TLP **15** (1): 35–78, 2015. © Cambridge University Press 2013
doi:10.1017/S1471068413000598 First published online 17 December 2013

Generating explanations for biomedical queries

ESRA ERDEM and UMUT OZTOK
Sabancı University, Orhanlı, Tuzla, İstanbul 34956, Turkey
(*e-mail:* {esraerdem,uoztok}@sabanciuniv.edu)

submitted 25 October 2012; revised 17 July 2013; accepted 19 August 2013

Abstract

We introduce novel mathematical models and algorithms to generate (shortest or k different) explanations for biomedical queries, using answer set programming. We implement these algorithms and integrate them in BioQuery-ASP. We illustrate the usefulness of these methods with some complex biomedical queries related to drug discovery, over the biomedical knowledge resources PharmGKB, DrugBank, BioGRID, CTD, SIDER, Disease Ontology, and Orphadata.

KEYWORDS: answer set programming, explanation generation, query answering, biomedical queries

1 Introduction

Recent advances in health and life sciences have led to generation of a large amount of biomedical data, represented in various biomedical databases or ontologies. That these databases and ontologies are represented in different formats and constructed/maintained independently from each other at different locations, have brought about many challenges for answering complex biomedical queries that require integration of knowledge represented in these ontologies and databases. One of the challenges for the users is to be able to represent such a biomedical query in a natural language, and get its answers in an understandable form. Another challenge is to extract relevant knowledge from different knowledge resources, and integrate them appropriately using also definitions, such as, chains of gene–gene interactions, cliques of genes based on gene–gene relations, or similarity/diversity of genes/drugs. Furthermore, once an answer is found for a complex query, the experts may need further explanations about the answer.

Table 1 displays a list of complex biomedical queries that are important from the point of view of drug discovery. In the queries, drug–drug interactions present negative interactions among drugs, and gene–gene interactions present both negative and positive interactions among genes. Consider, for instance, the query Q6. New molecule synthesis by changing substitutes of parent compound may lead to different biochemical and physiological effects; and each trial may lead to different indications. Such studies are important for fast inventions of new molecules. For example, while

Table 1. *A list of complex biomedical queries*

Q1	What are the drugs that treat the disease Asthma and that target the gene ADRB1?
Q2	What are the side effects of the drugs that treat the disease Asthma and that target the gene ADRB1?
Q3	What are the genes that are targeted by the drug Epinephrine and that interact with the gene DLG4?
Q4	What are the genes that interact with at least three genes and that are targeted by the drug Epinephrine?
Q5	What are the drugs that treat the disease Asthma or that react with the drug Epinephrine?
Q6	What are the genes that are targeted by all the drugs that belong to the category Hmg-coa reductase inhibitors?
Q7	What are the cliques of five genes, that contain the gene DLG4?
Q8	What are the genes that are related to the gene ADRB1 via a gene–gene interaction chain of length at most 3?
Q9	What are the three most similar genes that are targeted by the drug Epinephrine?
Q10	What are the genes that are related to the gene DLG4 via a gene–gene interaction chain of length at most 3 and that are targeted by the drugs that belong to the category Hmg-coa reductase inhibitors?
Q11	What are the drugs that treat the disease Depression and that do not target the gene ACYP1?
Q12	What are the symptoms of diseases that are treated by the drug Triadimefon?
Q13	What are the three most similar drugs that target the gene DLG4?
Q14	What are the three closest drugs to the drug Epinephrine?

developing the drug Lovastatin (a member of the drug class of Hmg-coa reductase inhibitors, used for lowering cholesterol) from *Aspergillus terreus* (a sort of fungus) in 1979, scientists at Merck derived a new molecule named Simvastatin that also belongs to the same drug category (a hypolipidemic drug used to control elevated cholesterol) targeting the same gene. Therefore, identifying genes targeted by a group of drugs automatically by means of queries like Q6 may be useful for experts.

Once an answer to a query is found, the experts may ask for an explanation to have a better understanding. For instance, an answer for the query Q3 in Table 1 is "ADRB1." A shortest explanation for this answer is as follows:

> The drug Epinephrine targets the gene ADRB1 according to CTD.
> The gene DLG4 interacts with the gene ADRB1 according to BioGRID.

An answer for the query Q8 is "CASK." A shortest explanation for this answer is as follows:

> The distance of the gene CASK from the start gene is 2.
> The gene CASK interacts with the gene DLG4 according to BioGRID.
> The distance of the gene DLG4 from the start gene is 1.
> The gene DLG4 interacts with the gene ADRB1 according to BioGRID.
> ADRB1 is the start gene.

(Statements with more indentations provide explanations for statements with less indentations.)

https://doi.org/10.1017/S1471068413000598 Published online by Cambridge University Press

To address the first two challenges described above (i.e., representing complex queries in natural language and finding answers to queries efficiently), novel methods and a software system, called BioQuery-ASP (Erdem *et al.* 2011) (Fig. 1), have been developed using answer set programming (ASP) (Marek and Truszczyński 1999; Niemelä 1999; Lifschitz 2002; Baral 2003; Lifschitz 2008; Brewka *et al.* 2011):

- Erdem and Yeniterzi (2009) developed a controlled natural language, BioQuery-CNL, for expressing biomedical queries related to drug discovery. For instance, queries Q1–Q10 in Table 1 are in this language. Recently, this language has been extended (called BioQuery-CNL*) to cover queries Q11–Q13 (Oztok 2012). Some algorithms have been introduced to translate a given query in BioQuery-CNL (respectively BioQuery-CNL*) to a program in ASP as well.
- Bodenreider *et al.* (2008) introduced methods to extract biomedical information from various knowledge resources and integrate them by a rule layer. This rule layer not only integrates those knowledge resources but also provides definitions of auxiliary concepts.
- Erdem *et al.* (2011) have introduced an algorithm for query answering by identifying the relevant parts of the rule layer and the knowledge resources with respect to a given query.

The details of representing biomedical queries in natural language and answering them using ASP are explained in a companion article. The focus of this article is the last challenge: generating explanations for biomedical queries.

Most of the existing biomedical querying systems (e.g., web services built over the available knowledge resources) support keyword search but not complex queries like the queries in Table 1. None of the existing systems can provide informative explanations about the answers, but point to related web pages of the knowledge resources available online.

The contributions of this article can be summarized as follows.

- We have formally defined "explanations" in ASP, utilizing properties of programs and graphs. We have also defined variations of explanations, such as "shortest explanations" and "k different explanations."
- We have introduced novel generic algorithms to generate explanations for biomedical queries. These algorithms can compute shortest or k different explanations. We have analyzed the termination, soundness, and complexity of those algorithms.
- We have developed a computational tool, called ExpGen-ASP, that implements these explanation generation algorithms.
- We have showed the applicability of our methods to generate explanations for answers of complex biomedical queries related to drug discovery.
- We have embedded ExpGen-ASP into BioQuery-ASP so that the experts can obtain explanations regarding the answers of biomedical queries, in a natural language.

The rest of the article is organized as follows. In Section 2, we provide a summary of ASP. Next, in Section 3, we give an overview of BioQuery-ASP, in particular

https://doi.org/10.1017/S1471068413000598 Published online by Cambridge University Press

the earlier work done on answering biomedical queries in ASP. Then, in Sections 4–6, we provide some definitions and algorithms related to generating shortest or k different explanations for an answer, also in ASP. Next, Section 7 illustrates the usefulness of these algorithms on some complex queries over the biomedical knowledge resources PharmGKB (McDonagh *et al.* 2011)[1], DrugBank (Knox *et al.* 2010)[2], BioGRID (Stark *et al.* 2006)[3], CTD (Davis *et al.* 2011)[4], SIDER (Kuhn *et al.* 2010)[5], Disease Ontology (Schriml *et al.* 2012)[6], and Orphadata[7]. In Sections 8 and 9, we discuss how to present explanations to the user in a natural language, and embedding of these algorithms in BioQuery-ASP. In Section 10, we provide a detailed analysis of the related work on "justifications" (Pontelli *et al.* 2009) in comparison to explanations; and in Section 11, we briefly discuss other related work. We conclude in Section 12 by summarizing our contributions and pointing out some possible future work. Proofs are provided in the online Appendix of the article.

2 Answer set programming

ASP (Marek and Truszczyński 1999; Niemelä 1999; Lifschitz 2002; Baral 2003; Lifschitz 2008; Brewka *et al.* 2011) is a form of declarative programming paradigm oriented towards solving combinatorial search problems as well as knowledge-intensive problems. The idea is to represent a problem as a "program" whose models (called "answer sets" (Gelfond and Lifschitz 1988; Gelfond and Lifschitz 1991)) correspond to the solutions. The answer sets for the given program can then be computed by special implemented systems called answer set solvers. ASP has a high-level representation language that allows recursive definitions, aggregates, weight constraints, optimization statements, and default negation.

ASP also provides efficient solvers, such as clasp (Gebser *et al.* 2007). Due to the continuous improvement of the ASP solvers and highly expressive representation language of ASP that is supported by a strong theoretical background that results from years of intensive research, ASP has been applied fruitfully to a wide range of areas. Here are, for instance, three applications of ASP used in industry:

- *Decision Support Systems*: An ASP-based system was developed to help flight controllers of space shuttle solve some planning and diagnostic tasks (Nogueira *et al.* 2001) (used by United Space Alliance).
- *Automated Product Configuration*: A web-based commercial system uses an ASP-based product configuration technology (Tiihonen *et al.* 2003) (used by Variantum Oy).

[1] http://www.pharmgkb.org/
[2] http://www.drugbank.ca/
[3] http://thebiogrid.org/
[4] http://ctd.mdibl.org/
[5] http://sideeffects.embl.de/
[6] http://disease-ontology.org
[7] http://www.orphadata.org

https://doi.org/10.1017/S1471068413000598 Published online by Cambridge University Press

- *Workforce Management*: An ASP-based system is developed to build teams of employees to handle incoming ships by taking into account a variety of requirements, e.g., skills, fairness, regulations (Ricca *et al.* 2012) (used by Gioia Tauro seaport).

Let us briefly explain the syntax and semantics of ASP programs and describe how a computational problem can be solved in ASP.

2.1 Programs

Syntax The input language of ASP programs is composed of three sets namely *constant symbols*, *predicate symbols*, and *variable symbols*, where intersection of constant symbols and variable symbols is empty. The basic elements of the ASP programs are *atoms*. An atom $p(\vec{t})$ is composed of a predicate symbol p and *terms* $\vec{t} = t_1, \ldots, t_k$ where each t_i $(1 \leqslant i \leqslant k)$ is either a constant or a variable. A *literal* is either an atom $p(\vec{t})$ or its negated form *not* $p(\vec{t})$.

An ASP program is a finite set of *rules* of the form:

$$A \leftarrow A_1, \ldots, A_k, not\ A_{k+1}, \ldots, not\ A_m \tag{1}$$

where $m \geqslant k \geqslant 0$ and each A_i is an atom, whereas A is an atom or $\bot$.

For a rule r of the form (1), A is called the *head* of the rule and denoted by $H(r)$. The conjunction of the literals $A_1, \ldots, A_k, not\ A_{k+1}, \ldots, not\ A_m$ is called the *body* of r. The set $\{A_1, ..., A_k\}$ of atoms (called the positive part of the body) is denoted by $B^+(r)$, and the set $\{A_{k+1}, ..., A_m\}$ of atoms (called the negative part of the body) is denoted by $B^-(r)$, and all the atoms in the body are denoted by $B(r) = B^+(r) \cup B^-(r)$.

We say that a rule r is a *fact* if $B(r) = \emptyset$, and we usually omit the $\leftarrow$ sign. Furthermore, we say that a rule r is a *constraint* if the head of r is $\bot$, and we usually omit the $\bot$ sign.

Semantics (Answer Sets) Answer sets of a program are defined over *ground programs*. We call an atom, rule, or program *ground*, if it does not contain any variables. Given a program Π, the set $\mathscr{U}_\Pi$ represents all the constants in Π, and the set $\mathscr{B}_\Pi$ represents all the ground atoms that can be constructed from atoms in Π with constants in $\mathscr{U}_\Pi$. Also, *Ground*(Π) denotes the set of all the ground rules that are obtained by substituting all variables in rules with the set of all possible constants in $\mathscr{U}_\Pi$.

A subset I of $\mathscr{B}_\Pi$ is called an *interpretation* for Π. A ground atom p is true with respect to an interpretation I if $p \in I$; otherwise, it is false. Similarly, a set S of atoms is true (respectively, false) with respect to I if each atom $p \in S$ is true (respectively, false) with respect to I. An interpretation I *satisfies* a ground rule r, if $H(r)$ is true with respect to I whenever $B^+(r)$ is true and $B^-(r)$ is false with respect to I. An interpretation I is called a *model* of a program Π if it satisfies all the rules in Π.

The *reduct* Π^I of a program Π with respect to an interpretation I is defined as follows:

$$\Pi^I = \{H(r) \leftarrow B^+(r) \mid r \in Ground(\Pi)\ s.t.\ I \cap B^-(r) = \emptyset\}.$$

https://doi.org/10.1017/S1471068413000598 Published online by Cambridge University Press

An interpretation I is an *answer set* for a program Π, if it is a subset-minimal model for Π^I, and $AS(\Pi)$ denotes the set of all the answer sets of a program Π.

For example, consider the following program Π_1:

$$p \leftarrow not\ q \tag{2}$$

and take an interpretation $I = \{p\}$. The reduct Π_1^I is as follows:

$$p. \tag{3}$$

The interpretation I is a model of the reduct (3). Let us take a strict subset I' of I, which is $\emptyset$. Then, the reduct $\Pi_1^{I'}$ is again equal to (3); however, I' does not satisfy (3). Therefore, $I = \{p\}$ is a subset-minimal model; hence an answer set of Π_1. Note also that $\{p\}$ is the only answer set of Π.

2.2 Generate-and-test representation methodology with special ASP constructs

The idea of ASP (Lifschitz 2008) is to represent a computational problem as a program whose answer sets correspond to the solutions of the problem, and to find the answer sets for that program using an answer set solver.

When we represent a problem in ASP, two kinds of rules play an important role: those that "generate" many answer sets corresponding to "possible solutions," and those that can be used to "eliminate" the answer sets that do not correspond to solutions. The rules

$$\begin{array}{l} p \leftarrow not\ q \\ q \leftarrow not\ p \end{array} \tag{4}$$

are of the former kind: they generate the answer sets $\{p\}$ and $\{q\}$. Constraints are of the latter kind. For instance, adding the constraint

$$\leftarrow p$$

to program (4) eliminates the answer sets for the program that contain p.

In ASP, we use special constructs of the form

$$\{A_1, \ldots, A_n\}^c \tag{5}$$

(called *choice expressions*), and of the form

$$l \leqslant \{A_1, \ldots, A_m\} \leqslant u \tag{6}$$

(called *cardinality expressions*) where each A_i is an atom and l and u are nonnegative integers denoting the "lower bound" and the "upper bound" (Simons *et al.* 2002). Programs using these constructs can be viewed as abbreviations for normal nested programs defined in Ferraris and Lifschitz (2005). Expression (5) describes subsets of $\{A_1, \ldots, A_n\}$. Such expressions can be used in heads of rules to generate many answer sets. For instance, the answer sets for the program

$$\{p, q, r\}^c \ \leftarrow \tag{7}$$

are arbitrary subsets of $\{p, q, r\}$. Expression (6) describes the subsets of the set $\{A_1, \ldots, A_m\}$ whose cardinalities are at least l and at most u. Such expressions can be used in constraints to eliminate some answer sets. For instance, adding the constraint

https://doi.org/10.1017/S1471068413000598 Published online by Cambridge University Press

$$\leftarrow \ 2 \leqslant \{p, q, r\}$$

to program (7) eliminates the answer sets for (7) whose cardinalities are at least 2. We abbreviate the rules

$$\{A_1, \ldots, A_m\}^c \leftarrow Body$$
$$\leftarrow not\ (l \leqslant \{A_1, \ldots, A_m\})$$
$$\leftarrow not\ (\{A_1, \ldots, A_m\} \leqslant u)$$

by the rule

$$l \leqslant \{A_1, \ldots, A_m\}^c \leqslant u \leftarrow Body.$$

In ASP, there are also special constructs that are useful for optimization problems. For instance, to compute answer sets that contain the maximum number of elements from the set $\{A_1, \ldots, A_m\}$, we can use the following optimization statement:

$$maximize\langle\{A_1, \ldots, A_m\}\rangle.$$

2.3 Presenting programs to answer set solvers

Once we represent a computational problem as a program whose answer sets correspond to the solutions of the problem, we can use an answer set solver to compute the solutions of the problem. To present a program to an answer set solver, like CLASP, we need to make some syntactic modifications.

Recall that answer sets for a program are defined over ground programs. Thus, the input of ASP solvers should be ground instantiations of the programs. For that, programs go through a "grounding" phase in which variables in the program (if exists) are substituted by constants. For CLASP, we use the "grounder" GRINGO (Gebser *et al.* 2011).

Although the syntax of the input language of GRINGO is somewhat more restricted than the class of programs defined above, it provides a number of useful special constructs. For instance, the head of a rule can be an expression of one of the forms

$$\{A_1, \ldots, A_n\}^c$$
$$l \leqslant \{A_1, \ldots, A_n\}^c$$
$$\{A_1, \ldots, A_n\}^c \leqslant u$$
$$l \leqslant \{A_1, \ldots, A_n\}^c \leqslant u$$

but the superscript ^{c} and the sign $\leqslant$ are dropped. The body can also contain cardinality expressions but the sign $\leqslant$ is dropped. In the input language of GRINGO, `:-` stands for $\leftarrow$, and each rule is followed by a period. For facts $\leftarrow$ is dropped. For instance, the rule

$$1 \leqslant \{p, q, r\}^c \leqslant 1 \leftarrow$$

can be presented to GRINGO as follows:

```
1{p,q,r}1.
```

Variables in a program are represented by strings whose initial letters are capitalized. The constants and predicate symbols, on the other hand, start with

https://doi.org/10.1017/S1471068413000598 Published online by Cambridge University Press

a lowercase letter. For instance, the program Π_n

$$p_i \leftarrow not \quad p_{i+1} \qquad (1 \leqslant i \leqslant n)$$

can be presented to GRINGO as follows:

```
index(1..n).
p(I) :- not p(I+1), index(I).
```

Here, the auxiliary predicate `index` is a "domain predicate" used to describe the ranges of variables. Variables can be also used "locally" to describe the list of formulas. For instance, the rule

$$1 \leqslant \{p_1, \ldots, p_n\} \leqslant 1$$

can be expressed in GRINGO as follows

```
index(1..n).
1{p(I) : index(I)}1.
```

3 Answering biomedical queries

We have earlier developed the software system BIOQUERY-ASP (Erdem *et al.* 2011) (see Fig. 1) to answer complex queries that require appropriate integration of relevant knowledge from different knowledge resources and auxiliary definitions such as chains of drug–drug interactions, cliques of genes based on gene–gene relations, or similar/diverse genes. As depicted in Figure 1, BIOQUERY-ASP takes a query in a controlled natural language and transforms it into ASP. Meanwhile, it extracts knowledge from biomedical databases and ontologies, and integrates them in ASP. Afterwards, it computes an answer to the given query using an ASP solver.

Let us give an example to illustrate these stages; the details of representing biomedical queries in natural language and answering them using ASP are explained in a companion article though.

First of all, let us mention that knowledge related to drug discovery is extracted from the biomedical databases/ontologies and represented in ASP. If the biomedical ontology is in RDF(S)/OWL, then we can extract such knowledge using the ASP solver DLVHEX (Eiter *et al.* 2006) by making use of external predicates. For instance, consider as an external theory a Drug Ontology described in RDF. All triples from this theory can be exported using the external predicate `&rdf`:

```
triple_drug(X,Y,Z) :- &rdf["URI for Drug Ontology"](X,Y,Z).
```

Then the names of drugs can be extracted by DLVHEX using the rule:

```
drug_name(A) :- triple_drug(_,"drugproperties:name",A).
```

Some knowledge resources are provided as relational databases, or more often as a set of triples (probably extracted from ontologies in RDF). In such cases, we use short scripts to transform the relations into ASP.

To relate the knowledge extracted from the biomedical databases or ontologies and also provide auxiliary definitions, a rule layer is constructed in ASP. For instance,

https://doi.org/10.1017/S1471068413000598 Published online by Cambridge University Press

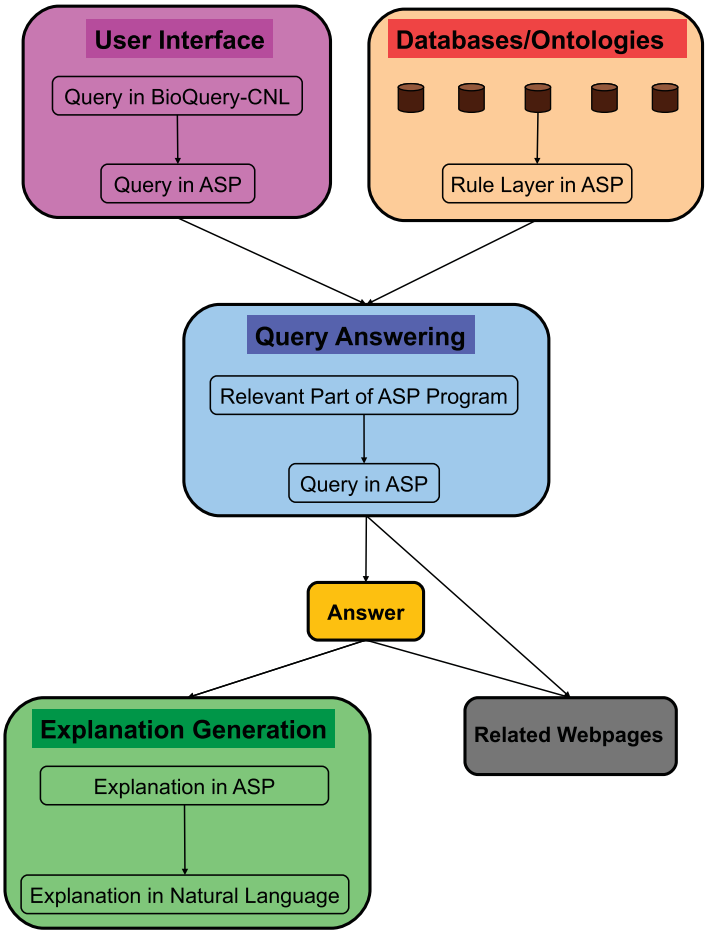

Fig. 1. System overview of BioQuery-ASP.

drugs targeting genes are described by the relation `drug_gene` defined in the rule layer as follows:

```
drug_gene(D,G) :- drug_gene_pharmgkb(D,G).
drug_gene(D,G) :- drug_gene_ctd(D,G).
```

where `drug_gene_pharmgkb` and `drug_gene_ctd` are relations for extracting knowledge from relevant knowledge resources. The auxiliary concept of reachability of a gene from another gene by means of a chain of gene–gene interactions is defined in the rule layer as well:

```
gene_reachable_from(X,1) :- gene_gene(X,Y), start_gene(Y).
gene_reachable_from(X,N+1) :- gene_gene(X,Z),
   gene_reachable_from(Z,N), 0 < N, N < L,
   max_chain_length(L).
```

Now, consider, for instance, the query Q11 from Table 1.

https://doi.org/10.1017/S1471068413000598 Published online by Cambridge University Press

Q11 What are the drugs that treat the disease Depression and that do not target the gene ACYP1?

This type of queries might be important in terms of drug repurposing (Chong and Sullivan 2007) that has achieved a number of successes in drug development, including the famous example of Pfizer's Viagra (Gower 2009).

This query is then translated into the following program in the language of GRINGO:

```
what_be_drugs(DRG) :-  cond1(DRG), cond2(DRG).
cond1(DRG) :- drug_disease(DRG,"Depression").
cond2(DRG) :- drug_name(DRG), not drug_gene(DRG,"ACYP1").
answer_exists :- what_be_drugs(DRG).
:- not answer_exists.
```

where `cond1` and `cond2` are invented relations, and `drug_name`, `drug_disease`, and `drug_gene` are defined in the rule layer.

Once the query and the rule layer are in ASP, the parts of the rule layer that are relevant to the given query are identified by an algorithm (Erdem *et al.* 2011). For some queries, the relevant part of the program is almost 100 times smaller than the whole program (considering the number of ground rules).

Then, given the query as an ASP program and the relevant knowledge as an ASP program, we can find answers to the query by computing an answer set for the union of these two programs using CLASP. For the query above an answer computed in this way is "Fluoxetine."

4 Explaining an answer for a query

Once an answer is found for a complex biomedical query, the experts may need informative explanations about the answer, as discussed in the Introduction. With this motivation, we study generating explanations for complex biomedical queries. Since the queries, knowledge extracted from databases and ontologies, and the rule layer are in ASP, our studies focus on explanation generation within the context of ASP.

Before we introduce our methods to generate explanations for a given query, let us introduce some definitions regarding explanations in ASP.

Let Π be the relevant part of a ground ASP program with respect to a given biomedical query Q (also a ground ASP program) that contains rules describing the knowledge extracted from biomedical ontologies and databases, the knowledge integrating them, and the background knowledge. Rules in $\Pi \cup Q$ generally do not contain cardinality/choice expressions in the head; therefore, we assume that in $\Pi \cup Q$ only bodies of rules contain cardinality expressions. Let X be an answer set for $\Pi \cup Q$. Let p be an atom that characterizes an answer to the query Q. The goal is to find an "explanation" as to why p is computed as an answer to the query Q, i.e., why is p in X? Before we introduce a definition of an explanation, we need the following notations and definitions.

https://doi.org/10.1017/S1471068413000598 Published online by Cambridge University Press

We say that a set X of atoms *satisfies* a cardinality expression C of the form

$$l \leqslant \{A_1, \ldots, A_m\} \leqslant u$$

if the cardinality of $X \cap \{A_1, \ldots, A_m\}$ is within the lower bound l and upper bound u. Also X *satisfies* a set SC of cardinality expressions (denoted by $X \models SC$), if X satisfies every element of SC.

Let Π be a ground ASP program, r be a rule in Π, p be an atom in Π, and Y and Z be two sets of atoms. Let $B_{card}(r)$ denote the set of cardinality expressions that appear in the body of r. We say that r *supports* an atom p using atoms in Y but not in Z (or with respect to Y but Z), if the following hold:

$$\begin{aligned} &H(r) = p, \\ &B^+(r) \subseteq Y \backslash Z, \\ &B^-(r) \cap Y = \emptyset, \\ &Y \models B_{card}(r). \end{aligned} \tag{8}$$

We denote the set of rules in Π that support p with respect to Y but Z, by $\Pi_{Y,Z}(p)$.

We now introduce definitions about explanations in ASP. We first define a generic tree whose vertices are labeled by either atoms or rules.

Definition 1 (*Vertex-labeled tree*)
A *vertex-labeled tree* $\langle V, E, l, \Pi, X\rangle$ for a program Π and a set X of atoms is a tree $\langle V, E\rangle$ whose vertices are labeled by a function l that maps V to $\Pi \cup X$. In this tree, the vertices labeled by an atom (respectively, a rule) are called *atom vertices* (respectively, *rule vertices*).

For a vertex-labeled tree $T = \langle V, E, l, \Pi, X\rangle$ and a vertex v in V, we introduce the following notations:

- $anc_T(v)$ denotes the set of atoms that are labels of ancestors of v.
- $des_T(v)$ denotes the set of rule vertices that are descendants of v.
- $child_E(v)$ denotes the set of children of v.
- $sibling_E(v)$ denotes the set of siblings of v.
- $out_E(v)$ denotes the set of out-going edges of v.
- $deg_E(v)$ denotes the degree of v and equals to $|out_E(v)|$.
- If $deg_E(v) = 0$, then v is a *leaf* vertex.
- $leaf(T)$ denotes the set of leaves of T.
- The *root* of T is the root of $\langle V, E\rangle$.
- T is *empty* if $\langle V, E\rangle = \langle\emptyset, \emptyset\rangle$.

We now define a specific class of vertex-labeled trees that contains all possible "explanations" for an atom.

Definition 2 (*And-or explanation tree*)
Let Π be a ground ASP program, X be an answer set for Π, and p be an atom in X. The *and-or explanation tree* for p with respect to Π and X is a vertex-labeled tree $T = \langle V, E, l, \Pi, X\rangle$ that satisfies the following:

(i) for the root $v \in V$ of the tree, $l(v) = p$;

https://doi.org/10.1017/S1471068413000598 Published online by Cambridge University Press

(*ii*) for every atom vertex $v \in V$,

$$out_E(v) = \{(v, v') \mid (v, v') \in E,\ l(v') \in \Pi_{X, anc_T(v')}(l(v))\};$$

(*iii*) for every rule vertex $v \in V$,

$$out_E(v) = \{(v, v') \mid (v, v') \in E,\ l(v') \in B^+(l(v))\};$$

(*iv*) each leaf vertex is a rule vertex.

Let us explain Conditions $(i) - (iv)$ in Definition 2 in detail.

(*i*) The root of the and-or explanation tree T is labeled by the atom p. Intuitively, T contains all possible explanations for p.

(*ii*) For every atom vertex $v \in V$, there is an out-going edge (v, v') to a rule vertex $v' \in V$ under the following conditions: the rule that labels v' supports the atom that labels v, using atoms in X but not any atom that labels an ancestor of v'. We want to exclude the atoms labeling ancestors of v' to ensure that the height of the and-or explanation tree is finite (e.g., otherwise, due to cyclic dependencies the tree may be infinite).

(*iii*) For every rule vertex $v \in V$, there is an out-going edge (v, v') to an atom vertex if the atom that labels v' is in the positive body of the rule that labels v. In this way, we make sure that every atom in the positive body of the rule that labels v takes part in explaining the head of the rule that labels v.

(*iv*) Together with Conditions (*ii*) and (*iii*) above, this condition guarantees that the leaves of the and-or explanation tree are rule vertices that are labeled by facts in the reduct of the given ASP program Π with respect to the given answer set X. Intuitively, this condition expresses that the leaves are self-explanatory.

Example 1
Let Π be the program

$$a \leftarrow b, c$$
$$a \leftarrow d$$
$$d \leftarrow$$
$$b \leftarrow c$$
$$c \leftarrow$$

and $X = \{a, b, c, d\}$. The and-or explanation tree for a with respect to Π and X is shown in Figure 2. Intuitively, the and-or explanation tree includes all possible "explanations" for an atom. For instance, according to Figure 2, the atom a has two explanations:

- One explanation is characterized by the rules that label the vertices in the left subtree of the root: a is in X because the rule

  $$a \leftarrow b, c$$

  supports a. Moreover, this rule can be "applied to generate a" because b and c, the atoms in its positive body, are in X. Further, b is in X because the rule

  $$b \leftarrow c$$

https://doi.org/10.1017/S1471068413000598 Published online by Cambridge University Press

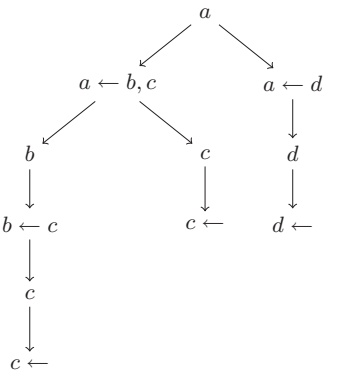

Fig. 2. The and-or explanation tree for Example 1.

supports b. Further, c is in X because c is supported by the rule

$$c \leftarrow$$

that is self-explanatory.

- The other explanation is characterized by the rules that label the vertices in the right subtree of the root: a is in X because the rule

$$a \leftarrow d$$

supports a. Further, this rule can be "applied to generate a" because d is in X. In addition, d is in X because d is supported by the rule

$$d \leftarrow$$

that is self-explanatory.

Proposition 1
Let Π be a ground ASP program and X be an answer set for Π. For every p in X, the and-or explanation tree for p with respect to Π and X is not empty.

Note that in the and-or explanation tree, atom vertices are the "or" vertices, and rule vertices are the "and" vertices. Then, we can obtain a subtree of the and-or explanation tree that contains an explanation, by visiting only one child of every atom vertex and every child of every rule vertex, starting from the root of the and-or explanation tree. Here is precise definition of such a subtree, called an explanation tree.

Definition 3 (*Explanation tree*)
Let Π be a ground ASP program, X be an answer set for Π, p be an atom in X, and $T = \langle V, E, l, \Pi, X\rangle$ be the and-or explanation tree for p with respect to Π and X. An *explanation tree* in T is a vertex-labeled tree $T' = \langle V', E', l, \Pi, X\rangle$ such that

(i) $\langle V', E'\rangle$ is a subtree of $\langle V, E\rangle$;
(ii) the root of $\langle V', E'\rangle$ is the root of $\langle V, E\rangle$;
(iii) for every atom vertex $v' \in V'$, $deg_{E'}(v') = 1$;
(iv) for every rule vertex $v' \in V'$, $out_E(v') \subseteq E'$.

https://doi.org/10.1017/S1471068413000598 Published online by Cambridge University Press

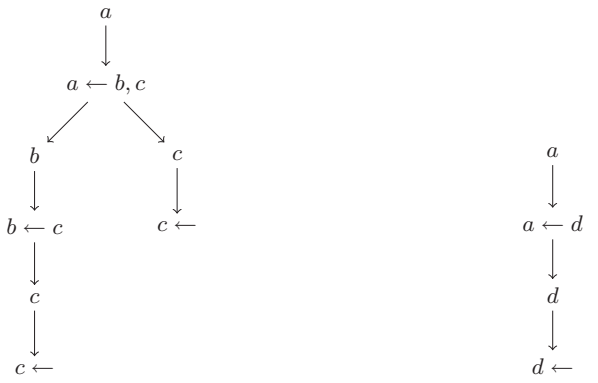

Fig. 3. Explanation trees for Example 2.

Fig. 4. Explanations for Example 3.

Example 2
Let T be the and-or explanation tree in Figure 2. Then, Figure 3 illustrates the explanation trees in T. These explanation trees characterize the two explanations for a explained in Example 1.

After having defined the and-or explanation tree and an explanation tree for an atom, let us now define an explanation for an atom.

Definition 4 (*Explanation*)
Let Π be a ground ASP program, X be an answer set for Π, and p be an atom in X. A vertex-labeled tree $\langle V', E', l, \Pi, X\rangle$ is an *explanation* for p with respect to Π and X if there exists an explanation tree $\langle V, E, l, \Pi, X\rangle$ in the and-or explanation tree for p with respect to Π and X such that

(i) $V' = \{v \mid v \text{ is a rule vertex in } V\}$;
(ii) $E' = \{(v_1, v_2) \mid (v_1, v), (v, v_2) \in E, \text{ for some atom vertex } v \in V\}$.

Intuitively, an explanation can be obtained from an explanation tree by "ignoring" its atom vertices.

Example 3
Let Π and X be defined as in Example 1. Then, Figure 4 depicts two explanations for a with respect to Π and X, described in Example 1.

So far, we have considered only positive programs in the examples. Our definitions can also be used in programs that contain negation and aggregates in the bodies of rules.

https://doi.org/10.1017/S1471068413000598 Published online by Cambridge University Press

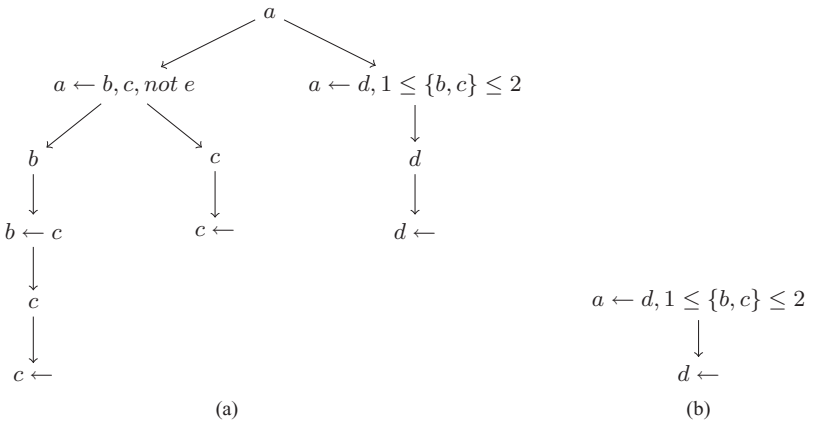

Fig. 5. (a) The and-or explanation tree for a and (b) an explanation for a.

Example 4
Let Π be the program

$$a \leftarrow b, c, not\ e$$
$$a \leftarrow d, not\ b$$
$$a \leftarrow d, 1 \leqslant \{b, c\} \leqslant 2$$
$$d \leftarrow$$
$$b \leftarrow c$$
$$c \leftarrow$$

and $X = \{a, b, c, d\}$. The and-or explanation tree for a with respect to Π and X is shown in Figure 5(a). Here, the rule $a \leftarrow d, not\ b$ is not included in the tree as b is in X, whereas the rule $a \leftarrow b, c, not\ e$ is in the tree as e is not in X and b and c are in X. Also, the rule $a \leftarrow d, 1 \leqslant \{b, c\} \leqslant 2$ is in the tree as d is in X and the cardinality expression $1 \leqslant \{b, c\} \leqslant 2$ is satisfied by X. An explanation for a with respect to Π and X is shown in Figure 5(b).

Note that our definition of an and-or explanation tree considers positive body parts of the rules only to provide explanations. Therefore, explanation trees do not provide further explanations for negated literals (e.g., why an atom is not included in the answer set), or aggregates (e.g., why a cardinality constraint is satisfied) as seen in the example above.

5 Generating shortest explanations

As can be seen in Figure 4, there might be more than one explanation for a given atom. Hence, it is not surprising that one may prefer some explanations to others. Consider biomedical queries about chains of gene–gene interactions like the query Q8 in Table 1. Answers of such queries may contain chains of gene–gene interactions with different lengths. For instance, an answer for this query is "CASK." Figure 6 shows an explanation for this answer. Here, "CASK" is related to "ADRB1" via a gene–gene chain interaction of length 2 (the chain

https://doi.org/10.1017/S1471068413000598 Published online by Cambridge University Press

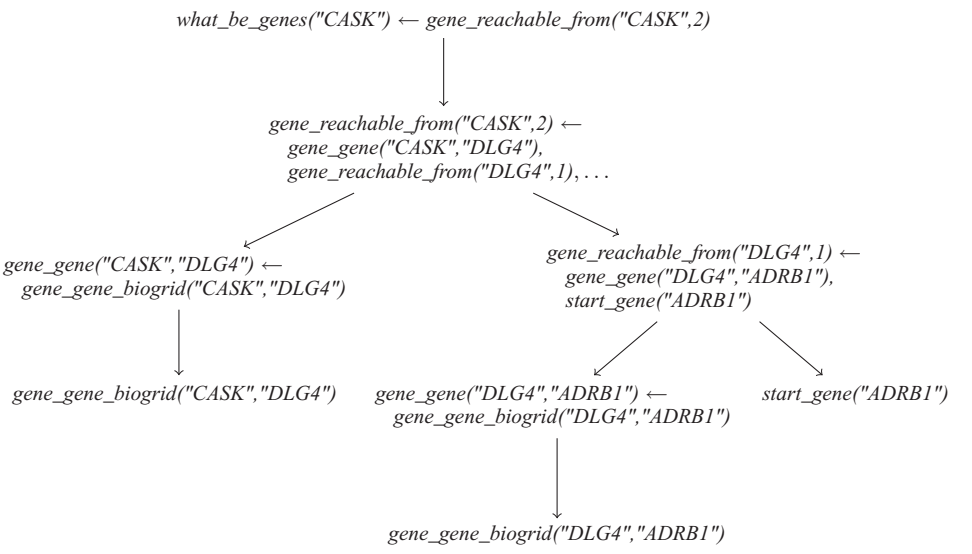

Fig. 6. Another explanation for Q8.

"CASK"–"DLG4"–"ADRB1"). Another explanation is partly shown in Figure 7. Now, "CASK" is related to "ADRB1" via a gene–gene chain interaction of length 3 (the chain "CASK"–"DLG1"–"DLG4"–"ADRB1"). Since gene–gene interactions are important for drug discovery, it may be more desirable for the experts to reason about chains with shortest lengths.

With this motivation, we consider generating shortest explanations. Intuitively, an explanation S is shorter than another explanation S' if the number of rule vertices involved in S is less than the number of rule vertices involved in S'. Then we can define shortest explanations as follows.

Definition 5 (*Shortest explanation*)
Let Π be a ground ASP program, X be an answer set for Π, p be an atom in X, and S be an explanation (with vertices V) for p with respect to Π and X. Then, S is a *shortest explanation* for p with respect to Π and X if there exists no explanation S' (with vertices V') for p with respect to Π and X such that $|V'| < |V|$.

Example 5
Let Π and X be defined as in Example 1. Then, Figure 4(b) is the shortest explanation for a with respect to Π and X.

To compute shortest explanations, we define a weight function that assigns weights to the vertices of the and-or explanation tree. Basically, the weight of an atom vertex ("or" vertex) is equal to the minimum weight among weights of its children and the weight of a rule vertex ("and" vertex) is equal to sum of weights of its children plus 1. Then the idea is to extract a shortest explanation by propagating the weights of the leaves up and then traversing the vertices that contribute to the weight of the root. Let us define the weight of vertices in the and-or explanation tree.

https://doi.org/10.1017/S1471068413000598 Published online by Cambridge University Press

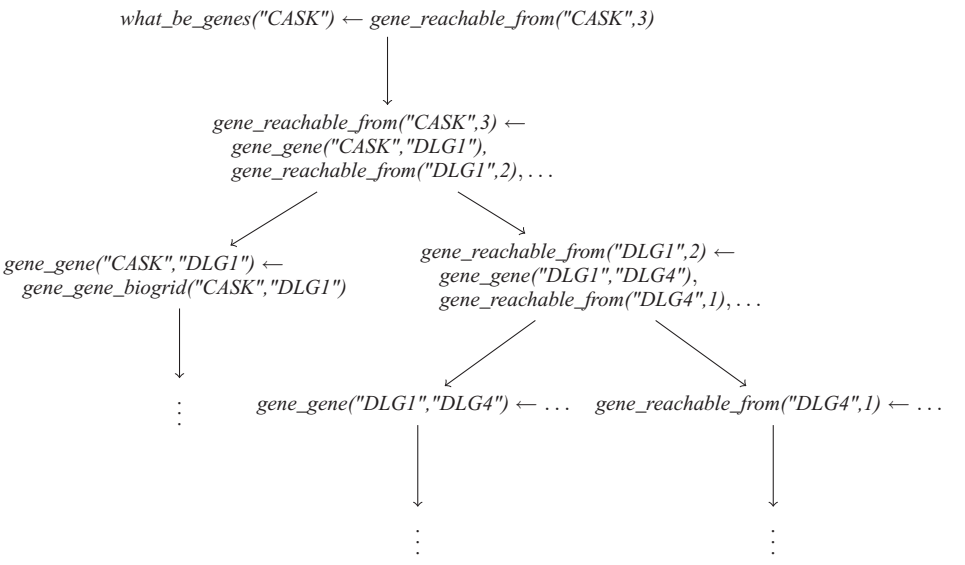

Fig. 7. A shortest explanation for Q8.

Definition 6 (*Weight function*)
Let Π be a ground ASP program, X be an answer set for Π, p be an atom in X, and $T = \langle V, E, l, \Pi, X\rangle$ be the and-or explanation tree for p with respect to Π and X. The weight function W_T for T maps vertices in V to a positive integer and it is defined as follows.

$$W_T(v) = \begin{cases} \min\{W_T(c) \mid c \in child_E(v)\} & \text{if v is an atom vertex in } V; \\ 1 + \sum_{c \in child_E(v)} W_T(c) & \text{otherwise.} \end{cases}$$

Input: Π : ground ASP program, X : answer set for Π, p : atom in X.
Output: a shortest explanation for p w.r.t Π and X, or an empty vertex-labeled tree.
$\langle V, E, l, \Pi, X\rangle$:= createTree($\Pi, X, p, \{\}$);
if $\langle V, E\rangle$ *is not empty* **then**
  $v \leftarrow$ root of $\langle V, E\rangle$;
  calculateWeight(Π, X, V, l, v, E, W_T);
  $\langle V', E', l, \Pi, X\rangle$:= extractExp($\Pi, X, V, l, v, E, W_T, \emptyset, min$);
  return $\langle V', E', l, \Pi, X\rangle$;
end
else
  return $\langle \emptyset, \emptyset, l, \Pi, X\rangle$;
end

Algorithm 1: Generating Shortest Explanations

Using this weight function, we develop Algorithm 1 to generate shortest explanations. Let us describe this algorithm. Algorithm 1 starts by creating the

https://doi.org/10.1017/S1471068413000598 Published online by Cambridge University Press

and-or explanation tree T for p with respect to Π and X (Line 1); for that it uses Algorithm 2. If T is not empty, then Algorithm 1 assigns weights to the vertices of T (Line 4), using Algorithm 3. As the final step, Algorithm 1 extracts a shortest explanation from T (Line 5), using Algorithm 4. The idea is to traverse an explanation tree of T, by the help of the weight function, and construct an explanation, which would be a shortest one, by contemplating only the rule vertices in the traversed explanation tree. If T is empty, Algorithm 1 returns an empty vertex-labeled tree.

Algorithm 2 (with the call createTree$(\Pi, X, p, \{\})$) creates the and-or explanation tree for p with respect to Π and X recursively. With a call createTree(Π, X, d, L), where L intuitively denotes the atoms labeling the atom vertices created so far, the algorithm considers two cases: d being an atom or a rule. In the former case, (1) the algorithm creates an atom vertex v for d, (2) it identifies the rules that support d, (3) for each such rule, it creates a vertex labeled tree (i.e., a subtree of the resulting and-or explanation tree), and (4) it connects these trees to the atom vertex v. In the latter case, if d is a rule in Π, (1) the algorithm creates a rule vertex v for d, (2) it identifies the atoms in the positive part of the rule, (3) it creates the and-or explanation tree for each such atom, and (4) it connects these trees to the rule vertex v.

Once the and-or explanation tree is created, Algorithm 3 assigns weights to all vertices in the tree by propagating the weights of the leaves (i.e., 1) up to the root in a bottom-up fashion using the weight function definition (i.e., Definition 6).

After that, Algorithm 1 (with the call extractExp$(\Pi, X, V, l, v, E, W_T, \emptyset, min)$) extracts a shortest explanation in a top-down fashion starting from the root by examining the weights of the vertices. In particular, if a visited vertex v is an atom vertex then the algorithm proceeds with the child of v with the minimum weight; otherwise, it considers all the children of v.

The execution of Algorithm 1 is also illustrated in Figure 8. First, the and-or explanation tree is generated, which has a generic structure as in Figure 8(a). Here, yellow vertices denote atom vertices and blue vertices denote rule vertices. Then, this tree is weighted as in Figure 8(b). Then, starting from the root, a subtree of the and-or explanation tree is traversed by visiting minimum weighted child of every atom vertex and every child of every rule vertex. This process is shown in Figure 8(c), where red vertices form the traversed subtree. From this subtree, an explanation is extracted by ignoring atom vertices and keeping the parent–child relationship of the tree as it is. The resulting explanation is depicted in Figure 8(d).

Proposition 2
Given a ground ASP program Π, an answer set X for Π, and an atom p in X, Algorithm 1 terminates.

Proposition 3
Given a ground ASP program Π, an answer set X for Π, and an atom p in X, Algorithm 1 either finds a shortest explanation for p with respect to Π and X or returns an empty vertex-labeled tree.

https://doi.org/10.1017/S1471068413000598 Published online by Cambridge University Press

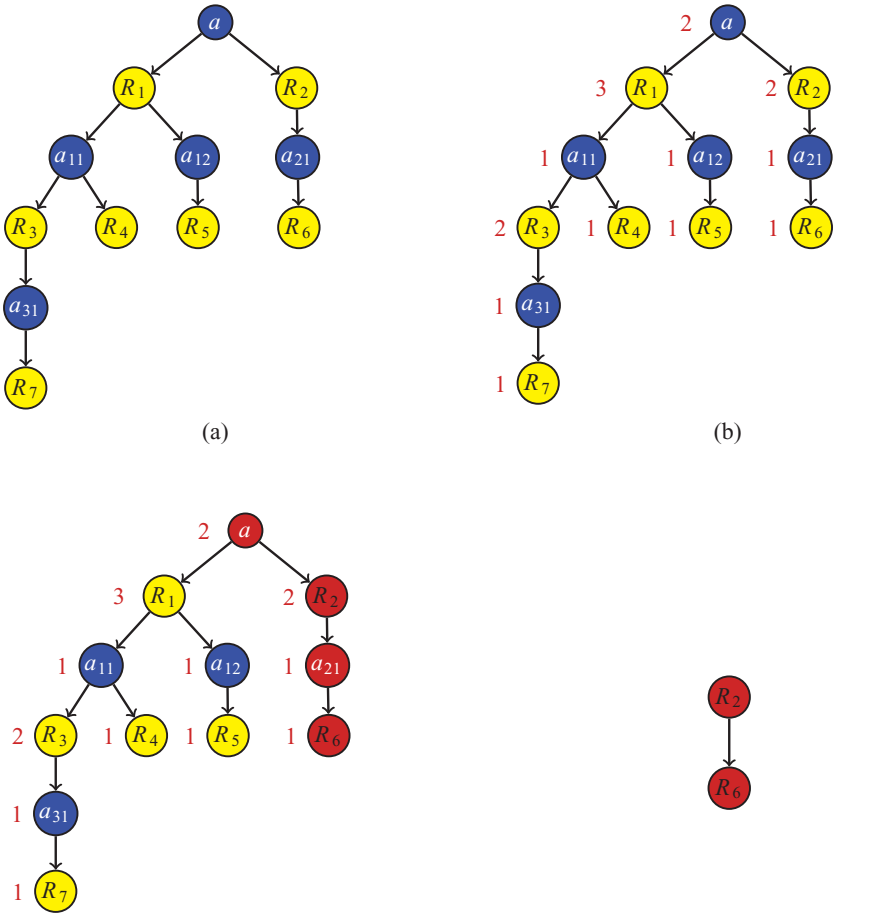

Fig. 8. A generic execution of Algorithm 1.

Proposition 4
Given a ground ASP program Π, an answer set X for Π, and an atom p in X, the time complexity of Algorithm 1 is $O(|\Pi|^{|X|} \times |\mathscr{B}_\Pi|)$.

We generate the complete and-or explanation tree while finding a shortest explanation. In fact, we can find a shortest explanation by creating a partial and-or explanation tree using a branch-and-bound idea. In particular, the idea is to compute the weights of vertices during the creation of the and-or explanation tree and, in case there exists a branch of the and-or explanation tree that exceeds the weight of a vertex computed so far, to stop branching on unnecessary parts of the and-or explanation tree. Then, a shortest explanation can be extracted by the same method used previously, i.e., by traversing a subtree of the and-or explanation tree and ignoring the atom vertices in this subtree. For instance, consider Figure 8(b). Assume that we first create the right branch of the root. Since the weight of an atom vertex is equal to the minimum weight among its children weights, we know that

https://doi.org/10.1017/S1471068413000598 Published online by Cambridge University Press

Input: Π : ground ASP program, X : answer set for Π, d : an atom in X or a rule in Π, L : set of atoms in X.
Output: A vertex-labeled tree.
$V := \emptyset,\ E := \emptyset$;
if $d \in X \setminus L$ **then**
  $v \leftarrow$ Create an atom vertex s.t. $l(v) = d$;
  $L := L \cup \{d\},\quad V := V \cup \{v\}$;
  foreach $r \in \Pi_{X,L}(d)$ **do**
    $\langle V', E', l, \Pi, X\rangle := \text{createTree}(\Pi, X, r, L)$;
    if $\langle V', E'\rangle \neq \langle \emptyset, \emptyset\rangle$ **then**
      $v' \leftarrow$ root of $\langle V', E'\rangle$ s.t. $l(v') = r$;
      $V := V \cup V',\quad E := E \cup \{(v, v')\} \cup E'$;
    end
  end
  if $E = \emptyset$ **then return** $\langle \emptyset, \emptyset, l, \Pi, X\rangle$;
end
else if $d \in \Pi$ **then**
  $v \leftarrow$ Create a rule vertex s.t. $l(v) = d$;
  foreach $a \in B^+(d)$ **do**
    $\langle V', E', l, \Pi, X\rangle := \text{createTree}(\Pi, X, a, L)$;
    if $\langle V', E'\rangle = \langle \emptyset, \emptyset\rangle$ **then return** $\langle \emptyset, \emptyset, l, \Pi, X\rangle$;
    $v' \leftarrow$ root of $\langle V', E'\rangle$ s.t. $l(v') = a$;
    $V := V \cup V',\ E := E \cup \{(v, v')\} \cup E'$;
  end
end
return $\langle V, E, l, \Pi, X\rangle$;

Algorithm 2: createTree

the weight of the root is at most 2. Now, we check whether it is necessary to branch on the left child of the root. Note that the weight of a rule vertex is equal to 1 plus the sum of its children weights. As R_1 has two children, its weight is at least 3. Therefore, it is redundant to branch on the left child of the root. This improvement is not implemented and is a future work.

6 Generating k different explanations

When there is more than one explanation for an answer of a query, it might be useful to provide the experts with several more explanations that are different from each other. For instance, consider the query Q5 in Table 1.

Q5 What are the drugs that treat the disease Asthma or that react with the drug Epinephrine?

An answer for this query is "Doxepin." According to one explanation, "Doxepin" reacts with "Epinephrine" with respect to DrugBank. At this point, the expert may

https://doi.org/10.1017/S1471068413000598 Published online by Cambridge University Press

Input: Π : ground ASP program, X : answer set for Π, V : set of vertices, $l : V \to \Pi \cup X$, v : vertex in V, E : set of edges, W_T : candidate weight function.
Output: Weight of v.
if $l(v) \in X$ **then**
  foreach $c \in child_E(v)$ **do** $W_T(c) :=$ calculateWeight$(\Pi, X, V, l, c, E, W_T)$;
  $W_T(v) := min\{W_T(c) \mid c \in child_E(v)\}$;
end
else if $l(v) \in \Pi$ **then**
  $W_T(v) := 1$;
  foreach $c \in child_E(v)$ **do**
  $W_T(v) := W_T(v) +$ calculateWeight$(\Pi, X, V, l, c, E, W_T)$;
end
return $W_T(v)$;

Algorithm 3: calculateWeight

Input: Π : ground ASP program, X : answer set for Π, V_t : set of vertices, $l : V_t \to \Pi \cup X$, v : vertex in V_t, E_t : set of edges, W_T : weight function of T, r : rule vertex in V_t or $\emptyset$, op: string *min* or *max*.
Output: A vertex-labeled tree $\langle V, E, l, \Pi, X \rangle$.
$V := \emptyset,\ E := \emptyset$;
if $l(v) \in X$ **then**
  $c \leftarrow$ Pick op weighted child of v ;
  if $r \neq \emptyset$ **then** $E := E \cup \{(r, c)\}$;
  $\langle V', E', l, \Pi, X \rangle :=$ extractExp$(\Pi, X, V_t, l, c, E_t, W_T, r, op)$;
  $V := V \cup V',\ E := E \cup E'$;
end
else if $l(v) \in \Pi$ **then**
  $V := V \cup \{v\}$;
  foreach $c \in child_{E_t}(v)$ **do**
    $\langle V', E', l, \Pi, X \rangle :=$ extractExp$(\Pi, X, V_t, l, c, E_t, W_T, v, op)$;
    $V := V \cup V',\ E := E \cup E'$;
  end
end
return $\langle V, E, l, \Pi, X \rangle$;

Algorithm 4: extractExp

not be convinced and ask for a different explanation. Another explanation for this answer is that "Doxepin" treats "Asthma" according to CTD. Motivated by this example, we study generating different explanations.

We introduce an algorithm (Algorithm 5) to compute k different explanations for an atom p in X with respect to Π and X. For that, we define a distance measure Δ_D between a set Z of (previously computed) explanations, and an (to be computed) explanation S. We consider the rule vertices R_Z and R_S contained in Z and S,

https://doi.org/10.1017/S1471068413000598 Published online by Cambridge University Press

Input: Π: ground ASP program, X: answer set for Π, p: atom in X, k : a positive integer. Assume there are n different explanations for p w.r.t Π and X.
Output: $min\{n,k\}$ different explanations for p with respect to Π and X.
$K := \emptyset$, $R_0 := \emptyset$;
$\langle V, E, l, \Pi, X\rangle$:= createTree($\Pi, X, p, \{\}$);
$v \leftarrow$ root of $\langle V, E\rangle$;
for $i = 1, 2, \ldots k$ **do**
  calculateDifference($\Pi, X, V, l, v, E, R_{i-1}, W_{T,R_{i-1}}$);
  if $W_{T,R_{i-1}}(v) = 0$ **then return** K;
  $\langle V', E', l, \Pi, X\rangle$:= extractExp($\Pi, X, V, l, v, E, W_{T,R_{i-1}}, \emptyset, max$);
  $K_i \leftarrow \langle V', E', l, \Pi, X\rangle$;
  $K := K \cup \{K_i\}$;
  $R_i := R_{i-1} \cup \{v \mid \text{rule vertex } v \in V'\}$;
end
return K;

Algorithm 5: Generating k Different Explanations

respectively. Then, we define the function Δ_D that measures the distance between Z and S as follows:

$$\Delta_D(Z, S) = |R_S \backslash R_Z|.$$

In the following, we sometimes use R_Z and R_S instead of Z and S in Δ_D. Also, we denote by $RVertices(S)$ the set of rule vertices of a vertex-labeled tree S.

Let us now explain Algorithm 5. It computes a set K of k different explanations iteratively. Initially, $K = \emptyset$. First, we compute the and-or explanation tree T (Line 2). Then, we enter into a loop that iterates at most k times (Line 4). At each iteration i, an explanation K_i that is most distant from the previously computed $i-1$ explanations is extracted from T. Let us denote the rule vertices included in the previously computed $i-1$ explanations by R_{i-1}. Then, essentially, at each iteration we pick an explanation K_i such that $\Delta_D(R_{i-1}, RVertices(K_i))$ is maximum. To be able to find such a K_i, we need to define the "contribution" of each vertex v in T to the distance measure $\Delta_D(R_{i-1}, RVertices(K_i))$ if v is included in explanation K_i:

$$W_{T,R_{i-1}}(v) = \begin{cases} max\{W_{T,R_{i-1}}(v') \mid v' \in child_E(v)\} & \text{if } v \text{ is an atom vertex;} \\ \sum_{v' \in child_E(v)} W_{T,R_{i-1}}(v') & \text{if } v \text{ is a rule vertex and } v \in R_{i-1}; \\ 1 + \sum_{v' \in child_E(v)} W_{T,R_{i-1}}(v') & \text{otherwise.} \end{cases}$$

Note that this function is different from W_T. Intuitively, v contributes to the distance measure if it is not included in R_{i-1}. The contributions of vertices in T are computed by Algorithm 6 (Line 5) by propagating the contributions up in the spirit of Algorithm 3. Then, K_i is extracted from weighted-T by using Algorithm 4 (Line 7).

The execution of Algorithm 5 is also illustrated in Figure 9. Similar to Algorithm 1, which generates shortest explanations, first the and-or explanation tree is created,

https://doi.org/10.1017/S1471068413000598 Published online by Cambridge University Press

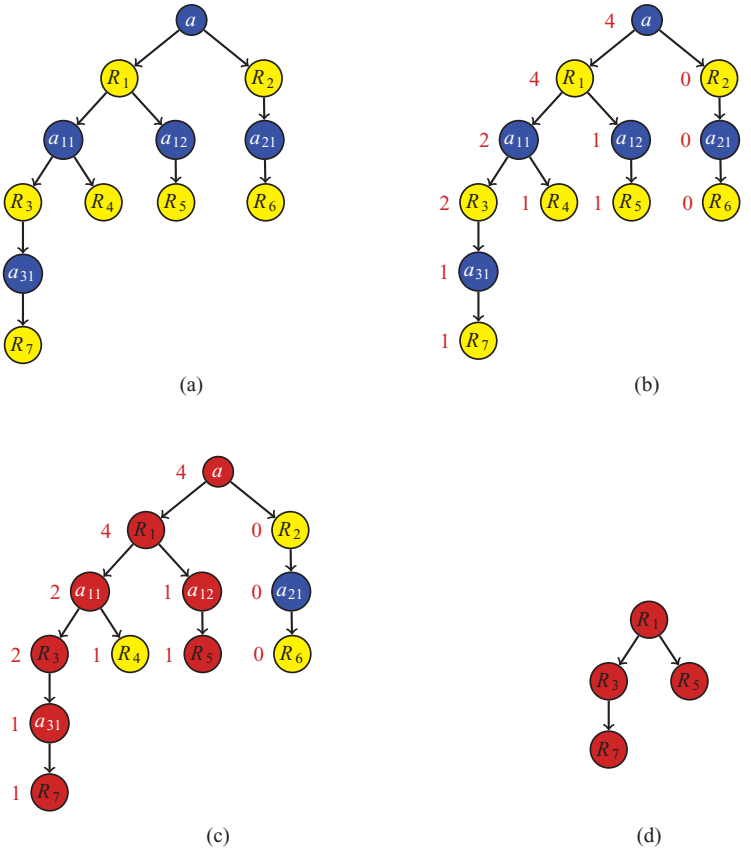

Fig. 9. A generic execution of Algorithm 5.

which has a generic structure as shown in Figure 9(a). Recall that yellow vertices denote atom vertices and blue vertices denote rule vertices. For the sake of example, assume that $R = \{R_2, R_6\}$. Then, the goal is to generate an explanation that contains different rule vertices from the rule vertices in R as much as possible. For that, the weights of vertices are assigned according to the weight function $W_{T,R}$ as depicted in Figure 9(b). Here, the weight of the root implies that there exists an explanation that contains four different rule vertices from the rule vertices in R and this explanation is the most different one. Then, starting from the root, a subtree of the and-or explanation tree is traversed by visiting maximum weighted child of every atom vertex, and every child of every rule vertex. This subtree is shown in Figure 9(c) by red vertices. Finally, an explanation is extracted by ignoring the atom vertices and keeping the parent–child relationship as it is, from this subtree. This explanation is illustrated in Figure 9(d).

Proposition 5

Given a ground ASP program Π, an answer set X for Π, an atom p in X, and a positive integer k, Algorithm 5 terminates.

https://doi.org/10.1017/S1471068413000598 Published online by Cambridge University Press

Input: Π : ground ASP program, X : answer set for Π, V : set of vertices, $l : V \rightarrow \Pi \cup X$, v : vertex in V, E : set of edges, R : set of rule vertices in V, D_R : candidate distance function.
Output: distance of v.
if $l(v) \in X$ **then**
  foreach $c \in child_E(v)$ **do**
    $D_R(c) :=$ calculateDifference$(\Pi, X, V, l, c, E, R, D_R)$;
  end
  $D_R(v) := max\{D_R(c) \mid c \in child_E(v)\}$;
end
else if $l(v) \in \Pi$ **then**
  if $v \notin R$ **then** $D_R(v) := 1$;
  else $D_R(v) := 0$;
  foreach $c \in child_E(v)$ **do**
    $D_R(v) := D_R(v) +$ calculateDifference$(\Pi, X, V, l, c, E, R, D_R)$;
  end
end
return $D_R(v)$;

Algorithm 6: calculateDifference

Proposition 6
Let Π be a ground ASP program, X be an answer set for Π, p be an atom in X, and k be a positive integer. Let n be the number of different explanations for p with respect to Π and X. Then, Algorithm 5 returns $min\{n,k\}$ different explanations for p with respect to Π and X.

Furthermore, at each iteration i of the loop in Algorithm 5 the distance $\Delta_D(R_{i-1}, K_i)$ is maximized.

Proposition 7
Let Π be a ground ASP program, X be an answer set for Π, p be an atom in X, and k be a positive integer. Let n be the number of explanations for p with respect to Π and X. Then, at the end of each iteration i ($1 \leqslant i \leqslant min\{n,k\}$) of the loop in Algorithm 5, $\Delta_D(R_{i-1}, RVertices(K_i))$ is maximized, i.e., there is no other explanation K' such that $\Delta_D(R_{i-1}, RVertices(K_i)) < \Delta_D(R_{i-1}, RVertices(K'))$.

This result leads us to some useful consequences. First, Algorithm 5 computes "longest" explanations if $k = 1$. The following corollary shows how to compute longest explanations.

Corollary 1
Let Π be a ground ASP program, X be an answer set for Π, p be an atom in X, and $k = 1$. Then, Algorithm 5 computes a longest explanation for p with respect to Π and X.

Next, we show that Algorithm 5 computes k different explanations such that for every i ($1 \leqslant i \leqslant k$) the ith explanation is the most distant explanation from the previously computed $i-1$ explanations.

https://doi.org/10.1017/S1471068413000598 Published online by Cambridge University Press

Table 2. *Experimental results for generating shortest explanations for some biomedical queries, using Algorithm 1*

Query	CPU time	Explanation size	Answer set size	And-Or tree size	GRINGO calls
Q1	52.78*s*	5	1.964.429	16	0
Q2	67.54*s*	7	2.087.219	233	1
Q3	31.15*s*	6	1.567.652	15	0
Q4	1245.83*s*	6	19.476.119	6690	4
Q5	41.75*s*	3	1.465.817	16	0
Q8	40.96*s*	14	1.060.288	28	4
Q10	1601.37*s*	14	1.612.128	3419	193
Q11	113.40*s*	6	2.158.684	5528	5
Q12	327.22*s*	5	10.338.474	10	1

Corollary 2
Let Π be a ground ASP program, X be an answer set for Π, p be an atom in X, and k be a positive integer. Let n be the number of explanations for p with respect to Π and X. Then, Algorithm 5 computes $min\{n,k\}$ different explanations $K_1,\ldots,K_{min\{n,k\}}$ for p with respect to Π and X such that for every j $(2 \leqslant j \leqslant min\{n,k\})$ $\Delta_D(\bigcup_{z=1}^{j-1} RVertices(K_z), K_j)$ is maximized.

The following proposition shows that the time complexity of Algorithm 5 is exponential in the size of the given answer set.

Proposition 8
Given a ground ASP program Π, an answer set X for Π, an atom p in X, and a positive integer k, the time complexity of Algorithm 5 is $O(k \times |\Pi|^{|X|+1} \times |\mathscr{B}_\Pi|)$.

7 Experiments with biomedical queries

Our algorithms for generating explanations are applicable to the queries Q1, Q2, Q3, Q4, Q5, Q8, Q10, Q11, and Q12 in Table 1. The ASP programs for the other queries involve choice expressions. For instance, the query Q7 asks for cliques of five genes. We use the following rule to generate a possible set of five genes that might form a clique.

```
5{clique(GEN):gene_name(GEN)}5.
```

Our algorithms apply to ASP programs that contain a single atom in the heads of the rules, and negation and cardinality expressions in the bodies of the rules. Therefore, our methods are not applicable to the queries that are presented by ASP programs that include choice expressions.

In Table 2, we present the results for generating shortest explanations for the queries Q1, Q2, Q3, Q4, Q5, Q8, Q10, Q11, and Q12. In this table, the second column denotes the CPU timings to generate shortest explanations in seconds. The third column consists of the sizes of explanations, i.e., the number of rule vertices

https://doi.org/10.1017/S1471068413000598 Published online by Cambridge University Press

in an explanation. In the fourth column, the sizes of answer sets, i.e., the number of atoms in an answer set, are given. The fifth column presents the sizes of the and-or explanation trees, i.e., the number of vertices in the tree.

Before telling what the last column presents, let us clarify an issue regarding the computation of explanations. Since answer sets contain millions of atoms, the relevant ground programs are also huge. Thus, first grounding the programs and then generating explanations over those grounded programs is an overkill in terms of computational efficiency. To this end, we apply another method and do grounding when it is necessary. To better explain the idea, let us present our method by an example. At the beginning, we have a ground atom for which we are looking for shortest explanations. Assume that this atom is *what_be_genes("ADRB1")*. Then, we find the rules whose heads are of the form *what_be_genes(GN)*, and instantiate *GN* with "ADRB1." For instance, assume that the following rule exists in the program:

$$what\_be\_genes(GN) \leftarrow drug\_gene(DRG,GN).$$

Then, by such an instantiation, we obtain the following instance of this rule:

$$what\_be\_genes("ADRB1") \leftarrow drug\_gene(DRG,"ADRB1").$$

Next, if the rules that we obtain by instantiating their heads are not ground, we ground them using the grounder GRINGO considering the answer set. We apply the same method for the atoms that are now ground, to find the relevant rules and ground them if necessary. This allows us to deal with a relevant subset of the rules while generating explanations. The last column of Table 2 presents the number of times GRINGO is called for such incremental grounding. For instance, for the queries Q1, Q3, and Q5, GRINGO is never called. However, GRINGO is called 193 times during the computation of a shortest explanation for the query Q10.

As seen from the results presented in Table 2, the computation time is not very much related to the size of the explanation. As also suggested by the complexity results of Algorithm 1 (i.e., $O(|\Pi|^{|X|} \times |\mathscr{B}_\Pi|)$), the computation time for generating shortest explanations greatly depends on the sizes of the answer set and the and-or explanation tree. For instance, for the query Q4, the answer set contains approximately 19 million atoms, the size of the and-or explanation tree is 6,690, and it takes 1,245 CPU seconds to compute a shortest explanation, whereas for the query Q8, the answer set approximately contains 1 million atoms, the and-or explanation tree has 28 vertices, and it takes 40 CPU seconds to compute a shortest explanation. Also, the number of times GRINGO is called during the computation affects the computation time. For instance, for the query Q10 the answer set approximately contains 1.6 million atoms, the and-or explanation tree has 3,419 vertices, and it takes 1,600 CPU seconds to compute a shortest explanation.

Table 3 shows the computation times for generating different explanations for the answers of the same queries, if exists. As seen from these results, the time for computing two and four different explanations is slightly different than the time for computing shortest explanations.

https://doi.org/10.1017/S1471068413000598 Published online by Cambridge University Press

Table 3. *Experimental results for generating different explanations for some biomedical queries, using Algorithm 5*

	CPU time		
Query	2 different	4 different	Shortest
Q1	53.73*s*	...	52.78*s*
Q2	66.88*s*	67.15*s*	67.54*s*
Q3	31.22*s*	...	31.15*s*
Q4	1248.15*s*	1251.13*s*	1245.83*s*
Q5	...	...	41.75*s*
Q8	...	...	40.96*s*
Q10	1600.49*s*	1602.16*s*	1601.37*s*
Q11	113.25*s*	112.83*s*	113.40*s*
Q12	...	...	327.22*s*

8 Presenting explanations in a natural language

An explanation for an answer of a biomedical query may not be easy to understand, since the user may not know the syntax of ASP rules neither the meanings of predicates. To this end, it is better to present explanations to the experts in a natural language.

Observe that leaves of an explanation denote facts extracted from the biomedical resources. Also some internal vertices contain informative explanations such as the position of a drug in a chain of drug–drug interactions. Therefore, there is a corresponding natural language explanation for some vertices in the tree. Such a correspondence can be stored in a predicate look-up table, like Table 4. Given such a look-up table, a pre-order depth-first traversal of an explanation and generating natural language expressions corresponding to vertices of the explanation lead to an explanation in natural language (Oztok 2012).

For instance, the explanation in Figure 6 is expressed in natural language as illustrated in the introduction.

9 Implementation of explanation generation algorithms

Based on the algorithms introduced above, we have developed a computational tool called ExpGen-ASP (Oztok 2012), using the programming language C++. Given an ASP program and its answer set, ExpGen-ASP generates shortest explanations as well as k different explanations.

The input of ExpGen-ASP are

- an ASP program Π,
- an answer set X for Π,
- an atom p in X,
- an option that is used to generate either a shortest explanation or k different explanations, and
- a predicate look-up table,

https://doi.org/10.1017/S1471068413000598 Published online by Cambridge University Press

Table 4. *Predicate look-up table used while expressing explanations in natural language*

Predicate	Expression in natural language
gene_gene_biogrid(x,y)	The gene x interacts with the gene y according to BioGRID.
drug_disease_ctd(x,y)	The disease y is treated by the drug x according to CTD.
drug_gene_ctd(x,y)	The drug x targets the gene y according to CTD.
gene_disease_ctd(x,y)	The disease y is related to the gene x according to CTD.
disease_symptom_do(x,y)	The disease x has the symptom y according to Disease Ontology.
drug_category_drugbank(x,y)	The drug x belongs to the category y according to DrugBank.
drug_drug_drugbank(x,y)	The drug x reacts with the drug y according to DrugBank.
drug_sideeffect_sider(x,y)	The drug x has the side effect y according to SIDER.
disease_gene_orphadata(x,y)	The disease x is related to the gene y according to Orphadata.
drug_disease_pharmgkb(x,y)	The disease y is treated by the drug x according to PharmGKB.
drug_gene_pharmgkb(x,y)	The drug x targets the gene y according to PharmGKB.
disease_gene_pharmgkb(x,y)	The disease x is related to the gene y according to PharmGKB.
start_drug(x)	The drug x is the start drug.
start_gene(x)	The gene x is the start gene.
drug_reachable_from(x,l)	The distance of the drug x from the start drug is l.
gene_reachable_from(x,l)	The distance of the gene x from the start gene is l.

and the output are

- a shortest explanation for p with respect to Π and X in a natural language (if shortest explanation option is chosen), and
- k different explanations for p with respect to Π and X in a natural language (if k different explanations option is chosen).

For generating shortest explanations (respectively, k different explanations), ExpGen-ASP utilizes Algorithm 1 (respectively, Algorithm 5).

To provide experts with further informative explanations about the answers of biomedical queries, we have embedded ExpGen-ASP into BioQuery-ASP by utilizing Table 4 as the predicate look-up table of the system. Figure 10 shows a snapshot of the explanation generation mechanism of BioQuery-ASP.

10 Relating explanations to justifications

The most similar work to ours is Pontelli *et al.* (2009) that study the question "why is an atom p in an answer set X for an ASP program Π." As an answer to this question, Pontelli *et al.* (2009) find a "justification", which is a labeled graph that provides an explanation for the truth values of atoms with respect to an answer set.

https://doi.org/10.1017/S1471068413000598 Published online by Cambridge University Press

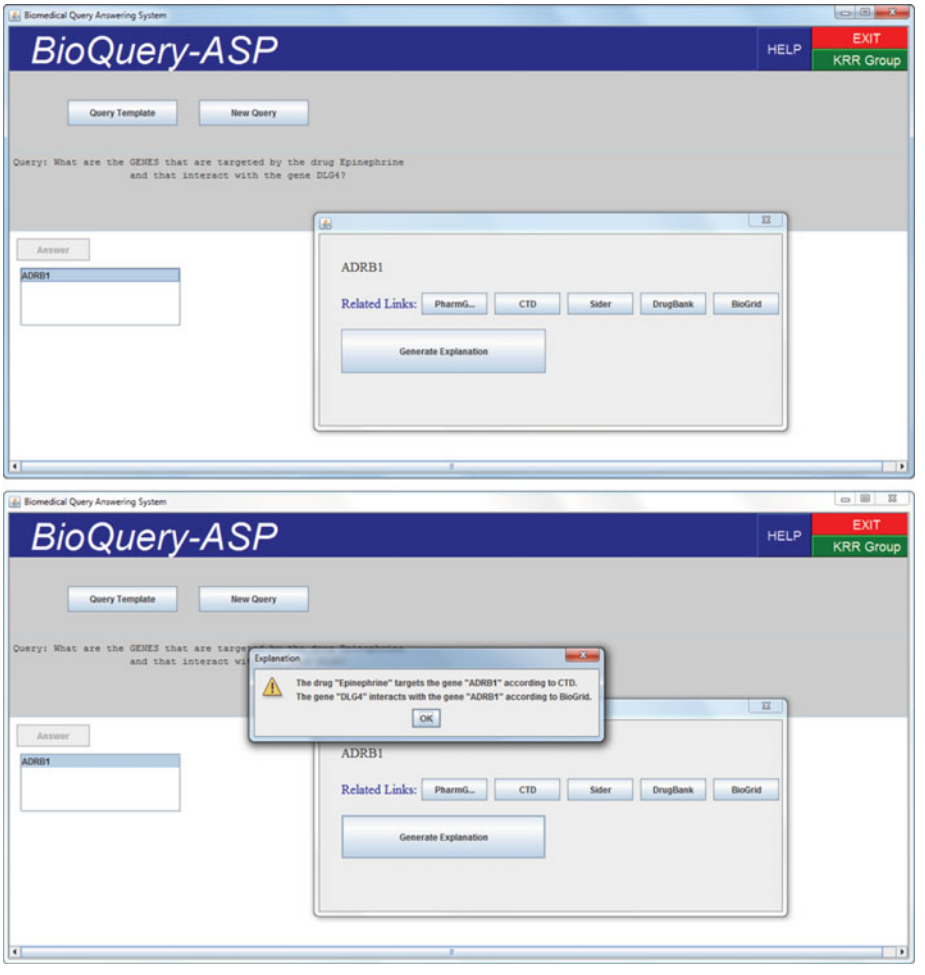

Fig. 10. A snapshot of BioQuery-ASP showing its explanation generation facility.

Example 6

Let Π be the program presented in Example 1:

$$a \leftarrow b, c$$
$$a \leftarrow d$$
$$d \leftarrow$$
$$b \leftarrow c$$
$$c \leftarrow$$

and $X = \{a, b, c, d\}$. Figure 11 is an offline justification of a^+ with respect to X and $\emptyset$. Intuitively, a is in X since b and c are also in X and there is a rule in Π that supports a using the atoms b and c. Furthermore, b is in X since c is in X and there is a rule in Π that supports b using the atom c. Finally, c is in X as it is a fact in Π.

https://doi.org/10.1017/S1471068413000598 Published online by Cambridge University Press

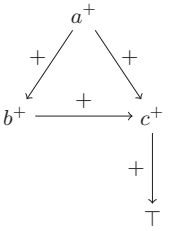

Fig. 11. An offline justification for Example 6.

To relate offline justifications and explanations, we need to introduce the following definitions and notations about justifications defined in Pontelli *et al.* (2009).

10.1 Offline justifications

First, let us introduce notations related to ASP programs used in Pontelli *et al.* (2009). The class of ASP programs studied is normal programs, i.e., programs that consist of the rules of the form

$$A \leftarrow A_1, \ldots, A_k, not\ A_{k+1}, \ldots, not\ A_m$$

where $m \geqslant k \geqslant 0$ and A and each A_i is an atom. Therefore, the programs we consider are more general. Let Π be a normal ASP program. Then, $\mathscr{A}_\Pi$ is the Herbrand base of Π. An *interpretation* I for a program Π is defined as a pair $\langle I^+, I^- \rangle$, where $I^+ \cup I^- \subseteq \mathscr{A}_\Pi$ and $I^+ \cap I^- = \emptyset$. Intuitively, I^+ denotes the set of atoms that are true, while I^- denotes the set of atoms that are false. I is a *complete interpretation* if $I^+ \cup I^- = \mathscr{A}_\Pi$. The *reduct* Π^I of Π with respect to I is defined as

$$\Pi^I = \{H(r) \leftarrow B^+(r) \mid r \in \Pi,\ B^-(r) \cap I^+ = \emptyset\}$$

A complete interpretation M for a program Π is an answer set for Π if M^+ is an answer set for Π^M. Also, a *literal* is either an atom or a formula of the form *not a* where a is an atom. The set of atoms that appears as negated literals in Π is denoted by $NANT(\Pi)$. For an atom a, a^+ denotes that the atom a is true and a^- denotes that a^- is false. Then, a^+ and a^- are called the *annotated* versions of a. Moreover, it is defined that $atom(a^+) = a$ and $atom(a^-) = a$. For a set S of atoms, the following sets of annotated atoms are defined:

- $S^p = \{a^+ \mid a \in S\}$
- $S^n = \{a^- \mid a \in S\}$.

Finally, the set *not S* is defined as $not\ S = \{not\ a \mid a \in S\}$.

Apart from the answer set semantics, there is another important semantics of logic programs, called the well-founded semantics (Gelder *et al.* 1991). Since this semantics is important to build the notion of a justification, we now briefly describe the well-founded semantics. We consider the definition proposed in Apt and Bol (1994), instead of the original definition proposed in Gelder *et al.* (1991), as considered by Pontelli *et al.* (2009).

https://doi.org/10.1017/S1471068413000598 Published online by Cambridge University Press

Definition 7 (*Immediate consequence*)
Let Π be a normal ASP program, and S and V be two sets of atoms from $\mathscr{A}_\Pi$. Then, the *immediate consequence* of S with respect to Π and V, denoted by $T_{\Pi,V}(S)$, is the set defined as follows:

$$T_{\Pi,V}(S) = \{a \mid \exists\, r \in \Pi,\, H(r) = a,\, B^+(r) \subseteq S,\, B^-(r) \cap V = \emptyset\}.$$

We denote by $lfp(T_{\Pi,V})$ the least fixpoint of $T_{\Pi,V}$ when V is fixed.

Definition 8 (*The well-founded model*)
Let Π be a normal ASP program, $\Pi^+ = \{r \mid r \in \Pi,\, B^-(r) = \emptyset\}$. The sequence $\langle K_i, U_i\rangle_{i\geqslant 0}$ is defined as follows:

$$K_0 = lfp(T_{\Pi^+}), \quad U_0 = lfp(T_{\Pi,K_0}),$$
$$K_i = lfp(T_{\Pi,U_{i-1}}),\ U_i = lfp(T_{\Pi,K_i}).$$

Let j be the first index of the computation such that $\langle K_j, U_j\rangle = \langle K_{j+1}, U_{j+1}\rangle$. Then, *the well-founded model* of Π is $WF_\Pi = \langle W^+, W^-\rangle$ where $W^+ = K_j$ and $W^- = \mathscr{A}_\Pi \backslash U_j$.

Example 7
Let Π be the program

$$a \leftarrow b, not\ d$$
$$d \leftarrow b, not\ a$$
$$b \leftarrow c$$
$$c \leftarrow$$

Then, the well-founded model of Π is computed as follows:

$$K_0 = \{b, c\},$$
$$U_0 = \{a, b, c, d\},$$
$$K_1 = \{b, c\},$$
$$U_0 = \{a, b, c, d\}.$$

Thus, $WF_\Pi = \langle\{b, c\}, \emptyset\rangle$.

We now provide definitions regarding the notion of an offline justification. First, we introduce the basic building of an offline justification, a labeled graph called as e-graph.

Definition 9 (*e-graph*)
Let Π be a normal ASP program. An *e-graph* for Π is a labeled, directed graph (N, E), where $N \subseteq \mathscr{A}^p_\Pi \cup \mathscr{A}^n_\Pi \cup \{assume, \top, \bot\}$ and $E \subseteq N \times N \times \{+, -\}$, which satisfies following properties:

(i) the only sinks (i.e., nodes without out-going edges) in the graph are $assume, \top, \bot$;
(ii) for every $b \in N \cap \mathscr{A}^p_\Pi$, $(b, assume, -) \notin E$ and $(b, \bot, -) \notin E$;
(iii) for every $b \in N \cap \mathscr{A}^n_\Pi$, $(b, assume, +) \notin E$ and $(b, \top, +) \notin E$;
(iv) for every $b \in N$, if for some $l \in \{assume, \top, \bot\}$ and $s \in \{+, -\}$, $(b, l, s) \in E$, then (b, l, s) is the only out-going edge originating from b.

https://doi.org/10.1017/S1471068413000598 Published online by Cambridge University Press

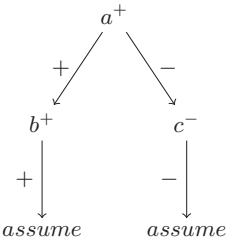

Fig. 12. An e-graph for Example 8.

According to this definition, an edge of an e-graph connects two annotated atoms or an annotated atom with one of the nodes in $\{assume, \top, \bot\}$ and is marked by a label from $\{+, -\}$. An edge is called as *positive* (respectively, *negative*) if it is labeled by $+$ (respectively, $-$). Also, a path in an e-graph is called as *positive* if it has only positive edges, whereas it is called as *negative* if it has at least one negative edge. The existence of a positive path between two nodes v_1 and v_2 is denoted by $(v_1, v_2) \in E^{*,+}$. In the offline justification, $\top$ is used to explain facts, $\bot$ to explain atoms that do not have defining rules, and *assume* is for atoms for which explanations are not needed, i.e., they are assumed to be true or false.

Example 8
Let Π be the program presented in Example 1, and $X = \{a, b, c, d\}$. Then, Figure 12 is an e-graph for Π. Intuitively, the true state of a depends on the true state of b and the false state of c, where b is assumed to be true and c is assumed to be false.

In an e-graph, a set of elements that directly contributes to the truth value of an atom can be obtained through the out-going edges of a corresponding node. This set is defined as follows.

Definition 10 (*support*(b, G))
Let Π be a normal ASP program, $G = (N, E)$ be an e-graph for Π and $b \in N \cap (\mathscr{A}^p_\Pi \cup \mathscr{A}^n_\Pi)$ be a node in G. Then, *support*(b, G) is defined as follows.

- *support*$(b, G) = \{l\}$, if for some $l \in \{assume, \top, \bot\}$ and $s \in \{+, -\}$, (b, l, s) is in E;
- *support*$(b, G) = \{atom(c) \mid (b, c, +) \in E\} \cup \{not\ atom(c) \mid (b, c, -) \in E\}$, otherwise.

Example 9
Let G be the e-graph in Figure 12. Then, *support*$(a, G) = \{b, not\ c\}$, *support*$(b, G) = \{assume\}$, and *support*$(c, G) = \{assume\}$.

To define the notion of a justification, an e-graph should be enriched with explanations of truth values of atoms that are derived from the rules of the program. For that, the concept of one step justification of a literal is defined as follows.

Definition 11 (*Local Consistent Explanation (LCE)*)
Let Π be a normal ASP program, b be an atom, J be a possible interpretation for Π, $U \subseteq \mathscr{A}_\Pi$ be a set of atoms, and $S \subseteq \mathscr{A}_\Pi \cup not\ \mathscr{A}_\Pi \cup \{assume, \top, \bot\}$ be a set of literals. We say that

https://doi.org/10.1017/S1471068413000598 Published online by Cambridge University Press

- S is an *LCE* of b^+ with respect to (J, U), if $b \in J^+$ and
  - $S = \{assume\}$ or
  - $S \cap \mathscr{A}_\Pi \subseteq J^+$, $\{c \mid \text{not } c \in S\} \subseteq J^- \cup U$, and there is a rule $r \in \Pi$ such that $H(r) = b$ and $B(r) = S$. In case, $B(r) = \emptyset$, S is denoted by the set $\{\top\}$.
- S is an *LCE* of b^- with respect to (J, U), if $b \in J^- \cup U$ and
  - $S = \{assume\}$ or
  - $S \cap \mathscr{A} \subseteq J^- \cup U$, $\{c \mid not\ c \in S\} \subseteq J^+$, and S is a minimal set of literals such that for every rule $r \in \Pi$ if $H(r) = b$, then $B^+(r) \cap S \neq \emptyset$ or $B^-(r) \cap \{c \mid \text{not } c \in S\} \neq \emptyset$. In case, $S = \emptyset$, S is denoted by the set $\{\bot\}$.

The set of all the LCEs of b^+ with respect to (J, U) is denoted by $\text{LCE}^p_\Pi(b, J, U)$ and the set of all the LCEs of b^- with respect to (J, U) is denoted by $\text{LCE}^n_\Pi(b, J, U)$.

Here, a possible interpretation J denotes an answer set. The set U consists of atoms that are assumed to be false (which will be called as Assumptions in the notion of justification later on). The need for U comes from the fact that the truth value of some atoms is first guessed while computing answer sets. Intuitively, if an atom a is true, an LCE consists of the body of a rule that is satisfied by J and has a in its head; if a is false, an LCE consists of a set of literals that are false in J and falsify all rules whose head are a.

Example 10
Let Π and X be defined as in Example 8. Then, the LCEs of the atoms with respect to $(X, \emptyset)$ is as follows:

$$\text{LCE}^p_\Pi(a, X, \emptyset) = \{\{b, c\}, \{d\}, \{assume\}\}$$
$$\text{LCE}^p_\Pi(b, X, \emptyset) = \{\{c\}, \{assume\}\}$$
$$\text{LCE}^p_\Pi(c, X, \emptyset) = \{\{\top\}, \{assume\}\}$$
$$\text{LCE}^p_\Pi(d, X, \emptyset) = \{\{\top\}, \{assume\}\}.$$

Accordingly, a class of e-graphs where edges represent LCEs of the corresponding nodes is defined as follows.

Definition 12 ((J, U)*-based e-graph*)
Let Π be a normal ASP program, J be a possible interpretation for Π, $U \subseteq \mathscr{A}_\Pi$ be a set of atoms, and b be an element in $\mathscr{A}^p_\Pi \cup \mathscr{A}^n_\Pi$. A (J, U)*-based e-graph* $G = (N, E)$ of b is an e-graph such that

(i) every node $c \in N$ is reachable from b,
(ii) for every $c \in N \backslash \{assume, \top, \bot\}$, $support(c, G)$ is an LCE of c with respect to (J, U);

A (J, U)-based e-graph (N, E) is *safe* if for all $b^+ \in N, (b^+, b^+) \notin E^{*,+}$, i.e., there is no positive cycle in the graph.

We now introduce a special class of (J, U)-based e-graphs where only false elements can be assumed.

https://doi.org/10.1017/S1471068413000598 Published online by Cambridge University Press

Definition 13 (*Offline e-graph*)
Let Π be a normal ASP program, J be a partial interpretation for Π, $U \subseteq \mathscr{A}_\Pi$ be a set of atoms, and b be an element in $\mathscr{A}^p \cup \mathscr{A}^n$. An *offline e-graph* $G = (N, E)$ of b with respect to J and U is a (J, U)-based e-graph of b that satisfies following properties:

(i) there exists no $p^+ \in N$ such that $(p^+, assume, +) \in E$;
(ii) $(p^-, assume, -) \in E$ if and only if $p \in U$.

$\mathscr{E}(b, J, U)$ is the set of all offline e-graphs of b with respect to J and U.

Here, the roles of J and U are the same as their roles in Definition 11. Observe that true atoms cannot be assumed due to the first condition and only elements in the set U are assumed due to the second condition.

We said earlier that in a (J, U)-based e-graph J represents an answer set and U consists of atoms that are assumed to be false. Here, U is chosen based on some characteristics of J. In particular, we want U to be a set of atoms such that when its elements are assumed to be false, the truth value of other atoms in the program can be uniquely determined and leads to J. We now introduce relevant definitions formally.

Definition 14 (*Tentative Assumptions*)
Let Π be a normal ASP program, M be an answer set for Π, and $WF_\Pi =< WF_\Pi^+, WF_\Pi^- >$ be the well-founded model of Π. The *tentative assumptions* $\mathscr{TA}_\Pi(M)$ of Π with respect to M are defined as

$$\mathscr{TA}_\Pi(M) = \{a \mid a \in NANT(\Pi) \wedge a \in M^-, a \notin (WF_\Pi^+ \cup WF_\Pi^-)\}. \tag{9}$$

Example 11
Let Π be the program:

$$c \leftarrow a, not\ d$$
$$d \leftarrow a, not\ c$$
$$a \leftarrow b$$
$$b \leftarrow$$

Then, $X = \{a, b, c\}$ is an answer set for Π and $\langle \{a, b\}, \emptyset \rangle$ is the well-founded model of Π. Given that, $\mathscr{TA}_\Pi(X) = \{d\}$ as $d \in NANT(\Pi)$, $d \notin X$ and $d \notin (WF_\Pi^+ \cup WF_\Pi^-)$.

In fact, tentative assumptions is a set of atoms whose subsets might "potentially" form U.

We provide a definition that would allow one to obtain a program from a given program Π and a set V of atoms by assuming all the atoms in V as false.

Definition 15 (*Negative Reduct*)
Let Π be a normal ASP program, M be an answer set for Π, and $U \subseteq \mathscr{TA}_\Pi(M)$ be a set of tentative assumption atoms. The *negative reduct* $NR(\Pi, U)$ of Π with respect to U is the set of rules defined as

$$NR(\Pi, U) = \Pi \backslash \{r \mid H(r) \in U\}. \tag{10}$$

https://doi.org/10.1017/S1471068413000598 Published online by Cambridge University Press

Finally, the concept of assumptions can be introduced formally.

Definition 16 (*Assumption*)
Let Π be a normal ASP program, and M be an answer set for Π. An *assumption* of Π with respect to M is a set U of atoms that satisfies the following properties:

(i) $U \subseteq \mathscr{T}\mathscr{A}_{\Pi}(M)$;
(ii) the well-founded model of $NR(\Pi, U)$ is equal to M.

Assumptions(Π, M) is the set of all assumptions of Π with respect to M.

Example 12
Let Π and X be defined as in Example 11. Let $U = \{d\}$. Then, $NR(\Pi, U)$ is:

$$c \leftarrow a, not\ d$$
$$a \leftarrow b$$
$$b \leftarrow$$

and $\langle\{a, b, c\}, \emptyset\rangle$ is the well-founded model of $NR(\Pi, U)$. Thus, U is an assumption of Π with respect to X.

Note that assumptions are nothing but subsets of tentative assumptions that would allow to obtain the answer set J.

At last, we are ready to define the notion of offline justification.

Definition 17 (*Offline Justification*)
Let Π be a normal ASP program, M be an answer set for Π, U be an assumption in *Assumptions*(Π, M), and b be an annotated atom in $\mathscr{A}^p \cup \mathscr{A}^n$. An *offline justification* of b with respect to M and U is an element (N, E) of $\mathscr{E}(b, M, U)$, which is safe.

According to the definition, a justification is a (J, U)-based e-graph where J is an answer set and U is an assumption. Also, justifications do not allow the creation of positive cycles in the justification of true atoms. For instance, for Π and X defined as in Example 1, Figure 11 illustrates an offline justification of a^+ with respect to X and $\emptyset$.

Pontelli *et al.* (2009) prove the following proposition that shows that for every atom in the program, there exists an offline justification.

Proposition 9
Let Π be a ground normal ASP program, and X be an answer for Π. Then, for each atom a in Π, there is an offline justification with respect to X and $X^- \setminus WF_{\Pi}^-$ that does not contain negative cycles.

10.2 From justifications to explanations

We relate a justification to an explanation. In particular, given an offline justification, we show that one can obtain an explanation tree whose atom vertices are formed by utilizing the "annotated atoms" of the justification and rule vertices are formed by utilizing the "support" of annotated atoms. To compute such explanation trees, we develop Algorithm 7.

https://doi.org/10.1017/S1471068413000598 Published online by Cambridge University Press

Input: Π : ground normal ASP program, X : answer set for Π, p : atom in X,
(V, E) : justification of p^+ w.r.t X and some $U \in \mathit{Assumptions}(\Pi, X)$.
Output: A vertex-labeled tree $\langle V', E', l, \Pi, X \rangle$.
$V' := \emptyset$, $E' := \emptyset$;
$v \leftarrow$ Create a vertex v s.t. $l(v) = p$;
$Q \leftarrow v$;
while $Q \neq \emptyset$ **do**
  $v' \leftarrow$ Dequeue an element from Q;
  $V' := V' \cup \{v'\}$;
  if $l(v') \in \Pi$ **then** // v' is a rule vertex
    foreach $a \in B^+(l(v'))$ **do**
      $v'' \leftarrow$ Create a vertex v'' s.t. $l(v'') = a$;
      $E' := E' \cup \{(v', v'')\}$ // edge from rule vertex to atom vertex;
      Enqueue v'' to Q;
    end
  end
  else if $l(v') \in X$ **then** // v' is an atom vertex
    $r \leftarrow$ Create a rule r s.t. $H(r) = l(v')$ and $B(r) = \mathit{support}(l(v')^+, G)$;
    $v'' \leftarrow$ Create a vertex v'' s.t. $l(v'') = r$;
    $E' := E' \cup \{(v', v'')\}$ // edge from atom vertex to rule vertex;
    Enqueue v'' to Q;
  end
end
return $\langle V', E', l, \Pi, X \rangle$;

Algorithm 7: Justification to Explanation

Let us now explain the algorithm in detail. Algorithm 7 takes as input a ground normal ASP program Π, an answer set X for Π, an atom p in X, and a justification (V, E) of p^+ with respect to X and some $U \in \mathit{Assumptions}(\Pi, X)$. Our goal is to obtain an explanation tree in the and-or explanation tree for p with respect to Π and X from the justification (V, E). The algorithm starts by creating two sets V' and E' (Line 1). Here, V' and E' corresponds to the set of vertices and the set of edges of the explanation tree, respectively. By Condition (*ii*) in Definition 3 and Condition (*i*) in Definition 2, we know that the label of the root of an explanation tree for p with respect to Π and X is p. Thus, a vertex v with label p is defined (Line 2), and added into the queue Q (Line 3). Then, the algorithm enters into a "while" loop that executes until Q becomes empty. At every iteration of the loop, an element v from Q is first extracted (Line 5) and added into V' (Line 6). This implies that every element added into Q is also added into V'. For instance, the vertex defined at Line 2 is the first vertex extracted from Q and also added into V', which makes sense since we know that the root of an explanation tree is an atom vertex with label v. Then, according to the type of the extracted vertex, its out-going edges are defined. Let v' be a vertex extracted from Q at Line 5 in some iteration of the loop. Consider the following two cases.

https://doi.org/10.1017/S1471068413000598 Published online by Cambridge University Press

Case 1 Assume that v' is an atom vertex. Then, the algorithm directly goes to Line 13. By Condition (*i*) in Definition 3, we know that an explanation tree is a subtree of the and-or explanation tree. Hence, we need to define outgoing edges of v' by taking into account Condition (*ii*) in Definition 2, which implies that a child of v' must be a rule vertex v'' such that the rule that labels v'' "supports" the atom that labels v'. Thus, a rule r that supports the atom that labels v' is created (Line 13). We ensure "supportedness" property by utilizing the annotated atoms in the given offline justification that supports the annotated version of the atom that labels v'. Then, a vertex v'' with label r is created (Line 14), and a corresponding child of v' is added into E' (Line 15). By Condition (*iii*) in Definition 3, we know that every atom vertex of an explanation tree has a single child. Therefore, another child of v' is not created. Then, before finishing the iteration of the loop, the child v'' of v' is added into Q so that its children can be formed in the next iterations of the loop.

Case 2 Assume that v' is a rule vertex. Then, the condition at Line 7 is satisfied and the algorithm goes to Line 8. In this case, while forming the children of v', we should consider Condition (*iii*) in Definition 2, which implies that a child v'' of v' must be an atom vertex such that the atom that labels v'' is in the positive body of the rule that labels v'. Also, by Condition (*iv*) in Definition 3, we should ensure that for every atom a in the positive body of the rule that labels v', there exists a child v_a of v' such that the atom that labels v_a is equal to a. Thus, the loop between Lines 8 and 11 iterates for every atom a in the positive body of the label of v' and a vertex v'' with label a is created (Line 9). Then, v'' becomes a child of v' (Line 10). To form the children of v'' in the next iterations of the "while" loop, the child v'' of v' is added into Q (Line 11).

When the algorithm finishes processing the elements of Q, i.e., Q becomes empty, the "while" loop terminates. Then, the algorithm returns a vertex-labeled tree (Line 17). We now provide the proposition about the soundness of Algorithm 7.

Proposition 10

Given a ground normal ASP program Π, an answer set X for Π, an atom p in X, an assumption U in $Assumption(\Pi, X)$, and an offline justification $G = (V, E)$ of p^+ with respect to X and U, Algorithm 7 returns an explanation tree $\langle V', E', l, \Pi, X\rangle$ in the and-or explanation tree for p with respect to Π and X.

Example 13

Let Π and X be defined as in Example 8. Figure 13(a) is an offline justification of a^+ with respect to X and $\emptyset$. Figure 13(b) shows a corresponding explanation tree in the and-or explanation tree for a with respect to Π and X that is obtained by using Algorithm 7.

https://doi.org/10.1017/S1471068413000598 Published online by Cambridge University Press

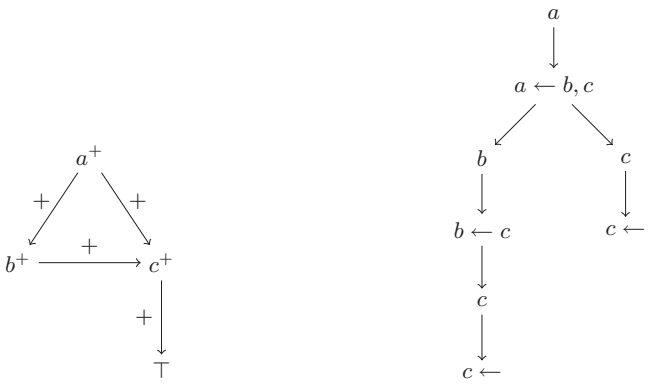

Fig. 13. (a) An offline justification and (b) its corresponding explanation tree obtained by using Algorithm 7.

Input: Π : ground normal ASP program, X : answer set for Π, p : atom in X, $\langle V', E', l, \Pi, X\rangle$: an explanation tree in the and-or explanation tree for p w.r.t Π and X.
Output: (V, E) : justification of p^+ w.r.t X and $\emptyset$.
$V := \emptyset$, $E := \emptyset$;
$Q \leftarrow$ root of $\langle V', E'\rangle$;
while $Q \neq \emptyset$ **do**
  $v \leftarrow$ Dequeue an element from Q;
  $V := V \cup \{l(v)^+\}$;
  $v' \leftarrow$ child of v in $\langle V', E'\rangle$;
  if $l(v')$ *is a fact in* Π^X **then**
    $E := E \cup \{(l(v)^+, \top, +)\}$;
  end
  foreach $v'' \in child_{E'}(v')$ **do**
    $E := E \cup \{(l(v)^+, l(v'')^+, +)\}$;
    Enqueue v'' to Q;
  end
end
$V := V \cup \{\top\}$;
return (V, E);

Algorithm 8: Explanation to Justification

10.3 From explanations to justifications

We relate an explanation to a justification. In particular, given an explanation tree whose labels of vertices are unique, we show that one can obtain an offline justification by utilizing the labels of atom vertices of the explanation tree. For that, we design Algorithm 8.

Let us now describe the algorithm in detail. Algorithm 8 takes as input a ground normal ASP program Π, an answer set X for Π, an atom p in X, and an explanation

https://doi.org/10.1017/S1471068413000598 Published online by Cambridge University Press

tree $T' = \langle V', E', l, \Pi, X \rangle$ in the and-or explanation tree for p with respect to Π and X. Our goal is to obtain an offline justification (V, E) of p^+ in Π^X with respect to X and $\emptyset$. The reason to obtain the offline justification in the reduct of Π with respect to X is that our definition of explanation is not defined for the atoms that are not in the answer set. Algorithm 8 starts by creating two sets V and E that will correspond to the set of nodes and the set of edges of the offline justification, respectively (Line 1). Then, the root of $\langle V', E' \rangle$ is added into the queue Q (Line 2) and we enter into a "while" loop that iterates until Q becomes empty. At every iteration of the loop, first an element v is extracted from Q (Line 4) and $l(v)^+$ is added into V (Line 5). Then, we form the out-going edges of $l(v)^+$. Due to Condition (*iii*) in Definition 3, every atom vertex in an explanation tree has a single child, which is a rule vertex due to Condition (*i*) in Definition 3 and Condition (*ii*) in Definition 2. Then, we extract the child v' of v at Line 6 and consider two cases. Note that v' is a rule vertex.

Case 1 Assume that $l(v')$ is a fact in Π^X. Then, $l(v')$ satisfies the condition at Line 7 and we add $(l(v)^+, \top, +)$ into E at Line 8. The key insight behind that is as follows. Due to Condition (*ii*) in Definition 12, $support(l(v)^+, (V, E))$ must be an LCE of $l(v)^+$. Due to Condition (*i*) in Definition 3 and Condition (*ii*) in Definition 2, the head of $l(v')$ is $l(v)$. As $l(v')$ is a fact in Π^X, i.e., its body is empty in Π^X, $\{\top\}$ becomes an LCE of $l(v)^+$ with respect to $(X, \emptyset)$, due to Definition 11. Thus, by adding $(l(v)^+, \top, +)$ to E, $support(l(v)^+, (V, E))$ becomes $\{\top\}$.

Case 2 Assume that $l(v')$ is not a fact in Π^X. Then, for every child v'' of v', we add $(l(v)^+, l(v'')^+, +)$ into E at Line 10. The intuition behind this is to make sure that $support(l(v)^+, (V, E))$ is an LCE of $l(v)^+$. Due to Condition (*i*) in Definition 3, an explanation tree is a subtree of the corresponding and-or explanation tree. Then, due to Condition (*ii*) in Definition 2, for every atom vertex v in an explanation tree, the atoms in the positive body of the rule that labels the child v' of v are in the given answer set X. Thus, due to Definition 11, adding $(l(v)^+, l(v'')^+, +)$ into E for every child v'' of v' ensures that $support(l(v)^+, (V, E))$ is an LCE of $l(v)^+$ with respect to $(X, \emptyset)$. Also, we add v'' into V so that its children in V are formed in the next iterations of the "while" loop.

Due to Line 8, there are incoming edges of $\top$. But, $\top$ is not added into V inside the "while" loop. Thus, when the "while" loop terminates, before returning (V, E) at Line 13, we add $\top$ into V.

Algorithm 8 creates an offline justification of the given atom in the reduct of the given ASP program with respect to the given answer set, provided that labels of the vertices of the given explanation tree are unique.

Proposition 11

Given a ground normal ASP program Π, an answer set X for Π, an atom p in X, and an explanation tree $\langle V', E', l, \Pi, X \rangle$ in the and-or explanation tree for p with respect to Π and X such that for every $v, v' \in V'$, $l(v) = l(v')$ if and only if

https://doi.org/10.1017/S1471068413000598 Published online by Cambridge University Press

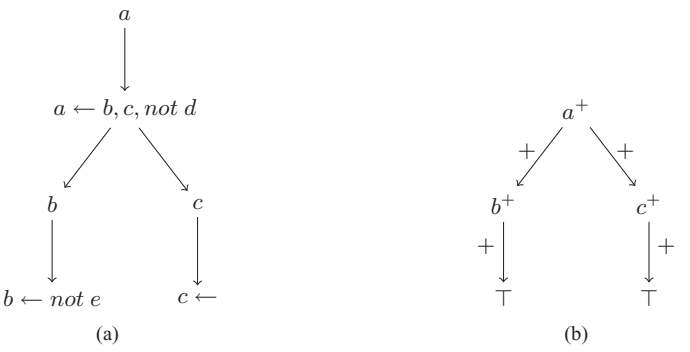

Fig. 14. (a) An explanation tree and (b) its corresponding offline justification obtained by using Algorithm 8.

$v = v'$, Algorithm 8 returns an offline justification of p^+ in Π^X with respect to X and $\emptyset$.

Example 14
Let Π be the program:

$$a \leftarrow b, c, not\ d$$
$$b \leftarrow not\ e$$
$$c \leftarrow$$

and $X = \{a, b, c\}$. Figure 14(a) is an explanation tree T in the and-or explanation tree for a with respect to Π and X. Then, given Π, X, a, and T, Algorithm 8 creates an offline justification of a^+ in Π^X with respect to Π and $\emptyset$ as in Figure 14(b).

11 Other related work

The most recent work related to explanation generation in ASP are Brain and Vos (2005), Syrjanen (2006), Gebser *et al.* (2008), Pontelli *et al.* (2009), and Oetsch *et al.* (2010; 2011), in the context of debugging ASP programs. Among those, Syrjanen (2006) studies why a program does not have an answer set, and Gebser *et al.* (2008) and Oetsch *et al.* (2010) study why a set of atoms is not an answer set. As we study the problem of explaining the reasons why atoms are in the answer set, our work differs from those two works.

In Brain and Vos (2005), similar to our work, the question "why is an atom p in an answer set X for an ASP program Π" is studied. As an answer to this question, Brain and Vos (2005) provide the rule in Π that supports X with respect to Π; whereas we compute shortest or k different explanations (as a tree whose vertices are labeled by rules).

Pontelli *et al.* (2009) also introduce the notion of an online justification that aims to justify the truth values of atoms during the computation of an answer set. In Oetsch *et al.* (2011), a framework where the users can construct interpretations through an interactive stepping process is introduced. As a result, Pontelli *et al.* (2009) and Oetsch *et al.* (2011) can be used together to provide the users with

https://doi.org/10.1017/S1471068413000598 Published online by Cambridge University Press

justifications of the truth values of atoms during the construction of interpretations interactively through stepping.

12 Conclusion

We have formally defined explanations in the context of ASP. We have also introduced variations of explanations, such as "shortest explanations" and "k different explanations."

We have proposed generic algorithms to generate explanations for biomedical queries. In particular, we have presented algorithms to compute shortest or k different explanations. We have analyzed termination, soundness, and complexity of these algorithms. In particular, the complexity of generating a shortest explanation for an answer (in an answer set X) is $O(|\Pi|^{|X|} \times |\mathscr{B}_\Pi|)$ where $|\Pi|$ is the number of ASP rules representing the query, the knowledge extracted from biomedical resources, and the rule layer, and $|\mathscr{B}_\Pi|$ is the number of atoms in Π. The complexity of generating k different explanations is $O(k \times |\Pi|^{|X|+1} \times |\mathscr{B}_\Pi|)$. For k different explanations, we have defined a distance measure based on the number of different ASP rules between explanations.

We have developed a computational tool ExpGen-ASP that implements these algorithms. We have embedded ExpGen-ASP into BioQuery-ASP to generate explanations regarding the answers of complex biomedical queries. We have proposed a method to present explanations in a natural language. No existing biomedical query answering system is capable of generating explanations; our methods have fulfilled this need in biomedical query answering.

We have illustrated the applicability of our methods to answer complex biomedical queries over large biomedical knowledge resources about drugs, genes, and diseases, such as PharmGKB, DrugBank, BioGRID, CTD, SIDER, Disease Ontology, and Orphadata. The total number of the facts extracted from these resources to answer queries is approximately 10.3 million.

It is important to emphasize here that our definitions and methods for explanation generation are general, so they can be applied to other applications (e.g., debugging, query answering in other domains).

One line of future work is to generalize the notion of an explanation to queries (like Q7) that contain choice expressions.

Acknowledgements

We thank Yelda Erdem (Sanovel Pharmaceutical Inc.) for her help in identifying biomedical queries relevant to drug discovery, Halit Erdogan (Sabanci University) for his help with an earlier version of BioQuery-ASP that he implemented as part of his MS thesis studies, and Hans Tompits (Vienna University of Technology) for his useful comments about the work presented in the article, and pointing out relevant references in the context of debugging ASP programs. We also thank anonymous reviewers for their useful comments and suggestions on an earlier draft. This work was partially supported by TUBITAK Grant 108E229.

References

Apt, K. R. and Bol, R. N. 1994. Logic programming and negation: A survey. *Journal of Logic Programming 19/20*, 9–71.

Baral, C. 2003. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press, Cambridge, UK.

Bodenreider, O., Coban, Z. H., Doganay, M. C., Erdem, E. and Kosucu, H. 2008. A preliminary report on answering complex queries related to drug discovery using answer set programming. In *Proc. of the Third International Workshop on Applications of Logic Programming to the (Semantic) Web and Web Services (ALPSWS 2008)*.

Brain, M. and Vos, M. D. 2005. Debugging logic programs under the answer set semantics. In *Proc. of the Third International Workshop on Answer Set Programming (ASP 2005)*, M. De Vos and A. Provetti (Eds.). Online CEUR-WS.org/Vol-142/page141.pdf.

Brewka, G., Eiter, T. and Truszczynski, M. 2011. Answer set programming at a glance. *Communications of the ACM 54,* 92–103.

Chong, C. R. and Sullivan, D. J. 2007. New uses for old drugs. *Nature 448*, 645–646.

Davis, A. P., King, B. L., Mockus, S., Murphy, C. G., Saraceni-Richards, C., Rosenstein, M., Wiegers, T. and Mattingly, C. J. 2011. The comparative toxicogenomics database: Update 2011. *Nucleic Acids Research 39,* D1067–D1072.

Eiter, T., Ianni, G., Schindlauer, R. and Tompits, H. 2006. Effective integration of declarative rules with external evaluations for Semantic-Web reasoning. In *Proc. of the Third European Conference on the Semantic Web: Research and Applications (ESWC 2006)*, Y. Sure and J. Domingue (Eds.). Springer-Verlag, Berlin, 273–287.

Erdem, E., Erdem, Y., Erdogan, H. and Oztok, U. 2011. Finding answers and generating explanations for complex biomedical queries. In *Proc. of the Twenty-Fifth AAAI Conference on Artificial Intelligence (AAAI 2011)*, W. Burgard and D. Roth (Eds.). AAAI Press, 785–790.

Erdem, E., Erdogan, H. and Oztok, U. 2011. BIOQUERY-ASP: Querying biomedical ontologies using answer set programming. In *Proc. of the Fifth International RuleML2011@BRF Challenge*, S. Bragaglia, C. Viegas Damasio, M. Montali, A. Preece, C. Petrie, M. Proctor and U. Straccia (Eds.). Online CEUR-WS.org/Vol-560/paper8.pdf.

Erdem, E. and Yeniterzi, R. 2009. Transforming controlled natural language biomedical queries into answer set programs. In *Proc. of the BioNLP 2009 Workshop*, K. Bretonnel Cohen, D. Demner-Fushman, S. Ananiadou, J. Pestian, J. Tsujii and B. Webber (Eds.). Association for Computational Linguistics, 117–124.

Ferraris, P. and Lifschitz, V. 2005. Weight constraints as nested expressions. *Theory and Practice of Logic Programming 5*, 45–74.

Gebser, M., Kaminski, R., Koenig, A. and Schaub, T. 2011. Advances in *gringo* series 3. In *Proc of the 11th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2011)*, J. P. Delgrande and W. Faber (Eds.). Springer-Verlag, Berlin, 345–351.

Gebser, M., Kaufmann, B., Neumann, A. and Schaub, T. 2007. Clasp: A conflict-driven answer set solver. In *Proc of the 9th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2007)*, C. Baral, G. Brewka and J. Schlipf (Eds.). Springer-Verlag, Berlin, 260–265.

Gebser, M., Puehrer, J., Schaub, T. and Tompits, H. 2008. A meta-programming technique for debugging answer-set programs. In *Proc. of the 23rd National Conference on Artificial Intelligence (AAAI 2008)*, D. Fox and C. Gomes (Eds.). AAAI Press, 448–453.

Gelder, A. V., Ross, K. A. and Schlipf, J. S. 1991. The well-founded semantics for general logic programs. *Journal of the ACM 38,* 620–650.

Gelfond, M. and Lifschitz, V. 1988. The stable model semantics for logic programming. In *Proc. of the Fifth International Conference and Symposium on Logic Programming (ICLP 1988)*, R. A. Kowalski and K. A. Bowen (Eds.). MIT Press, 1070–1080.

Gelfond, M. and Lifschitz, V. 1991. Classical negation in logic programs and disjunctive databases. *New Generation Computing 9*, 365–385.

Gower, T. 2009. Born again. *Proto Magazine Summer*, 14–19.

Knox, C., Law, V., Jewison, T., Liu, P., Ly, S., Frolkis, A., Pon, A., Banco, K., Mak, C., Neveu, V., Djoumbou, Y., Eisner, R., Guo, A. C. and Wishart, D. S. 2010. Drugbank 3.0: A comprehensive resource for "omics" research on drugs. *Nucleic Acids Research 39,* D1035–D1041.

Kuhn, M., Campillos, M., Letunic, I., Jensen, L. J. and Bork, P. 2010. A side effect resource to capture phenotypic effects of drugs. *Molecular Systems Biology 6*, 343.

Lifschitz, V. 2002. Answer set programming and plan generation. *Artificial Intelligence 138*, 39–54.

Lifschitz, V. 2008. What is answer set programming? In *Proc. of the Twenty-Third AAAI Conference on Artificial Intelligence (AAAI 2008)*, D. Fox and C. Gomes (Eds.). AAAI Press, 1594–1597.

Marek, V. and Truszczyński, M. 1999. Stable models and an alternative logic programming paradigm. In *The Logic Programming Paradigm: A 25-Year Perspective*, Springer, Berlin, 375–398.

McDonagh, E. M., Whirl-Carrillo, M., Garten, Y., Altman, R. B. and Klein, T. E. 2011. From pharmacogenomic knowledge acquisition to clinical applications: The PharmGKB as a clinical pharmacogenomic biomarker resource. *Biomarkers in Medicine 5,* 795–806.

Niemelä, I. 1999. Logic programs with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence 25*, 241–273.

Nogueira, M., Balduccini, M., Gelfond, M., Watson, R. and Barry, M. 2001. An A-Prolog decision support system for the space shuttle. In *Proc. of the Third International Symposium on Practical Aspects of Declarative Languages (PADL 2001)*, I. V. Ramakrishnan (Ed.). Springer-Verlag, Berlin, 169–183.

Oetsch, J., Puehrer, J. and Tompits, H. 2010. Catching the ouroboros: On debugging non-ground answer-set programs. *Theory and Practice of Logic Programming 10,* 513–529.

Oetsch, J., Puehrer, J. and Tompits, H. 2011. Stepping through an answer-set program. In *Proc. of the Eleventh International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2011)*, J. P. Delgrande and W. Faber (Eds.). Springer-Verlag, Berlin, 134–147.

Oztok, U. 2012. Generating explanations for complex biomedical queries. MS Thesis, Sabanci University.

Pontelli, E., Son, T. C. and El-Khatib, O. 2009. Justifications for logic programs under answer set semantics. *Theory and Practice of Logic Programming 9*, 1–56.

Ricca, F., Grasso, G., Alviano, M., Manna, M., Lio, V., Iiritano, S. and Leone, N. 2012. Team-building with answer set programming in the Gioia-Tauro seaport. *Theory and Practice of Logic Programming 12*, 361–381.

Schriml, L. M., Arze, C., Nadendla, S., Chang, Y.-W. W., Mazaitis, M., Felix, V., Feng, G. and Kibbe, W. A. 2012. Disease ontology: A backbone for disease semantic integration. *Nucleic Acids Research 40,* D940–D946.

Simons, P., Niemelä, I. and Soininen, T. 2002. Extending and implementing the stable model semantics. *Artificial Intelligence 138*, 181–234.

Stark, C., Breitkreutz, B.-J., Reguly, T., Boucher, L., Breitkreutz, A. and Tyers, M. 2006. BioGRID: A general repository for interaction datasets. *Nucleic Acids Research 34,* D535–D539.

https://doi.org/10.1017/S1471068413000598 Published online by Cambridge University Press

Syrjanen, T. 2006. Debugging inconsistent answer set programs. In *Proc. of the Eleventh International Workshop on Non-Monotonic Reasoning (NMR 2006)*, J. Dix and A. Hunter (Eds.). Online http://www.in.tu-clausthal.de/uploads/media/NMR_Proc_TR4.pdf.

Tiihonen, J., Soininen, T. and Sulonen, R. 2003. A practical tool for mass-customising configurable products. In *Proc. of the 14th International Conference on Engineering Design (ICED03)*, A. Folkeson, K. Gralén, M. Norell and U. Sellgren (Eds.). Design Society, 1290–1299.