Finding Answers and Generating Explanations for Complex Biomedical Queries

Esra Erdem[a] and **Yelda Erdem**[b] and **Halit Erdogan**[a] and **Umut Oztok**[a]
[a]Faculty of Engineering and Natural Sciences, Sabancı University, İstanbul, Turkey
[b]Research and Development Department, Sanovel Pharmaceutical Inc., İstanbul, Turkey

Abstract

We present new methods to efficiently answer complex queries over biomedical ontologies and databases considering the relevant parts of these knowledge resources, and to generate shortest explanations to justify these answers. Both algorithms rely on the high-level representation and efficient solvers of Answer Set Programming. We apply these algorithms to find answers and explanations to some complex queries related to drug discovery, over PHARMGKB, DRUGBANK, BIOGRID, CTD and SIDER.

1 Introduction

Recent advances in health and life sciences have led to generation of a large amount of biomedical data. To facilitate access to its desired parts, such a big mass of data has been represented in structured forms, like biomedical ontologies and databases. On the other hand, representing these biomedical ontologies and databases in different forms, constructing them independently from each other, and storing them at different locations have brought about many challenges for answering queries about the knowledge represented in these ontologies and databases.

In this paper, we study the following challenging problems: 1) to represent complex biomedical queries that require appropriate integration of relevant knowledge from different knowledge resources and/or that require auxiliary definitions, such as, chains of drug-drug interactions, cliques of genes based on gene-gene relations, or similarity/diversity of genes/drugs; 2) to automatically find answers and informative explanations to these queries.

Consider, for instance, the following queries:

Q1 What are the genes that are targeted by the drug Epinephrine and that interact with the gene DLG4?
Q2 What are the genes that are targeted by all the drugs that belong to the category Hmg-coa reductase inhibitors?
Q3 What are the cliques of 5 genes, that contain the gene DLG4?
Q4 What are the genes related to the gene ADRB1 via a gene-gene relation chain of length at most 3?
Q5 What are the most similar 3 genes that are targeted by the drug Epinephrine?

These queries are important from the point of view of drug discovery. For instance, consider the query Q2. 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 developing Lovastatin (a member of the drug class of statins, used for lowering cholesterol) from Aspergillus terreus (a sort of fungus) in 1979, scientists at Merck derived a new molecule named Simvastatin (a hypolipidemic drug used to control elevated cholesterol). Therefore, identifying genes targeted by a group of drugs automatically by means of queries like Q2 may be useful for experts.

Some of these complex queries, such as Q1 or Q2, can be represented in a formal query language (e.g., SQL/SPARQL) and then answered using Semantic Web technologies. However, queries, like Q4, that require auxiliary recursive definitions (such as transitive closure) cannot be directly represented in these languages; and thus such queries cannot be answered directly using Semantic Web technologies. The experts usually compute auxiliary relations externally, for instance, by enumerating all drug-drug interaction chains or gene cliques, and then use these auxiliary relations to represent and answer a query like Q3 or Q4. Similarity/diversity queries, like Q5, cannot be represented directly in these languages either, and require a sophisticated reasoning algorithm.

We address the challenges described above using Answer Set Programming (ASP) (Lifschitz 2008) as follows:

- The high-level representation language of ASP allows us to formalize auxiliary recursive definitions, aggregates, and defaults; and thus we represent queries in ASP.
- Well-studied foundations and existing computational methods of ASP allow us to develop efficient computational methods for answering complex queries. In particular, we define the relevant part of an ASP program with respect to a query and a set of auxiliary definitions in ASP. Based on the relevancy definition, we develop a method to efficiently compute an answer to a given complex query considering relevant parts of knowledge resources and relevant definitions. The efficiency of our methods is verified experimentally.
- We introduce a method to generate shortest explanations for answers. In particular, we define an explanation as a sequence of rules from the ASP program, that are used to justify an answer; and an explanation tree whose every branch from the root to a leaf denotes an explanation.

Copyright © 2011, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

Based on these definitions, we generate a weighted explanation tree for an answer to a query and find the shortest explanation by finding a least-weighted explanation.

We show the applicability of our methods to answer queries over large biomedical knowledge resources about genes, drugs and diseases, such as PHARMGKB,[1] DRUGBANK,[2] BIOGRID,[3] CTD,[4] and SIDER,[5] using efficient solvers of ASP. For queries that are not concerned about similarity/diversity of genes/drugs, we use the ASP solver CLASP (Gebser et al. 2007). For similarity/diversity queries, we utilize the online methods of (Eiter et al. 2009) for finding similar/diverse solutions, and thus use the ASP solver CLASP-NK, a variant of CLASP.

2 Answer Set Programming

We study representing complex biomedical queries, query answering, and explanation generation in the context of Answer Set Programming (ASP) (Lifschitz 2008)—a declarative programming paradigm with a high-level representation language and efficient solvers. The idea of ASP is to represent knowledge as a "program" and to reason about the knowledge by computing models (called "answer sets" (Gelfond and Lifschitz 1991)) of the program using "ASP solvers" like CLASP.

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

$$F \leftarrow G$$

where F is a propositional atom or $\bot$, and G is a formula; F is called the *head* and G is called the *body* of the rule. We denote the head of a rule r by $H(r)$ and the body by $B(r)$. A rule of the form $F \leftarrow \top$ will be identified with the formula F. A rule of the form $\bot \leftarrow F$ (called a *constraint*) will be abbreviated as $\leftarrow F$.

If every rule of a program is of the form

$$A_0 \leftarrow A_1, \ldots, A_k, \text{not } A_{k+1}, \ldots, \text{not } A_m \qquad (1)$$

where $m \geq k \geq 0$ and each A_i is an atom, then the program is a *normal program*. For a rule r of the form (1), 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 $B^-(r)$.

A program is *positive* if it does not contain any negation. A normal program is *stratified* if there exists a level mapping λ such that, for every rule (1) in that program, $\lambda(A_0) \geq \lambda(A_i)$ for every $1 \leq i \leq k$, and $\lambda(A_0) > \lambda(A_i)$ for every $k < i \leq m$.

When we represent a problem in ASP, we use special constructs of the form $l\{A_1, \ldots, A_k\}u$ (called *cardinality expressions*) where each A_i is an atom and l and u are non-negative integers denoting the "lower bound" and the "upper bound" (Simons, Niemelä, and Soininen 2002). Programs using these constructs can be viewed as abbreviations for normal programs (Ferraris and Lifschitz 2005). Such an expression describes the subsets of the set $\{A_1, \ldots, A_k\}$ whose cardinalities are at least l and at most u. Such expressions when used in heads of rules generate many answer sets whose cardinality is at least l and at most u, and when used in constraints eliminate some answer sets.

A group of rules that follow a pattern can be often described in a compact way using "schematic variables". For instance, we can write the program

$$p_i \leftarrow \textit{not } p_{i+1} \qquad (1 \leq i \leq 7)$$

as follows

$$\textit{index}(1).\ \textit{index}(2).\ \ldots\ \textit{index}(7).$$
$$p(i) \leftarrow \textit{not } p(i+1), \textit{index}(i).$$

The auxiliary predicate $\textit{index}(i)$ is introduced to describe the ranges of variables. ASP solvers compute an answer set for a given program that contains variables, after "grounding" the program, e.g., by the "grounder" GRINGO (Gebser, Schaub, and Thiele 2007). The "definitions" of such auxiliary predicates tell GRINGO how to substitute specific values for variables in schematic expressions. Variables can be also used "locally" to describe the list of formulas. For instance, the rule $1\{p_1, \ldots, p_7\}1$ can be represented as follows: $1\{p(i) : \textit{index}(i)\}1$.

With schematic variables and "constants", an atom can be of the form $p(\vec{t})$ where p is a predicate symbol and $\vec{t} = t_1, ..., t_k$ ($k \geq 0$) with each t_i being either a constant or a schematic variable. Then, for a rule r (1), let $HP(r) = \{p_0\}$, $BP^+(r) = \{p_1, ..., p_k\}$, $BP^-(r) = \{p_{k+1}, ..., p_m\}$, and $BP(r) = BP^+(r) \cup BP^-(r)$. The set of predicate names of a rule r (resp. a normal ASP program Π) is denoted by $\textit{pred}(r) = HP(r) \cup BP(r)$ (resp. $\textit{pred}(\Pi)$). For an ASP program Π, we denote by $\mathcal{B}_\Pi$ the set of all ground instances of the atoms in Π; and by $\textit{Ground}(\Pi)$ the set of all ground instances of the rules in Π.

3 Efficient Query Answering

Consider a normal ASP program Π consisting of facts that describe relations extracted from the knowledge resources, and rules that integrate various concepts from different knowledge resources or that describe some background knowledge.

Suppose that we represent a given biomedical query as an ASP program Q. Therefore, this program may contain auxiliary definitions required to represent the query itself.

The goal is to find the relevant part R of Π with respect to Q, and then efficiently compute an answer to the query (i.e., find an answer set X for R, and extract the answer from X).

Defining relevancy: For a set X of atoms and a set P of predicate symbols, let us denote by X^P the atoms in X with predicate symbols in P.

Let Π_1 and Π_2 be two ASP programs and let P be a set of predicate names. We say that Π_1 is *equivalent* to Π_2 modulo P (denoted $\Pi_1 \equiv_P \Pi_2$) if

- for each answer set $S_1 \in AS(\Pi_1)$, there exists an answer set $S_2 \in AS(\Pi_2)$ such that $S_1^P = S_2^P$, and
- for each answer set $S_2 \in AS(\Pi_2)$, there exists an answer set $S_1 \in AS(\Pi_1)$ such that $S_2^P = S_1^P$.

[1] http://www.pharmgkb.org/
[2] http://www.drugbank.ca/
[3] http://thebiogrid.org/
[4] http://ctd.mdibl.org/
[5] http://sideeffects.embl.de/

Let Π and Q be two ASP programs. Intuitively, Π represents the rule layer over various knowledge resources, and Q describes the query. We say that a subset R of Π is the *relevant* part of Π with respect to Q if the following hold:

- If $R \cup Q$ has no answer set then $\Pi \cup Q$ has no answer set.
- If $R \cup Q$ has an answer set then $R \cup Q \equiv_{pred(Q)} \Pi \cup Q$.

Note that, $R \subseteq \Pi$ is a relevant part of Π with respect to Q if and only if *Ground*(R) is a relevant part of *Ground*(Π) with respect to *Ground*(Q).

Finding the relevant part of a program: Most of the time, the ASP program that integrates various concepts from different knowledge resources or that describes some background knowledge is a stratified ASP program. Therefore, we introduce a method for computing the relevant part of a stratified ASP program with respect to a given biomedical query, which is a general ASP program that may contain auxiliary definitions, constraints, aggregates and so forth. For that, we need the following definitions.

The *predicate dependency graph* DG_Π of a normal ASP program Π (as defined in (Baral 2003)) is a directed graph, whose vertices denote the predicate names of atoms in Π, and edges $\langle p_i, p_j \rangle$ denote the existence of a rule r (1) in Π where p_i (resp. p_j) is the predicate symbol of the atom in $H(r)$ (resp. $B(r)$); edges are labeled by $+$ (resp. $-$) if p_j appears in $B^+(r)$ (resp. $B^-(r)$).

Let Π be a normal ASP program and Q be an ASP program. The set $S_{\Pi,Q}$ of *relevant predicate symbols* of Π with respect to Q is the set of all predicate names reachable (in $DG(\Pi)$) from some vertex in *pred*(Q). Here we consider reachability as the reflexive transitive closure of edges in $DG(\Pi)$. The set $Rel_{\Pi,Q}$ of *relevant rules* of Π with respect to Q is the set of rules r in Π such that $HP(r) \subseteq S_{\Pi,Q}$ whenever $BP(r) \subseteq S_{\Pi,Q}$.

The following proposition allows us to compute the relevant part of a stratified program with respect to a query, as in Algorithm 1.

Proposition 1. *For a stratified normal ASP program* Π *and an ASP program* Q, $Rel_{\Pi,Q}$ *is the relevant part of* Π *with respect to* Q.

Algorithm 1 Answering Queries

Input: A stratified normal ASP program Π and a query Q
Output: An answer set for $\Pi \cup Q$ projected on the predicate names in $R \cup Q$

$DG_\Pi :=$ *DependencyGraph*(Π);
pred$(Q) :=$ *PredicateNames*(Q);
$S_{\Pi,Q} :=$ *RelevantPredicateNames*$(DG_\Pi,$ *pred*$(Q))$;
$Rel_{\Pi,Q} :=$ *RelevantRules*$(\Pi, S_{\Pi,Q})$;
Answer := *ComputeAnswerSet*$(Rel_{\Pi,Q} \cup Q)$;
return *Answer*;

The time complexity of constructing the predicate dependency graph, getting the predicate names, and finding the relevant predicate names and rules is $O(|\Pi| \times |\mathcal{B}_{\Pi \cup Q}| + |$*pred*$(\Pi \cup Q)|^2)$. However, computing an answer set of a given program is NP-hard (Dantsin et al. 2001).

Similarity/diversity queries: Some queries may have too many answers. In such cases, it might be more desirable to compute a subset of answers (e.g., genes) which are similar/diverse to each other with respect to some given distance measure (e.g., based on the functional distance between genes). Also, to be able to analyze relations between genes/drugs, the experts may want to find a set of genes/drugs that are distant/close to a previously computed/known set of genes/drugs. However, no existing software provides such a utility to be able to reason about biomedical knowledge resources.

In (Eiter et al. 2009), the authors introduce two online methods: 1) to compute n answer sets that are similar/diverse to each other by a distance of k; 2) to compute n answer sets that are distant/close to a given set of answer sets by a distance of k. Both methods compute distances with respect to a distance function implemented in the programming language C++, and consider these distances while computing an answer set incrementally by a branch-and-bound approach. Therefore, these methods avoid computing all answer sets to find similar/diverse or distant/close solutions. Utilizing these methods and their implementation CLASP-NK (based on the ASP solver CLASP), we can find similar/diverse or distant/close genes/drugs without computing all answers to queries like Q5.

4 Generating Shortest Explanations

Let Π be the relevant part of a ground normal ASP program with respect to a given query Q (also a ground normal ASP program), that contains rules describing the knowledge extracted from biomedical ontologies/databases, the knowledge integrating them, and the background knowledge. Let X be an answer set for $\Pi \cup Q$, that contains an atom x that characterizes an answer to query Q as described above. The goal is to understand why x is computed as an answer to the query Q, i.e., why x is in X.

Explanations: Let us first define the positive dependency graph of a program: The *positive dependency graph* of a ground normal ASP program Π is a directed graph, whose vertices denote the atoms in Π, and edges $\langle x, y \rangle$ denote the existence of a rule r (1) in Π where $x \in H(r)$ and $y \in B^+(r)$.

Let Π be a ground normal ASP program, X be an answer set for Π, and x be an atom in X. An *explanation* for x with respect to Π and X is a finite sequence $\langle R_1, \ldots, R_n \rangle$ of rules in Π such that the following hold:

- $H(R_i) \cup B^+(R_i) \subseteq X$ and $B^-(R_i) \cap X = \emptyset$ for $1 \leq i \leq n$;
- $H(R_i) \neq H(R_j)$ for $1 \leq i < j \leq n$;
- $H(R_n) = x$,
- for every R_i, for every atom $a \in B^+(R_i)$, there exists a rule R_k $(i \neq k)$ such that $H(R_k) = a$;
- for every R_i $(1 \leq i < n)$, $H(R_i) \in B^+(R_j)$ for some $j \neq i$;
- The positive dependency graph of the ASP program $\{R_1, \ldots, R_n\}$ is acyclic.

Informally, an explanation for an atom is a sequence of rules which are the reasons of that atom being in an answer set.

Consider a program Π, consisting of three rules, $a,\ b \leftarrow a, \mathit{not}\ c$ and b, labeled by $R_1, ...R_3$ respectively. Let $I = \{a, b\}$. Then, the sequence $\langle R_1, R_2\rangle$ is an explanation of b with respect to Π and I; the sequence $\langle R_3\rangle$ is another explanation. As it is illustrated by this example, there may be different explanations for an atom. Among them, we are interested in shortest explanations, with the minimum number of rules. In the example above, the sequence $\langle R_3\rangle$ is a shortest explanation for b.

Explanation trees: To compute shortest explanations, we use "explanation trees" whose vertices correspond to atoms/rules and edges describe dependencies between atoms/rules. Intuitively, an explanation tree whose root corresponds to an atom x represents all possible explanations for x; the idea is then to extract a shortest explanation among all explanations.

An *explanation tree* of x with respect to Π and X is a tree $\langle V, E\rangle$, where V is the set of vertices and E is the set of edges, with the following properties:

- $V \subseteq X \cup \Pi$;
- x is the root of the tree;
- for every vertex $v \in X$, the set of outgoing-edges from v is $E_v = \{\langle v, r\rangle : r \in \Pi,\ H(r) = v,\ B^+(r) \subseteq X,\ B^-(r) \cap X = \emptyset,\ v \notin \mathit{descendant}(r)\}$ (here $\mathit{descendant}(r)$ denotes the descendants of the vertex r);
- for every vertex $v \in \Pi$, the set of outgoing-edges from v is $E_v = \{\langle v, a\rangle : a \in B^+(v)\}$.

By weighting this tree appropriately (as described next), we can extract a shortest explanation for the atom denoted by the root of the tree.

Finding shortest explanations: Our algorithm to find a shortest explanation for an atom x with respect to a normal ASP program Π and an answer set X for Π such that $x \in X$ is summarized in Algorithm 2.

Algorithm 2 Generating Shortest Explanations

Input: A normal ASP program Π, an answer set X for Π, an atom $x \in X$.
Output: A shortest explanation for x wrt Π and X.

$T :=$ createTree$(\Pi, X, x, \{\}, \{\})$;
$T_w :=$ labelTree(Π, X, T, x);
$L :=$ extractShortExp(Π, X, T_w, x);
return L;

First, it generates an explanation tree T. For every vertex v in T that denotes an atom x, it finds every rule R in Π that support x (i.e., every rule $R \in \Pi$ such that $H(R) = x$, $B^+(R) \subseteq X$, and $B^- \cap X = \emptyset$) and assigns these rules as the children of v in T. For every vertex v in T that denotes a rule R, it assigns the atoms in $B^+(R)$ as the children of v.

Next, Algorithm 2 assigns weights to the vertices of the explanation tree T so that we can find a shortest explanation. Since the length of an explanation depends on the number of rules it contains, we need to assign weights to the vertices of the tree in such a way that the weight of a vertex $v \in X$ in the tree corresponds to the number of rules in a shortest explanation for v and the weight of a vertex $u \in \Pi$ in the tree corresponds to the sum of the weights of its children. As a child of a vertex $u \in \Pi$ is a vertex $u' \in X$ and the weight of u' is the number of rules in a shortest explanation of u', the weight of u becomes the least number of rules needed to explain atoms in its body. So, starting at a vertex $v \in X$ and traversing a path by moving towards the vertex which has the smallest weight until a leaf is reached, we can find a shortest explanation of v (vertices $v' \in \Pi$ on the path). Thus, the weight $W(v)$ of a vertex v can be defined as follows:

- If $v \in \Pi$, $W(v) = 1 + \sum_{i=1}^{i=n} W(c_i)$, where c_i is the ith child of v.
- If $v \in X$, $W(v) = min_{1 \le i \le n}(W(c_i))$, where c_i is the ith child of v.

As the final step, Algorithm 2 extracts a shortest explanation for x with respect to Π and X from a given weighted explanation tree T. The explanation computed by Algorithm 2 is indeed the shortest .

Proposition 2. *Let Π be a normal ASP program, X be an answer set for Π, and x be an atom in X. Algorithm 2 finds a shortest explanation for x with respect to Π and X.*

The time complexity of generating a weighted explanation tree and extracting a shortest explanation from that tree, in the worst case, is $h \times b$, where h is the maximum height of the tree and b is the maximum branching factor of a vertex in the tree. Since the set of vertices of the explanation tree is a subset of $X \cup \Pi$, $h = |X| + |\Pi|$ and $b = max\{|X|, |\Pi|\}$. Therefore, the time complexity of Algorithm 2 is $O((|X| + |\Pi|) \times max\{|X|, |\Pi|\})$.

5 Biomedical Queries for Drug Discovery

We applied our methods to find answers to the following biomedical queries, in addition to Q1–Q5:

- Q6 What are the drugs that treat the disease Asthma and target the gene ADRB1?
- Q7 What are the side effects of the drugs that treat the disease Asthma and that target the gene ADRB1?
- Q8 What are the genes that interact with at least 3 genes and that are targeted by the drug Epinephrine?
- Q9 What are the drugs that treat the disease Asthma or that react with 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 a drug in the Statin group?

To answer these queries, we considered the biomedical knowledge resources about genes, drugs and diseases, such as PHARMGKB, DRUGBANK, BIOGRID, CTD and SIDER. In particular, we extracted 347965 triples (as ASP facts) from BIOGRID, 17266 triples from DRUGBANK, 61102 triples from SIDER, 1809 triples from PHARMGKB, 1877799 triples from CTD.

We defined a "rule layer" over these knowledge resources, in the language of GRINGO. This ASP program contains rules to integrate the knowledge resources, such as:

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

which integrates the knowledge extracted from PHARMGKB and CTD, about "which drug targets which gene." The rule layer also defines auxiliary definitions, such as chains of gene-gene relations from a starting gene Y whose length is at most L:

```
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).
```

We represented each query as an ASP program as well. For instance, query Q4 is represented in ASP as follows:

```
start_gene("ADRB1"). max_chain_length(3).
related_gene(G) :- gene_reachable_from(G,L).
answer_exists :- related_gene(G).
:- not answer_exists.
```

Query Q4 has an answer if the ASP program consisting of the rule layer and the program describing the query above has an answer set. If the ASP program has an answer set, answers to Q4 are extracted from the atoms of the form `related_gene(G)`. For instance, the gene CD53 is an answer to Q4 extracted in this way.

Query Q2 is represented in ASP as follows:

```
gene_not_targeted_by_some_drug(GN) :-
  gene_name(GN), not drug_gene(DRG,GN),
  drug_category(DRG,"Hmg-coa ... inhibitors").
gene_targeted_by_all_drugs(GN) :- gene_name(GN),
  not gene_not_targeted_by_some_drug(GN).
answer_exists :- gene_targeted_by_all_drugs(GN).
:- not answer_exists.
```

Query Q3 on the other hand is represented as follows:

```
clique("DLG4").
4{clique(N):gene_gene("DLG4",N)}4.
:- clique(N1), clique(N2),
   not gene_gene(N1,N2), N1 > N2.
answer_exists :- clique(GN).
:- not answer_exists.
```

To answer these queries that are not concerned about similarity/diversity of genes/drugs, we used the ASP solver CLASP(Version 1.3.6).

The similarity query Q5, on the other hand, is represented in ASP as follows:

```
1{pick_a_gene(G):drug_gene("Epinephrine",G)}1.
answer_exists :- pick_a_gene(G).
:- not answer_exists.
```

For the computation of 3 most similar genes, we considered the distance function Δ for a set S of genes, as the maximum distance over the pairwise gene distances in S; the distance between two genes is taken as the number of drugs that target both of these genes subtracted from a large number. Therefore, the less the distance between two genes, the more similar the genes are. This distance function is implemented in C++; and CLASP-NK considers distance of a set of genes while computing an answer, and thus does not even have to compute the gene-gene distance matrix in advance. (Note that, since we consider about 29750 genes, the computation of such a matrix may not be feasible in terms of the time/memory required.) CLASP-NK finds a set S of 3 similar genes targeted by Epinephrine, whose distance is at most k as follows: first it picks a gene g_1 targeted by Epinephrine; after that, it searches for another gene g_2 targeted by Epinephrine such that the distance of g_2 to g_1 is at most k (here the distance is computed on-the-fly), in a branch-and-bound fashion (e.g., if the distance is greater than k, CLASP-NK does not proceed with the search along this branch of the search tree); the search proceeds in this way until 3 genes are computed. With a binary search, one can try to minimize the value of k; however, the underlying online method of CLASP-NK is not complete (due to the gene picked at the first iteration).

Table 1: Experimental Results

Query	with the complete program	with the relevant part
Q1	36.1 sec. Rules: 3662195	7.3 sec. Rules:797129
Q2	36 sec. Rules: 3662689	8.6 sec. Rules: 919827
Q3	36.3 sec. Rules: 3662155	8.5 sec. Rules: 1065150
Q4	36.5 sec. Rules: 3662187	30 sec. Rules: 2991820
Q5	36.2 sec. Rules: 3663880	5.1 sec. Rules: 527064
Q6	36.3 sec. Rules: 3703031	29.9 sec. Rules: 3041876
Q7	36.9 sec. Rules: 3664914	5.7 sec. Rules: 764244
Q8	36.7 sec. Rules: 3665821	6.2 sec. Rules: 765155
Q9	43.8 sec. Rules: 3662159	5.9 sec. Rules: 309488
Q10	36.5 sec. Rules: 3672390	29.9 sec. Rules: 1084597

Table 1 shows, for each query, the computation times (CPU sec.s for a workstation with two 1.60GHz Intel Xeon E5310 Quad-Core Processor and 16GB RAM) and the program sizes, with the complete rule layer and with the relevant part of the rule layer. For instance, for the query Q2, CLASP takes 36 seconds to find an answer with the complete program containing 3662698 rules, whereas it takes 8.6 seconds to find an answer with the relevant part of the program containing 919827 rules. As seen from the other results, it is advantageous to apply our method of query answering with respect to the relevant part of the program.

After we computed answers to these queries, we also generated shortest explanations for queries whose representation does not involve choice/cardinality expressions, as described in the previous section. For instance, the following is a part of the explanation as to why the gene CD53 is an answer to query Q4:

```
gene_reachable_from("PRKCA",2) :-
  gene_gene("PRKCA","DLG4"),
  gene_reachable_from("DLG4",1),
  max_chain_length(3).
gene_reachable_from("CD53",3) :-
  gene_gene("CD53","PRKCA"),
  gene_reachable_from("PRKCA",2),
  max_chain_length(3).
```

```
related_gene("CD53") :-
  gene_reachable_from("CD53",3).
```

According to this explanation, CD53 is an answer to Q4 because it is reachable from ADRB1 with a chain of length 3: CD53 is related to PRKCA which is related to DLG4 which is further related to ADRB1.

Expressing queries and explanations in a natural language: We can express queries Q1, Q6–Q9 in the controlled natural language BIOQUERYCNL of (Erdem and Yeniterzi 2009), as shown above. These queries then are automatically translated into an ASP program, as described in (Erdem and Yeniterzi 2009), after matching the predicate names with the ones appearing in the rule layer. The ASP programs for queries Q1, Q6–Q9 are obtained in this way.

For these queries, we can also express explanations in BIOQUERYCNL. For instance, the following is an explanation for why ADRB1 is an answer to the query Q1:

> the drug Epinephrine targets the gene ADRB1 according to CTD and the gene DLG4 interacts with the gene ADRB1 according to BIOGRID.

Extending BIOQUERYCNL to other queries is part of our ongoing work.

6 Discussion

One of the recent related work on efficient query answering in ASP is (Alviano and Faber. 2010), which describes DLVMAGIC as an extension of DLV based on dynamic magic sets. DLVMAGIC checks whether a given query (as a single atom) is a cautious/brave consequence of a stratified ASP programs. The underlying idea is to modify the given program by means of additional rules for a more efficient computation of an answer. On the other hand, our system does not modify the given program but extracts some part of it for a more efficient computation of an answer. Also our method applies to queries which contain auxiliary definitions with unstratified negation, constraints, aggregates, etc..

The most recent work related to explanation generation in ASP are (Pontelli, Son, and El-Khatib 2009) and (Brain and Vos 2005), in the context of debugging normal ASP programs. (Brain and Vos 2005) studies the question "why is an atom x in an answer set X for an ASP program Π." As an answer to this question, the authors find the rule in Π that supports x with respect to X; whereas we compute a shortest explanation (a sequence of rules) to answer this question.

(Pontelli, Son, and El-Khatib 2009) studies also the same question, and, as an answer, finds a "justification", which is a labeled graph that provides an explanation for the truth values of atoms with respect to an answer set. There is a provable correspondence between a justification and an explanation as we defined above:

Proposition 3. *Let Π be a normal ASP program, X be an answer set for Π, and x be an atom in X. An offline justification of x with respect to X and some $U \in$ Assumptions(Π, X) can be translated into an explanation for x with respect to Π and X. An explanation for an atom x with respect to Π and X can be transformed into a subgraph of an offline justification of x with respect to X and some $U \in$ Assumptions(Π, X).*

The reason for not being able to transform an explanation into a complete justification is the lack of information about why an atom is not in the given answer set. Although there is such a correspondence between a justification and an explanation, (Pontelli, Son, and El-Khatib 2009) finds an explanation whereas we find a shortest explanation.

Also related to our work are (Syrjanen 2006) that studies why a program does not have an answer set, and (Gebser et al. 2008) that studies why a set of atoms is not an answer set. Extending our work to such questions as part of generating explanations for biomedical queries is under consideration.

Acknowledgments This work has been supported by TUBITAK Grant 108E229.

References

Alviano, M., and Faber., W. 2010. Dynamic magic sets for super-consistent answer set programs. In *Proc. of ASPOCP*.

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

Brain, M., and Vos, M. D. 2005. Debugging logic programs under the answer set semantics. In *Proc. of ASP'05*.

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

Eiter, T.; Erdem, E.; Erdogan, H.; and Fink, M. 2009. Finding similar or diverse solutions in answer set programming. In *Proc. of ICLP*, 342–356.

Erdem, E., and Yeniterzi, R. 2009. Transforming controlled natural language biomedical queries into answer set programs. In *Proc. of the Workshop on BioNLP*, 117–124.

Ferraris, P., and Lifschitz, V. 2005. Weight constraints as nested expressions. *TPLP* 5:45–74.

Gebser, M.; Kaufmann, B.; Neumann, A.; and Schaub, T. 2007. clasp: A Conflict-Driven Answer Set Solver. In *Proc. of LPNMR*, 260–265.

Gebser, M.; Phrer, J.; Schaub, T.; and Tompits, H. 2008. A meta-programming technique for debugging answer-set programs. In *Proc. of AAAI'08*.

Gebser, M.; Schaub, T.; and Thiele, S. 2007. Gringo : A new grounder for answer set programming. In *Proc. of LPNMR*, 266–271.

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

Lifschitz, V. 2008. What is answer set programming? In *Proc. of AAAI*.

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

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

Syrjanen, T. 2006. Debugging inconsistent answer set programs. In *Proc. of NMR'06*.