Generating explanations for answer set programming applications

Ly Ly Trieu[a], Tran Cao Son[a], Enrico Pontelli[a], and Marcello Balduccini[b]

[a]New Mexico State University, MSC CS, PO Box 30001, Las Cruces, New Mexico, USA
[b]Saint Joseph's University, 5600 City Avenue, Philadelphia, PA, USA

ABSTRACT

We present an explanation system for applications that leverage Answer Set Programming (ASP). Given a program P, an answer set A of P, and an atom a in the program P, our system generates all explanation graphs of a which help explain why a is true (or false) given the program P and the answer set A. We illustrate the functionality of the system using some examples from the literature.

Keywords: Explainable AI, Answer Set Programming, Artificial Intelligence

1. INTRODUCTION

In recent years, explainable AI has been introduced to help users gain confidence in the AI's decisions and conclusions. We aim at developing AI systems capable of explaining their responses to a request from a user until the user finds that the answers are acceptable as illustrated in Fig. 1. It starts with the user sending a request (Step 1). The system responds with an initial answer (2). Thereafter, a dialog between the two ensues with user sending questions and information—that the user believes might be different than those owned by the system—to the system (3), and the system responds with explanations and information—that the system believes the user might not have or have in incorrect form (4). This process continues until the user agrees with the AI system. During this process, the AI will need to take into consideration the human model of the situation and the information sent by the user, differentiate it from its own, and inform the user about missing or false information. In addition, the AI's model of the human will need to be learned and updated as exchanges between the two sides occur. Most current AI systems lack components for dealing with the steps (3)-(4).

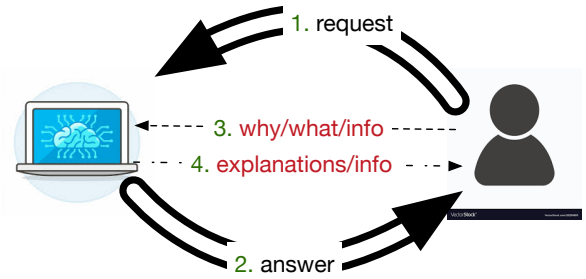

Figure 1: Interactions between AI System and Human User

In this paper, we focus on a key component needed in steps (3)-(4) in the above architecture. We propose an *explainable Answer Set Programming (ASP)* system. ASP is a programming paradigm that has been applied in several applications such as planning, diagnosis, robotics, etc. ASP is attractive as it is declarative, non-monotonic, and elaboration tolerant, and has free and scalable solvers. Thus far, only limited attention has been paid to explaining the output of an ASP execution. The proposed system will take a program P and a model A of P and explain "why an atom is true/false in A." In addition, the system will also produce explanation graphs for atoms in an ASP program.

The paper is organized as follows. We will start with a short illustration of answer set programming with focus on why ASP is an appropriate paradigm for the development of AI applications and when is it most appropriate for using ASP. Afterwards, we will define the notion of a justification (explanation graph) of an atom in a given answer set. We will then describe a system for generating explanations for answer set programming and present some initial applications.

2. PRELIMINARIES

2.1 Answer Set Programming (ASP)

(ASP)[1,2] is a declarative programming paradigm based on logic programming under the answer set semantics. A logic program P is a set of rules of the form

$$c \leftarrow a_1, \ldots, a_m, \; not\; b_1, \ldots, \; not\; b_n \tag{1}$$

where c, a_i's, and b_j's are atoms of a propositional language* and *not* represents (default) negation. Intuitively, a rule states that if all a_i are believed to be true and none of the b_j is believed to be true then c will be true. For a rule r, r^+ and r^- denote the sets $\{a_1, \ldots, a_m\}$ and $\{b_1, \ldots, b_n\}$, respectively. We write $head(r)$ and $body(r)$ to denote c and the right side of a rule r. Both the head and the body or a rule r can be empty; when the body is empty, the rule is called a fact; when the head is empty, it is a constraint. We use H to denote the Herbrand base of a logic program P, which is the set of all ground atoms in P.

Let P be a program. An interpretation I of P is a subset of H. I satisfies an atom a ($I \models a$) if $a \in I$. The body of a rule r is satisfied by I if $r^+ \subseteq I$ and $r^- \cap I = \emptyset$. A rule r is satisfied by I if $I \models head(r)$ or $I \not\models body(r)$. I is a model of P if it satisfies all rules in P.

For an interpretation I and a program P, the *reduct* of P w.r.t. I (denoted by P^I) is the program obtained from P by deleting *(i)* each rule r such that $r^- \cap I \neq \emptyset$, and *(ii)* all elements of the form $not\; a$ in the bodies of the remaining rules. Given an interpretation I, observe that the program P^I is a program with no occurrences of $not\; a$. An interpretation I is an *answer set* of P if I is the least model (w.r.t. $\subseteq$) of P^I.[3] Answer sets of logic programs can be computed using efficient and scalable answer set solvers, such as *clingo*.[4]

We illustrate the concepts of answer set programming by showing how the 3-coloring problem of a undirected graph G can be solved using ASP. Let the three colors be red (r), blue (b), and green (g), and the vertices of G be $0, 1, \ldots, n$. Let $P(G)$ be the program consisting of

- the set of atoms $edge(u, v)$ and $edge(v, u)$ for every pair of connected vertices u, v in G,
- for each vertex u of G, three rules stating that u must be assigned one of the colors red, blue, or green ($colored(x, c)$ denotes that the node x is colored with the color c):

$$\begin{aligned} colored(u, g) &\leftarrow not\; colored(u, b), \; not\; colored(u, r) \\ colored(u, r) &\leftarrow not\; colored(u, b), \; not\; colored(u, g) \\ colored(u, b) &\leftarrow not\; colored(u, r), \; not\; colored(u, g) \end{aligned}$$

and

- for each edge (u, v) of G, three rules representing the constraint that u and v must have different color:

$$\begin{aligned} &\leftarrow colored(u, r), colored(v, r), edge(u, v) \\ &\leftarrow colored(u, b), colored(v, b), edge(u, v) \\ &\leftarrow colored(u, g), colored(v, g), edge(u, v) \end{aligned}$$

It can be shown that for each graph G, (i) $P(G)$ does not have answer sets iff the 3-coloring problem of G does not have a solution; and (ii) if $P(G)$ has solutions, then each answer set of $P(G)$ corresponds to a solution of the 3-coloring problem of G and vice versa.

*For simplicity, we often use first order logic atoms, as a representation of all of their ground instantiations.

2.2 Properties of ASP

Given a program P and an answer set A, the following properties hold:

1. If $c \in A$, there exists a rule r in P such that

   - $head(r) = c$;
   - $r^+ \subseteq A$; and,
   - $r^- \cap A = \emptyset$

   For such a rule r, we define $support(c, r) = \{p \mid p \in A \wedge p \in r^+\} \cup \{\sim n \mid n \notin A \wedge n \in r^-\}$ and refer to this set as a *supported set* of c for rule r.

2. If $c \notin A$, for every rules r such that $head(r) = c$, then

   - $r^+ \setminus A \neq \emptyset$; or,
   - $r^- \cap A \neq \emptyset$

   For such a rule r, we define $support(\sim c, r) \in \{\{p\} \mid p \in A \wedge p \in r^-\} \cup \{\{\sim n\} \mid n \notin A \wedge n \in r^+\}$ and refer to it as a supported set of $\sim c$ for rule r.

2.3 Explanation Graph

We rely on the notion of an off-line justification introduced by[5] as an explanation of an atom a given an answer set A of a program P. $NANT(P) = \{a \mid a \in r^- \wedge r \in P\}$ is the set of all negation atoms in P. Let $C(P)$ denote the set of cautious consequences of P, i.e., $C(P) = C^+ \cup C^-$ where C^+ is the set of atoms belonging to all answer sets of P and C^- is the set of atoms which do not belong to any answer set of P. A set of atoms U such that $U \subseteq NANT(P) \setminus (A \cup C(P))$ is called a set of assumptions with respect to A if $A = C(P \setminus \{r \in P \mid head(r) \in U\})$.

Intuitively, given an answer set A of a program P, an atom $a \in A$ ($a \notin A$) is considered to be true (false) given A. An off-line justification for an atom a presents a possible reason for the truth value of a, i.e., it answers the question "*why $a \in A$ (or $\notin A$)?*". If a is true in A, an off-line justification of a represents a derivation of a from the set of assumptions U and the set of facts in P. If a is false in A, an off-line justification encodes the reason why it is not supported by A, which can be that it is assumed to be false (being an assumption in U) or there exists no possible derivation of it given A. An off-line justification of a is analogous to the well-known SLDNF tree in that it represents the derivation for a. The key difference between these two notions is that an off-line justification might contain a cycle consisting of negative atoms, i.e., atoms not belonging to A. For this reason, a justification is represented as an explanation graph, defined as follows.

Definition 2.1 (Explanation Graph). *Let us consider a program P, an answer set A, and a set of assumptions U with respect to A. Let $N = \{x \mid x \in A\} \cup \{\sim x \mid x \notin A\} \cup \{\top, \bot, \texttt{assume}\}$ where $\top$ and $\bot$ represent true and false, respectively. An* explanation graph *of an atom a occurring in P is a finite labeled and directed graph $DG_a = (N_a, E_a)$ with $N_a \subseteq N$ and $E_a \subseteq N_a \times N_a \times \{+, -, \circ\}$, where $(x, y, z) \in E_a$ represents a link from x to y with the label z, and satisfies the following conditions:*

- *if $a \in A$ then $a \in N_a$ and every node in N_a must be reachable from a;*
- *if $a \notin A$ then $\sim a \in N_a$ and every node in N_a must be reachable from $\sim a$;*
- *if $(x, \top, +) \in E_a$ then x is a fact in P;*
- *if $(\sim x, \texttt{assume}, \circ) \in E_a$ then $x \in U$;*
- *if $(\sim x, \bot, +) \in E_a$ then there exists no rule in P whose head is x;*
- *there exists no x, y such that $(\top, x, y) \in E_a$, $(\bot, x, y) \in E_a$, or $(\texttt{assume}, x, y) \in E_a$;*
- *for every $x \in N_a \cap A$ and x is not a fact in P,*

- *there exists no* $y \in N_a \cap A$ *such that* $(x, y, -)$ *or* $(x, y, \circ)$ *belong to* E_a*;*
- *there exists no* $\sim y \in N_a \cap \{\sim u \mid u \notin A\}$ *such that* $(x, \sim y, +)$ *or* $(x, \sim y, \circ)$ *belong to* E_a*;*
- *if* $X^+ = \{a \mid (x, a, +) \in E_a\}$ *and* $X^- = \{a \mid (x, \sim a, -) \in E_a\}$ *then* $X^+ \subseteq A$*,* $X^- \cap A = \emptyset$*, and there is a rule* $r \in P$ *whose head is* x *such that* $r^+ = X^+$ *and* $r^- = X^-$*; and*
- DG_a *contains no cycle containing* x*.*

- *for every* $\sim x \in N_a \cap \{\sim u \mid u \notin A\}$ *and* $x \notin U$*,*
  - *there exists no* $y \in N_a \cap A$ *such that* $(\sim x, y, +)$ *or* $(\sim x, y, \circ)$ *belong to* E_a*;*
  - *there exists no* $\sim y \in N_a \cap \{\sim u \mid u \notin A\}$ *such that* $(\sim x, \sim y, -)$ *or* $(\sim x, \sim y, \circ)$ *belong to* E_a*;*
  - *if* $X^+ = \{a \mid (\sim x, a, -) \in E_a\}$ *and* $X^- = \{a \mid (\sim x, \sim a, +) \in E_a\}$ *then* $X^+ \subseteq A$*,* $X^- \cap A = \emptyset$*, and for every rule* $r \in P$ *whose head is* x *we have that* $r^+ \cap X^- \neq \emptyset$ *or* $r^- \cap X^+ \neq \emptyset$*; and*
  - *any cycle containing* $\sim x$ *in* DG_a *contains only nodes in* $N_a \cap \{\sim u \mid u \notin A\}$*.*

Given an explanation graph G and a node x in G, if x is an atom a then the nodes directly connected to a—the nodes y such that $(x, y, \_)$ is an edge in G—represent a rule whose head is x and whose body is satisfied by A. If x is $\sim x$ for some atom x, then the set of nodes directly connected to $\sim x$ represents a set of atoms who truth values in A are such to make each rule whose head is x unsatisfied by A. In other words, the direct connections with a node represent the *support* for the node being in (or not in) the answer set under consideration. We refer the readers to the paper by Pontelli et al.[5] for an in-depth discussion of properties of off-line justifications and the proof of existence of such justifications for every atom in the program.

Given a program P , an answer set A of P, and an atom a occurring in P, explanation graphs for a can be generated by (i) determining the set of assumptions U with respect to A; (ii) generating explanation graphs for a using P, A, and U following its definition.

Fig. 2 illustrates the above definitions for the graph coloring program, given the graph $G = (\{1, 2, 3, 4\}, \{(1, 2), (1, 3), (2, 3), (2, 4), (3, 4)\})$ and a solution on the left, represented by the answer set containing $\{colored(1, red), colored(2, blue), colored(3, green), colored(4, red)\}$.

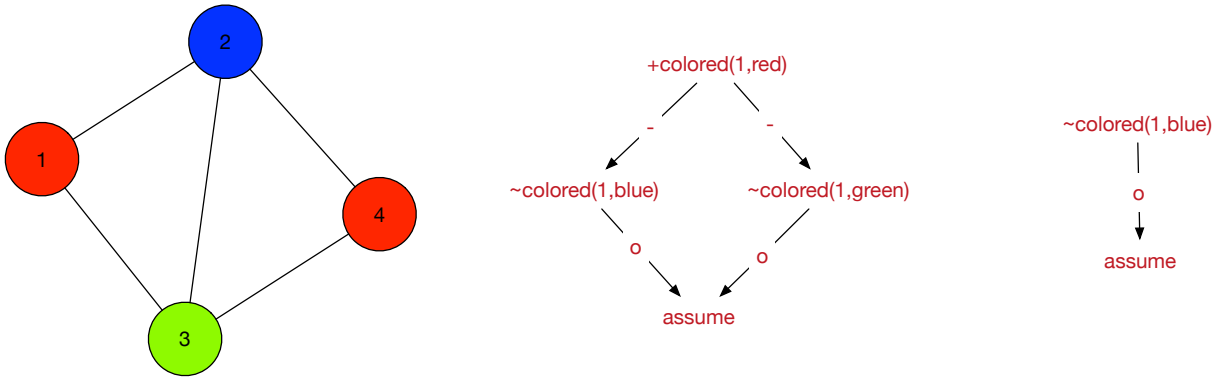

Figure 2. A solution to the 3-coloring problem for a graph on the (left), an explanation for positive atom $colored(1, red)$ (middle), and an explanation for negative atom $colored(1, blue)$ (middle)

For our computation in the next section, we define a *derivation path* of an atom a as follows.

Definition 2.2 (Derivation Path). *Given a program* P*, an answer set* A*, a derivation path of an atom* a *is a directed graph* $D_a = (N_a, E_a)$ *where* $N_a \subseteq \{x \mid x \in A\} \cup \{\sim x \mid x \notin A\} \cup \{\top, \bot\}$ *and* $E_a \subseteq N_a \times N_a \times \{+, -\}$*, where* $(x, y, z) \in E_a$ *represents a link from* x *to* y *with the label* z*, and satisfies the following conditions:*

- *if* $\{(\sim x, \bot, +) \in E_a\}$ *then there exists no rule in* P *whose head is* x*;*

- *there exists no* x, y *such that* $(\top, x, y) \in E_a$ *and* $(\bot, x, y) \in E_a$*;*
- *for every* $x \in N_a \cap A$ *and* x *is not a fact in* P*,*
  - *there exists no* $y \in N_a \cap A$ *such that* $(x, y, -)$*;*
  - *there exists no* $\sim y \in N_a \cap \{\sim u \mid u \notin A\}$ *such that* $(x, \sim y, +)$ *belongs to* E_a*;*
  - *if* $X^+ = \{a' \mid (x, a', +) \in E_a\}$ *and* $X^- = \{a' \mid (x, \sim a', -) \in E_a\}$ *then* $X^+ \subseteq A$ *and* $X^- \cap A = \emptyset$ *and there is a rule* r *whose head is* x *in* P *such that* $r^+ = X^+$ *and* $r^- = X^-$*;*
  - D_a *contains no cycle containing* x*.*
- *for every* $\sim x \in N_a \cap \{\sim u \mid u \notin A\}$*,*
  - *there exists no* $y \in N_a \cap A$ *such that* $(\sim x, y, +)$ *belongs to* E_a*;*
  - *there exists no* $\sim y \in N_a \cap \{\sim u \mid u \notin A\}$ *such that* $(\sim x, \sim y, -)$ *belongs to* E_a*;*
  - *if* $X^+ = \{a' \mid (\sim x, a', -) \in E_a\}$ *and* $X^- = \{a' \mid (\sim x, \sim a', +) \in E_a\}$ *then* $X^+ \subseteq A$ *and* $X^- \cap A = \emptyset$ *and for every rule* r *whose head is* x *in* P *we have that* $r^+ \cap X^- \neq \emptyset$ *or* $r^- \cap X^+ \neq \emptyset$*;*
  - *any cycle containing* $\sim x$ *in* D_a *contains only node in* $N_a \cap \{\sim u \mid u \notin A\}$*.*

Note that derivation path is different from the explanation graph. The purpose of derivation graph is to compute U in Section 3.2, in order to develop the explanation graphs in Section 3.3.

3. `exp(ASP)`: A SYSTEM FOR GENERATING EXPLANATIONS FOR ASP-PROGRAMS

In this section, we describe the algorithm implemented in the system `exp(ASP)` that generates explanations for ASP-programs (Section 3.1, 3.2, and 3.3). Given a program P, the preprocessing steps in Section 3.1 are used to obtain simplified ground rules from the answer set solver `clingo` and compute support sets for each rule. Given an answer set A of program P, the algorithms in Section 3.2 compute minimal assumption sets with respect to A. Algorithms in Section 3.3 use a minimal assumption set obtained from Section 3.2 to provide the explanation graphs for an atom in P. For simplicity of the presentation, the discussion in this section assumes an arbitrary but fixed program P, unless otherwise specified.

3.1 Preprocessing

In this development, `clingo` is utilized as an off-the-shelf tool. Given a program P, simplified ground rules of P are computed via the combination of the `--text` and `--keep-facts` options and an intermediate language *aspif*.[6] First, by directly utilizing the `--text` and `--keep-facts` options, plain text format are obtained. Then, a set of facts F is extracted from such plain text. Next, each fact $f \in F$ is modified to become an external statement $(\#external\ f)$ of the program. Facts in P are replaced by the external statements, creating a modified program P'. By doing this, we prevent `clingo` from simplifying rules when computing the *aspif* representation of P' via its grounder, `gringo`.[7] Let us illustrate this process using Example 1. The *aspif* statements are given in Listing 1.

EXAMPLE 1. *Let us consider the program* P_1 *containing the rules:*

(r_1) `a :- k, not b.` (r_2) `k :- e, not b.` (r_3) `c :- a, b.` (r_4) `b :- not a.` (r_5) `c :- k.` (r_6) `f :- e, not k, not c.` (r_7) `e.`

The program P_1' *is as follows:*

(r_1) `a :- k, not b.` (r_2) `k :- e, not b.` (r_3) `c :- a, b.` (r_4) `b :- not a.` (r_5) `c :- k.` (r_6) `f #external e.` (r_7) `:- e, not k, not c.`

A rule of the form (1) in Sec. 2.1, in *aspif* format, is of the form:

$$1\ H\ B$$

where H describes the head of the rule, starting with 0, and has the form:

$$0\ n\ i_c$$

where $n = \{0, 1\}$ is the number of head atoms and i_c is an integer identifying the atom c. If the head is empty, $n = 0$ and $i_c = 0$.

The body of the rule is described by B and has the form:

$$0\ k\ i_{a_1}\ ...\ i_{a_m}\ -i_{b_1}\ ...\ -i_{b_n}$$

where i_{a_i}'s and i_{b_i}'s are the integer identifiers of atoms a_i's and b_i's, respectively, and $k = n + m$. For instance, in Listing 1, line 3 represents rule r_4.

The lines starting with 4 represent the mapping of atoms to their unique integer identifiers and have the form:

$$4\ m\ a\ 1\ i$$

where m is the length in bytes of atom a and i is a's identifier. For instance, line 9 in Listing 1 shows that 2 is an integer identifying atom b.

Because we had changed facts to external statements in the modified program, facts are represented in lines starting with 5 and have the form:

$$5\ i\ 2$$

where i is an integer identifier of a fact. For instance, Line 2 in Listing 1 (and Line 12) shows that atom e is associated with the integer identifying 1 is a fact.

Listing 1. Representing P_1' in *aspif* format

```
1  asp 1 0 0
2  5 1 2
3  1 0 1 2 0 1 -3
4  1 0 1 4 0 2 -2 1
5  1 0 1 3 0 2 -2 4
6  1 0 1 5 0 2 2 3
7  1 0 1 5 0 1 4
8  1 0 1 6 0 3 -5 -4 1
9  4 1 b 1 2
10 4 1 k 1 4
11 4 1 a 1 3
12 4 1 e 1 1
13 4 1 c 1 5
14 4 1 f 1 6
15 0
```

We implement `exp(ASP)` using Python. Thus, from now on, whenever we mention a dictionary, we refer to a Python data structure dictionary. For a dictionary D, we use $D.keys()$ and $D[k]$ to the set of keys in D and the value associated with k in D, respectively.

We use the *aspif* representation of the modified program P' to create a dictionary $D_P = \{h \mapsto B \mid h \in H, B = \{body(r) \mid r \in P, head(r) = h\}\}$. For each rule r, $body(r)$ consists of r^+ and r^-. D_P is then used as an input in Algorithm 1 to compute the dictionary $E_P = \{k \mapsto V \mid k \in \{a \mid a \in A\} \cup \{\sim a \mid a \notin A\}, V = \{support(k, r) \mid r \in P\}\}$. Algorithm 1 first checks the truth value of atom $a \in H$ (recall that H is the Herbrand base of the given program). If a is true w.r.t A (Line 4), the function *true_atom_processing* is used to compute

Algorithm 1: *preprocessing*(D, F, A)

Input: D - dictionary of rules (this is D_P), F - facts of the program, A - an answer set

$E \leftarrow \{\emptyset \mapsto \emptyset\}$ // Initialize an empty dictionary E: $E.key() = \emptyset$
for $a \in H$ **do**
  $bodies \leftarrow D[a]$
  if $a \in A$ **then**
    $E[a] = []$ // Initialize an empty list for the value stored with a in E
    for $body \in bodies$ **do**
      $E \leftarrow true\_atoms\_processing(a, r^+, r^-, E, A, F)$
  else
    $E[\sim a] = []$ // Initialize an empty list for the value stored with $\sim a$ in E
    for $body \in bodies$ **do**
      $E \leftarrow false\_atoms\_processing(a, r^+, r^-, E, A)$
return E

function *true_atoms_processing*(a, r^+, r^-, E, A, F)
if $r^+ \subseteq A \wedge r^- \cap A = \emptyset$ **then**
  $S \leftarrow \{p \mid p \in r^+\} \cup \{\sim n \mid n \in r^-\}$
  if $a \in F \wedge r^+ = \emptyset \wedge r^- = \emptyset$ **then**
    $S \leftarrow \{\text{“T”}\}$
  Append S to $E[a]$
return E

function *false_atoms_processing*(a, r^+, r^-, E, A)
$L \leftarrow [\{\{\sim p\} \mid p \in r^+ \wedge p \notin A\} \cup \{\{n\} \mid n \in r^- \wedge n \in A\}]$
if $\sim a \notin E.keys()$ **then**
  $E[\sim a] \leftarrow L$
else
  $T = []$ // Initialize an empty list T
  for $s \in L$ **do**
    for $e \in E[\sim a]$ **do**
      $t \leftarrow s \cup e$
      Append t to a list T
  $E[\sim a] \leftarrow T$
return E

the supported set for a, following the first property in Sect. 2.2. Otherwise, the function *false_atom_processing* is used to compute the supported set for a, following the second property in Sect. 2.2.

EXAMPLE 2. *Let us reconsider the program P_1 of Example 1 which has an answer set $A = \{b, f, e\}$. The output of Algorithm 1, dictionary E_{P_1}, is as follows:*

```
E = {  f : [{~k, ~c, e}]
       b : [{~a}]
      ~c : [{~k, ~a}]
      ~k : [{b}]
      ~a : [{~k}, {b}]
       e : [{T}]
    }
```

As can be seen from Example 2, E_{P_1} contains supported sets for the keys in E_{P_1}. For instance, atom e in rule r_2 is not in any supported set for atom b, so e does not appear in $E_{P_1}[b]$'s value.

We will also use the *aspif* representation to compute $NANT(P)$ (Section 2.3). Note that each negative atom is represented by a negative integer whose absolute value is in the symbol table. For instance, Line 3 in Listing 1 shows that the atom associated with integer 3 is a negative atom, and Line 11 in Listing 1 shows that 3 is integer identifying atom a; thus, for P_1, we have $a \in NANT(P_1)$, and $NANT(P_1) = \{a, b, c, k\}$.

3.2 Minimal Assumption Set

In this section, we compute all minimal assumption sets U w.r.t A that satisfy conditions of an assumption set in Sect. 2.3. For every $u \in U$, u is false in A and does not belong to the well-founded model and it must be assumed to be false to guarantee that cycles in the explanation graphs are acceptable.[5]

Algorithm 2 shows the pseudo-code for computing TU which consists of all minimal assumption sets U. The cautious consequence $C(P)$ is computed using `clingo`. $NANT(P)$ and E are obtained in Sec. 3.1. First, $TA_P(A)$, called *the tentative assumption set*, is computed in Line 2. It is the set of negative atoms that are false in A and do not belong to $C(P)$.

T and DA are computed by the function *derivation_path* (Line 3) where $T = TA_P(A) \setminus T'$. T is the set of atoms from $TA_P(A)$ and do not belong to T' which is computed by the for-loop (Line 12–26). For each $x \in T'$, there exists one derivation path of x that either depends on other atoms in $TA_P(A)$ or satisfies the cycle conditions in the definition of explanation graph. Intuitively, the atoms in T must be assumed to be false and DA contains the dependencies among atoms in $TA_P(A)$.

D' is computed in Algorithm 3. It checks whether a derivation path of an atom a is acceptable and updates the set of dependency atoms in *the tentative assumption set* for a. A derivation path is unacceptable if it has a cycle containing true atoms. This is verified by Algorithm 4.

The computation of sets of minimal atoms that break all cycles in *the tentative assumption set* is then done by the function *dependency_assumption*(DA) (Lines 30–33).

EXAMPLE 3. *Let use reconsider the Examples 1 and 2.*

- *For the program P_1, we have:*
$$TA_P(A) \quad = \quad \{c, a, k\}$$
- *As can be seen from E in Example 2, c, a and k derive from other atoms in $\{k, a\}$, $\{k\}$ and $\{a\}$, respectively. Also, there is a cycle associated to k and a. Thus, we have*
$$\begin{array}{rcl} T & = & \emptyset \\ D & = & \{\{a\}\}, \{k\}\} \end{array}$$
- *Two minimal assumption sets are as follows:*
$$\begin{array}{rcl} U_1 & = & \{a\} \\ U_2 & = & \{k\} \end{array}$$

3.3 ASP-based explanation graph

In this section, we utilize the dictionary E_P and a minimal assumption set U to provide the explanation graphs of an atom a in the program P. Algorithm 5 shows the pseudo-code for computing the explanation graphs for a. Elements in $u \in U$ is {"`assume`"} (Lines 1–2). Supported sets of a are then computed and assigned to L (Line 4). Lines 5–9 create M, a collection of dictionaries of derivation paths for a. It is then used for computing explanation graphs for a via the function *get_graph* (Lines 17–40). If the derivation graph is an explanation graph, it will be displayed as a graph on the screen.

EXAMPLE 4. *For program P_1 in Example 1, the explanation graphs of f w.r.t U_1 and U_2 are shown on the left and right of Fig. 3, respectively.*

As can be seen from Fig. 3, a justification for f depends negatively on c and k, and positively on e. Based on the minimal assumption set we choose, we receive two different explanation graphs for f. The left figure contains b in its graph while the right figure does not.

Algorithm 2: $assumption\_func(C(P), NANT(P), E)$

Input: C(P) - A cautious consequence of a program P, NANT(P) - A set of negative atoms in P, E - A dictionary computed in Algorithm 1.

1 $TU = \emptyset$ // Initialize an empty set TU
2 $TA_P(A) = \{a \mid a \in NANT(P) \wedge a \notin A \wedge a \notin C(P)\}$
3 $(T, DA) = derivation\_path(TA_P(A), E)$
4 $D = dependency\_assumption(DA)$
5 **for** $M \in D$ **do**
6 $\quad U \leftarrow M \cup T$
7 $\quad TU \leftarrow TU \cup \{U\}$
8 **return** TU
9
10 **function** $derivation\_path(TA, E)$
11 $DA \leftarrow \{\emptyset \mapsto \emptyset\}$ and $T' \leftarrow \emptyset$ // Initialize an empty dictionary DA and an empty set T'
12 **for** $a \in TA$ **do**
13 $\quad O \leftarrow TA \setminus \{a\}$
14 $\quad L \leftarrow E[\sim a]$
15 $\quad$ **for** $S \in L$ **do**
16 $\quad\quad E_a \leftarrow \{\emptyset \mapsto \emptyset\}$ // Initialize an empty dictionary E_a
17 $\quad\quad E_a[\sim a] \leftarrow [S]$
18 $\quad\quad E_a \leftarrow get\_connection(S, E, E_a)$
19 $\quad\quad M \leftarrow \{N_i \mid N_i = k \mapsto V \mid V \in E_a[k] \wedge \forall k \in E_a.keys(), k \in N_i.keys() \wedge (k \mapsto V_1 \in N_i \wedge k \mapsto V_2 \in N_i) \Rightarrow V_1 = V_2\}$
20 $\quad\quad$ **for** $N_i \in M$ **do**
21 $\quad\quad\quad V \leftarrow [], R \leftarrow \emptyset, C = \{\emptyset \mapsto \emptyset\}$ and $D = \emptyset$
22 $\quad\quad\quad (safe, D') \leftarrow check\_derivation\_path(\sim a, N_i, O, V, R, C, D)$
23 $\quad\quad\quad$ **if** $safe = True$ **then**
24 $\quad\quad\quad\quad T' \leftarrow T' \cup \{a\}$
25 $\quad\quad\quad\quad DA[a] \leftarrow D'$
26 $\quad\quad\quad\quad$ **goto line** 12
27 $T \leftarrow TA \setminus T'$
28 **return** (T, DA)
29
30 **function** $dependency\_assumption(DA)$
31 $DC \leftarrow \{J \mid J = \{u_1, u_2, ..., u_n \mid u_i \in DA.keys() \wedge u_{i+1} \in DA[u_{i<n}]\} \wedge u_1 \in D[u_n]\}$
32 $B \leftarrow \{\{j_1, ..., j_n\} \mid (j_1, ..., j_n) \in J_1 \times .. \times J_n \wedge n = \left|DC\right| \wedge J_i \in DC\}$
33 $min(B) \leftarrow \{M \mid \forall C \in B, C \neq M \implies \left|M\right| \leq \left|C\right|\}$
34 **return** $min(B)$
35
36 **function** $get\_connection(S, E, E_a)$
37 **for** $e \in S$ **do**
38 $\quad$ **if** $e \notin E_a.keys() \wedge e \in E.keys()$ **then**
39 $\quad\quad E_a[e] \leftarrow E[e]$
40 $\quad\quad$ **for** $S_e \in E[e]$ **do**
41 $\quad\quad\quad E_a \leftarrow get\_connection(S_e, E, E_a)$
42 **return** E_a

Algorithm 3: *check_derivation_path*(k, N, O, V, R, C, Da)

```
I ← ∅                                                    // Initialize an empty set I
if k ∈ N.keys() then
    I ← N[k]
V ← V ∪ {k}
Append k to list R
for i ∈ I do
    C[k] ← i
    j ← i.remove(i.index(0))
    if i.index(0) = " ~ " ∧ j ∈ O then
        Da ← Da ∪ {j}
    else
        if i ∉ V then
            if check_derivation_path(i, N, O, V, R, C, Da)[0] = False then
                return (False,Da)
        else
            if i ∈ R then
                if i.index(0) = " ~ " then
                    if cycle_identification(C, i, i) = False then
                        return (False,Da)
                else
                    return (False,Da)
l = R.pop()
if l ∈ C.keys() then
    C.pop(l)
return (True,Da)
```

Algorithm 4: *cycle_identification*(C, s, e)

```
v ← C[s]
if v ≠ e ∧ s.index(0) = " ~ " ∧ v.index(0) = " ~ " then
    return cycle_identification(C, v, e)
if v = e ∧ s.index(0) = " ~ " ∧ v.index(0) = " ~ " then
    return True
return False
```

4. ILLUSTRATION

We have validated a prototype implementation of this system on a number of ASP programs, ranging from simple benchmarks to complex ASP applications (e.g., we used it to interpret the responses produced by a diagnosis system modeling the Three Mile Island disaster[8]). In this paper, we illustrate the use of `exp(ASP)` in two examples from the literature.

EXAMPLE 5. *Consider the following problem, which represents a variant of the agent problem described by Garcia et al.*[9] *An agent, named Bob, wants to make plan for the upcoming week. Bob has two choices: stay home or go to the opera. Bob will stay at home on Monday. He will also stay at home with his best friend because she has a*

Algorithm 5: *explanation_graph*(a, E, U)

```
for u ∈ U do
    E[∼u] ← ["assume"]
a′ ← a if a ∈ A and a′ ←∼a if a ∉ A
L ← E[a′]
for S ∈ L do
    E_a ← {∅ ↦ ∅}                                   // Initialize an empty dictionary E_a
    E_a[a′] ← [S]
    E_a ← get_connection(S, E, E_a)
    M ← {N_i | N_i = k ↦ V | V ∈ E_a[k] ∧ ∀k ∈ E_a.keys(), k ∈ N_i.keys() ∧ (k ↦ V_1 ∈ N_i ∧ k ↦ V_2 ∈
      N_i) ⟹ V_1 = V_2}
    for N_i ∈ M do
        V ← [], R ← ∅, C ← {∅ ↦ ∅}, and an empty graph G ← (∅, ∅)
        Add node a′ to G
        graph ← get_graph(a′, G, N_i, V, R, C)
        if graph ≠ False then
            Draw graph

function get_graph(k, G, N, V, R, C)
I ← ∅                                               // Initialize an empty set I
if k ∈ N.keys() then
    I ← N[k]
V ← V ∪ {k}
Append k to list R
for i ∈ I do
    Add node i to G
    e ← (k, i, sign)
    Add edge e to G
    if i ∉ V then
        if get_graph(i, G, N, V, R, C) = False then
            return False
    else
        if i ∈ R then
            if i.index(0) = " ∼ " then
                if cycle_identification(C, i, i) = False then
                    return False
            else
                return False
l = R.pop()
if l ∈ C.keys() then
    C.pop(l)
return G
```

baby, and the friend comes this Tuesday. The situation can be encoded as follows:

```
day(monday).        day(tuesday).       day(wednesday).
day(thursday).      day(friday).        day(saturday).
day(sunday).        home(monday).       baby(tuesday).
opera(D)      :-    day(D), not home(D).
home(D)       :-    day(D), not opera(D).
home(D)       :-    day(D), baby(D).
```

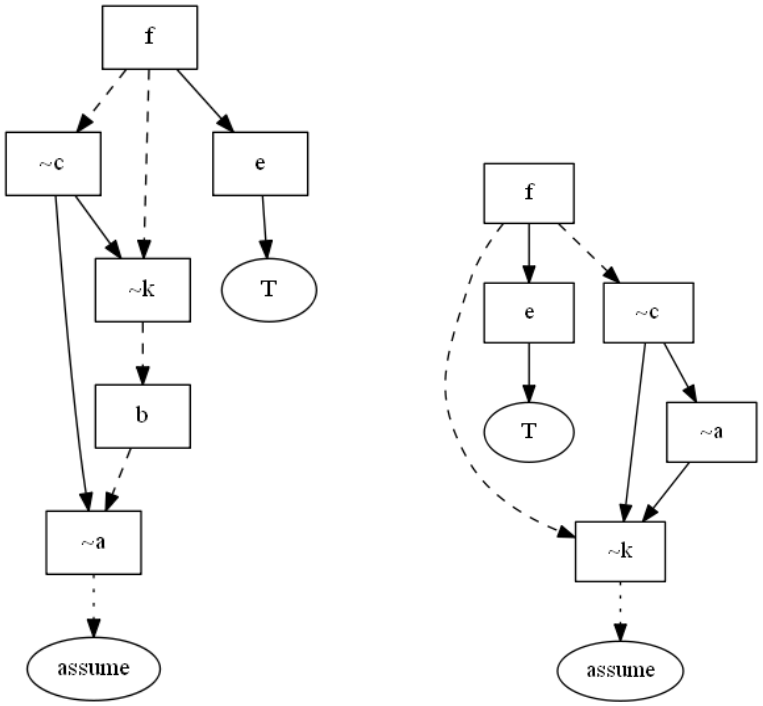

Figure 3. Explanation graph of a w.r.t U_1 (left) and U_2 (right)

The above program has 32 answer sets, i.e., Bob has 32 possible plans. Let us consider the answer set $A = \{day(monday), day(tuesday), day(wednesday), day(thursday), day(friday), day(saturday), day(sunday), home(monday), home(tuesday), baby(tuesday), opera(wednesday), opera(thursday), opera(friday), opera(saturday), opera(sunday)\}$.

Some explanation graphs w.r.t A are displayed below:

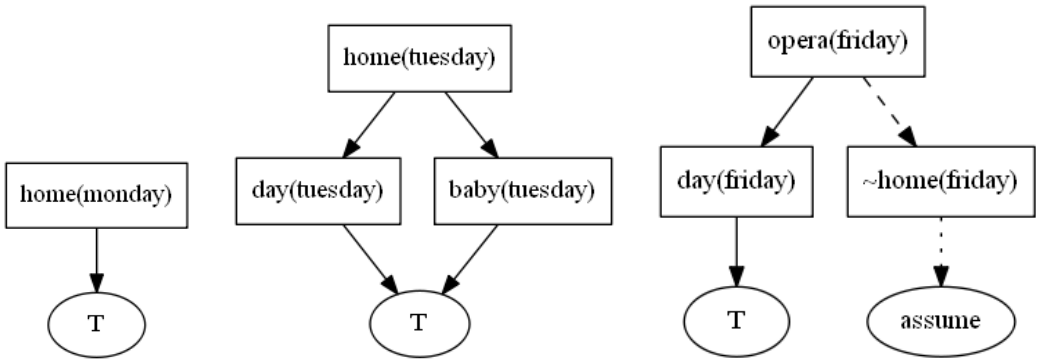

Figure 4. Explanation of $home(monday)$ (left), $home(tuesday)$ (middle) and $opera(friday)$ (right)

- *Fig. 4 (left) shows that Bob will stay at home on Monday because of the provided information in the situation that* $home(monday)$ *is a fact.*
- *Fig. 4 (middle) shows that Bob will stay at home on Tuesday because his best friend and her baby will come to his house on Tuesday (*$baby(tuesday)$ *is a fact).*

- *Fig. 4 (right) shows that Bob will go to opera on Friday because it is assumed to be false that he will not stay at home on Friday.*

EXAMPLE 6. *Consider the problem of decision making in an ophthalmologist diagnostic system.*[10] *We need to provide a suggestion to Peter. Peter is short-sighted. He is afraid to touch his eyes. He is a student and likes sport. The information about Peter is encoded by the following facts.*

```
shortSighted.      afraidToTouchEyes.
student.           likesSports.
```

Other information about Peter is given by the following program:

```
tightOnMoney       :-  student, not richParents.
caresPracticality  :-  likesSports.
correctiveLens     :-  shortSighted, not laserSurgery.
laserSurgery       :-  shortSighted, not tightOnMoney, not correctiveLens.
glasses            :-  correctiveLens, not caresPracticality,
                       not contactLens.
contactLens        :-  correctiveLens, not afraidToTouchEyes,
                       not longSighted, not glasses.
intraocularLens    :-  correctiveLens, not glasses, not contactLens.
```

This program, together with the facts, has an answer set

$A = \{tightOnMoney, shortSighted, caresPracticality, afraidToTouchEyes, student, likesSports, correctiveLens, intraocularLens\}$.

Therefore, intraocularLens is suggested to Peter. The explanation graph for this recommendation is depicted in Fig. 5.

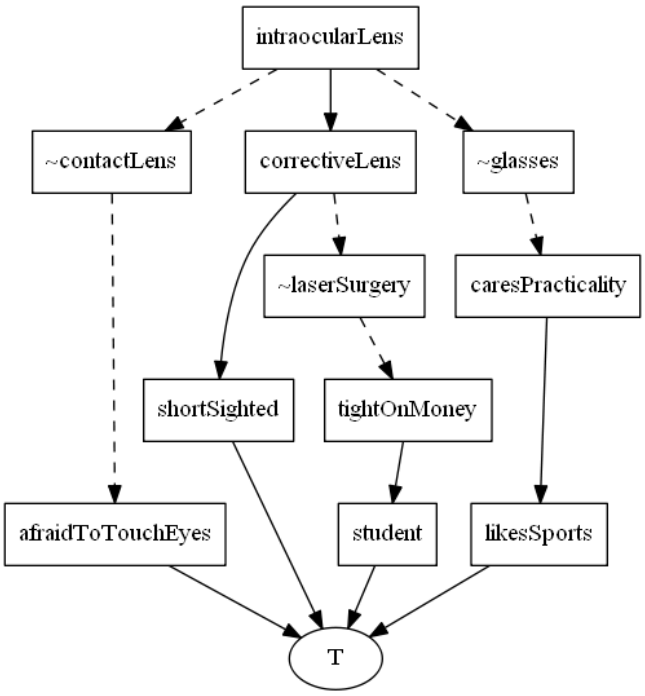

Figure 5. Explanation of *intraocularLens*

As can be seen from Fig. 5, intraocularLens is in the answer set because:

- *The truth value of correctiveLens is True, because*
  - *It has information that Peter is short-sighted (shortSighted is a fact).*
  - *Negatively it depends on the truth value of laserSurgery which is False. This latter truth value derives from the fact that laserSurgery depends negatively on the truth value of tightOnMoney which is True. This is because it has information that Peter is student (student is a fact).*
- *The truth value of contactLens is False, because*
  - *Negatively it depends on the truth value of afraidToTouchEyes which is a fact.*
- *The truth value of glasses is False, because*
  - *Negatively it depends on the truth value of caresPracticality which is True, because*
    - *It has information that Peter likes sports (likeSports is a fact).*

5. CONCLUSION

In this paper, we described an explanation generation system for ASP programs, `exp(ASP)`, and illustrate its use in some examples. Our future goal is to use the proposed system in explainable planning with the full cycle of four steps (1)-(4) in Fig. 1.

ACKNOWLEDGMENTS

This research is partially supported by NSF grants 1757207, 1914635, and 1812619/26. Portions of this publication and research effort are made possible through the help and support of NIST via cooperative agreement 70NANB19H102. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the sponsoring organizations, agencies, or the U.S. government.

REFERENCES

[1] Marek, V. and Truszczyński, M., "Stable models and an alternative logic programming paradigm," in [*The Logic Programming Paradigm: a 25-year Perspective*], 375–398 (1999).

[2] Niemelä, I., "Logic programming with stable model semantics as a constraint programming paradigm," *Annals of Mathematics and Artificial Intelligence* **25**(3,4), 241–273 (1999).

[3] Gelfond, M. and Lifschitz, V., "Logic programs with classical negation," in [*Logic Programming: Proceedings of the Seventh International Conference*], Warren, D. and Szeredi, P., eds., 579–597 (1990).

[4] Gebser, M., Kaufmann, B., Neumann, A., and Schaub, T., "clasp: A conflict-driven answer set solver," in [*Proceedings of the Ninth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'07)*], Baral, C., Brewka, G., and Schlipf, J., eds., *Lecture Notes in Artificial Intelligence* **4483**, 260–265, Springer-Verlag (2007).

[5] Pontelli, E., Son, T., and El-Khatib, O., "Justifications for logic programs under answer set semantics," *TPLP* **9**(1), 1–56 (2009).

[6] Kaminski, R., Schaub, T., and Wanko, P., "A tutorial on hybrid answer set solving with clingo," in [*Reasoning Web International Summer School*], 167–203, Springer (2017).

[7] Gebser, M., Schaub, T., and Thiele, S., "Gringo: A new grounder for answer set programming," in [*International Conference on Logic Programming and Nonmonotonic Reasoning*], 266–271, Springer (2007).

[8] Hanna, B. N., T Trieu, L. L., Son, T. C., and Dinh, N. T., "An application of asp in nuclear engineering: Explaining the three mile island nuclear accident scenario," *Theory and Practice of Logic Programming* **20**(6), 926–941 (2020).

[9] García, A. J., Chesñevar, C. I., Rotstein, N. D., and Simari, G. R., "Formalizing dialectical explanation support for argument-based reasoning in knowledge-based systems," *Expert Systems with Applications* **40**(8), 3233–3247 (2013).

[10] Schulz, C. and Toni, F., "Justifying answer sets using argumentation," *Theory and Practice of Logic Programming* **16**(1), 59–110 (2016).