ORB ACADEMIC PUBLISHER

Int. J. Comput. Sci. Theor. App., 2016, vol. 5, no. 2., p. 28-34.
Available online at www.orb-academic.org

International Journal of
Computer Science:
Theory and Application
ISSN: 2336-0984

Bisimilarity, Datalog and Negation

Antoun Yaacoub

Lebanese University, Lebanon
Email: antoun.yaacoub@ul.edu.lb

ABSTRACT

We extend the concept of bisimilarity relation between datalog goals from positive datalog programs to stratified and restricted Datalog programs with negation. The introduction of negation forced us to reconsider the search space and the semantics in order to guarantee and preserve soundness and completeness results. We address the problem of deciding whether two given goals are bisimilar with respect to a given program. When the given programs are stratified or restricted with negation, this problem is decidable.

KEYWORDS

Logic programming — Equivalence of goals — Datalog — Decision problem — Computational complexity — Deductive database

1. Introduction

In [1, 2, 3, 4, 5], a formal framework is given for deciding the bisimilarity relation between Datalog goals. However, all of these papers deal with positive programs only. This paper extends the framework of [1] to interpreters for Datalog programs with negation, i.e. logic programs allowing negative literals in clauses bodies.

We restrict our attention to stratified and restricted Datalog programs, for which a clear semantics is available [6, 7, 8]. One of the well known problems is the occurrence of floundering [9, 10]: a nonground negative goal cannot be answered properly. Thus, goals of the form $\leftarrow \texttt{not}\ p(x)$ are not allowed. Since the SLDNF-trees do not present enough detail in the treatment of negative literals, these trees are augmented and show the construction of subsidiary SLDNF-trees of $\leftarrow G$ when $\texttt{not}\ G$ is selected [11].

The goal of this paper is to suggest the use of equivalence relations between logic programs that take into account the shape of the SLDNF-trees that these programs give rise to. This idea is not new: in automata theory, for instance, many variants of the equivalence relation of bisimilarity have been defined in order to promote the idea that automata with the same trace-based semantics should sometimes not be considered as equivalent if they are not bisimilar [12].

In this paper, we consider Datalog programs with negation. Furthermore, comparing two given Datalog programs with negation and taking into account the shape of the SLDNF-trees they give rise to, necessitates the comparison of infinitely many SLDNF-trees. Thus, we restrict our study to the comparison of two given Datalog goals. We will say that, with respect to a fixed Datalog program P with negation, two given goals are equivalent when their SLDNF-trees are bisimilar.

In this paper, we investigate the computability of the equivalence problem between Datalog goals. In particular, we examine the following decision problems:

- given two Datalog goals F, G and a stratified Datalog program P, determine if the SLDNF-trees of $P \cup F$ and $P \cup G$ are bisimilar.
- given two Datalog goals F, G and a restricted Datalog program with negation P, determine if the SLDNF-trees of $P \cup F$ and $P \cup G$ are bisimilar.

In section 2 of this paper, we will present some basic notions about Datalog programs with negated literals, syntax and semantics. In section 3, we will introduce the concept of bisimulation between Datalog goals with negated literals. In section 4, we will address the problem of deciding whether two given goals are bisimilar with respect to a given stratified Datalog program. In section 5, we will address the same question as in section 4 by considering here restricted Datalog programs with negation. These programs allow a specific kind of recursion in the clauses. Section 6 concludes this paper.

2. Datalog Programs with negated literals

Datalog [13, 14, 15, 16] is a simplified version of Prolog. A Datalog program consists of a finite set of Horn clauses of the form $A_0 \leftarrow A_1, \cdots, A_n$, where each A_i is a literal of the form $p(t_1, \cdots, t_k)$ such that p is a predicate symbol of arity k and the $t_1, \cdots, t_k$ are terms. A term is either a constant or a variable.

The left-hand side of a Datalog-clause is called its head and the right-hand side is called its body. Any Datalog program must satisfy the following condition: each variable which occurs in the head of a clause must also occur in the body of the same clause. A Datalog program P is said to be stratified if it forbids recursion inside negation. For example, any program containing a clause of the form $p \leftarrow q, \texttt{not}\ p$ is not stratified. Nor is any program containing clauses of the form:
$p \leftarrow q, \texttt{not}\ r$
$r \leftarrow s, p$

We say that if P contains a clause of the form $A \leftarrow \cdots, p(\cdots), \cdots,$ then predicate p occurs positively in the clause, and if P contains a clause of the form $A \leftarrow \cdots, \texttt{not}\, p(\cdots), \cdots$ then predicate p occurs negatively in the clause. Note that, a predicate could occur both positively and negatively, even in the same clause.

`not` is negation by failure (NBF): `not` B succeeds when all attempts to prove B fail after a finite number of resolution steps.

A normal logic program P is stratified when there is a partition $P = P_0 \cup P_1 \cup \cdots \cup P_n$ (P_i and P_j disjoint for all $i \neq j$) such that, for every predicate p:

- the definition of p (all clauses with p in the head) is contained in one of the partitions/strata P_i

and, for each $1 \leq i \leq n$:

- if a predicate occurs positively in a clause of P_i then its definition is contained within $\cup_{j \leq i} P_j$
- if a predicate occurs negatively in a clause of P_i then its definition is contained within $\cup_{j < i} P_j$

A program P is said to be stratified if there is any such partition.

Example 2.1. *Stratified program*
Let P be the following program:
$p(X) \leftarrow q(X), \texttt{not}\ r(X);$
$p(X) \leftarrow q(X), \texttt{not}\ t(X);$
$r(X) \leftarrow s(X), \texttt{not}\ t(X);$
$t(a) \leftarrow;$
$s(a) \leftarrow;$
$s(b) \leftarrow;$
$q(a) \leftarrow;$

A possible stratification of P is:
$P = \{p(X) \leftarrow q(X), \texttt{not}\ r(X); p(X) \leftarrow q(X), \texttt{not}\ t(X)\} \cup$
$\{r(X) \leftarrow s(X), \texttt{not}\ t(X)\} \cup$
$\{t(a) \leftarrow; s(a) \leftarrow; s(b) \leftarrow; q(a) \leftarrow\}$

Example 2.2. *Non-stratified program*
Let P be the following program:
$p \leftarrow q, \texttt{not}\ r;$
$r \leftarrow s, \texttt{not}\ p;$
$q \leftarrow;$
$s \leftarrow;$

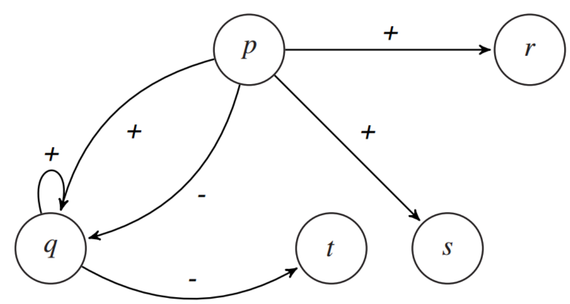
This program cannot be stratified since it contains a recursion through negation.

The dependency graph of a stratified Datalog program P is the graph (N, E) where N is the set of all predicate symbols occurring in P and E is the adjacency relation (edges labeled +/-) defined on N as follows:

- pE^+q (*p refers + to q*) iff P contains a clause of the form $p(\cdots) \leftarrow \cdots, q(\cdots), \cdots$
- pE^-q (*p refers - to q*) iff P contains a clause of the form $p(\cdots) \leftarrow \cdots, \texttt{not}\ q(\cdots), \cdots$

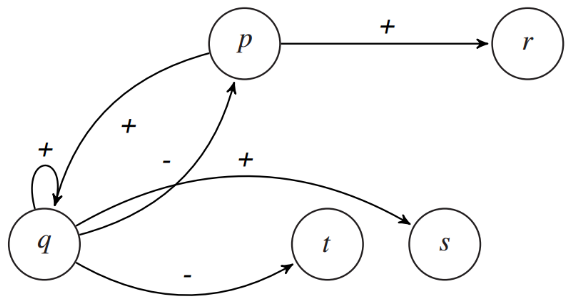
Let E^* be the reflexive transitive closure of E.
A logic program P is stratified iff the dependency graph for P contains no cycles containing a negative edge.
A Datalog program P is said to be restricted iff for all clauses $A_0 \leftarrow A_1, \cdots, A_n$ in P and for all $1 \leq i \leq n-1$, if A_0 is of the form $p(\cdots)$ and A_i is of the form $q(\cdots)$, then `not` $qE * p$

Example 2.3. *Dependency Graph of a Stratified program The dependency graph of the following program is:*
$p \leftarrow q, r;$
$p \leftarrow \texttt{not}\ q, s;$
$q \leftarrow q, \texttt{not}\ t;$



Example 2.4. *Dependency Graph of a non-stratified program The dependency graph of the following program is:*
$p \leftarrow q, r;$
$q \leftarrow \texttt{not}\ p, s;$
$q \leftarrow q, \texttt{not}\ t;$
The logic program P is not stratified since the dependency graph for P contains a cycle containing a negative edge (note the existence of the negative edge directed from node q to node p).

The inference rule we rely on in this paper, is the so-called SLDNF proof procedure. In fact, the computation of a goal/query $G \leftarrow L_1, \cdots, L_m$ is a series of derivation steps:



$$\begin{array}{l} \quad \leftarrow G \\ \theta_1 | \leftarrow G_1 \\ \theta_2 | \leftarrow G_2 \\ \quad \vdots \\ \quad \leftarrow G_{n-1} \\ \theta_n | \Box \end{array}$$

The θ_i are the unifiers (m.g.u.'s) produced by each derivation step. The answer computed θ is the composition of these unifiers $\theta = \theta_1 \cdots \theta_n$.
There are two kinds of derivation steps.

1. Computation rule selects a positive literal $L_i = B$:

$$\theta \left| \begin{array}{l} L_1, \cdots, L_{i-1}, B, L_{i+1}, \cdots L_n \\ \\ \text{resolve with clause } B' \leftarrow M_1, \cdots, M_k \text{ where } B.\theta = B'.\theta \\ \\ \leftarrow (L_1, \cdots, L_{i-1}, M_1, \cdots, M_k, L_{i+1}, \cdots, L_n)\theta \end{array} \right.$$

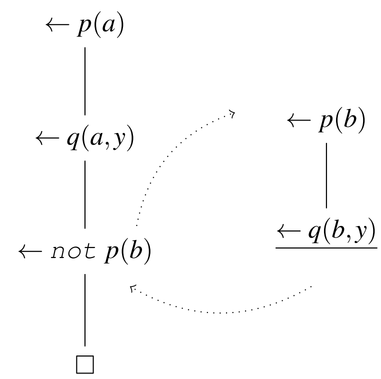
2. Computation rule selects a negative literal $L_i = \texttt{not}\ B$:

$$\iota \left| \begin{array}{l} L_1, \cdots, L_{i-1}, \texttt{not}\ B, L_{i+1}, \cdots L_n \\ \\ \text{sub-computation: all ways of computing B fail} \\ \\ \leftarrow (L_1, \cdots, L_{i-1}, L_{i+1}, \cdots, L_n)\theta \end{array} \right.$$

   ι is the identity substitution (the NBF sub-computation does not generate bindings for variables, it only succeeds or fails. NBF is purely a test.)

The SLDNF-tree of $P \cup G$ is a labeled tree satisfying the above derivation steps.

Example 2.5. *SLDNF-tree*
Let P be the following program:
$p(x) \leftarrow q(x,y), \texttt{not}\ p(y)$,
$q(a,b) \leftarrow$,
and let G be the goal $\leftarrow p(a)$. *The next figure shows the SLDNF-tree for this goal. The success and failure branches are discussed:*



Since the goal $\leftarrow p(b)$ *fails (its associated tree shows a failure branch),* $\leftarrow \texttt{not}\ p(b)$ *succeeds and thus* $\leftarrow p(a)$ *succeeds.*

The computation rule picks out one of the literals of the current goal (query). The computation rule must be *safe*: it must not pick a negative literal containing a variable. This is necessary for soundness. If the current goal contains only negative literals with variables then the computation cannot proceed: it flounders.

The SLDNF proof procedure, which is the most common way of executing normal logic programs is sound, but not complete (because of the possibility of floundering and of infinite computation trees) with respect to this semantics.

3. Bisimulation with negated literals

As stated in [1], a bisimulation is a binary relation between goals such that related goals, even NBF ones, have "equivalent" SLD-trees.

Let P be a Datalog program with negated literals. A binary relation $\mathscr{Z}$ between Datalog goals is said to be a P-bisimulation iff it satisfies the following conditions for all Datalog goals F_1, G_1 such that $F_1 \mathscr{Z} G_1$:

- $F_1 = \Box$ iff $G_1 = \Box$,
- For each resolvent F_2 of F_1 and a clause in P, there exists a resolvent G_2 of G_1 and a clause in P such that $F_2 \mathscr{Z} G_2$,
- For each resolvent G_2 of G_1 and a clause in P, there exists a resolvent F_2 of F_1 and a clause in P such that $F_2 \mathscr{Z} G_2$.
- For the SLD-tree Φ associated to $F_1 = \leftarrow \texttt{not}\ A_1, \cdots$, there exists an SLD-tree Φ' associated to $G_1 = \leftarrow \texttt{not}\ B_1, \cdots$ such that $A_1 \mathscr{Z} B_1$.
- For the SLD-tree Φ' associated to $G_1 = \leftarrow \texttt{not}\ B_1, \cdots$, there exists an SLD-tree Φ associated to $F_1 = \leftarrow \texttt{not}\ A_1, \cdots$ such that $B_1 \mathscr{Z} A_1$.

Note that the main difference between this definition of bisimulation and the one proposed for datalog programs without negated literals is the introduction of two supplementary tests.

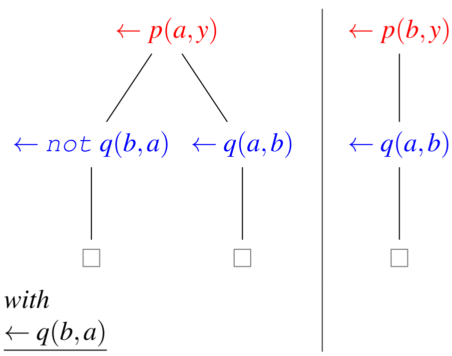
These tests check whether the SLD-trees of the two leftmost negated atoms (if they exist) in a goal are bisimilar. One can check easily that the set of all P-bisimulations is closed under taking arbitrary unions.

Proposition 3.1. *There exists a maximal P-bisimulation, namely the binary relation* $\mathscr{Z}^P_{max}$ *between Datalog goals with negated literals defined as follows:* $F_1 \mathscr{Z}^P_{max} G_1$ *iff there exists a P-bisimulation* $\mathscr{Z}$ *such that* $F_1 \mathscr{Z} G_1$*.* $\mathscr{Z}^P_{max}$ *is an equivalence relation on the set of all Datalog goals.*

Example 3.2. *Let P be the following program:*

$$p(a,b) \leftarrow \mathtt{not}\ q(b,a), \qquad p(a,b) \leftarrow q(a,b),$$
$$q(a,b) \leftarrow, \qquad p(b,b) \leftarrow q(a,b)$$

and let $F = \leftarrow p(a,y)$ *and* $G = \leftarrow p(b,y)$*.*



$\mathscr{Z}$ is not a *P-bisimulation* due the presence of an associated tree for the goal $\leftarrow q(b,a)$ in the left tree and the absence of an associated tree for the goal $\leftarrow q(a,b)$ in the right tree. Since $F \not\mathscr{Z} G$, then $F \not\mathscr{Z}^P_{max} G$.

It was shown in [1] that, in the general case, it is undecidable, given a Prolog program P and Prolog goals F_1, G_1, to determine whether $F_1 \mathscr{Z}^P_{max} G_1$. Thus, we will restrict our language and consider stratified and restricted Datalog programs with negation in order to restore the decidability of our decision problem.

4. Decidability of Bisimulation for Stratified Datalog Programs with Negation

We now study the computational decidability of the following decision problem: $(\pi_{str_{neg}})$ given an stratified Datalog program P with negation and Datalog goals F_1, G_1, determine whether $F_1 \mathscr{Z}^P_{max} G_1$. In this respect, let P be a stratified Datalog program with negation. In `Algorithm 1`, `bothempty(`F_1`,`G_1`)` is a Boolean function returning true iff $F_1 = \Box$ and $G_1 = \Box$, whereas `bothfail(`F_1`,`G_1`)` is a Boolean function returning true iff $F_1 \neq \Box$, `successor(`F_1`)`$=\emptyset$, $G_1 \neq \Box$ and `successor(`G_1`)`$=\emptyset$. Moreover, `successor(.)` is a function returning the set of all resolvents of its argument with a clause of P whereas `get-element(.)` is a function removing one element from the set of elements given as input and returning it.

In order to demonstrate the decidability of $(\pi_{str_{neg}})$, we need to prove the following lemmas for all Datalog goals F_1, G_1:

```
begin
  if (SLDNF-tree(F1) exists and SLDNF-tree(G1) not
    exists)
  or
  (SLDNF-tree(F1) not exists and SLDNF-tree(G1)
    exists)
  or
  (SLDNF-tree(F1) exists with root A and
    SLDNF-tree(G1) exists with root B and
    bisim1(A,B) = false) then
    | return false
  end
  else
    if bothempty(F1,G1) or bothfail(F1,G1) then
      | return true
    end
    else
      SF <-- successor(F1)
      SG <-- successor(G1)
      if SF ≠ ∅ and SG ≠ ∅ then
        SF' <-- SF
        while SF' ≠ ∅ do
          F2 <-- get-element(SF')
          found-bisim <-- false
          SG' <-- SG
          while SG' ≠ ∅ and
            found-bisim = false do
              G2 <-- get-element(SG')
              found-bisim <-- bisim1(F2,G2)
          end
          if found-bisim = false then
            | return false
          end
        end
        SG' <-- SG
        while SG' ≠ ∅ do
          G2 <-- get-element(SG')
          found-bisim <-- false
          SF' <-- SF
          while SF' ≠ ∅ and
            found-bisim = false do
              F2 <-- get-element(SF')
              found-bisim <-- bisim1(G2,F2)
          end
          if found-bisim = false then
            | return false
          end
        end
        return true
      end
      else
        | return false
      end
    end
  end
end
```

Algorithm 1: function bisim1(F_1,G_1)

Lemma 4.1 (Termination)**.** `bisim1(`F_1`,`G_1`)` *terminates.*

Lemma 4.2 (Completeness)**.** *If* $F_1 \mathscr{Z}^P_{max} G_1$*, then* `bisim1(`$F_1$`,`$G_1$`)` *returns* `true`.

Lemma 4.3 (Soundness)**.** *If* `bisim1(`F_1`,`G_1`)` *returns* `true`*, then* $F_1 \mathscr{Z}^P_{max} G_1$.

In order to prove the termination, we will introduce two notions. The first is the level of a goal and the second is a measure which calculates the maximum number of steps needed to reach a leaf from some node in a SLDNF-tree.

Let $F = \leftarrow A_1, \cdots, A_n, \texttt{not } B_1, \cdots, \texttt{not } B_m$, the level of F is defined as follows:

$$level(F) = \begin{cases} max\{level(A_1), \cdots, level(A_n), level(B_1), \cdots, level(B_m)\} \\ 0 \text{ iff } \begin{cases} \forall A_i \in F, i = 1, \cdots, n; & level(A_i) = 0 \\ \forall B_j \in F, j = 1, \cdots, m; & level(b_j) = 0 \end{cases} \end{cases}$$

Consider also H_1 a Datalog goal with negated atoms. Let:
$M(H_1) = max\{L(D) |\ D = (H_1 \Rightarrow \cdots \Rightarrow H_n)$ a derivation from $H_1\}$
and let:

$$L(D) = \begin{cases} 0 & \text{if} \quad D = (H_1) \\ \sum_{i=1}^{n-1} C(H_i, H_{i+1}) & \text{if} \quad D = (H_1 \Rightarrow \cdots \Rightarrow H_n) \end{cases}$$

where

$$C(H_i, H_{i+1}) = \begin{cases} 1 & \text{if} \quad H_i = \leftarrow A, \cdots \\ M(\leftarrow A) + 1 & \text{if} \quad H_i = \leftarrow \texttt{not } A, \cdots \end{cases}$$

We can prove by induction on the level of goals, that the M-function is well-defined.

Proof. Let $F = \leftarrow H_1, \cdots, H_n$ be a goal of level zero, then for every $i = 1, \cdots, n$, $level(H_i) = 0$. Thus, each H_i is either a fact or/and an atom that appears in the body of some clause(s). Thus, the function C for goals of level 0 is always equal to 1. Consequently, $M(F) = n - 1$.
Suppose that $M(F)$ is defined for all goals of level $\leq k$.
Consider a goal F of level $k+1$. Consider also an atom A (positive or negative) in the goal F of level $k+1$. Since A is of level $k+1$, then A will be resolved into resolvents of level $\leq k$. Thus F of level $k+1$ will be resolved to some goals of level $\leq k$. □

For the previous example, $M(\leftarrow p(a,y)) = 3$, and $M(\leftarrow p(b,y)) = 2$.

Let $\ll$ be the binary relation on the set of all pairs of Datalog goals defined by: $(F_2, G_2) \ll (F_1, G_1)$ iff

- $M(F_2) < M(F_1)$,
- $M(G_2) < M(G_1)$,

$\ll$ is a well-founded partial order on the set of all pairs of goals.

Proof of Lemma 1. The proof is done by $\ll$-induction on (F_1, G_1). Let (F_1, G_1) be such that for all (F_2, G_2), if $(F_2, G_2) \ll (F_1, G_1)$ then `bisim1(`F_2`,`G_2`)` terminates. Since every recursive call to `bisim1` that is performed along the execution of `bisim1(`F_1`,`G_1`)` is done with respect to a pair (F_2, G_2) of goals such that $(F_2, G_2) \ll (F_1, G_1)$, then `bisim1(`$F_1$`,`$G_1$`)` terminates.□

Proof of Lemma 2. Let us consider the following property: $(Prop_1(F_1, G_1))$ if $F_1 \mathscr{Z}^P_{max} G_1$ then `bisim1(`F_1`,`G_1`)` returns `true`. Again, we proceed by $\ll$-induction. Suppose (F_1, G_1) is such that for all (F_2, G_2), if $(F_2, G_2) \ll (F_1, G_1)$ then $Prop_1(F_2, G_2)$. Let us show that $Prop_1(F_1, G_1)$. Suppose $F_1 \mathscr{Z}^P_{max} G_1$. Hence, for all successors F_2 of F_1, there exists a successor G_2 of G_1 such that $F_2 \mathscr{Z}^P_{max} G_2$, and conversely. Seeing that the logic program is stratified, then $(F_2, G_2) \ll (F_1, G_1)$. By induction hypothesis, $Prop_1(F_2, G_2)$. Since $F_2 \mathscr{Z}^P_{max} G_2$, then `bisim1(`$F_2$`,`$G_2$`)` returns `true`. As a result, one sees that `bisim1(`F_1`,`G_1`)` returns `true`.□

Proof of Lemma 3. It suffices to demonstrate that the binary relation $\mathscr{Z}$ defined as follows between Datalog goals is a bisimulation: $F_1 \mathscr{Z} G_1$ iff `bisim1(`F_1`,`G_1`)` returns `true`. Let F_1, G_1 be Datalog goals such that $F_1 \mathscr{Z} G_1$. Hence, `bisim1(`F_1`,`G_1`)` returns `true`. Thus, obviously, $F_1 = \Box$ iff $G_1 = \Box$, and the first condition characterizing bisimulations holds for $\mathscr{Z}$. Now, suppose that F_2 is a resolvent of F_1 and a clause in P. Since `bisim1(`F_1`,`G_1`)` returns `true`, then there exists a resolvent G_2 of G_1 and a clause in P such that `bisim1(`F_2`,`G_2`)` returns `true`, i.e. $F_2 \mathscr{Z} G_2$. As a result, the second condition characterizing bisimulations holds for $\mathscr{Z}$. The third condition characterizing bisimulations holds for $\mathscr{Z}$ too, as the reader can quickly check. The same discussion applies for any couple of associated trees to negated bisimilar goals, verifying thus the fourth and fifth conditions. Thus $\mathscr{Z}$ is a bisimulation. □

As a consequence of lemmas 1 – 3, we have:

Theorem 4.4. `Algorithm 1` *is a sound and complete decision procedure for* $(\pi_{str_{neg}})$.

It follows that $(\pi_{str_{neg}})$ is decidable.

5. Decidability of Bisimulation for Restricted Datalog Programs with Negation

In this section, we will consider restricted programs with negation with a restriction that we will not allow any dependencies between the rightmost negative atom (if it is exists) in a clause and the atom in its head.

We study here the computational decidability of the following decision problem: $(\pi_{res_{neg}})$ given a restricted Datalog program P with negation and Datalog goals F_1, G_1, determine whether $F_1 \mathscr{Z}^P_{max} G_1$. In this respect, let P be a restricted Datalog program with negation. In `Algorithm 2`, `bothempty(`F_i`,`G_i`)`, `bothfail(`F_i`,`G_i`)` and `occur((`$F_1 \Rightarrow \cdots \Rightarrow F_i$`),(`$G_1 \Rightarrow \cdots \Rightarrow G_i$`))` are similar to the corresponding functions used in

```
begin
  if (SLDNF-tree(F_1) exists and SLDNF-tree(G_1) not
    exists) or (SLDNF-tree(F_1) not exists and
    SLDNF-tree(G_1) exists) or (SLDNF-tree(F_1) exists
    with root A and SLDNF-tree(G_1) exists with root B
    and bisim2(A,B) = false) then
      return false
  end
  else
    if bothempty(F_i,G_i) or bothfail(F_i,G_i) or
      occur((F_1 ⇒ ··· ⇒ F_i),(G_1 ⇒ ··· ⇒ G_i)) then
        return true
    end
    else
      SF ⟵ successor(F_i)
      SG ⟵ successor(G_i)
      if SF ≠ ∅ and SG ≠ ∅ then
        SF' ⟵ SF
        while SF' ≠ ∅ do
          F' ⟵ get-element(SF')
          found-bisim ⟵ false
          SG' ⟵ SG
          while SG' ≠ ∅ and
            found-bisim = false do
              G' ⟵ get-element(SG')
              found-bisim ⟵ bisim2((F_1 ⇒
                ··· ⇒ F_i ⇒ F'),(G_1 ⇒ ··· ⇒
                G_i ⇒ G'))
          end
          if found-bisim = false then
            return false
          end
        end
        SG' ⟵ SG
        while SG' ≠ ∅ do
          G' ⟵ get-element(SG')
          found-bisim ⟵ false
          SF' ⟵ SF
          while SF' ≠ ∅ and
            found-bisim = false do
              F' ⟵ get-element(SF')
              found-bisim ⟵ bisim2((G_1 ⇒
                ··· ⇒ G_i ⇒ G'),(F_1 ⇒ ··· ⇒
                F_i ⇒ F'))
          end
          if found-bisim = false then
            return false
          end
        end
        return true
      end
      else
        return false
      end
    end
  end
end
```

Algorithm 2: function bisim2(($F_1 \Rightarrow \cdots \Rightarrow F_i$),($G_1 \Rightarrow \cdots \Rightarrow G_i$))

`Algorithm 1`. As the reader can see, `Algorithm 2` is very similar to the other algorithms.
In order to demonstrate the decidability of ($\pi_{res_{neg}}$), we need to prove the following lemmas for all Datalog goals F_1, G_1.

Lemma 5.1 (Termination)**.** `bisim2((`F_1`),(`G_1`))` *terminates.*

Lemma 5.2 (Completeness)**.** *If* $F_1 \mathscr{Z}^P_{max} G_1$*, then* `bisim2((`$F_1$`),(`$G_1$`))` *returns* `true`.

Lemma 5.3 (Soundness)**.** *If* `bisim2((`F_1`),(`G_1`))` *returns* `true`*, then* $F_1 \mathscr{Z}^P_{max} G_1$.

One can follow the same proofs presented for the positive case [1] except that we need to prove that for a restricted program P with negation and a goal F_1, all its derived subgoals in all the derivations must be bounded.
For this, suppose that the derived subgoals are not bounded and suppose that $F_1 = \leftarrow A$.
Consider a derivation D of the form $D = (F_1 \Rightarrow F_2 \Rightarrow \cdots \Rightarrow F_n)$.
One can decompose each $F_i (i \geq 2)$ into the concatenation of suffixes of body of clauses:
$F_i = L_1^1, \cdots, L_1^{k_1}, \cdots, L_i^1, \cdots, L_i^{k_i}, L_{i+1}^1, \cdots, L_{i+1}^{k_{i+1}}, \cdots, L_n^1, \cdots, L_n^{k_n}$, where $1 \leq i < n$, $k_1 > 0, \cdots, k_n > 0$. Note that, for example, $L_i^1, \cdots, L_i^{k_i}$ is a suffix of the body of the clause $p_i \leftarrow Q_i$ in the program P, and that $L_{i+1}^1, \cdots, L_{i+1}^{k_{i+1}}$ is a suffix of the body of the clause $p_{i+1} \leftarrow Q_{i+1}$. Since F_i can be decomposed as described, then the head atom p_i of the clause $p_i \leftarrow Qi$ can be unified with the atom L_{i+1}^0 which precedes L_{i+1}^1 contained in Q_{i+1}.
Since $level(L_{i+1}^0) < level(p_{i+1})$ (as p_{i+1} depends on L_{i+1}^0), thus $level(p_i) < level(p_{i+1})$. This contradicts the fact that F_i is not bounded.
As a consequence of lemmas 4 – 6, we have:

Theorem 5.4. `Algorithm 4` *is a sound and complete decision procedure for* ($\pi_{res_{neg}}$).

It follows that ($\pi_{res_{neg}}$) is decidable.

6. Conclusion

In this paper, we have introduced the concept of bisimulation between datalog goals with negation: two Datalog goals are bisimilar with respect to a given program when their SLDNF-trees are bisimilar. As proved, when the given logic program is stratified or restricted with negation, the problem of deciding whether two given goals are bisimilar is decidable. The proof of decidability of bisimulation problem for restricted logic program with negation that we presented in Section 5 is based on techniques that were developed in [17] for detecting loops in logic programming.

We have shown the decidability of the bisimilarity relation between negated Datalog goals. We are currently working toward evaluating the complexity of our algorithms.

Future work can be dedicated also to the study of the adaptation of the flow definitions, presented in [1, 2, 3, 4] for positive Datalog programs, to Datalog programs with negation. These flows could also be studied for Datalog programs with negation equiped with bottom up evaluation techniques.

References

[1] BALBIANI, P., AND YAACOUB, A. Deciding the bisimilarity relation between Datalog goals (regular paper). In *European Conference on Logics in Artificial Intelligence (JELIA), Toulouse, 26/09/2012-28/09/2012* (http://www.springerlink.com, septembre 2012), L. Fariñas del Cerro, A. Herzig, and J. Mengin, Eds., Springer, pp. 67–79.

[2] YAACOUB, A. Towards an information flow in logic programming. *International Journal of Computer Science Issues (IJCSI) 9*, 2 , 2012.

[3] YAACOUB, A., AND AWADA, A. Inference Control On Information Flow In Logic Programming. *International Journal of Computer Science: Theory and Application (IJCSTA)*, 2015, Vol. 3, Issue.: 1, p. 13-22.

[4] YAACOUB, A. Flux de l'information en programmation logique *Université Paul Sabatier - Toulouse III*, thèse de doctorat, 2012.

[5] YAACOUB, A. AWADA, A. AND KOBEISSI, H. Information Flow in Concurrent Logic Programming. *British Journal of Mathematics & Computer Science*, 2015, Vol. 5, Issue.: 3, p. 367-382

[6] VAN GELDER, A., ROSS, K., AND SCHLIPF, J.S. Unfounded Sets and Well-Founded Semantics for General Logic Programs *Proceedings of the 7th Symposium on Principles of Databases*, 1988, pp. 221–230 ACM SIGACT-SIGMOD

[7] GELFOND, M. AND LIFSCHITZ, V. The Stable Model Semantics for Logic Programming *,in: R. Kowalski, K. Bowen (Eds.), Proceedings of the 5th International Conference on Logic ProgrammingMIT Press, Cambridge, MA*, 1988, pp. 1070–1080

[8] PRZYMUSINSKI, T.C. On the Declarative Semantics of Deductive Databases and Logic Programs *,in: J. Minker (Ed.), Foundations of Deductive Databases and Logic ProgrammingMorgan Kaufmann, Los Altos, CA*, 1987, pp. 193–216

[9] KEMP, D.B. AND TOPOR, R.W. Completeness of a Top-Down Query Evaluation Procedure for Stratified Databases *,in: R. Kowalski, K. Bowen (Eds.), Proceedings of the 5th International Conference on Logic ProgrammingMIT Press, Cambridge, MA*, 1988, pp. 178–194

[10] SEKI, H. AND ITOH, H. A Query Evaluation Method for Stratified Programs under the Extended CWA *,in: R.A. Kowalski, K.A. Bowen (Eds.), Proceedings of the 5th International Conference on Logic ProgrammingMIT Press, Cambridge, MA*, 1988, pp. 195–211

[11] PRZYMUSINSKI, T.C. On the Declarative and Procedural Semantics of Logic Programs *J. Automated Reasoning, 5*, 1989, pp. 167–205

[12] Sangiorgi, D.: On the origins of bisimulation and coinduction. In: ACM Trans. Program. Lang. Syst., pp. 1–41. ACM, USA, 2009

[13] STEFANO, C., AND GEORG, G., AND LETIZIA, T. What you always wanted to know about Datalog (and never dared to ask). *Knowledge and Data Engineering, IEEE Transactions*, 1989, Vol. 1, Issue.: 1, p. 146–166.

[14] CLARK K.L. Negation as Failure *,in: H. Gallaire, J. Minker (Eds.)*, Logic and Data Bases, Plenum, New York, 1978, pp. 293–322

[15] LLOYD, J.W. *Foundations of Logic Programming*, 2nd Edition. Springer, 1987.

[16] ULLMAN, J. *Principles of Databases and Knowledge base Systems, Volume I and II*. Computer Science Press, 1988.

[17] Bol, R.N., Apt, K.R., Klop, J.W.: An Analysis of Loop Checking Mechanisms for Logic Programs. In: Theoretical Computer Science, vol. 86, pp. 35–79, 1991