Figure 3. An example of the just defined recipe is available at https://asp-chef.alviano.net/s/CC#markducks/ASP-Q-recipes).

Strong Nash Equilibria. Nash Equilibria are an important concept in the theory of strategic games [34]. Intuitively, a game $\mathcal{G}$ is played by a set of players P. In the game $\mathcal{G}$ we have different information for each player $p \in P$:

- an action set A_p that contains the actions that can be played by p;
- a set of players N_p that are the neighbors of p; and
- a utility function $u_p : A_p \times_{j \in N_p} A_j \to \mathbb{R}$ describing the utility associated to an action played by p w.r.t. the actions played by all its neighbors.

Thus, a game $\mathcal{G}$ is formally defined as a tuple $\langle P, Neigh, Act, U \rangle$, where P is the set of players, $Neigh = \{N_p \mid p \in P\}$, $Act = \{A_p \mid p \in P\}$, and $U = \{u_p \mid p \in P\}$. For each player $p \in P$, the action played by P is referred to as *strategy*. Given a subset of players $P' \subseteq P$, a *combined strategy* for P' contains one strategy for each player $p \in P'$. The set of combined strategy for P' is denoted by $St(P')$. A combined strategy of the full set of players P is referred to as *global strategy*. A set of players $K \subseteq P$ is referred to as a *coalition*. Let x be a global strategy, $K \subseteq P$ be a coalition and y a combined strategy for K. Then, we denote by $x_{\_K}[y]$ the global strategy obtained from x by replacing the strategy of each player in $p \in K$ with the strategy played by p in y. If K is a singleton $\{p\}$, we will simply write $x_{\_p}[y]$. Let x be a global strategy, p be a player, and u_p be their utility function. Then, with a slight abuse of notation, we use $u_p(x)$ to denote the $u_p(s)$ where s denotes the strategy of p in x and is the combined strategy by neighbors of p in x. A global strategy x is a *Strong Nash Equilibrium*(SNE) for $\mathcal{G}$ if $\forall K \subseteq P$, $\forall y \in St(K)$, $\exists p \in K$ such that $u_p(x) \geq u_p(x_{\_K}[y])$, or equivalently, if $\forall K \subseteq P$, $\nexists y \in St(K)$ such that $\forall p \in K$, $u_p(x) < u_p(x_{\_K}[y])$.

Intuitively, for a global strategy x to be a SNE, it must not exist a coalition K that can design a combined strategy y in which all the players in K increase their utility w.r.t. the utility they had over x.

A game $\mathcal{G}$ is in *graphical normal form* (GNF) if the utility function u is represented as a list of separate tables, one for each player $p \in P$, containing a cell for each combined strategy $x \in St(Neigh(p) \cup \{p\})$. Deciding whether a game $\mathcal{G}$ in GNF has a strong Nash equilibria is Σ_2^P-complete [34].

This problem can be modeled in ASP(Q) as an existential program of the form $\exists^{st} P_1 \forall^{st} P_2 : C$. Let us proceed step by step in the construction of such an ASP(Q) program. First, we need an ASP program P_1 which identifies all possible global strategies of $\mathcal{G}$. The following program serves this purpose.

```
%@exists // ∃st P1
play(P, A) :- act(P, A), not nPlay(P, A).
nPlay(P, A) :- act(P, A), not play(P, A).

:- player(P), play(P, A1), play(P, A2), A1 != A2.
played(P) :- play(P, A).
:-player(P), not played(P).

missingSt(P1,A1,Id) :- playerResp(Id,P1,A1,P2,A2), not play(P2, A2).
utilityU(P,C) :- play(P,A), combinedScore(Id,P,A,C),
                 not missingSt(P,A,Id).
```

The input of P_1 is encoded as facts over the predicates $player/1$, $act/2$, $playerResp/5$, and $combinedScore/4$. Specifically, $player(p)$ encodes that p is a player of the game and $act(p, a)$ encodes that a is a possible action from the action set of player p. Then, we introduce some identifiers to model the utility function of each player. Specifically, $id\_p\_a\_y$ identifies a player p playing an action a against a combined strategy y of their neighbors. Then, for each identifier $id = id\_p\_a\_y$, the facts of the form $playerResp(id, p, a, n, a_n)$ denote that the neighbor n of p is playing the action a_n in y; whereas $combinedScore(id, p, a, c)$ denotes that c is the utility of p playing a against y. Given this input encoding, rules in P_1 guess a possible global strategy x and compute the utility of each player (i.e., $u_p(x)$, for each $p \in P$). More in detail, the first block of rules guesses whether a player P plays the action A or not in x; the second block enforces that each player has to play exactly one action in x; and the last block of rules identifies the combined strategy played by players' neighbors in x and computes the associated utility C for each player in the game. Then we need a program P_2 which identifies all possible coalitions and all possible combined strategies for such coalitions.

```
%@forall // ∀st P2
inC(X) :- player(X), not outC(X).
outC(X) :- player(X), not inC(X).

playC(P, A) :- inC(P), act(P, A), not nPlayC(P, A).
nPlayC(P, A) :- inC(P), act(P, A), not playC(P, A).
:- inC(P), playC(P, A1), playC(P, A2), A1 != A2.
playedC(P) :- playC(P, A).
:-playC(P), not playedC(P).

playerAct(P,A):- inC(P),playC(P,A).
playerAct(P,A):- play(P,A),not inC(P).

missingStC(P1,A1,Id) :- playerResp(Id,P1,A1,P2,A2),
                        not playerAct(P2,A2).
utilityCU(P,C):- playerAct(P,A),combinedScore(Id,P,A,C),
                 not missingStC(P,A,Id).
```

In particular, the first two rules guess a coalition K of players, whereas the remaining rules are specular w.r.t. the rules of P_1 and are used to: (i) guess a combined strategy y for the coalition K; (ii) inherit actions played in x for players not in the coalition K (i.e., obtaining $x_{\_K}[y]$); and (iii) compute the value of the utility function for each player p w.r.t. $x_{\_K}[y]$. Finally, we need an ASP program C for comparing the value of the utility function of each player in the coalition K according to the global strategy x and the value of the utility function computed on $x_{\_K}[y]$ (i.e., the global strategy where players in K changed their strategy).

```
%@constraint // C
betterInC(P) :- inC(P), utilityU(P, C1),
                utilityCU(P, C2), C2>C1.
noImprove(P) :- inC(P), player(P), not betterInC(P).
allImproved :- inC(P), not noImprove(P).
:- allImproved.
```

In particular, the first rule of C is used to derive $betterInC(P)$ for each player in the coalition K that improves its utility function (i.e. $u_p(x_{\_K}[y]) > u_p(x)$). The second rule, is used to derive $noImprove(P)$ for all the players in the coalition K that do not improve their utility. Finally, the last two rules impose that at least one player from K must not improve their utility.

By leveraging the proposed ASP(Q) encoding, we can solve arbitrary instances of the SNE problem by encoding the input instance in the program P_1 and then solving the resulting program.

We now describe an ASP Chef recipe for solving SNE instances and visualize their solutions. A

#3. Encode

Height 1000 | Predicate __program__ | Language

```
%@exists
player(f). act(f, m). act(f, o).
player(g). act(g, m). act(g, o).
player(p). act(p, m). act(p, o).
player(r). act(r, m). act(r, o).
player(m). act(m, m). act(m, o).

%player f move m
playerResp(1, f, m, p, m). playerResp(1, f, m, r, m). combinedScore(1, f, m, 2).
playerResp(2, f, m, p, m). playerResp(2, f, m, r, o). combinedScore(2, f, m, 2).
playerResp(3, f, m, p, o). playerResp(3, f, m, r, m). combinedScore(3, f, m, 1).
playerResp(4, f, m, p, o). playerResp(4, f, m, r, o). combinedScore(4, f, m, 0).

play(P, A) :- act(P, A), not nPlay(P, A).
nPlay(P, A) :- act(P, A), not play(P, A).
:- player(P), play(P, A1), play(P, A2), A1 != A2.
played(P) :- play(P, A).
:-player(P), not played(P).
missingSt(P1,A1,Id) :- playerResp(Id,P1,A1,P2,A2), not play(P2, A2).
utilityU(P,C) :- play(P,A), combinedScore(Id,P,A,C), not missingSt(P,A,Id).

current_scenario(Id) :- combinedScore(Id, P, A, _), play(P, A), not missingSt(P, A, Id).

%@forall
inC(X) :- player(X), not outC(X).
outC(X) :- player(X), not inC(X).
playC(P, A) :- inC(P), act(P, A), not nPlayC(P, A).
nPlayC(P, A) :- inC(P), act(P, A), not playC(P, A).
:- inC(P), playC(P, A1), playC(P, A2), A1 != A2.
playedC(P) :- playC(P, A).
:-playC(P), not playedC(P).
playerAct(P,A):- inC(P),playC(P,A).
playerAct(P,A):- play(P,A),not inC(P).

missingStC(P1,A1,Id) :- playerResp(Id,P1,A1,P2,A2), not playerAct(P2,A2).
utilityCU(P,C):- playerAct(P,A),combinedScore(Id,P,A,C), not missingStC(P,A,Id).

%@constraint
betterInC(P) :- inC(P), utilityU(P, C1), utilityCU(P, C2), C2>C1.
noImprove(P) :- inC(P), player(P), not betterInC(P).
allImproved :- inC(P), not noImprove(P).
:- allImproved.
```

Figure 4: Encode ingredient containing the 2-ASP(Q) encoding of the SNE problem

possible recipe for SNE has the following sequence of ingredients:

1. An $Encode$ containing the ASP(Q) encoding of SNE;
2. a $@dumbo/casper$ (resp. $@dumbo/pyqasp$) for computing quantified answer sets;
3. An $Encode$ containing a Mustache template which generates the configuration for the visualization ingredient;
4. a $Tabulator$ to visualize the solutions.

Overall, the structure of the recipe is the same w.r.t. the one used for CC. However, the visualization ingredient is not the same since for this problem, we do not have a colored graph as output but we have a global strategy together with the output of the utility function of each player. Such kind of output fits well a table-based representation, where each row represents a combined strategy for the coalition made of a player p and all their neighbors.

Let us now go into the details of SNE recipe. Given a game $\mathcal{G} = \langle P, Neigh, Act, U \rangle$, we can encode $\mathcal{G}$ as a set of facts F over predicates $player/1$, $act/2$, $playerResp/5$, and $combinedScore/4$ according to the aforementioned description. At this point, the first Encode ingredient contains the ASP(Q) encoding $\Pi = \exists^{st} P_1 \cup F \forall^{st} P_2 : C$ of the problem together with the input game. Then, similarly as the CC recipe the *@dumbo/casper* (or alternatively *@dumbo/pyqasp*) ingredient computes a quantified answer set of Π. The result generated by *@dumbo/capser* is give to another *Encode* ingredient that leverages a Mustache template for generating a JSON configuration for the *Tabulator* ingredient that will visualize our result. Similarly to the previous recipe, the template uses the shortcut form for Mustache queries alongside f-strings to enable the direct interpolation of variables into the JSON structure. In this case,

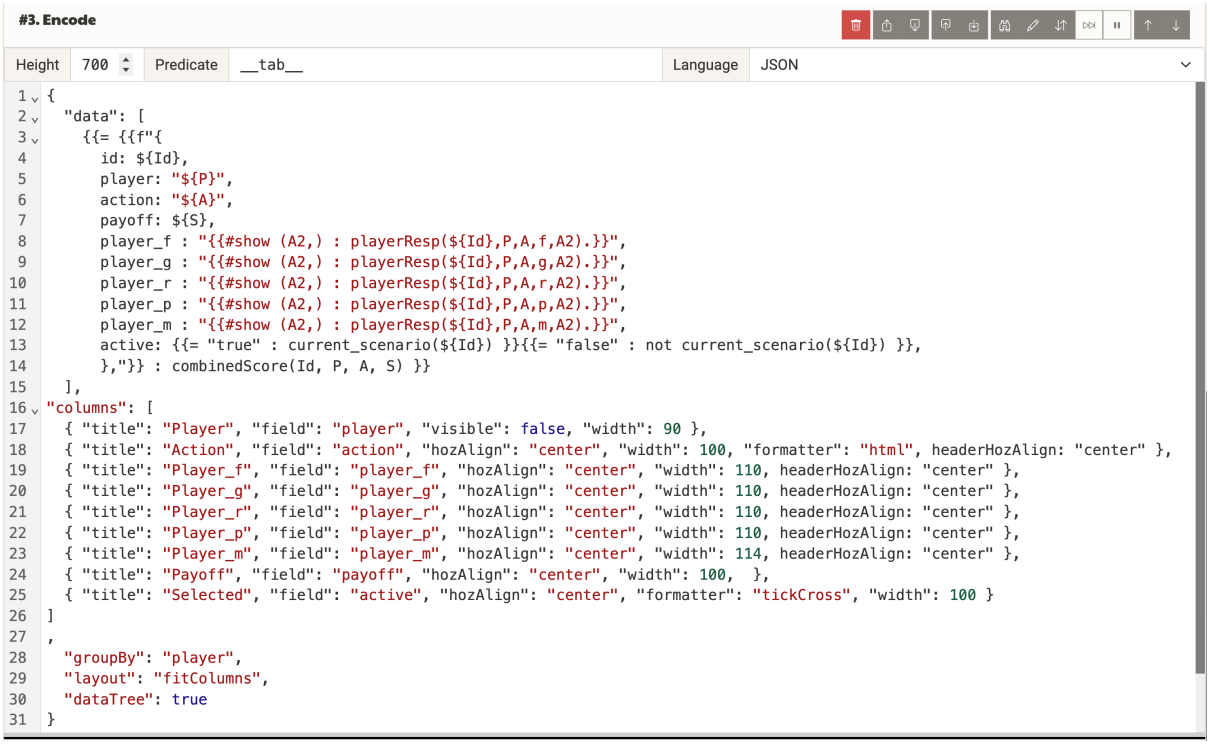

Figure 5: Template used to populate a Relaxed JSON configuration for visualizing quantified answer sets of the SNE problem

we need to populate an array of JSON objects, where each object represents a row of our table. Such objects should be of the following form:

```
{
   id: identifier,
   player: p,
   action: a,
   payoff: score,
   player_f : a_f,
   player_g : a_g,
   player_r : a_r,
   player_p : a_p,
   player_m : a_m,
   active: yes/no
}
```

To construct such kind of objects, multiple Mustache queries are required.

First, for each player p and each action $a \in A_p$, we need to consider all combined strategies of the remaining players together with the associated utility. More precisely, atoms of the form $combinedScore(id, p, a, score)$ indicate that there exists a combined strategy, identified by id, in which player p plays action a and obtains utility $score$. These atoms directly provide the values for the fields id, $player$, $action$, and $payoff$.

The actions of the other players can be retrieved from atoms of the form $playerResp(id, \_, \_, p', a')$, which specify that, in the combined strategy id, player p' plays action a'. Accordingly, each field corresponding to another player (e.g., $player\_f$) can be filled by issuing a Mustache query that matches atoms of the form $playerResp(id, \_, \_, f, \_)$. By applying such queries for all players in the game, all the required columns of the object can be populated.

Finally, the field $active$ can be populated with *yes* or *no* by issuing another Mustache query that

Action	Player_f	Player_g	Player_r	Player_p	Player_m	Payoff	Selected
f (8 items)							
m			m	m		2	✗
m			o	m		2	✓
...	...	...	...	...	...	...	...
g (8 items)							
m	m			m		2	✓
o	m			m		2	✗
...	...	...	...	...	...	...	...
r (4 items)							
o	m					2	✓
m	o					1	✗
...	...	...	...	...	...	...	...
p (4 items)							
m	m					2	✓
o	o					1	✗
...	...	...	...	...	...	...	...
m (4 items)							
o			o			2	✓
m			m			1	✗
...	...	...	...	...	...	...	...

Figure 6: Visualization of a SNE quantified answer set

checks for the presence of atoms of the form $current\_scenario(id)$. If such an atom is true in the quantified answer set, it indicates that the actions played by all players in the combined strategy identified by id coincide with those in the global strategy corresponding to the Strong Nash Equilibrium.

Other than the *data* array, we statically define:

- the `columns` array that contains all the information required to display columns headers and setup their look and feel;
- a configuration block to define the structure of the table and control its visual presentation.

The generated configuration is then given in input to the *Tabulator* ingredient, which leverages the Tabulator library (https://tabulator.info/) to generate interactive data tables. In our case, the operation produces a list of separate tables, one for each player, containing a row for each combined strategy $y \in St(Neigh(p) \cup \{p\})$ and highlights the strategy profile associated with the computed Strong Nash Equilibrium. More precisely, for each player in the game, the row corresponding to the utility of that player according to the global strategy is marked with a green tick, while the remaining rows are marked with a red cross. An example of visualization of a Strong Nash Equilibrium is provided in Figure 6. The instance we considered is the same considered by Gottlob et al. in Example 2.2. An example of the just defined recipe is available at https://asp-chef.alviano.net/s/SNE#markducks/ASP-Q-recipes).

4. Related Work

In this paper we extend the ASP Chef framework [28] to support the ASP(Q) formalism.

Several systems have been proposed to support the development of ASP applications [29, 30, 35, 36, 37]. Although none of them supports ASP(Q), they are closely related to our work, as their underlying features can be exploited also to support the development of ASP(Q) applications.

Among Integrated Development Environments (IDE) for ASP, ASPIDE [29] provides advanced editing features (e.g., syntax highlighting, auto-completion, and error checking), as well as graphical tools for

workspace management and dependency graph visualization, offering a comprehensive environment for ASP development. Similarly, SeaLion [30] provides advanced editing, visualization, and debugging support. LoIDE [36], instead, is a web-based IDE that emphasizes interoperability and flexibility by supporting multiple formalisms and solvers through a modular architecture. Unlike traditional desktop IDEs, LoIDE relies on a backend infrastructure to execute programs. In contrast, ASP Chef executes most operations directly in the browser, allowing recipes to be easily shared and reused.

Concerning debugging and validation, unit testing has also been investigated for ASP. Early support was introduced within ASPIDE and SeaLion, and later generalized through the LANA annotation language [38]. More recently, testing frameworks such as ASP-WIDE [39] have been proposed to support test-driven development, allowing tests to be specified alongside ASP programs while remaining transparent to their standard evaluation.

Finally, visual representation of answer sets of ASP programs has been also studied extensively. Early tools such as ASPVIZ [40], IDPD3 [41], and KARA [42] generate visualizations by encoding layout information as ASP facts. More recent approaches, including Clingraph [43], Clinguin [44], and ASPECT [45], provide high-quality visualizations, although they typically rely on external tools. These visualization tools are, however, orthogonal to our work, and find application within the ASP Chef framework.

5. Conclusions

In this paper, we addressed the lack of user-friendly environments for ASP(Q) by extending the ASP Chef framework with dedicated operations for ASP(Q) solving. In particular, we introduced the *@dumbo/pyqasp* and *@dumbo/casper* operations, enabling the seamless integration of state-of-the-art ASP(Q) solvers within ASP Chef workflows.

As a result, ASP(Q) can now be effectively used within ASP Chef, supporting the modeling of hard combinatorial problems across the entire Polynomial Hierarchy. We demonstrated the practical applicability of our approach through two case studies from graph and game theory, showing how complex ASP(Q) problems can be modeled, solved, and visualized within a unified and intuitive environment.

Overall, this integration significantly simplifies the development of ASP(Q) applications, fostering its adoption by a broader community. As future work, we plan to extend the set of supported operations, including the integration of ASP(Q) within Mustache queries, as well as the development of dedicated debugging techniques for ASP(Q) programs.

Acknowledgments

This work has been partially funded by the Italian Ministry of Industrial Development (MISE) under project EI-TWIN n. F/310168/05/X56 CUP B29J24000680005 and also partially by projects SIGENERA (CUP J29I24001770005), MOZART (J49I24001740005) and NextGenGuides (CUP J39I24002040005) selected within the framework of the PR FESR – FSE Calabria 2021/2027 and implemented with the support of the Italian State and the Calabria Region; by the Italian Ministry of Health (MSAL) under POS projects CAL.HUB.RIA (CUP H53C22000800006) and RADIOAMICA (CUP H53C22000650006); by the Italian Ministry of Enterprises and Made in Italy under project STROKE 5.0 (CUP B29J23000430005); under PN RIC project ASVIN “Assistente Virtuale Intelligente di Negozio” (CUP B29J24000200005); Mario Alviano is a member of Gruppo Nazionale Calcolo Scientifico-Istituto Nazionale di Alta Matematica (GNCS-INdAM).

Declaration on Generative AI

The author(s) have not employed any Generative AI tools.

References

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

[2] M. Gelfond, V. Lifschitz, Classical negation in logic programs and disjunctive databases, New Gener. Comput. 9 (1991) 365–386.

[3] T. C. Son, E. Pontelli, M. Balduccini, T. Schaub, Answer set planning: A survey, TPLP 23 (2023) 226–298.

[4] M. Balduccini, D. Magazzeni, M. Maratea, E. Leblanc, CASP solutions for planning in hybrid domains, Theory Pract. Log. Program. 17 (2017) 591–633.

[5] C. Dodaro, G. Galatà, M. Maratea, I. Porro, An ASP-based framework for operating room scheduling, Intelligenza Artificiale 13 (2019) 63–77.

[6] M. Alviano, R. Bertolucci, M. Cardellini, C. Dodaro, G. Galatà, M. K. Khan, M. Maratea, M. Mochi, V. Morozan, I. Porro, M. Schouten, Answer set programming in healthcare: Extended overview, in: IPS-RCRA@AI*IA, CEUR Workshop Proceedings, CEUR-WS.org, 2020.

[7] P. Bruno, C. Dodaro, G. Galatà, M. Maratea, M. Mochi, Improving asp-based ORS schedules through machine learning predictions, Theory Pract. Log. Program. 25 (2025) 558–578.

[8] F. Chiariello, V. Fionda, A. Ielo, F. Ricca, A direct ASP encoding for declare, in: PADL, Lecture Notes in Computer Science, Springer, 2024, pp. 116–133.

[9] V. Fionda, A. Ielo, F. Ricca, Ltlf2asp: Ltlf bounded satisfiability in ASP, in: LPNMR, Lecture Notes in Computer Science, Springer, 2024, pp. 373–386.

[10] T. Eiter, M. Fink, G. Greco, D. Lembo, Repair localization for query answering from inconsistent databases, ACM Trans. Database Syst. 33 (2008) 10:1–10:51.

[11] C. Dodaro, N. Leone, B. Nardi, F. Ricca, Allotment problem in travel industry: A solution based on ASP, in: RR, volume 9209 of *Lecture Notes in Computer Science*, Springer, 2015, pp. 77–92.

[12] M. Gebser, N. Leone, M. Maratea, S. Perri, F. Ricca, T. Schaub, Evaluation techniques and systems for answer set programming: a survey, in: IJCAI, ijcai.org, 2018, pp. 5450–5456.

[13] T. Eiter, G. Gottlob, On the computational cost of disjunctive logic programming: Propositional case, Ann. Math. Artif. Intell. 15 (1995) 289–323.

[14] M. Gebser, R. Kaminski, T. Schaub, Complex optimization in answer set programming, TPLP 11 (2011) 821–839.

[15] B. Bogaerts, T. Janhunen, S. Tasharrofi, Stable-unstable semantics: Beyond NP with normal logic programs, TPLP 16 (2016) 570–586. doi:10.1017/S1471068416000387.

[16] T. Janhunen, Representing normal programs with clauses, in: R. L. de Mántaras, L. Saitta (Eds.), Proceedings of ECAI'2004., IOS Press, 2004, pp. 358–362.

[17] J. Fandinno, F. Laferrière, J. Romero, T. Schaub, T. C. Son, Planning with incomplete information in quantified answer set programming, Theory Pract. Log. Program. 21 (2021) 663–679.

[18] G. Amendola, F. Ricca, M. Truszczynski, Beyond NP: quantifying over answer sets, Theory Pract. Log. Program. 19 (2019) 705–721.

[19] D. Azzolini, G. Mazzotta, F. Ricca, F. Riguzzi, 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, ijcai.org, 2025, pp. 9049–9057. URL: https://doi.org/10.24963/ijcai.2025/1006. doi:10.24963/IJCAI.2025/1006.

[20] D. Azzolini, G. Mazzotta, F. Ricca, F. Riguzzi, A novel framework for reasoning over optimization problems in probabilistic answer set programming, in: KR, 2025.

[21] W. Faber, M. Morak, L. Chrpa, Determining action reversibility in STRIPS using answer set programming with quantifiers, in: PADL, volume 13165 of *Lecture Notes in Computer Science*, Springer, 2022, pp. 42–56.

[22] P. Bellusci, G. Mazzotta, F. Ricca, Modelling the outlier detection problem in ASP(Q), in: PADL, volume 13165 of *Lecture Notes in Computer Science*, Springer, 2022, pp. 15–23.

[23] W. Faber, M. Morak, Evaluating epistemic logic programs via answer set programming with quantifiers, in: AAAI, AAAI Press, 2023, pp. 6322–6329.

[24] A. Cuteri, G. Mazzotta, F. Ricca, 2-asp(q) solving based on CEGAR, in: AAAI, AAAI Press, 2026, pp. 19030–19038.
[25] W. Faber, G. Mazzotta, F. Ricca, An efficient solver for ASP(Q), TPLP 23 (2023) 948–964.
[26] G. Amendola, B. Cuteri, F. Ricca, M. Truszczynski, Solving problems in the polynomial hierarchy with ASP(Q), in: LPNMR, Lecture Notes in Computer Science, Springer, 2022, pp. 373–386.
[27] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, H. Veith, Counterexample-guided abstraction refinement for symbolic model checking, J. ACM 50 (2003) 752–794.
[28] M. Alviano, L. A. R. Reiners, W. Faber, ASP chef grows mustache to look better, Theory Pract. Log. Program. 25 (2025) 417–436. URL: https://doi.org/10.1017/s1471068425100094. doi:10.1017/S1471068425100094.
[29] O. Febbraro, K. Reale, F. Ricca, ASPIDE: integrated development environment for answer set programming, in: LPNMR, Lecture Notes in Computer Science, Springer, 2011, pp. 317–330.
[30] P. Busoniu, J. Oetsch, J. Pührer, P. Skocovsky, H. Tompits, Sealion: An eclipse-based IDE for answer-set programming with advanced debugging support, Theory Pract. Log. Program. 13 (2013) 657–673. URL: https://doi.org/10.1017/S1471068413000410. doi:10.1017/S1471068413000410.
[31] F. Calimeri, W. Faber, M. Gebser, G. Ianni, R. Kaminski, T. Krennwallner, N. Leone, M. Maratea, F. Ricca, T. Schaub, Asp-core-2 input language format, Theory Pract. Log. Program. 20 (2020) 294–309.
[32] G. Amendola, G. Rotondaro, Modeling clique coloring via ASP(Q), in: ICLP Workshops, volume 2970 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2021.
[33] D. Marx, Complexity of clique coloring and related problems, Theor. Comput. Sci. 412 (2011) 3487–3500. URL: https://doi.org/10.1016/j.tcs.2011.02.038. doi:10.1016/J.TCS.2011.02.038.
[34] G. Gottlob, G. Greco, F. Scarcello, Pure nash equilibria: Hard and easy games, J. Artif. Intell. Res. 24 (2005) 357–406.
[35] C. Dodaro, P. Gasteiger, K. Reale, F. Ricca, K. Schekotihin, Debugging non-ground ASP programs: Technique and graphical tools, Theory Pract. Log. Program. 19 (2019) 290–316.
[36] F. Calimeri, S. Germano, E. Palermiti, K. Reale, F. Ricca, Developing ASP programs with ASPIDE and loide, Künstliche Intell. 32 (2018) 185–186. URL: https://doi.org/10.1007/s13218-018-0534-z. doi:10.1007/S13218-018-0534-Z.
[37] O. Febbraro, N. Leone, G. Grasso, F. Ricca, JASP: A framework for integrating answer set programming with java, in: KR, AAAI Press, 2012.
[38] M. De Vos, D. G. Kisa, J. Oetsch, J. Pührer, H. Tompits, Annotating answer-set programs in LANA, TPLP 12 (2012) 619–637.
[39] G. Amendola, G. Mazzotta, F. Ricca, T. Berei, Unit testing in ASP revisited: Language and test-driven development environment, Theory Pract. Log. Program. 24 (2024) 1078–1108.
[40] O. Cliffe, M. D. Vos, M. Brain, J. A. Padget, ASPVIZ: declarative visualisation and animation using answer set programming, in: ICLP, Lecture Notes in Computer Science, Springer, 2008, pp. 724–728.
[41] R. Lapauw, I. Dasseville, M. Denecker, Visualising interactive inferences with IDPD3, CoRR abs/1511.00928 (2015).
[42] C. Kloimüllner, J. Oetsch, J. Pührer, H. Tompits, Kara: A system for visualising and visual editing of interpretations for answer-set programs, in: INAP/WLP, Lecture Notes in Computer Science, Springer, 2011, pp. 325–344.
[43] S. Hahn, O. Sabuncu, T. Schaub, T. Stolzmann, *Clingraph*: A system for asp-based visualization, Theory Pract. Log. Program. 24 (2024) 533–559.
[44] A. Beiser, S. Hahn, T. Schaub, Asp-driven user-interaction with clinguin, in: ICLP, EPTCS, 2024, pp. 215–228.
[45] A. Bertagnon, M. Gavanelli, ASPECT: answer set representation as vector graphics in latex, J. Log. Comput. 34 (2024) 1580–1607.