ORB ACADEMIC PUBLISHER

Int. J. Comput. Sci. Theor. App., 2016, vol. 5, no. 2., p. 35-48.
Available online at www.orb-academic.org

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

Information Flow in Datalog using Very Naive and Semi Naive Bottom-Up Evaluation Techiques

Antoun Yaacoub[1], Ali Awada[2]

[1] *Lebanese University, Lebanon*
[2] *Laboratoire de Recherche en Informatique Fondamentale et Appliquées (LaRIFA) - Lebanese University, Lebanon*
Email: antoun.yaacoub@ul.edu.lb, al-awada@ul.edu.lb

ABSTRACT
Information flow in logic programming is well defined using the usual top-down evaluation approach. In this paper, we tackle the same question using a buttom up evaluation approach using two algorithms, Very Naive and Semi Naive algorithms. We will show that, using these techniques, the existence of the information flow is decidable, so that the flow of information can be detected in all cases. Finally, we will study the computational complexities of each decision problem and prove that it's EXPTIME-complete.

KEYWORDS
Logic programming — Information flow — Datalog — Decision problem — Computational complexity — Deductive database.

1. Introduction

Data security refers to protective measures that are applied to prevent unauthorized access, disclosure and modification to data stored on computers, databases and websites. Data security is the main priority for organizations of every size and genre. The control of information flow is one of the aspects of data security. Information flow describes controls that regulate the disclosure of information; controls to prevent leaking confidential data.

The theory of information flow is well defined for imperative programming [1, 2, 3, 4, 5, 6, 7]. Intuitively, information flows from an object x to an object y if the application of a sequence of commands causes the information initially in x to affect the information in y. For example, the sequence $tmp := x; y := tmp;$ has information flowing from x to y because the value of x at the beginning of the sequence is revealed when the value of y is determined at the end of the sequence.

As logic programming is considered as a declarative method of knowledge representation and programming, it is thus well-suited for both representing data and describing desired outputs. Logic programming was developed in the early 1970s based on work in automated theorem proving, in particular, on Robinson's resolution principle.

Many researchers related first order logic to data security. A framework that models access control using logic programming with set constraints of a computable set theory was presented by Wang *et al.* [10]. A logic-based access control approach for Web Service endpoint was defined by Coetzee *et al.* [12]. Open logic-based security language was introduced by DeTreville [8] that encodes security statements as components of communicating distributed logic programs, used to express security statements in a distributed system. Bai *et al.* [11] proposed a knowledge oriented formal language to specify the system security policies and their reasoning in response to system resource access request. The semantics of the language was provided by translating it into epistemic logic program in which knowledge related modal operators are employed to represent agents' knowledge in reasoning. Bertino *et al.* [9] proposed a formal framework for reasoning about access control models. The framework they proposed is based on a logical formalism and is general enough to model different access control models. Each instance of the proposed framework corresponded to a C-Datalog program, interpreted according to a stable model semantics.

Information flow in logic programming was introduced in [13, 14, 15]. Yaacoub *et al.* defined the information flow in logic programming and developed mechanisms to detect such flows and proposed the notion of indistinguishability of flow and elaborated definitions of protection mechanisms, secure mechanisms, precise mechanisms and confidentiality policies based on this notion. They gave a secure and precise protection mechanism that prohibits any undesirable inferences and minimizes the number of denials of legitimate actions. The main results of their researches have shown that, it is undecidable to check whether there is a flow of information or not in the general setting of logic program. But, if they restricted the language to Datalog programs using top-down evaluation, they achieved the decidability of the flow. They also worked on hierarchical,

single variable occurrence (svo), non variable introducing (nvi) Datalog programs.

In this paper, we study the flow of information in different evaluation such as bottom up evaluation. Trying to check if with bottom up algorithms, information flow can be decidable, without the need to restrict the language.

In section 2, we briefly discuss the syntax and semantics and highlight information flow detection mechanisms based on top down evaluation in Datalog logic programming along with some detection mechanims complexity results. At the end of this section, Very Naive and Semi Naive buttom-up evaluation techniques in Datalog are presented.

In section 3, we will present two information flow definitions in logic programming based on bottom up evaluation in Datalog programs. Moreover, we will highlight on the implications between these definitions, in addition to the implications between flow definitions of top down evaluation vs flow definitions of bottom up evaluation. Lastly, in section 4, we study the decidability and computational complexity of flow based on bottom up evaluation using Very Naive and Semi Naive evaluation techniques.

2. Framework

2.1 Syntax and Semantics

In our context, we will deal with the first order logic [17] and each clause is a Horn clause. A Horn clause is a clause of the form $A_1 \leftarrow B_1, B_2, \cdots, B_n$, where A_1 is called the head of the clause and $B_1, \cdots, B_n$ its body. The body is composed of positive literals (appear in a positive manner without negation). We will use $p, q, \cdots$ to represent predicates, $a, b, \cdots$ to represent constants, function symbols are of the form $f(x)$, a fact is of the form $A_1 \leftarrow$ where its body is empty, and a goal is of the form $\leftarrow B_1$. Herbrand Universe, denoted $U_{L(P)}$, is the set of constants that appear in some program P.

Prolog: Prolog is a declarative programming language (it is called prolog from the combination of programming and logic). It deals with first order logic; the logic program is expressed in terms of relations, represented as facts and rules. It allows the existence of function symbols in its relations. Prolog can be executed by running some query. Its evaluation is based on a top down approach. Let's list some of the characteristics of this approach:

- Goal directed: this means that the program starts from the given query and divides it into sub goals, then trying to solve each sub goal to reach the answer.
- It selects the leftmost atom in the body; for example, if we have $p(x,y) \leftarrow q(x,z), p(z,y)$, it selects $q(x,z)$ first and resolves it.
- It follows depth first search and backward chaining: this means that when it divides the goal into sub goals, and when it reaches the level of the facts, it starts to recollect the data in a backward manner.

On the other hand, Prolog has a drawback that it may stuck in an infinite loop so that it will not reach the desired goal. An SLD-tree for a goal G in a logic program P is a (possibly infinite) tree with each node labeled with a goal and each branch labeled with a substitution. The shape of the SLD-tree generated for a query depends on the particular computation rule employed in the SLD-resolution; indeed, the choice of rule can have a tremendous influence on the size of the corresponding SLD-tree.

Datalog: Datalog is a subset of Prolog. It is a declarative programming language as well. Datalog is the combination between database and logic (data-log) used for deductive database. It is characterized by:

- Function symbols are not allowed in Datalog whereas in Prolog they are allowed, for example: $p(x)$ is admissible while $p(f(x))$ is not.
- It is safe, this means that every variable appears in the head of the clause must appear in a non arithmetic positive literal (not negated) in the body of the clause. For instance: $p(x,y) \leftarrow q(x,z), p(z,y)$ is allowed whereas $p(x,y) \leftarrow q(x,x), p(x,z)$ is not (since variable y doesn't appear in the body).
- Terms are only variables or atoms.
- Does not have the cut operator(!)[1].
- Predicates that appear in the head of the clause are called intensional predicates(IDB), while those that appear only in the body of the clause are called extensional predicates (EDB) [19].

Example 2.1. *Datalog Let P be a Datalog program with the following input:*
$p(x,y) \leftarrow q(x,y)$
$q(a,b) \leftarrow$
Let the goal be $\leftarrow p(x,y)$.

IDB predicate: $\{p\}$ *(it appears in the head of the clause).*
EDB predicate: $\{q\}$ *(it appears in the body of the clause only).*

2.2 Information flow in Datalog Logic Programming

The theory of information flow in logic programming is based on the innovative work done by Yaacoub *et al*. [13, 14, 18]. They proposed several definitions for flow detection. These definitions correspond to what can be observed by the user when a query $G(x,y)$ is run on a logic program P.

The first definition is based on Success/Failure (SF) of the goals. Let P be a Datalog program and $G(x,y)$ be a two variables goal. There is a flow of information from x to y in P ($x \xrightarrow{SF}{}^{P}_{G} y$ based on SF in goal G and Program P) iff there exists $a, b \in U_{L(P)}$ such that $P \cup \{G(a,y)\}$[2] succeeds and $P \cup \{G(b,y)\}$

[1] Used to prevent unwanted backtracking.
[2] $P \cup \{G(a,y)\}$ means running the goal $G(a,y)$ on the program P.

fails. This means that when the user only sees the outputs of computations in terms of successes and failures, $\exists$ two different $a,b \in U_{L(P)}$ such that the user can distinguish between the outputs of the goals without seeing what concerns a and b. Now let us show an example to make it clearer.

Example 2.2. *Let P be a program:*
$p(a,b) \leftarrow$;
$p(c,b) \leftarrow$;
and let $G(x,y)$ *be the following goal:* $\leftarrow p(x,y)$
Since $P \cup \{G(a,y)\}$ *succeeds and* $P \cup \{G(b,y)\}$ *fails, then* $x \xrightarrow{SF}{}^{P}_{G} y$ *based on SF, goal G and Program P. In other words, if we hide a and b from the goals and since the first goal succeeds and the second one fails, we can distinguish by looking at the facts that the first constant is a whereas the second one is b, consequently the flow occurs.*

The second definition is based on the substitution answers of the goals. Let P be a Datalog program and $G(x,y)$ be a goal. We can say that there is an information flow from x to y in $G(x,y)$ with respect to substitution answers in P iff $\exists$ a,b $\in U_{L(P)}$ such that $\theta(P \cup \{G(a,y)\}) \neq \theta(P \cup \{G(b,y)\})$. In this definition, the user only sees the outputs of computations in terms of substitution answers. Consequently, there is a flow of information from x to y if this user can distinguish the output of $P \cup \{G(a,y)\}$ and the output of $P \cup \{G(b,y)\}$.

Example 2.3. *Let us consider the following example of program P:*
$p(a,b) \leftarrow$;
$p(c,d) \leftarrow$;
and let $G(x,y)$ *be the following goal:* $\leftarrow p(x,y)$
$\theta(P \cup \{G(a,y)\}) = \{y \rightarrow b\}$ *and* $\theta(P \cup \{G(b,y)\}) = \emptyset$*, then there is a flow from x to y* ($x \xrightarrow{SA}{}^{P}_{G} y$) *based on substitution answers in* $G(x,y)$ *and P. In other words, let us hide both a and b from the goals. The first answer is "b", by looking at the facts we know that y is substituted by "b" then "a" is the hidden constant. Whereas in the second goal, y is substituted by empty set, this means that the first constant is either "b" or "d", consequently, there is a flow.*

Result:

As for results obtained from [13], implications between the definitions of flows are as follows:

- The existence of a flow with respect to substitution answers does not entail the existence of a flow with respect to successes and failures.
- If $x \xrightarrow{SF}{}^{P}_{G} y$ (based on success/failure in goal $G(x,y)$ and program P) then $x \xrightarrow{SA}{}^{P}_{G} y$ (based on substitution answers in goal $G(x,y)$ and program P).

As in [13], the complexity results obtained for the following two decision problems

$$\pi_{SF} \begin{cases} \text{Input: A logic program } P\text{, a two variables goal } G(x,y) \\ \text{Output: Determine whether } x \xrightarrow{SF}{}^{P}_{G} y \end{cases}$$

$$\pi_{SA} \begin{cases} \text{Input: A logic program } P\text{, a two variables goal } G(x,y) \\ \text{Output: Determine whether } x \xrightarrow{SA}{}^{P}_{G} y \end{cases}$$

are as follows:

- In the general settings of Prolog, the two decision problems are undecidable.

If the language is restricted to Datalog programs then determining the existence of information flows becomes decidable.

- π_{SF} is EXPTIME-complete for Datalog programs.
- π_{SA} is EXPTIME-complete for Datalog programs.

2.3 Very Naive and Semi Naive Bottom Up Evaluation in Datalog

Datalog has two evaluation approaches, **top down** and **bottom up**. In what follows, we will focus on the bottom up evaluation. Now let's list some of its characteristics:

- Not goal directed: this means that it does not start searching for the desired answer from the query (the head of the clause), it starts from the body of the clause.
- Starts from known facts and tries to infer new facts.
- It stops when no more new facts can be generated, i.e it reaches a fixed point.
- Always terminates.
- The order of the rules is irrelevant, while in top down it is extremely important. For example, in bottom up evaluation $p(x,y) \leftarrow q(x,z), p(z,y)$ is the same as $p(x,y) \leftarrow p(x,z), q(z,y)$, whereas in top down it differs. For example, $p(x,y) \leftarrow p(x,z), q(z,y)$ will lead to an infinite loot using top down evaluation; but, using bottom up evaluation, it will terminate.

In the upcoming parts, we will highlight on two of the algorithms applied to the bottom up evaluation.

2.3.1 Very Naive Algorithm

Very Naive Algorithm is the first algorithm of the set of bottom up evaluation algorithms[20]. First, we list some points of how it works, then we write the evaluation algorithm and explain it, after that we will write an example to demonstrate its behavior by explaining in details every single step. Very Naive algorithm works as follows:

- It starts from scratch i.e it sets the values of all IDB predicates to $\emptyset$.
- It computes all the possible facts.
- It may infer irrelevant facts according to a certain query.
- The query is considered only at the end of the algorithm. For example, if we have a goal: $\leftarrow p(a,y)$, the answer of this goal is returned as soon as the algorithm terminates (i.e. after computing all the possible facts).

Require: Program P(

- List of EDB predicates $\{e_1,\cdots,e_m\}$
- List of IDB predicates $\{r_1,\cdots,r_k\}$
- List of relations $\{E_1,\cdots,E_m\}$ to serve as values of EDB predicates. For example, if we have $q(a,b)$. $\{q\}$ is e_1 and $\{(a,b)\}$ is E_1.
- List of relations $\{R_1,\cdots,R_k\}$ to serve as values of IDB predicates. For example, if we have $p(a,b)$. $\{p\}$ is r_1 and $\{(a,b)\}$ is R_1.
- Goal: $\leftarrow p(x,y)$

)
Ensure: returns the answer of the goal in the input
begin
  for $i=1$ *to* k **do**
    $R_i=\emptyset$
  end
  repeat
    for $i=1$ *to* k **do**
      $Q_i=R_i$ (**save the old values**)
    end
    for $i=1$ *to* k **do**
      $R_i=EVAL(r_i,E_1,\cdots,E_m,Q_i,\cdots,Q_k)$
    end
  until $R_i=Q_i\ \forall i, 1\leq i\leq k$
  return the answer of the goal in the input
end

Algorithm 1: Very Naive Algorithm

As we see in the first for loop, it sets all the IDB prdicates to $\emptyset$. Then, it repeats the following loops: first, it saves the old values of the IDB predicates, where in the second one it applies the EVAL function to each of the IDB's. It does so until all the IDB's do not change, then it returns the answer of the input goal.
Note: in the following example, we will explain more about the EVAL function.

Example 2.4. *Very Naive Algorithm Let P be a Datalog program with the following input:*
$q(a,b)\leftarrow;$
$q(b,c)\leftarrow;$
$q(c,d)\leftarrow;$
$p(x,y)\leftarrow q(x,y);$
$p(x,y)\leftarrow q(x,z),p(z,y);$
Let the goal be $\leftarrow p(a,y)$.

In this example, we have 3 facts $q(a,b)$, $q(b,c)$ *and* $q(c,d)$. *Also, q is the only EDB predicate since it appears only in the body of the rules, whereas p is the only IDB predicate. Moreover, we have two rules as shown earlier. These rules will be transformed into the following rule:*
$P(X,Y)=Q(X,Y)\cup(\pi_{X,Y}(Q(X,Z)\bowtie P(Z,Y)))$

This what the EVAL function does, it is a combination of select, project and join (database technique).

Now let's execute this program.
Initially, set all the IDB's to $\emptyset$, *here we only have p.*
So $P=\emptyset$, *and* $Q=\{(a,b),(b,c),(c,d)\}$

Round	*P*
1	$P=\{(a,b),(b,c),(c,d)\}$ ***Explanation****: Clearly, only the first rule works in the first round because P is* $\emptyset$, *so it takes all the tuples from Q.*
2	$P=\{(a,b),(b,c),(c,d),\underline{(a,c)},\underline{(b,d)}\}$ ***Explanation****: Here, the second rule also works. Applying the join between p and q, we get the newly added tuples. The algorithm continues because the values of P are changed.*
3	$P=\{(a,b),(b,c),(c,d),(a,c),(b,d),\underline{(a,d)}\}$ ***Explanation****: New facts are generated, so the algorithm continues.*
4	$P=\{(a,b),(b,c),(c,d),(a,c),(b,d),(a,d)\}$ ***Explanation****: No new facts are generated, the algorithm stops. Now it considers the query,* $\pi_y(\sigma_{x=a}(P(x,y))$. *The answer is* $y=\{b,c,d\}$.

We presented an example of the Very Naive Algorithm to show how it works. As noticed, in each round, it generated new facts. But, it recomputed previously generated facts. For example, in round 3, the new generated fact is (a,d), but, $(a,b),(b,c),(c,d),(a,c)$ and (b,d) are recomputed again. This algorithm is thus inefficient. In the next part, we will show the Semi Naive Algorithm.

2.3.2 Semi Naive Algorithm

There is a slight difference between Very Naive and Semi Naive Algorithms, in which the latter does less work. For instance, in the third round of the previous example, the tuple (a,d) was newly generated, but all the other tuples were recomputed again, which is really a wasted time. The Semi Naive Algorithm works in a different manner. Since the values of EDB predicates never change during the whole execution except for the IDB values, we can deduce the following:

at round i for example, if we get a new fact (let's say (a,b)), this means that we used a newly generated fact at round i-1 to infer (a,b) [20].

Require: Program P(

- List of EDB predicates $\{e_1,\cdots,e_m\}$
- List of IDB predicates $\{r_1,\cdots,r_k\}$
- List of relations $\{E_1,\cdots,E_m\}$ to serve as values of EDB predicates.
- List of incremental predicates $\{\Delta R_1,\cdots,\Delta R_k\}$ where, for each IDB predicate R_i, there is an incremental predicate ΔR_i that holds only the tuples added in the previous round
- Goal: $\leftarrow p(x,y)$

) **Ensure**: returns the answer of the goal in the input

```
begin
    for i = 1 to k do
        ΔR_i=EVAL(r_i, E_1, ..., E_m, ∅, ..., ∅)
        R_i=ΔR_i
    end
    repeat
        for i = 1 to k do
            ΔQ_i=ΔR_i
        end
        for i = 1 to k do
            ΔR_i=
            EVAL(r_i, E_1, ..., E_m, R_1, ..., R_k, ΔQ_1, ..., ΔQ_k)
            ΔR_i=ΔR_i-R_i
        end
        for i = 1 to k do
            R_i=R_i ∪ ΔR_i
        end
    until ΔR_i=∅ ∀i, 1 ≤ i ≤ k
    return the answer of the query
end
```

Algorithm 2: Semi Naive Algorithm

Indeed, this will avoid recomputing already generated facts in addition to returning the same answer as the Naive one.

Now let's list how this algorithm works:

- Starts from scratch i.e it sets the values all IDB predicates to $\emptyset$.
- Computes all the possible facts.
- It may infer irrelevant facts according to a certain query. For example, if the goal: $\leftarrow p(a,y)$, it may infer a fact of predicate q ($q(a,b)$ for instance).
- The query is considered only at the end of the algorithm. For example, if we have a goal: $\leftarrow p(a,y)$, the answer of this goal is returned as soon as the algorithm terminates (i.e. after computing all the possible facts).
- It introduces a new type of predicates called incremental predicates. The incremental predicates are those for handling the newly generated facts [20].
- It stops when all the incremental predicates become $\emptyset$.

Now let's write the algorithm of the Semi Naive evaluation. In the first for loop, we apply the EVAL function to the incremental predicates having EDB predicates only in the body. Then, we set all the IDB's equal to the incremental predicates. In the second loop, we save the old values of incremental predicates. While in the third loop we apply the EVAL function to the incremental predicates but including the newly generated facts from loop number 1. After that, we remove the newly generated facts. In the last loop, we add the new facts the original IDB predicate. We repeat loop 2 to loop 4 until all the incremental predicates become $\emptyset$. Then, the answer of the query is returned.

Example 2.5. *Semi Naive Algorithm*

Now let's consider the same example as before to show the difference between the two algorithms. The difference occurs in the rules:

1. $\Delta p(x,y) \leftarrow q(x,y)$
2. $\Delta p(x,y) \leftarrow q(x,z), \Delta p(z,y)$

Now let's run this program. In the first loop, the incremental predicates take values from the rules that have only EDB's in the body (here rule 1).
Thus, $\Delta P = \{(a,b),(b,c),(c,d)\}$ and P has the same values of ΔP.

Round	
1	$\Delta P = \{(a,b),(b,c),(c,d),(a,c),(b,d)\}$ ΔP=ΔP-P = $\{(a,c),(b,d)\}$ P=$P \cup \Delta P$ = $\{(a,b),(b,c),(c,d),(a,c),(b,d)\}$ ***Explanation****: Here we removed the newly added tuples, then we will add those tuples to the original predicate which is P. $\Delta P \neq \emptyset$, the algorithm continues.*
2	$\Delta P = \{(a,c),(b,d),(a,d)\}$ ΔP=ΔP-P = $\{(a,d)\}$ P=$P \cup \Delta P$ = $\{(a,b),(b,c),(c,d),(a,c),(b,d),(a,d)\}$ ***Explanation****: Notice the difference here, ΔP contains only two tuples (not 5 as in the Very Naive Algorithm, here is the optimization).*
3	$\Delta P = \{(a,d)\}$ ΔP=ΔP-P = $\emptyset$ P=$P \cup \Delta P$ = $\{(a,b),(b,c),(c,d),(a,c),(b,d),(a,d)\}$ ***Explanation****: Consequently, the algorithm stops since all the incremental predicates became $\emptyset$. The answer of the query is: $y = \{b,c,d\}$. Same answer as before but more efficient since it didn't recompute already generated facts in each round.*

3. Information Flow Definitions Revisited using Bottom Up Evaluation Techniques

In this section, we will revisit information flow defintions in Datalog programs but using a bottom up evaluation technique using the two algorithms (Very Naive and Semi Naive). We recall the first two definitions of detecting whether there is a flow in a Datalog program based on bottom up evaluation approach. Then, we show if there is any implication between the definitions, for instance, if the existence of the first definition entails the existence of the second one, and vice versa.

3.1 Definitions of Information Flow in Datalog Programs based on Bottom Up Evaluation

The first definition is based on Success/Failure of the goals. Let P be a Datalog program and $G(x,y)$ be a two variables goal. There is a flow of information from x to y in P ($x \xrightarrow{SF}{}^{P}_{G} y$ based on SF in goal G and Program P) iff there exists a, $b \in U_{L(P)}$ such that $P \cup \{G(a,y)\}$ succeeds and $P \cup \{G(b,y)\}$ fails. This means that when the user only sees the outputs of computations in terms of successes and failures, $\exists$ two different a, $b \in U_{L(P)}$ such that the user can distinguish between the outputs of the goals without seeing what concerns a and b. Now let's show an example to make it clearer.

Example 3.1. *Information Flow based on Success/Failure using Bottom Up Evaluation Let P be a program:*
$p(a,b) \leftarrow$;
and let $G(x,y)$ be the following goal: $\leftarrow p(x,y)$.
The set of generated facts using bottom up evaluation consists of $R = \{p(a,b)\}$. Clearly, $P \cup \{G(a,y)\}$ succeeds and $P \cup \{G(b,y)\}$ fails (since we can find a fact $p(a,.) \in R$ but no fact of the form $p(b,.)$ exists in R), then $x \xrightarrow{SF}{}^{P}_{G} y$ based on SF, goal G and Program P. We retreive the same intuition as in the case where the evaluation was based on a top down approach: if we hide a and b from the goals and since the first goal succeeds and the second one fails, we can distinguish by looking at the facts that the first constant is "a" whereas the second one is "b", consequently the flow occurs.

The second definition is based on the substitution answers of the goals. Let P be a Datalog program and $G(x,y)$ be a goal. We can say that there is an information flow from x to y in $G(x,y)$ with respect to substitution answers in P iff $\exists$ a, b $\in U_{L(P)}$ such that $\theta(P \cup \{G(a,y)\}) \neq \theta(P \cup \{G(b,y)\})$. In this definition, the user only sees the outputs of computations in terms of substitution answers. Consequently, there is a flow of information from x to y if this user can distinguish the output of $P \cup \{G(a,y)\}$ and the output of $P \cup \{G(b,y)\}$.

Example 3.2. *Information Flow based on Substitution Answers using Bottom Up Evaluation Let's consider the previous example of program P:*
$p(a,b) \leftarrow$;
$p(c,d) \leftarrow$;
and let $G(x,y)$ be the following goal: $\leftarrow p(x,y)$
The set of generated facts using bottom up evaluation consists of $R = \{p(a,b), p(c,d)\}$. Clearly, $\theta(P \cup \{G(a,y)\}) = \{y \rightarrow b\}$ and $\theta(P \cup \{G(b,y)\}) = \emptyset$, then there is a flow from x to y($x \xrightarrow{SA}{}^{P}_{G}$ y based on substitution answers in $G(x,y)$ and P).

3.2 Implications between Definitions based on Bottom Up Evaluation

We retrieve the same results for the implication between the definitions if it were based on a top down evaluation approach.

- The existence of a flow with respect to substitution answers does not entail the existence of a flow with respect to success and failures. To show this, let's consider this example.

  Example 3.3. *Existence of a flow w.r.t. SA doesn't entail the existence of a flow w.r.t SF Let P be the following program:*
  $q(a) \leftarrow$;
  $q(b) \leftarrow$;
  $p(a,b) \leftarrow$;
  $p(x,c) \leftarrow q(x)$;
  and let $G(x,y)$ be the goal: $\leftarrow p(x,y)$.
  The set of generated facts using bottom up evaluation is $R = \{q(a), q(b), p(a,b), p(a,c), p(b,c)\}$. Clearly, we find that $\theta(P \cup \{G(a,y)\}) = \{y \rightarrow b, y \rightarrow c\}$ and $\theta(P \cup \{G(b,y)\}) = \{y \rightarrow c\}$, then $x \xrightarrow{SA}{}^{P}_{G} y$ based on substitution answers in goal $G(x,y)$ and program P. Since $P \cup \{G(a,y)\}$ and $P \cup \{G(b,y)\}$ both succeed, then not $x \xrightarrow{SF}{}^{P}_{G} y$ based on success/failure in goal $G(x,y)$ and program P.

- Let P be a logic program and $G(x,y)$ be a two variables goal. If $x \xrightarrow{SF}{}^{P}_{G} y$ based on success/failure in goal $G(x,y)$ and program P, then $x \xrightarrow{SA}{}^{P}_{G} y$ based on substitution answers in goal $G(x,y)$ and program P. The proof is as follows:
  Suppose that $x \xrightarrow{SF}{}^{P}_{G} y$ based on success/failure in goal $G(x,y)$ and program P, then $\exists a$, $b \in U_{L(P)}$ such that $P \cup \{G(a,y)\}$ succeeds and $P \cup \{G(b,y)\}$ fails, i.e the set of generated facts contains $p(a,.)$, but does not contain any $p(b,.)$. Therefore, $\theta(P \cup \{G(a,y)\}) \neq \emptyset$ and $\theta(P \cup \{G(b,y)\}) = \emptyset$. Consequently, $x \xrightarrow{SA}{}^{P}_{G} y$ based on substitution answers in goal $G(x,y)$ and program P.

3.3 Flow Definitions based on Bottom Up Evaluation VS Flow Definitions based on Top Down Evaluations

In this part, we study the implications between the flow definitions of the two evaluations (Bottom Up and Top Down).

- The existence of a flow with respect to success/failure using bottom up evaluation doesn't entail the existence of a flow with respect to success/failure using top down evaluation. To show this, let's consider this example.

  Example 3.4. *Existence of a flow w.r.t. success/failure using bottom up evaluation doesn't entail the existence of a flow w.r.t. success/failure using top down evaluation Let P be the following program:*

  $p(a,b) \leftarrow$;

  $p(x,y) \leftarrow p(x,y)$;
  and let $G(x,y)$ *be the goal:* $\leftarrow p(x,y)$.
  The set of generated facts using bottom up evaluation R $= \{p(a,b)\}$.
  Clearly, we find that $P \cup \{G(a,y)\}$ *succeeds and* $P \cup \{G(b,y)\}$ *fails, then* $x \xrightarrow{SF}{}^{P}_{G} y$. *But, using top down evaluation, this program does not terminate. Consequently, flow can not be detected in top down.*

- Let P be a logic program and $G(x,y)$ be a two variables goal. If $x \xrightarrow{SF}{}^{P}_{G} y$ using top down evaluation, then $x \xrightarrow{SF}{}^{P}_{G} y$ using bottom up evaluation. The proof is as follows:
  Suppose that $x \xrightarrow{SF}{}^{P}_{G} y$ using top down evaluation, then there $\exists a,b \in U_{L(P)}$ such that $P \cup \{G(a,y)\}$ succeeds and $P \cup \{G(b,y)\}$ fails.
  Suppose that there is no flow using bottom up evaluation. This means that $P \cup \{G(b,y)\}$ succeeds. Therefore, there $\exists$ a fact f for example, led to the success of $P \cup \{G(b,y)\}$. Consequently, the fact f should lead to the success of $P \cup \{G(b,y)\}$ in top down. Contradiction, therefore, there is a flow using bottom up evaluation.

- The existence of a flow with respect to substitution answers using bottom up evaluation doesn't entail the existence of a flow with respect to substitution answers using top down evaluation. To show this, let's consider this example.

  Example 3.5. *Existence of a flow w.r.t. substitution answers using bottom up evaluation doesn't entail the existence of a flow w.r.t. substitution answers using top down evaluation Let P be the following program:*

  $p(a,b) \leftarrow$;

  $p(x,y) \leftarrow p(x,y)$;
  and let $G(x,y)$ *be the goal:* $\leftarrow p(x,y)$.
  The set of generated facts using bottom up evaluation R $= \{p(a,b)\}$.
  Clearly, we find that $\theta(P \cup \{G(a,y)\}) = \{y \rightarrow b\}$ *and* $\theta(P \cup \{G(b,y)\}) = \emptyset$, *then* $x \xrightarrow{SA}{}^{P}_{G} y$. *But, using top down evaluation, this program does not terminate. Consequently, flow can not be detected in top down.*

- Let P be a logic program and $G(x,y)$ be a two variables goal. If $x \xrightarrow{SA}{}^{P}_{G} y$ using top down evaluation, then $x \xrightarrow{SA}{}^{P}_{G} y$ using bottom up evaluation. The proof is as follows:
  Suppose that $x \xrightarrow{SA}{}^{P}_{G} y$ using top down evaluation, then there $\exists a,b \in U_{L(P)}$ such that $\theta(P \cup \{G(a,y)\}) \neq \theta(P \cup \{G(b,y)\})$.
  Suppose that there is no flow using bottom up evaluation. This means that $\theta(P \cup \{G(a,y)\}) = \theta(P \cup \{G(b,y)\})$. Therefore, there $\exists$ a generated fact of the form $p(a,\theta_a^1)$, $\cdots, p(a,\theta_a^n), p(b,\theta_b^1), \cdots, p(b,\theta_b^m)$ such that $p(a,\theta_a^i) = p(b,\theta_a^j)$ $\forall\ 1 \leq i \leq n$ and $\forall\ 1 \leq j \leq m$. Contradiction since $\theta(P \cup \{G(a,y)\}) \neq \theta(P \cup \{G(b,y)\})$ in top down.

4. Decidability / Complexity

We study the computational complexity of the following decision problems:

$$\pi_{SF} \begin{cases} \text{Input: A logic program } P\text{, a two variables goal } G(x,y) \\ \text{Output: Determine whether } x \xrightarrow{SF}{}^{P}_{G} y \end{cases}$$

$$\pi_{SA} \begin{cases} \text{Input: A logic program } P\text{, a two variables goal } G(x,y) \\ \text{Output: Determine whether } x \xrightarrow{SA}{}^{P}_{G} y \end{cases}$$

Thus, we present algorithms for detecting whether there is a flow of information in Datalog program based on the bottom up technique (Naive and Semi Naive respectively). Proofs of this algorithm will be given such as Termination, Soundness, Completeness; hardness and the complexity of the decision problem will be studied, in addition to the running example to demonstrate the detection of the flow.

4.1 Decidability and Computational Complexity of Flow based on Bottom Up Evaluation Naive Algorithm

We now study the decidability and computational complexity of (π_{SF}) based on Bottom Up Evaluation Naive Algorithm. The main point of this algorithm is to skim over the set of herbrand universe and to check, at the end of the algorithm if there are unremoved constants. If it is the case, then there is a flow. For this, consider algorithm 3.

In order to demonstrate the decidability of (π_{SF}), we need to prove the following lemmas:

Lemma 4.1 (Termination)**.** `Algorithm` *3 terminates.*

Lemma 4.2 (Completeness)**.** *If* $x \xrightarrow{SF}{}^{P}_{G} y$, *then* `Algorithm` *3 returns* `true`.

Lemma 4.3 (Soundness)**.** *If* `Algorithm` *3 returns* `true`, *then* $x \xrightarrow{SF}{}^{P}_{G} y$.

Proof of Lemma 4.1. According to Jeffrey D.Ullman [20], suppose we have an upper limit of arity, let's say a, and number of symbols, let it be b. So, we have b^a different tuples. Also, let m be the number of IDB predicates, so, this algorithm needs at most mb^a rounds to reach a fixed point and terminates. □

Require: Program P(

- List of EDB predicates $\{e_1, \cdots, e_m\}$
- List of IDB predicates $\{r_1, \cdots, r_k\}$
- List of relations $\{E_1, \cdots, E_m\}$ to serve as values of EDB predicates.
- Distinct Herbrand Universe $U_{L(P)}$ $\{a_1, \cdots, a_n\}$
- Goal: $\leftarrow g(x,y)$

)
Ensure: $x \xrightarrow{SF}{}^{P}_{G} y$.
begin
  succeed = false
  for $i = 1$ *to* k **do**
    $R_i = \emptyset$
  end
  repeat
    for $i = 1$ *to* k **do**
      $Q_i = R_i$
    end
    for $i = 1$ *to* k **do**
      $R_i = EVAL(r_i, E_1, \cdots, E_m, Q_i, \cdots, Q_k)$
      if *predicate(R_i,'g') and arity(R_i,2)* **then**
        succeed = true
        remove(constant(R_i,U_L))
      end
    end
  until $R_i = Q_i \; \forall i, 1 \leq i \leq k$
  if *succeed == true and U_L contains at least one element* **then**
    return true
  end
  return false
end

Algorithm 3: Very Naive - Flow Detection based on Success/Failure

Proof of Lemma 4.2. If $x \xrightarrow{SF}{}^{P}_{G} y$, thus there $\exists$ two constants a, $b \in U_L$ with $a \neq b$ such that $G(a,y)$ succeeds whereas $G(b,y)$ fails. According to this algorithm, constant a will be removed from U_L and *succeed* will be set to `true`, while b will remain in U_L. Consequently, the `if` condition will be `true` and the algorithm will return `true`.

On the other hand, if there is no flow $x \xrightarrow{SF}{}^{P}_{G} y$, we have two cases:
(1) $\forall$ constant $a_i \in U_L (1 \leq i \leq n)$, $G(a_i,y)$ fails, OR
(2) $\forall$ constant $a_i \in U_L (1 \leq i \leq n)$, $G(a_i,y)$ succeed and a_i will be removed from U_L.
In both cases, this will lead to the falseness of the `if` condition and the algorithm will return `false`. □

Proof of Lemma 4.3. If the algorithm returns `true`, thus the `if` condition is `true`, consequently *succeed* = `true` (1) and U_L contains at least one element (2).
From (1), there $\exists$ constant $a \in U_L$ such that $G(a,y)$ succeeded and a is removed from U_L.
From (2), there $\exists$ another constant $b \neq a \in U_L$ such that $G(b,y)$ failed. ($b \neq a$ because U_L is distinct).
Thus, $x \xrightarrow{SF}{}^{P}_{G} y$.

On the other hand, if the algorithm returns `false`, we have two cases:
(1) *succeed* = `false` which leads to the fact that $\forall$ constant $a_i \in U_L (1 \leq i \leq n)$ $G(a_i,y)$ failed.
(2) U_L contains zero elements, which means that $\forall$ constant $a_i \in U_L (1 \leq i \leq n)$, $G(a_i,y)$ succeeded and a_i removed from U_L.
In both cases $\nexists$ two constants $a, b \in U_L$ with $a \neq b$ such that $G(a,y)$ succeeds and $G(b,y)$ fails. Thus, there is no flow. □

As a consequence of lemmas 4.1 – 4.3, we have:

Theorem 4.4. `Algorithm` *3 is a sound and complete decision procedure for* (π_{SF}).

It follows that (π_{SF}) is decidable. Moreover,

Theorem 4.5. (π_{SF}) *is EXPTIME-Complete*

Proof. (*Membership*) Now, we will explain briefly about the complexity of this algorithm by estimating the time needed for the loops and certain instructions.

The following for loop needs about $\theta(k)$ as a time complexity.
for $i = 1$ to k **do** $R_i = \emptyset$
EVAL function is a database technique $(\sigma, \pi, \bowtie)$.
Suppose we have a table of length k.
$\sigma : \theta(k)$
$\pi : \theta(K)$
Suppose we have two tables of length k, l respectively.
$\bowtie$: $\theta(k \times l)$
Also, we have "repeat - until" and an inner for loop, so, this algorithm can be executed in EXPTIME in the worst case.

(*Hardness*) In order to prove EXPTIME-hardness, we consider the following decision problem known to be EXPTIME-hard [21]:

$$\pi \begin{cases} \text{Input: A Datalog program } P, \text{ a ground atom } A \\ \text{Output: } P \cup A \ (A \text{ is a logical consequence of } P) \end{cases}$$

Let (P,A) be an instance of π and let $(P', g(x,y))$ be the instance of π_{SF} defined by $P' = P \cup \{g(a,y) \leftarrow A\}$, where g is a new predicate symbol. Thus $P \cup A$ iff $x \xrightarrow{SF}{}^{P'}_{g} y$ based on success/failure in goal g and program P'.
$(\rightarrow)$ Suppose that A is a logical consequence of P, thus, $P' \cup \{\leftarrow g(a,y)\}$ succeeds and $P' \cup \{\leftarrow g(b,y)\}$ fails. Consequently, $x \xrightarrow{SF}{}^{P'}_{g} y$ based on success/failure in goal g and program P'.
$(\leftarrow)$ Suppose that $x \xrightarrow{SF}{}^{P'}_{g} y$ based on success/failure in goal g and program P'. Then $\exists a, b \in U_{L(P)}$ such that $P' \cup \{\leftarrow g(a',y)\}$ succeeds and $P' \cup \{\leftarrow g(b',y)\}$ fails. Hence, it follows that $a' = a$ and $b' \neq a$. Thus, $P \cup A$. □

Example 4.6. *Flow Detection based on SF using Naive Algorithm in Bottom Up Evaluation This example will show how algorithm 3 detects whether there is a flow or not based on success/failure definition.*
Let P be the following program:
$g(a,b) \leftarrow$;
And let the goal be $g(x,y)$.
Clearly, the herbrand universe $U_{L(P)} = \{a,b\}$, *and we have only one predicate which is g.*
When we reach the if condition, the predicate is 'g' and the arity is 2 also, then the boolean flag (succeed) will be set to `true`. *The function "constant" returns the first argument of the predicate 'g' so that the function "remove" removes it from* $U_{L(P)}$. *Now* $U_{L(P)} = \{b\}$.
Now, the algorithm reaches the last `if` *condition since no more new facts can be generated. succeed is set to* `true` *and* U_L *contains at least one element, consequently, it returns* `true` $\rightarrow$ ***Flow Detected***.

In contrast, if the program has one more fact $g(b,a)$ *in addition to the old one*($g(a,b)$), *this will lead to the emptiness of* U_L *in some round. Consequently, this will lead to the falseness of the* `if` *condition. The algorithm will returns* `false` $\rightarrow$ ***NO Flow***.

We now study the decidability and computational complexity of (π_{SA}) based on Bottom Up Evaluation Naive Algorithm. For this, consider algorithm 4.

In algorithm 4, A is a set of (key/value) pairs representing the substitution answers of each of the constants in U_L. For example, if we have A[a] = $\{a,b,c\}$ this means that a is the key and $\{a,b,c\}$ is the value representing the substitution answer of a.

In order to demonstrate the decidability of (π_{SA}), we need to prove the following lemmas:

Lemma 4.7 (Termination)**.** `Algorithm` *4 terminates.*

Lemma 4.8 (Completeness)**.** *If* $x \xrightarrow{SA}{}^{P}_{G} y$, *then* `Algorithm` *4 returns* `true`.

Lemma 4.9 (Soundness)**.** *If* `Algorithm` *4 returns* `true`, *then* $x \xrightarrow{SA}{}^{P}_{G} y$.

Proof of Lemma 4.7. Same proof as 4.1. □

Proof of Lemma 4.8. If $x \xrightarrow{SA}{}^{P}_{G} y$, thus there $\exists$ constants a,$b \in U_L$ with $a \neq b$ such that $\theta(g(a,y)) \neq \theta(g(b,y))$. So, according to this algorithm, we have two cases:
(1) one of the substitution answers is $\emptyset$, this means that this constant is not removed from U_L, while the other one has a non empty set and removed from U_L and added to A (A now is not empty). Thus, A is not empty and U_L contains at least one element not in keys of A. So, the `if` condition will be false, consequently this algorithm will return `true`.
(2) All the constants are removed from U_L. This means that U_L contains zero elements not in keys of A. Also, A is not empty and contains two different substitution answers. Consequently, the `if` condition is false, thus, the algorithm will return `true`.

On the other hand, if there is no flow $x \xrightarrow{SA}{}^{P}_{G} y$, we have two cases:
(1) $\forall$ constant $a_i \in U_L (1 \leq i \leq n), \theta(g(a_i,y)) = \emptyset$, OR
(2) $\forall$ constant $a_i, a_j \in U_L (1 \leq i \leq n), \theta(g(a_i,y)) = \theta(g(a_j,y))$.
In both cases, this will lead to the truth of the `if` condition. Consequently, the algorithm will return `false`.

□

Require: Program P (
- List of EDB predicates $\{e_1, \cdots, e_m\}$
- List of IDB predicates $\{r_1, \cdots, r_k\}$
- List of relations $\{E_1, \cdots, E_m\}$ to serve as values of EDB predicates.
- Distinct Herbrand Universe $U_{L(P)}$ $\{a_1, \cdots, a_n\}$
- Goal: $\leftarrow g(x,y)$

)
Ensure: $x \xrightarrow{SA}{}^{P}_{G} y$.
begin
  for $i = 1$ *to* k **do**
    $R_i = \emptyset$
  end
  repeat
    for $i = 1$ *to* k **do**
      $Q_i = R_i$
    end
    for $i = 1$ *to* k **do**
      $R_i = EVAL(r_i, E_1, \cdots, E_m, Q_i, \cdots, Q_k)$ **if** *predicate(R_i,'g') and arity(R_i,2)* **then**
        A[constant(R_{i1})] = constant(R_{i2})
        remove(constant(R_{i1},U_L)
      end
    end
  until $R_i = Q_i \; \forall i, 1 \leq i \leq k$
  if *A is empty or (U_L contains zero elements not in keys of A and A doesn't contain two different substitution answers)* **then**
    return false
  end
  return true
end

Algorithm 4: Very Naive - Flow Detection based on Substitution Answers

Proof of Lemma 4.9. If the algorithm returns `true`, thus, the `if` condition is false. This means that A is not empty and (U_L contains at least one element not in keys of A or A contains two different substitution answers).
(1) (A is not empty and U_L contains at least one element not in keys of A), OR
(2) (A is not empty and A contains two different substitution answers).
In (1), A is not empty, then there $\exists$ a constant $a \in U_L$ such that $\theta(g(a,y)) \neq \emptyset$. And U_L contains at least one element not in keys of A means, there $\exists$ another constant $b \neq a \in U_L$ such that $\theta(g(b,y)) = \emptyset$. Consequently, $x \xrightarrow{SA}{}^{P}_{G} y$.
In (2) as in (1) for A is not empty. And A contains two different substitution answers means that there $\exists$ two constants $a,b \in U_L$ with $a \neq b$ such that $\theta(g(a,y)) \neq \theta(g(b,y))$. Thus, $x \xrightarrow{SA}{}^{P}_{G} y$.

On the other hand, if the algorithm returns `false`, we have two cases:
(1) A is empty. Therefore, $\forall$ constants $a_i, a_j \in U_L$ with $a_i \neq a_j$, we have $\theta(g(a_i,y)) = \theta(g(a_j,y)) = \emptyset$. Consequently, there is no flow.
(2) U_L contains zero elements not in keys of A and A doesn't contain two different substitution answers. This means that $\forall$ constant $a_i \in U_L (1 \leq i \leq n), \theta(g(a_i,y)) \neq \emptyset$. But A doesn't contain two different substitution answers, so, $\forall$ constant $a_i, a_j \in U_L$ with $a_i \neq a_j, \theta(g(a_i,y)) = \theta(a_j,y))$. Thus, there is now flow. □

As a consequence of lemmas 4.7 – 4.9, we have:

Theorem 4.10. `Algorithm` *4 is a sound and complete decision procedure for* (π_{SA}).

It follows that (π_{SA}) is decidable. Moreover,

Theorem 4.11. (π_{SA}) *is EXPTIME-Complete*

Proof. (*Membership*) Concerning the time complexity of this algorithm, our study will be based on the set of pairs A.
First, we have to sort the set of substitution answers which is called A using merge-sort algorithm:
worst case:
Suppose the length of keys of A is n. And each entry of A (i.e values of substitution answers) is n also.
To sort one row it is about $\theta(nlogn)$.
To sort all rows, time complexity will be about $\theta(n^2logn)$.
To search an element in a sorted set, complexity will be $\theta(log_2 n)$.
To compare all rows with each other $\theta(n^2logn)$.
Consequently, this algorithm is executed in EXPTIME.

(*Hardness:*) In order to prove EXPTIME-hardness, we consider the following decision problem known to be EXPTIME-hard [16, 21]:

$$\pi \begin{cases} \text{Input: A Datalog program } P, \text{ a ground atom } A \\ \text{Output: } P \cup A \ (A \text{ is a logical consequence of } P) \end{cases}$$

Let (P,A) an instance of π and let $(P', g(x,y))$ be the instance of π_{SA} defined by $P' = P \cup \{g(a,y) \leftarrow A\}$, where g is a new predicate symbol. Thus $P \cup A$ iff $x \xrightarrow{SA}{}^{P'}_{g} y$.
($\rightarrow$) Suppose that A is a logical consequence of P, thus, $\theta(P' \cup \{\leftarrow g(a,y)\}) \neq \emptyset$ and $\theta(P' \cup \{\leftarrow g(b,y)\}) = \emptyset$. Consequently, $x \xrightarrow{SA}{}^{P'}_{g} y$.

($\leftarrow$) Suppose that $x \xrightarrow{SA}{}^{P'}_{g} y$. Then $\exists a,b \in U_{L(P)}$ such that $\theta(P' \cup \{\leftarrow g(a',y)\}) \neq \emptyset$ and $\theta(P' \cup \{\leftarrow g(b',y)\}) = \emptyset$. Hence, it follows that $a' = a$ and $b' \neq a$. Thus, $P \cup A$. □

Example 4.12. *Flow Detection based on SA using naive bottom up evaluation Let's consider the previous example.*
Let P be the following program:
$g(a,b) \leftarrow$
And let the goal be $g(x,y)$.
Clearly, the herbrand universe $U_{L(P)} = \{a,b\}$*, and we have only one predicate which is g.*

When we reach the if condition, the predicate is 'g' and the arity is 2 also, then the first constant of g will be added to the keys of A, whereas the second constant of g will be added to the substitution answers of the first constant. And the function "constant" returns the first argument of the predicate 'g' so that the function "remove" removes it from $U_{L(P)}$*. Now* $U_{L(P)} = \{b\}$ *and A[a] =* $\{b\}(A \neq \emptyset)$.
Now, the algorithm reaches the last if condition since no more new facts can be generated. We have A is not empty and U_L *contains at least one element not in keys of A, consequently, it returns 1* $\rightarrow$ ***Flow Detected***.

In contrast, if we add another fact to the program, let it be $g(b,b)$*. At some round, A will get a second entry which is for constant b which in turn will be removed from* U_L*. Now,* $U_L = \emptyset$ *and* $A[b]$=$\{b\}$*. As soon as the algorithm stops, it reaches the last if condition. Here, we have* $A \neq \emptyset$ *but all entries of A are of the same substitution answers in addition to* $U_L = \emptyset$*. Thus, the algorithm returns zero. Consequently,* ***NO Flow***.

4.2 Decidability and Computational Complexity of Flow based on Bottom Up Evaluation Semi Naive Algorithm

We now study the decidability and computational complexity of (π_{SF}) based on Bottom Up Evaluation Semi Naive Algorithm. For this, consider algorithm 5.

In order to demonstrate the decidability of (π_{SF}), we need to prove the following lemmas:

Lemma 4.13 (Termination)**.** `Algorithm` *5 terminates.*

Lemma 4.14 (Completeness)**.** *If* $x \xrightarrow{SF}{}^{P}_{G} y$*, then* `Algorithm` *5 returns* `true`.

Lemma 4.15 (Soundness)**.** *If* `Algorithm` *5 returns* `true`*, then* $x \xrightarrow{SF}{}^{P}_{G} y$.

Require: Program P (

- List of EDB predicates $\{e_1, \cdots, e_m\}$
- List of IDB predicates $\{r_1, \cdots, r_k\}$
- List of relations $\{E_1, \cdots, E_m\}$ to serve as values of EDB predicates.
- List of incremental predicates $\{R_1, \cdots, R_k\}$ where, for each IDB predicate P_i, there is an incremental predicate ΔP_i that holds only the tuples added in the previous round
- Distinct Herbrand Universe $U_{L(P)}\{a_1, \cdots, a_n\}$
- Goal: $\leftarrow g(x,y)$

)

Ensure: $x \xrightarrow{SF}{}^{P}_{G} y$.

begin
  succeed = false
  for $i = 1$ *to* k **do**
    ΔR_i=$EVAL(r_i, E_1, \cdots, E_m, \emptyset, \cdots, \emptyset)$
    if *predicate*(ΔR_i,'g') *and arity*(ΔR_i,2) **then**
      succeed = true
      remove(constant(ΔR_i),U_L)
    end
    R_i=ΔR_i
  end
  repeat
    for $i = 1$ *to* k **do**
      ΔQ_i=ΔR_i
    end
    for $i = 1$ *to* k **do**
      ΔR_i=
      $EVAL(r_i, E_1, \cdots, E_m, R_1, \cdots, R_k, \Delta Q_1, \cdots, \Delta Q_k)$
      ΔR_i=ΔR_i-R_i
      if *predicate*(ΔR_i,'g') *and arity*(ΔR_i,2) **then**
        succeed = true
        remove(constant(ΔR_i),U_L)
      end
    end
    for $i = 1$ *to* k **do**
      R_i=$R_i \cup \Delta R_i$
    end
  until ΔR_i=$\emptyset$ $\forall i$, $1 \leq i \leq k$
  if *succeed == true and* U_L *contains at least one element* **then**
    return true
  end
  return false
end

Algorithm 5: Semi Naive - Flow Detection based on Success/Failure

Proof of Lemma 4.13. Same proof as 4.1. □

Proof of Lemma 4.14. Same proof as 4.2. □

Proof of Lemma 4.15. Same proof as 4.3. □

As a consequence of lemmas 4.13 – 4.15, we have:

Theorem 4.16. `Algorithm 5` *is a sound and complete decision procedure for* (π_{SF}).

It follows that (π_{SF}) is decidable. Moreover,

Theorem 4.17. (π_{SF}) *is EXPTIME-Complete*

Proof. (*Membership*) Now, we will explain briefly about the complexity of this algorithm by estimating the time needed for the loops and certain instructions. This following for loop needs about $\theta(k)$ as a time complexity.
for $i = 1$ to k **do** $R_i = \emptyset$
EVAL function is a database technique $(\sigma, \pi, \bowtie)$.
Suppose we have a table of length k.
$\sigma : \theta(k)$
$\pi : \theta(K)$
Suppose we have two tables of length k, l respectively.
$\bowtie$: $\theta(k \times l)$
Also, we have "repeat - until" and an inner for loop, so, this algorithm can be executed in EXPTIME in the worst case.

(*Hardness*) In order to prove EXPTIME-hardness, we consider the following decision problem known to be EXPTIME-hard [16, 21]:

$$\pi \begin{cases} \text{Input: A Datalog program } P\text{, a ground atom } A \\ \text{Output: } P \cup A \text{ (} A \text{ is a logical consequence of } P\text{)} \end{cases}$$

Let (P,A) be an instance of π and let $(P', g(x,y))$ be the instance of π_{SF} defined by $P' = P \cup \{g(a,y) \leftarrow A\}$, where g is a new predicate symbol. Thus $P \cup A$ iff $x \xrightarrow{SF}{}^{P'}_{g} y$ based on success/failure in goal g and program P'.
$(\rightarrow)$ Suppose that A is a logical consequence of P, thus, $P' \cup \{\leftarrow g(a,y)\}$ succeeds and $P' \cup \{\leftarrow g(b,y)\}$ fails. Consequently, $x \xrightarrow{SF}{}^{P'}_{g} y$ based on success/failure in goal g and program P'.
$(\leftarrow)$ Suppose that $x \xrightarrow{SF}{}^{P'}_{g} y$ based on success/failure in goal g and program P'. Then $\exists a, b \in U_{L(P)}$ such that $P' \cup \{\leftarrow g(a',y)\}$ succeeds and $P' \cup \{\leftarrow g(b',y)\}$ fails. Hence, it follows that $a' = a$ and $b' \neq a$. Thus, $P \cup A$. □

We now study the decidability and computational complexity of (π_{SA}) based on Bottom Up Evaluation Semi Naive Algorithm. For this, consider algorithm 6:

Where, A is a set of (key/value) pairs representing the substitution answers of each of the constants in U_L. For example, if we have A[a] = $\{a,b,c\}$ this means that a is the key and $\{a,b,c\}$ is the value representing the substitution answer of a.

Require: Program P (

- List of EDB predicates $\{e_1, \cdots, e_m\}$
- List of IDB predicates $\{r_1, \cdots, r_k\}$
- List of relations $\{E_1, \cdots, E_m\}$ to serve as values of EDB predicates.
- List of incremental predicates $\{R_1, \cdots, R_k\}$ where, for each IDB predicate P_i, there is an incremental predicate ΔP_i that holds only the tuples added in the previous round
- Distinct Herbrand Universe $U_{L(P)}\{a_1, \cdots, a_n\}$
- Goal: $\leftarrow g(x,y)$

)
Ensure: $x \xrightarrow[SA]{P}_G y$.
begin
  for $i = 1$ *to* k **do**
    ΔR_i=$EVAL(r_i, E_1, \cdots, E_m, \emptyset, \cdots, \emptyset)$
    if *predicate(ΔR_i,'g') and arity(ΔR_i,2)* **then**
      A[constant(ΔR_{i1})] = constant(ΔR_{i2})
      remove(constant(ΔR_{i1}),U_L)
    end
    R_i=ΔR_i
  end
  repeat
    for $i = 1$ *to* k **do**
      ΔQ_i=ΔR_i
    end
    for $i = 1$ *to* k **do**
      ΔR_i=
      $EVAL(r_i, E_1, \cdots, E_m, R_1, \cdots, R_k, \Delta Q_1, \cdots, \Delta Q_k)$
      ΔR_i=ΔR_i-R_i
      if *predicate(ΔR_i,'g') and arity(ΔR_i,2)* **then**
        A[constant(ΔR_{i1})]=constant(ΔR_{i2})
        remove(constant(ΔR_{i1}),U_L)
      end
    end
    for $i = 1$ *to* k **do**
      R_i=$R_i \cup \Delta R_i$
    end
  until ΔR_i=$\emptyset$ $\forall i$, $1 \leq i \leq k$
  if *A is empty or (U_L contains zero elements not in keys of A and A doesn't contain two different substitution answers)* **then**
    return false
  end
  return true
end

Algorithm 6: Semi Naive - Flow Detection based on Substitution Answers

In order to demonstrate the decidability of (π_{SA}), we need to prove the following lemmas:

Lemma 4.18 (Termination)**.** `Algorithm 6` *terminates.*

Lemma 4.19 (Completeness)**.** *If* $x \xrightarrow[SA]{P}_G y$, *then* `Algorithm 6` *returns* `true`.

Lemma 4.20 (Soundness)**.** *If* `Algorithm 6` *returns* `true`, *then* $x \xrightarrow[SA]{P}_G y$.

Proof of Lemma 4.18. Same proof as 4.1. □

Proof of Lemma 4.19. Same proof as 4.8. □

Proof of Lemma 4.20. Same proof as 4.9. □

As a consequence of lemmas 4.18 – 4.20, we have:

Theorem 4.21. `Algorithm 6` *is a sound and complete decision procedure for* (π_{SA}).

It follows that (π_{SA}) is decidable. Moreover,

Theorem 4.22. (π_{SA}) *is EXPTIME-Complete*

Proof. (*Membership*) Concerning the time complexity of this algorithm, our study will be based on the set of pairs A.
First, we have to sort the set of substitution answers which is called A using merge-sort algorithm:
worst case:
Suppose the length of keys of A is n. And each entry of A (i.e values of substitution answers) is n also.
To sort one row it is about $\theta(nlogn)$.
To sort all rows, time complexity will be about $\theta(n^2logn)$.
To search an element in a sorted set, complexity will be $\theta(log_2 n)$.
To compare all rows with each other $\theta(n^2logn)$.
Consequently, this algorithm is executed in EXPTIME.

(*Hardness:*) In order to prove EXPTIME-hardness, we consider the following decision problem known to be EXPTIME-hard [16, 21]:

$$\pi \begin{cases} \text{Input: A Datalog program } P, \text{ a ground atom } A \\ \text{Output: } P \cup A \ (A \text{ is a logical consequence of } P) \end{cases}$$

Let (P,A) an instance of π and let $(P', g(x,y))$ be the instance of π_{SA} defined by $P' = P \cup \{g(a,y) \leftarrow A\}$, where g is a new predicate symbol. Thus $P \cup A$ iff $x \xrightarrow[g]{P'} y$.
$(\rightarrow)$ Suppose that A is a logical consequence of P, thus, $\theta(P' \cup \{\leftarrow g(a,y)\}) \neq \emptyset$ and $\theta(P' \cup \{\leftarrow g(b,y)\}) = \emptyset$. Consequently, $x \xrightarrow[g]{P'} y$.

$(\leftarrow)$ Suppose that $x \xrightarrow[g]{P'} y$. Then $\exists a, b \in U_{L(P)}$ such that $\theta(P' \cup \{\leftarrow g(a',y)\}) \neq \emptyset$ and $\theta(P' \cup \{\leftarrow g(b',y)\}) = \emptyset$. Hence, it follows that $a' = a$ and $b' \neq a$. Thus, $P \cup A$. □

5. Conclusion

in this paper, we presented an ewest approach of information flow detection in Datalog based using bottom up evaluation. Implications between definitions were studied, in addition to the implications between definitions of bottom up and top down evaluations. We studied the decidability and computational complexity of the flow detection using each the two algorithms, Very Naive and Semi Naive respectively.

We showed that information flow detection in logic programming using the two aforementioned algorithms is decidable and it is EXPTIME-Complete.

Future work can be dedicated to the study of the third information flow definition in Datalog programs based on bottom up evaluation, which is the bisimilarity between goals.

Magic Sets algorithm is an efficient algorith for bottom-up evalation for Dalaog goals, since it is query based algorithm. In other words, it returns the same answer when the same query is run using top down evaluation. A future study could be based on studying the complexity of information flow detection.

Moreover, we can base our research on another type of Datalog programs such as stratified Datalog, which is a Datalog program that accepts negative literals in its rules, i.e. $p(x,y) \leftarrow \neg q(x,y)$.

References

[1] BELL,D. AND LAPADULA,L. Secure Computer Systems: Mathematical Foundations and Model. *The MITRE Corporation Bedford MA Technical Report M74244*, 1973, vol. 1.

[2] FOLEY, S. A model for secure information flow. *Security and Privacy, 1989. Proceedings., 1989 IEEE Symposium on*, 1989, p. 248-258.

[3] DENNING, D. A lattice model of secure information flow. *Commun. ACM 19, 5 (May 1976)*, 1976, p. 236-243.

[4] GOGUEN, J. AND MESEGUER, J. Security policies and security models. *IEEE Symposium on Security and Privacy*, 1982, p. 11-20.

[5] DENNING, D. AND DENNING, P. Certification of programs for secure information flow. *Commun. ACM 20, 7 (July 1977)*, 1977, p. 504-513.

[6] FENTON, J. Memoryless subsystems *The Computer Journal, 17*, 1974, p. 143-147.

[7] DENNING, D. *Cryptography and Data Security*, Addison-Wesley, 1982.

[8] DETREVILLE, J.. Binder, a Logic-Based Security Language. *Proceedings of the 2002 IEEE Symposium on Security and Privacy (SP '02). IEEE Computer Society, Washington, DC, USA*, 2002.

[9] BERTINO, L. CATANIA, B. FERRARI, E. AND PERLASCA, P. A logical framework for reasoning about access control models. *Proceedings of the sixth ACM symposium on Access control models and technologies (SACMAT '01)*, 2001, p. 41-52.

[10] WANG, L. WIJESEKERA, D. AND JAJODIA, S. A logic-based framework for attribute based access control. *Proceedings of the 2004 ACM workshop on Formal methods in security engineering (FMSE '04)*, 2004, p. 45-55.

[11] BAI, Y. AND KHAN, K. A modal logic for information system security. *Proceedings of the Ninth Australasian Information Security Conference - Volume 116 (AISC '11)*, 2011, p. 51-56.

[12] COETZEE, M. AND ELOFF, J.H.P. A logic-based access control approach for Web Services *ISSA 2004 (Information Security for South Africa)*, 2004.

[13] 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.

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

[15] 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.

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

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

[18] 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

[19] 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.

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

[21] VARDI, M. The Complexity of Relational Query Languages (Extended Abstract). *Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing*, 1982, p. 137–146.