Query–subquery nets for Horn knowledge bases in first-order logic
Son Thanh Cao and Linh Anh Nguyen

Faculty of Information Technology, Vinh University, Vinh, Nghe An, Vietnam; Division of Knowledge and System Engineering for ICT, Faculty of Information Technology, Ton Duc Thang University, Ho Chi Minh City, Vietnam; Institute of Informatics, University of Warsaw, Warsaw, Poland

ABSTRACT
We formulate query–subquery nets and use them to create the first framework for developing algorithms for evaluating queries to Horn knowledge bases with the properties that: the approach is goal-directed; each subquery is processed only once and each supplement tuple, if desired, is transferred only once; operations are done set-at-a-time; and any control strategy can be used. Our intention is to increase efficiency of query processing by eliminating redundant computation, increasing adjustability (i.e. easiness in adopting advanced control strategies) and reducing the number of accesses to the secondary storage. For this purpose, we transform a logic program into an equivalent net structure and use it to determine which set of tuples or subqueries should be evaluated at each step, in an efficient way. The framework forms a generic evaluation method called QSQN, which is sound and complete and has polynomial time data complexity when the term-depth bound is fixed. The experimental results confirm the efficiency and usefulness of this method.

1. Introduction
Query processing is an important research area in computer science and information technology. Huang, Green, and Loo (2011) wrote we are witnessing an exciting revival of interest in recursive Datalog queries in a variety of emerging application domains such as data integration, information extraction, networking, program analysis, security, and cloud computing. During the last decade, rule-based query languages, including languages related to Datalog, were also intensively studied for the Semantic Web (e.g. in Cao, Nguyen, & Szalas, 2014; Eiter, Ianni, Lukasiewicz, & Schindlauer, 2011; Ruckhaus, Ruiz, & Vidal, 2008). In general, since deductive databases and knowledge bases are widely used in practical applications, improvements for processing recursive queries are always desirable. Due to the importance of the topic, it is worth doing further research on the topic.

Horn knowledge bases are extensions of Datalog deductive databases without the range-restrictedness and function-free conditions. As argued by Madalińska-Bugaj and Nguyen...
(2012), the Horn fragment of first-order logic plays an important role in knowledge representation and reasoning. A Horn knowledge base consists of a positive logic program for defining intensional predicates and an instance of extensional predicates. When the knowledge base is too big, not all of the extensional and intensional relations can be totally kept in the computer memory and query evaluation cannot be totally done in the computer memory. In such cases, the system usually has to load (resp. unload) relations from (resp. to) the secondary storage. Thus, in contrast to logic programming, for Horn knowledge bases efficient access to the secondary storage is a very important aspect.

This work studies query processing for Horn knowledge bases, which is a topic that has not been well studied as query processing for Datalog-like deductive databases or the theory and techniques of logic programming. The survey by Ramakrishnan and Ullman (1995) provides a good overview of deductive database systems, with a focus on implementation techniques. The book by Abiteboul, Hull, and Vianu (1995) is also a good source for references. We refer the reader to Madalińska-Bugaj and Nguyen (2012) for a discussion on query processing for Horn knowledge bases.

The most well-known methods for evaluating queries to Datalog deductive databases or Horn knowledge bases are QSQR (Madalińska-Bugaj & Nguyen, 2012; Vieille, 1989) and Magic-Sets (Bancilhon, Maier, Sagiv, & Ullman, 1986; Beeri & Ramakrishnan, 1991; Rohmer, Lescouer, & Kerisit, 1986). By Magic-Sets we mean the evaluation method that combines the magic-set transformation with the improved semi-naive bottom-up evaluation method. Both of these methods are goal-directed. As observed by Vieille (1989), the QSQR approach is like iterative deepening search. It allows redundant recomputations (Madalińska-Bugaj & Nguyen, 2012, Remark 3.2). On the other hand, the Magic-Sets method applies breadth-first search. The following example shows that the breadth-first approach is not always efficient.

**Example 1.1** The order of program clauses and the order of atoms in the bodies of program clauses may be essential, for example, when the positive logic program that defines intensional predicates is specified using the Prolog programming style. In such cases, the top-down depth-first approach may be much more efficient than the breadth-first approach. Here is such an example, in which \( p, q_1 \) and \( q_2 \) are intensional predicates, \( r_1 \) and \( r_2 \) are extensional predicates, \( x, y \) and \( z \) are variables, \( a_i \) and \( b_{i,j} \) are constant symbols:

- the positive logic program:
  
  \[
  p \leftarrow q_1(a_0, a_m) \\
  p \leftarrow q_2(a_0, a_m) \\
  q_1(x, y) \leftarrow r_1(x, y) \\
  q_1(x, y) \leftarrow r_1(x, z), q_1(z, y) \\
  q_2(x, y) \leftarrow r_2(x, y) \\
  q_2(x, y) \leftarrow r_2(x, z), q_2(z, y),
  \]

- the extensional instance (illustrated in Figure 1):
  
  \[
  l(r_1) = \{(a_i, a_{i+1}) \mid 0 \leq i < m\} \\
  l(r_2) = \{(a_0, b_{1,j}) \mid 1 \leq j \leq n\} \\
  \cup \{(b_{i,j}, b_{i+1,j}) \mid 1 \leq i < m - 1 \text{ and } 1 \leq j \leq n\} \\
  \cup \{(b_{m-1,j}, a_m) \mid 1 \leq j \leq n\},
  \]

- the query: \( p \).
Notice that the depth-first approach needs only $\Theta(m)$ steps for evaluating the query, while the breadth-first approach performs $\Theta(m \cdot n)$ steps. When $n$ is comparable to $m$, the difference is too big. The magic-sets transformation does not help for this case.

Our postulate is that the breadth-first approach (including the Magic-Sets evaluation method) is inflexible and not always efficient. Of course, depth-first search is not always good either. The aim of this work is to develop an evaluation method for Horn knowledge bases that is more efficient than the QSQR evaluation method and more adjustable than the Magic-Sets evaluation method. In particular, a good method should be not only set-oriented and goal-directed but should also reduce computational redundancy as much as possible and allow various control strategies.

This paper is a revised and extended version of our conference paper (Nguyen & Cao, 2012) and forms a chapter of the Ph.D. dissertation (Cao, 2016). In this work, we formulate query-subquery nets and use them to develop the first framework for developing algorithms for evaluating queries to Horn knowledge bases with the following properties:

- the approach is goal-directed,
- each subquery is processed only once,
- each supplement tuple, if desired, is transferred only once,
- operations are done set-at-a-time,
- any control strategy can be used.

The intention of our framework is to increase efficiency of query processing by eliminating redundant computation, increasing adjustability\(^1\) and reducing the number of

---

\(^1\) Adjustability refers to the ability of a method to adapt to different situations and requirements.
accesses to the secondary storage. The framework forms a generic evaluation method called QSQN. As a supplement, the Ph.D. dissertation (Cao, 2016) also contains:

- proofs of soundness and completeness of the QSQN method,
- data complexity analysis for the QSQN method,
- a control strategy called the Improved Depth-First Control Strategy (IDFS), which together with QSQN forms the QSQN-IDFS method,
- experiments for comparing the QSQN-IDFS, Magic-Sets and QSQR methods w.r.t.
  - the number of read/write operations on relations,
  - the maximum number of tuples/subqueries kept in the computer memory,
  - the number of accesses to the secondary storage when the memory is limited.

To deal with function symbols, we use a term-depth bound for atoms and substitutions occurring in the computation and propose to use iterative deepening search which iteratively increases the term-depth bound. Similar to the work by Madalińska-Bugaj and Nguyen (2012) but in contrast to the QSQ framework for Datalog queries (Abiteboul et al., 1995), our framework for Horn knowledge bases does not use adornments and annotations, but uses substitutions instead. This is natural for the case with function symbols and without the range-restrictedness condition.

Our experiments show that the QSQN-IDFS evaluation method is more efficient than the QSQR evaluation method and as competitive as the Magic-Sets evaluation method. In the case when the order of program clauses and the order of atoms in the bodies of program clauses are essential as in Prolog programming, the QSQN-IDFS evaluation method usually outperforms the Magic-Sets method. As QSQN-IDFS is just an instance of the generic QSQN evaluation method, we claim that this generic method is useful.

The rest of this paper is structured as follows. Section 2 recalls the most important notation and definitions of first-order logic, logic programming and Horn knowledge bases. Section 3 presents our QSQN evaluation method for Horn knowledge bases. The preliminary experiments are discussed in Section 4. Conclusions are given in Section 5.

2. Preliminaries

First-order logic is considered in this work and we assume that the reader is familiar with it. We recall only the most important definitions for our work and refer the reader to Lloyd (1987) and Madalińska-Bugaj and Nguyen (2012) for further reading.

A signature for first-order logic consists of constant symbols, function symbols, variable symbols and predicate symbols. Terms, atoms and formulas are defined in the usual way. An expression is either a term, a tuple of terms, a formula without quantifiers or a list of formulas without quantifiers. A simple expression is either a term or an atom.

A substitution is a finite set of the form \( \theta = \{x_1/t_1, \ldots, x_k/t_k\} \), where \( x_1, \ldots, x_k \) are pairwise distinct variables, \( t_1, \ldots, t_k \) are terms, and \( t_i \neq x_i \) for all \( 1 \leq i \leq k \). We denote \( \varepsilon \) the empty substitution.

The domain of a substitution \( \theta \) is the set \( \text{dom}(\theta) = \{x_1, \ldots, x_k\} \), and the range of \( \theta \) is the set \( \text{range}(\theta) = \{t_1, \ldots, t_k\} \). The restriction of a substitution \( \theta \) to a set \( X \) of variables is the substitution \( \theta |_X = \{(x/t) \in \theta \mid x \in X\} \).
Let \( \theta = \{ x_1/t_1, \ldots, x_k/t_k \} \) be a substitution and \( E \) be an expression. Then \( E\theta \), the instance of \( E \) by \( \theta \), is the expression obtained from \( E \) by simultaneously replacing all occurrences of the variable \( x_i \) in \( E \) by the term \( t_i \), for \( 1 \leq i \leq k \).

Let \( \theta = \{ x_1/t_1, \ldots, x_k/t_k \} \) and \( \delta = \{ y_1/s_1, \ldots, y_h/s_h \} \) be substitutions (where \( x_1, \ldots, x_k \) are pairwise distinct variables, and \( y_1, \ldots, y_h \) are also pairwise distinct variables). Then the composition \( \theta \delta \) of \( \theta \) and \( \delta \) is the substitution obtained from the sequence \( \{ x_1/(t_1\delta), \ldots, x_k/(t_k\delta), y_1/s_1, \ldots, y_h/s_h \} \) by deleting any binding \( x_i/(t_i\delta) \) for which \( x_i = (t_i\delta) \) and deleting any binding \( y_j/s_j \) for which \( y_j \in \{ x_1, \ldots, x_k \} \).

A substitution \( \theta \) is idempotent if \( \theta\theta = \theta \). We have that \( \theta = \{ x_1/t_1, \ldots, x_k/t_k \} \) is idempotent if none of \( x_1, \ldots, x_k \) occurs in any \( t_1, \ldots, t_k \).

If \( \theta \) and \( \delta \) are substitutions such that \( \theta\delta = \delta\theta = \varepsilon \), then we call them renaming substitutions. We say that an expression \( E \) is a variant of an expression \( E' \) if there exist substitutions \( \theta \) and \( \gamma \) such that \( E = E'\theta \) and \( E' = E\gamma \).

A substitution \( \theta \) is more general than a substitution \( \delta \) if there exists a substitution \( \gamma \) such that \( \delta = \theta\gamma \). Let \( \Gamma \) be a set of simple expressions. A substitution \( \theta \) is called a unifier for \( \Gamma \) if \( \Gamma\theta \) is a singleton. If \( \Gamma\theta = \{ \varphi \} \) then we say that \( \theta \) unifies \( \Gamma \) (into \( \varphi \)). A unifier \( \theta \) for \( \Gamma \) is called a most general unifier (mgu) for \( \Gamma \) if \( \theta \) is more general than every unifier of \( \Gamma \).

The term-depth of an expression (resp. a substitution) is the maximal nesting depth of function symbols occurring in that expression (resp. substitution). If \( E \) is an expression or a substitution then by \( \text{Vars}(E) \) we denote the set of variables occurring in \( E \). If \( \varphi \) is a formula then by \( \forall(\varphi) \) we denote the universal closure of \( \varphi \), which is the formula obtained by adding a universal quantifier for every variable having a free occurrence in \( \varphi \).

A (positive or definite) program clause is a formula of the form \( \forall(A \lor \neg B_1 \lor \ldots \lor \neg B_k) \) with \( k \geq 0 \), written as \( A \leftarrow B_1, \ldots, B_k \), where \( A, B_1, \ldots, B_k \) are atoms (i.e., atomic formulas). \( A \) is called the head, and \( B_1, \ldots, B_k \) the body of the program clause. If \( p \) is the predicate of \( A \) then the program clause is called a program clause defining \( p \).

A positive (or definite) logic program is a finite set of program clauses.

A goal (also called a negative clause) is a formula of the form \( \forall(\neg B_1 \lor \ldots \lor \neg B_k) \), written as \( \leftarrow B_1, \ldots, B_k \), where \( B_1, \ldots, B_k \) are atoms. If \( k=1 \) then the goal is called a unary goal. If \( k=0 \) then the goal stands for falsity and is called the empty goal (or the empty clause) and denoted by \( \square \).

A fresh variant of a formula \( \varphi \), where \( \varphi \) can be an atom, a goal \( \leftarrow A \) or a program clause \( A \leftarrow B_1, \ldots, B_k \) (written without quantifiers), is a formula \( \varphi\theta \), where \( \theta \) is a renaming substitution such that \( \text{dom}(\theta) = \text{Vars}(\varphi) \) and \( \text{range}(\theta) \) consists of variables that were not used in the computation.

Similarly as for deductive databases, we classify each predicate either as intensional or as extensional. A generalized tuple is a tuple of terms, which may contain function symbols and variables. A generalized relation is a set of generalized tuples of the same arity. A Horn knowledge base is defined to be a pair consisting of a positive logic program for defining intensional predicates and a generalized extensional instance, which is a function mapping each extensional \( n \)-ary predicate to an \( n \)-ary generalized relation. Note that intensional predicates are defined by a positive logic program which may contain function symbols and not be range-restricted. From now on, we use the term ‘relation’ to mean a generalized relation, and the term ‘extensional instance’ to mean a generalized extensional instance.

Given a Horn knowledge base specified by a positive logic program \( P \) and an extensional instance \( I \), a query to the knowledge base is a positive formula \( \varphi(\bar{x}) \) without quantifiers,
where $\bar{x}$ is a tuple of all the variables of $\varphi$. A (correct) answer for the query is a tuple $\bar{t}$ of terms of the same length as $\bar{x}$ such that $P \cup I \models \forall(\varphi(\bar{t}))$. When measuring data complexity, we assume that $P$ and $\varphi$ are fixed, while $I$ varies. Thus, the pair $(P, \varphi(\bar{x}))$ is treated as a query to the extensional instance $I$. We will use the term ‘query’ in this meaning.

It can be shown that, every query $(P, \varphi(\bar{x}))$ can be transformed in polynomial time to an equivalent query of the form $(P', q(\bar{x}))$ over a signature extended with new intensional predicates, including $q$. The equivalence means that, for every extensional instance $I$ and every tuple $\bar{t}$ of terms of the same length as $\bar{x}$, $P \cup I \models \forall(\varphi(\bar{t}))$ iff $P' \cup I \models \forall(q(\bar{t}))$.

The transformation is based on introducing new predicates for defining complex subformulas occurring in the query. For example, if $\varphi = p(x) \land r(x, y)$, then $P' = P \cup \{q(x, y) \leftarrow p(x), r(x, y)\}$, where $q$ is a new intensional predicate.

Without loss of generality, we will consider only queries of the form $(P, q(\bar{x}))$, where $q$ is an intensional predicate. Answering such a query on an extensional instance $I$ is to find (correct) answers for $P \cup I \cup \{\leftarrow q(\bar{x})\}$.

### 3. The query–subquery net evaluation method

In this section, we generalize the QSQ approach for Horn knowledge bases. Given a positive logic program, we make a query–subquery net structure and use it as a flow control network to determine which subqueries in which nodes should be processed next. We show how the data are transferred through edges of the net. We also propose an algorithm together with related procedures and functions for this framework. The algorithm repeatedly selects an active edge and fires the operation for the edge to transfer unprocessed data. Such a selection is decided by the adopted control strategy, which can be arbitrary. In addition, the processing is divided into smaller steps which can be delayed to maximize adjustability and allow various control strategies. The intention is to increase efficiency of query processing by eliminating redundant computation, increasing adjustability and reducing the number of accesses to the secondary storage.

In what follows, $P$ is a positive logic program and $\varphi_1, \ldots, \varphi_m$ are all the program clauses of $P$, with $\varphi_i = (A_i \leftarrow B_{i,1}, \ldots, B_{i,n_i})$, for $1 \leq i \leq m$ and $n_i \geq 0$. The following definition shows how to make a QSQ-net structure from the given logic program $P$.

**Definition 3.1 (Query–Subquery Net Structure):** A query–subquery net structure (QSQ-net structure for short) of $P$ is a tuple $(V, E, T)$ such that:

- $V$ is a set of nodes that consists of:
  - $input_p$ and $ans_p$, for each intensional predicate $p$ of $P$,
  - $pre_{filter}_i$, $filter_{i,1}, \ldots, filter_{i,n_i}$, $post_{filter}_i$, for each $1 \leq i \leq m$.
- $E$ is a set of edges that consists of:
  - $(filter_{i,1}, filter_{i,2}), \ldots, (filter_{i,n_{i-1}}, filter_{i,n_i})$, for each $1 \leq i \leq m$,
  - $(pre_{filter}_i, filter_{i,1})$ and $(filter_{i,n_i}, post_{filter}_i)$, for each $1 \leq i \leq m$ with $n_i \geq 1$,
  - $(pre_{filter}_i, post_{filter}_i)$, for each $1 \leq i \leq m$ with $n_i = 0$,
  - $(input_p, pre_{filter}_i)$ and $(post_{filter}_i, ans_p)$, for each $1 \leq i \leq m$, where $p$ is the predicate of $A_i$,
  - $(filter_{i,j}, input_p)$ and $(ans_p, filter_{i,j})$, for each intensional predicate $p$ and each $1 \leq i \leq m$ and $1 \leq j \leq n_i$ such that $B_{i,j}$ is an atom of $p$. 


• $T$ is a function, called the *memorizing type* of the net structure, mapping each node $\text{filter}_{i,j} \in V$ such that the predicate of $B_{i,j}$ is extensional to true or false. If $T(\text{filter}_{i,j}) = \text{false}$ (and the predicate of $B_{i,j}$ is extensional) then subqueries for $\text{filter}_{i,j}$ are always processed immediately, without being accumulated at $\text{filter}_{i,j}$.

If $(v, w) \in E$ then we call $w$ a *successor* of $v$, and $v$ a *predecessor* of $w$. Note that $V$ and $E$ are uniquely specified by $P$. We call the pair $(V, E)$ the *QSQ topological structure* of $P$.

**Example 3.2** Consider the following (recursive) positive logic program, where $x$, $y$ and $z$ are variables, $p$ is an intensional predicate, and $q$ is an extensional predicate:

\[
\begin{align*}
p(x, y) & \leftarrow q(x, y) \\
p(x, y) & \leftarrow q(x, z), p(z, y).
\end{align*}
\]

Its QSQ topological structure is illustrated in **Figure 2**.

![Figure 2](image_url)

**Figure 2.** The QSQ topological structure of the program given in Example 3.2.

**Example 3.3** Consider the following positive logic program, where $x$, $y$ and $z$ are variables, $p$ and $r$ are intensional predicates, $q$, $s$ and $t$ are extensional predicates:

\[
\begin{align*}
p(x, y) & \leftarrow q(x, z), r(z, y) \\
r(x, y) & \leftarrow s(x, y) \\
r(x, y) & \leftarrow t(x, y).
\end{align*}
\]

This program is a modified version of an example from Zhou and Sato (2003). **Figure 3** illustrates the QSQ topological structure of this program.

**Definition 3.4 (Query–Subquery Net):** A *query–subquery net* (QSQ-net for short) of $P$ is a tuple $N = (V, E, T, C)$ such that $(V, E, T)$ is a QSQ-net structure of $P$, $C$ is a mapping that associates each node $v \in V$ with a structure called the *contents* of $v$, and the following conditions are satisfied:

• $C(v)$, where $v = \text{input}_p$ or $v = \text{ans}_p$ for an intensional predicate $p$ of $P$, consists of:
  • tuples$(v)$: a set of generalized tuples of the same arity as $p$,
  • unprocessed$(v, w)$ for each $(v, w) \in E$: a subset of tuples$(v)$.

• $C(v)$, where $v = \text{prefilter}_i$, consists of:
  • atom$(v) = A_i$ and post_vars$(v) = \text{Vars}(B_{i,1}, \ldots, B_{i,n_i})$. 


• $C(v)$, where $v = \text{post}_i$ is empty, but we assume $\text{pre}_\text{vars}(v) = \emptyset$.
• $C(v)$, where $v = \text{filter}_i$ and $p$ is the predicate of $\text{Bi}_i$, consists of:
  • $\text{kind}(v) = \text{extensional}$ if $p$ is extensional, and $\text{kind}(v) = \text{intensional}$ otherwise,
  • $pred(v) = p$ and $\text{atom}(v) = \text{Bi}_i$,
  • $\text{pre}_\text{vars}(v) = \text{Vars}(\text{Bi}_1, \ldots, \text{Bi}_{i-1}, \text{Bi}_i, \text{Bi}_{i+1}, \ldots, \text{Bi}_n)$ and $\text{post}_\text{vars}(v) = \text{Vars}(\text{Bi}_1, \ldots, \text{Bi}_n),$
  • $\text{subqueries}(v)$: a set of pairs of the form $(\tilde{t}, \delta)$, where $\tilde{t}$ is a generalized tuple of the same arity as the predicate of $\text{Ai}_i$ and $\delta$ is an idempotent substitution such that $\text{dom}(\delta) \subseteq \text{pre}_\text{vars}(v)$ and $\text{dom}(\delta) \cap \text{Vars}(\tilde{t}) = \emptyset$,
  • $\text{unprocessed}_\text{subqueries}(v) \subseteq \text{subqueries}(v),$
  • in the case $p$ is intensional:
    • $\text{unprocessed}_\text{subqueries}_2(v) \subseteq \text{subqueries}(v),$
    • $\text{unprocessed}_\text{tuples}(v)$: a set of generalized tuples of the same arity as $p.$
• If $v = \text{filter}_i, \text{kind}(v) = \text{extensional}$ and $T(v) = \text{false}$ then $\text{subqueries}(v) = \emptyset.$

Figure 4 illustrates a QSQ-net of the positive logic program given in Example 3.2. By a subquery we mean a pair of the form $(\tilde{t}, \delta)$, where $\tilde{t}$ is a generalized tuple and $\delta$ is an idempotent substitution such that $\text{dom}(\delta) \cap \text{Vars}(\tilde{t}) = \emptyset.$

For $v = \text{filter}_i$ and $p$ being the predicate of $\text{Ai}_i$, the meaning of a subquery $(\tilde{t}, \delta) \in \text{subqueries}(v)$ is that: for processing a goal $\leftarrow p(s)$ with $s \in \text{tuples}(\text{input}, p)$ using the program clause $\varphi_i = (\text{Ai}_i \leftarrow \text{Bi}_1, \ldots, \text{Bi}_n)$, unification of $p(s)$ and $\text{Ai}_i$ as well as processing of the subgoals $\text{Bi}_1, \ldots, \text{Bi}_{i-1}$ were done, amongst others, by using a sequence of mgu’s $\gamma_0, \ldots, \gamma_{j-1}$ with the property that $\tilde{t} = \tilde{s}\gamma_0 \cdots \gamma_{j-1}$ and $\delta = (\gamma_0 \cdots \gamma_{j-1})|_{\text{Vars}(\text{Bi}_1, \ldots, \text{Bi}_n)}.$

An empty QSQ-net of $P$ is a QSQ-net of $P$ such that all the sets of the form $\text{tuples}(v), \text{unprocessed}(v, w), \text{subqueries}(v), \text{unprocessed}_\text{subqueries}(v), \text{unprocessed}_\text{subqueries}_2(v)$ or $\text{unprocessed}_\text{tuples}(v)$ are empty.

In a QSQ-net, if $v = \text{pre}_i$ or $v = \text{post}_i$ or $v = \text{filter}_i$ and $\text{kind}(v) = \text{extensional}$ then $v$ has exactly one successor, which we denote by $\text{succ}(v)$.

If $v$ is $\text{filter}_i$ with $\text{kind}(v) = \text{intensional}$ and $\text{pred}(v) = p$ then $v$ has exactly two successors. In that case, let

$$\text{succ}(v) = \begin{cases} \text{filter}_{i+1} & \text{if } n_i > j, \\ \text{post}_i & \text{otherwise}, \end{cases}$$
\[ p(x, y) \leftarrow q(x, y) \]
\[ p(x, y) \leftarrow q(x, z), p(z, y) \]

A QSQ-net of the above positive logic program has the form:

**Figure 4.** The QSQ-net of the program given in Example 3.2.
and $\text{succ}_2(v) = \text{input}_p$. The set $\text{unprocessed_subqueries}(v)$ is used for (i.e. corresponds to) the edge $(v, \text{succ}(v))$, while $\text{unprocessed_subqueries}_2(v)$ is used for the edge $(v, \text{succ}_2(v))$.

Note that if $\text{succ}(v) = w$ then $\text{post-vars}(v) = \text{pre-vars}(w)$. In particular, $\text{post-vars}(\text{filter}_{i,n}) = \text{pre-vars}(\text{post_filter}_i) = \emptyset$.

The formats of data transferred through edges of a QSQ-net are specified as follows:

- data transferred through an edge of the form $(\text{input}_p, v)$, $(v, \text{input}_p)$, $(v, \text{ans}_p)$ or $(\text{ans}_p, v)$ is a finite set of generalized tuples of the same arity as $p$,
- data transferred through an edge $(u, v)$ with $v = \text{filter}_{i,j}$ and $u$ not being of the form $\text{ans}_p$ is a finite set of subqueries that can be added to $\text{subqueries}(v)$,
- data transferred through an edge $(v, \text{post_filter}_i)$ is a set of subqueries $(\tilde{t}, \varepsilon)$ such that $\tilde{t}$ is a generalized tuple of the same arity as the predicate of $A_i$.

If $(\tilde{t}, \delta)$ and $(\tilde{t}', \delta')$ are subqueries that can be transferred through an edge to $v$ then we say that $(\tilde{t}, \delta)$ is more general than $(\tilde{t}', \delta')$ w.r.t. $v$, and that $(\tilde{t}', \delta')$ is less general than $(\tilde{t}, \delta)$ w.r.t. $v$, if there exists a substitution $\gamma$ such that $\tilde{t}\gamma = \tilde{t}'$ and $(\delta\gamma)|_{\text{pre-vars}(v)} = \delta'$.

Informally, a subquery $(\tilde{t}, \delta)$ transferred through an edge to $v$ is processed as follows:

- if $v = \text{filter}_{i,j}$, $\text{kind}(v) = \text{extensional}$ and $\text{pred}(v) = p$ then, for each $\tilde{t}' \in I(p)$, if $\text{atom}(v)\delta = B_{i,j}\delta$ is unifiable with a fresh variant of $p(\tilde{t}')$ by an mgu $\gamma$ then transfer the subquery $(\tilde{t}\gamma, (\delta\gamma)|_{\text{post-vars}(v)})$ through $(v, \text{succ}(v))$,
- if $v = \text{filter}_{i,j}$, $\text{kind}(v) = \text{intensional}$ and $\text{pred}(v) = p$ then
  - transfer the tuple $\tilde{t}'$ such that $p(\tilde{t}') = \text{atom}(v)\delta = B_{i,j}\delta$ through $(v, \text{input}_p)$ to add a fresh variant of it to $\text{tuples}(\text{input}_p)$,
  - for each currently existing $\tilde{t}' \in \text{tuples}(\text{ans}_p)$, if $\text{atom}(v)\delta = B_{i,j}\delta$ is unifiable with a fresh variant of $p(\tilde{t}')$ by an mgu $\gamma$ then transfer the subquery $(\tilde{t}\gamma, (\delta\gamma)|_{\text{post-vars}(v)})$ through $(v, \text{succ}(v))$,
  - store the subquery $(\tilde{t}, \delta)$ in $\text{subqueries}(v)$, and later, for each new $\tilde{t}'$ added to $\text{tuples}(\text{ans}_p)$, if $\text{atom}(v)\delta = B_{i,j}\delta$ is unifiable with a fresh variant of $p(\tilde{t}')$ by an mgu $\gamma$ then transfer the subquery $(\tilde{t}\gamma, (\delta\gamma)|_{\text{post-vars}(v)})$ through $(v, \text{succ}(v))$,
- if $v = \text{post_filter}_i$ and $p$ is the predicate of $A_i$ then transfer the tuple $\tilde{t}$ through $(\text{post_filter}_i, \text{ans}_p)$ to add it to $\text{tuples}(\text{ans}_p)$.

Formally, the processing of a subquery is designed more sophisticatedly so that:

- every subquery or input/answer tuple that is subsumed by another one or has a term-depth greater than a fixed bound $I$ is ignored,
- the processing is divided into smaller steps which can be delayed at each node to maximize adjustability and allow various control strategies,
- the processing is done set-at-a-time (e.g. for all the unprocessed subqueries accumulated in a given node).

The procedure $\text{transfer}(D, u, v)$ specifies the effects of transferring data $D$ through an edge $(u, v)$ of a QSQ-net. If $v$ is of the form $\text{pre_filter}_i$ or $\text{post_filter}_i$ or $(v = \text{filter}_{i,j}$ and $\text{kind}(v) = \text{extensional}$ and $T(v) = \text{false}$) then the input $D$ for $v$ is processed immediately and an appropriate data $I$ is produced and transferred through $(v, \text{succ}(v))$. Otherwise,
the input $D$ for $v$ is not processed immediately, but accumulated into the structure of $v$ in an appropriate way.

The function $\text{active-edge}(u, v)$ returns $true$ for an edge $(u, v)$ if data accumulated in $u$ can be processed to produce some data to transfer through $(u, v)$, and returns $false$ otherwise. If $\text{active-edge}(u, v)$ is $true$ then the procedure $\text{fire}(u, v)$ processes the data accumulated in $u$ that has not been processed before to transfer appropriate data through the edge $(u, v)$. This procedure uses the procedure $\text{transfer}(D, u, v)$. Both procedures $\text{fire}(u, v)$ and $\text{transfer}(D, u, v)$ use a parameter $l$ as a term-depth bound for tuples and substitutions.

Algorithm 1 presents our QSQN evaluation method for Horn knowledge bases. It repeatedly selects an active edge and fires the operation for the edge. Such a selection is decided by the adopted control strategy, which can be arbitrary.

**Algorithm 1**: for evaluating a query $(P, q(\bar{x}))$ on an extensional instance $I$.

1. let $(V, E, T)$ be a QSQ-net structure of $P$; // $T$ can be chosen arbitrarily
2. set $C$ so that $N = (V, E, T, C)$ is an empty QSQ-net of $P$;
3. let $\bar{x}'$ be a fresh variant of $\bar{x}$;
4. $\text{tuples}(\text{input}.q) := \{\bar{x}'\}$;
5. foreach $(\text{input}.q, v) \in E$ do $\text{unprocessed}(\text{input}.q, v) := \{\bar{x}'\}$;
6. while there exists $(u, v) \in E$ such that $\text{active-edge}(u, v)$ holds do
7.   select $(u, v) \in E$ such that $\text{active-edge}(u, v)$ holds;
8.     // any strategy is acceptable for the above selection
9.     $\text{fire}(u, v)$;
10. return $\text{tuples}(\text{ans}.q)$;

**Example 3.5** This example illustrates Algorithm 1 step by step. Consider the following Horn knowledge base $(P, l)$ and the query $s(x)$, where $p$ and $s$ are intensional predicates, $q$ is an extensional predicate, $x, y, z$ are variables, and $a – o, u$ are constant symbols:

- the positive logic program $P$:

  $$p(x, y) \leftarrow q(x, y)$$
  $$p(x, y) \leftarrow q(x, z), p(z, y)$$
  $$s(x) \leftarrow p(b, x),$$

- the extensional instance $l$ (illustrated in Figure 5):

  $$l(q) = \{(a, b), (b, c), (c, d), (d, e), (b, f), (f, g), (b, h), (h, g), (i, j), (j, k), (k, l), (m, n),$$
  $$\quad (n, u), (n, o)\},$$

- the query: $s(x)$.

![Figure 5](image.png)

**Figure 5.** A graph used for Example 3.5.
The QSQ topological structure of P is presented in Figure 6. We give below a trace of a run of Algorithm 1 that evaluates the query \((P, s(x))\) on the extensional instance \(I\), using term-depth bound \(l=0\) and the memorizing type \(T\) that maps each node \(v\) such that \(\text{kind}(v) = \text{extensional}\) (i.e. \(\text{filter}_{1,1}\) and \(\text{filter}_{2,1}\)) to false. For convenience, we denote the edges of the net with names \(E_1\)–\(E_{17}\) as shown in Figure 6.

Algorithm 1 starts with an empty QSQ-net. It then adds a fresh variant \((x_1)\) of \((x)\) to the empty sets \(\text{tuples}(\text{input}_{s})\) and \(\text{unprocessed}(E_{14})\). Next, it repeatedly selects an active edge and fires the edge. Assume that the selection is done as follows.

1. \(E_{14} \rightarrow E_{15}\)
   After processing \(\text{unprocessed}(E_{14})\), the algorithm empties this set and transfers \(((x_1), \{x/x_1\})\) through the edge \(E_{14}\). This produces \(((x_1), \{x/x_1\})\), which is then transferred through the edge \(E_{15}\) and added to the empty sets \(\text{subqueries}(\text{filter}_{3,1})\), \(\text{unprocessed_subqueries}(\text{filter}_{3,1})\) and \(\text{unprocessed_subqueries}_{2}(\text{filter}_{3,1})\).

2. \(E_{13}\)
   After processing \(\text{unprocessed_subqueries}_{2}(\text{filter}_{3,1})\), the algorithm empties this set and
transfers \(((b, x_1))\) through \(E_{13}\). This adds a fresh variant \((b, x_2)\) of the tuple \((b, x_1)\) to the empty sets \(\text{tuples}(\text{input}_p)\), \(\text{unprocessed}(E_1)\) and \(\text{unprocessed}(E_7)\).

(3) \(E_7 - E_8 - E_9\)

After processing \(\text{unprocessed}(E_7)\), the algorithm empties this set and transfers \(((b, x_2))\) through the edge \(E_7\). This produces \(((b, x_2), \{x/y, x_2\})\), which is then transferred through the edge \(E_8\), producing \(((b, x_2), \{y/x, z/c\})\), \(((b, x_2), \{y/x, z/f\})\), \(((b, x_2), \{y/x, z/h\})\), which in turn is then transferred through the edge \(E_9\) and added to the empty sets \(\text{subqueries}(\text{filter}_{2,2})\), \(\text{unprocessed}_{\text{subqueries}}((\text{filter}_{2,2})\) and \(\text{unprocessed}_{\text{subqueries}}_2(\text{filter}_{2,2})\).

(4) \(E_6\)

After processing \(\text{unprocessed}_{\text{subqueries}}_2(\text{filter}_{2,2})\), the algorithm empties this set and transfers \(((c, x_3), (f, x_4), (h, x_5))\) through the edge \(E_6\). This adds fresh variants of these tuples, namely \((c, x_3)\), \((f, x_4)\) and \((h, x_5)\), to the sets \(\text{tuples}(\text{input}_p)\), \(\text{unprocessed}(E_1)\) and \(\text{unprocessed}(E_7)\). After these steps, we have:

- \(\text{unprocessed}(E_1) = \text{tuples}(\text{input}_p) = \{(b, x_2), (c, x_3), (f, x_4), (h, x_5)\}\),
- \(\text{unprocessed}(E_7) = \{(c, x_3), (f, x_4), (h, x_5)\}\).

(5) \(E_7 - E_8 - E_9\)

After processing \(\text{unprocessed}(E_7)\), the algorithm empties this set and transfers \(((c, x_3), (f, x_4), (h, x_5))\) through the edge \(E_7\). This produces \(((c, x_3), \{x/c, y/x_3\})\), \(((f, x_4), \{x/f, y/x_4\})\), \(((h, x_5), \{x/h, y/x_5\})\), which is then transferred through the edge \(E_8\), producing \(((c, x_3), \{y/x_3, z/d\})\), \(((f, x_4), \{y/x_4, z/g\})\), \(((h, x_5), \{y/x_5, z/g\})\), which in turn is then transferred through the edge \(E_9\) and added to the sets \(\text{subqueries}(\text{filter}_{2,2})\), \(\text{unprocessed}_{\text{subqueries}}(\text{filter}_{2,2})\) and \(\text{unprocessed}_{\text{subqueries}}_2(\text{filter}_{2,2})\). After these steps, we have:

- \(\text{unprocessed}_{\text{subqueries}}(\text{filter}_{2,2}) = \text{subqueries}(\text{filter}_{2,2}) = \{(b, x_2), \{y/x_2, z/c\})\),
- \(((b, x_2), \{y/x_2, z/f\})\), \(((b, x_2), \{y/x_2, z/h\})\), \(((c, x_3), \{y/x_3, z/d\})\), \(((f, x_4), \{y/x_4, z/g\})\), \(((h, x_5), \{y/x_5, z/g\})\),
- \(\text{unprocessed}_{\text{subqueries}}_2(\text{filter}_{2,2}) = \{(c, x_3), \{y/x_3, z/d\})\), \(((f, x_4), \{y/x_4, z/g\})\), \(((h, x_5), \{y/x_5, z/g\})\).

(6) \(E_6\)

After processing \(\text{unprocessed}_{\text{subqueries}}_2(\text{filter}_{2,2})\), the algorithm empties this set and transfers \(((d, x_6), (g, x_7))\) through the edge \(E_6\). This adds fresh variants of these tuples, namely \((d, x_6)\) and \((g, x_7)\), to the sets \(\text{tuples}(\text{input}_p)\), \(\text{unprocessed}(E_1)\) and \(\text{unprocessed}(E_7)\). After these steps, we have:

- \(\text{unprocessed}(E_1) = \text{tuples}(\text{input}_p) = \{(b, x_2), (c, x_3), (f, x_4), (h, x_5), (d, x_6), (g, x_7)\}\),
- \(\text{unprocessed}(E_7) = \{(d, x_6), (g, x_7)\}\).

(7) \(E_7 - E_8 - E_9\)

After processing \(\text{unprocessed}(E_7)\), the algorithm empties this set and transfers \(((d, x_6), (g, x_7))\) through the edge \(E_7\). This produces \(((d, x_6), \{x/d, y/x_6\})\), \(((g, x_7), \{x/g, y/x_7\})\), which is then transferred through the edge \(E_8\), producing \(((d, x_6), \{y/x_6, z/e\})\), which in turn is then transferred through the edge \(E_9\) and added to the sets \(\text{subqueries}(\text{filter}_{2,2})\), \(\text{unprocessed}_{\text{subqueries}}(\text{filter}_{2,2})\) and \(\text{unprocessed}_{\text{subqueries}}_2(\text{filter}_{2,2})\). After these steps, we have:
• \( \text{unprocessed_subqueries}(\text{filter}_{2,2}) = \text{subqueries}(\text{filter}_{2,2}) = \{(b, x_2), (y/x_2, z/c), (b, x_2), (y/x_2, z/f), ((b, x_2), (y/x_2, z/h), ((c, x_3), (y/x_3, z/d)), ((f, x_4), (y/x_4, z/g)), ((h, x_5), (y/x_5, z/g)), ((d, x_6), (y/x_6, z/e))\}.

(8) \( E_6 \)

After processing \( \text{unprocessed_subqueries}_2(\text{filter}_{2,2}) \), the algorithm empties this set and transfers \( \{(e, x_8)\} \) through the edge \( E_6 \). This adds a fresh variant \( (e, x_8) \) of the tuple \( \{(e, x_8)\} \) to the sets \( \text{tuples}(\text{input}_p) \), \( \text{unprocessed}(E_1) \) and \( \text{unprocessed}(E_7) \). After these steps, we have:

• \( \text{unprocessed}(E_1) = \text{tuples}(\text{input}_p) = \{(b, x_2), (c, x_3), (f, x_4), (h, x_5), (d, x_6), (g, x_7), (e, x_8)\}, \)
• \( \text{unprocessed}(E_7) = \{(e, x_8)\} \).

(9) \( E_1 - E_2 - E_3 - E_4 \)

After processing \( \text{unprocessed}(E_1) \), the algorithm empties this set and transfers \( \{(b, x_2), (c, x_3), (f, x_4), (h, x_5), (d, x_6), (g, x_7), (e, x_8)\} \) through the edge \( E_1 \). This produces \( \{(b, x_2), \{x/b, y/x_2\}, \{(c, x_3), \{x/c, y/x_3\}, \{(f, x_4), \{x/f, y/x_4\}, \{(h, x_5), \{x/h, y/x_5\}, \{(d, x_6), \{x/d, y/x_6\}, \{(g, x_7), \{x/g, y/x_7\}, \{(e, x_8), \{x/e, y/x_8\}\}\), which is then transferred through the edge \( E_2 \), producing \( \{(b, c), (e), (b, f), (e), (b, h), (e), (c, d), (e), (f, g), (e), (h, g), (e), (d, e)\}\), which in turn is then transferred through the edge \( E_3 \), producing \( \{(b, c), (b, f), (b, h), (c, d), (f, g), (h, g), (d, e)\}\), which in turn is then transferred through the edge \( E_4 \) and added to the empty sets \( \text{tuples}(\text{ans}_p) \), \( \text{unprocessed}(E_5) \) and \( \text{unprocessed}(E_{12}) \).

(10) \( E_5 \)

After processing \( \text{unprocessed}(E_5) \), the algorithm empties this set and transfers \( \{(b, c), (b, f), (b, h), (c, d), (f, g), (h, g), (d, e)\} \) through the edge \( E_5 \) and adds these tuples to the empty set \( \text{unprocessed_tuples}(\text{filter}_{2,2}) \).

(11) \( E_{10} - E_{11} \)

After processing \( \text{unprocessed_tuples}(\text{filter}_{2,2}) \) and \( \text{unprocessed_subqueries}(\text{filter}_{2,2}) \), the algorithm empties these sets and transfers \( \{(b, d), (b, g), (c, e)\} \) through the edge \( E_{10} \). This produces \( \{(b, d), (b, g), (c, e)\} \), which is then transferred through the edge \( E_{11} \) and added to the sets \( \text{tuples}(\text{ans}_p) \), \( \text{unprocessed}(E_5) \) and \( \text{unprocessed}(E_{12}) \). After these steps, we have:

• \( \text{unprocessed}(E_{12}) = \text{tuples}(\text{ans}_p) = \{(b, c), (b, f), (b, h), (c, d), (f, g), (h, g), (d, e), (b, d), (b, g), (c, e)\}, \)
• \( \text{unprocessed}(E_5) = \{(b, d), (b, g), (c, e)\} \).

(12) \( E_5 \)

After processing \( \text{unprocessed}(E_5) \), the algorithm empties this set and transfers \( \{(b, d), (b, g), (c, e)\} \) through the edge \( E_5 \) and adds these tuples to the empty set \( \text{unprocessed_tuples}(\text{filter}_{2,2}) \).

(13) \( E_{10} - E_{11} \)

After processing \( \text{unprocessed_tuples}(\text{filter}_{2,2}) \), the algorithm empties this set and transfers \( \{((b, e), (e))\} \) through the edge \( E_{10} \). This produces \( \{(b, e)\} \), which is then
transferred through the edge $E_{11}$ and added to the sets $\text{tuples}(\text{ans}, p)$, $\text{unprocessed}(E_5)$ and $\text{unprocessed}(E_{12})$. After these steps, we have:

- $\text{unprocessed}(E_{12}) = \text{tuples}(\text{ans}, p) = \{(b, c), (b, f), (b, h), (c, d), (f, g), (h, g), (d, e), (b, d), (b, g), (c, e), (b, e)),$
- $\text{unprocessed}(E_5) = \{(b, e))$.  

$E_{12}$  
After processing $\text{unprocessed}(E_{12})$, the algorithm empties this set and transfers $\{(b, c), (b, f), (b, h), (c, d), (f, g), (h, g), (d, e), (b, d), (b, g), (c, e), (b, e))\}$ through the edge $E_{12}$ and adds these tuples to the empty set $\text{unprocessed\_tuples}(\text{filter}_{3,1})$.

$E_{16} - E_{17}$  
After processing $\text{unprocessed\_tuples}(\text{filter}_{3,1})$ and $\text{unprocessed\_subqueries}(\text{filter}_{3,1})$, the algorithm empties these sets and transfers $\{((c), (e)), ((f), (e)), ((h), (e)), ((d), (e)), ((g), (e)), ((e), (e))\}$ through the edge $E_{16}$. This produces $\{(c), (f), (h), (d), (g), (e))\}$, which is then transferred through the edge $E_{17}$ and added to the empty set $\text{tuples}(\text{ans}, 5)$.

$E_5, E_7, E_{10}$  
The edges $E_5$ and $E_7$ are still active, with $\text{unprocessed}(E_5) = \{(b, e))$ and $\text{unprocessed}(E_7) = \{(e, x_8))$. Firing the edge $E_5$ causes the edge $E_{10}$ to become active, but after that, firing the edges $E_7$ and $E_{10}$ does not create data to be transferred.

At this point, no edges are active (in particular, all the attributes $\text{unprocessed}$, $\text{unprocessed\_subqueries}$, $\text{unprocessed\_subqueries}_{2}$ and $\text{unprocessed\_tuples}$ of the nodes in the net are empty sets). The algorithm terminates and returns the set $\text{tuples}(\text{ans}, 5) = \{(c), (f), (h), (d), (g), (e))\}$.

Table 1 summarizes the effects of the steps of this trace. The numbers in bold font indicate the corresponding steps of the trace, which are listed in Example 3.5.

| Procedure | Purpose: add-subquery($\tilde{t}$, $\tilde{\delta}$, $\Gamma$, $v$) |
|-----------|-------------------------------------------------------------------|
|           | if term-depth($\tilde{t}$) $\leq l$ and term-depth($\tilde{\delta}$) $\leq l$ and no subquery in $\Gamma$ is more general than ($\tilde{t}$, $\tilde{\delta}$) w.r.t. $v$ then |
|           | delete from $\Gamma$ all subqueries less general than ($\tilde{t}$, $\tilde{\delta}$) w.r.t. $v$; |
|           | add ($\tilde{t}$, $\tilde{\delta}$) to $\Gamma$; |

| Procedure | Purpose: add-tuple($\tilde{t}$, $\Gamma$) |
|-----------|-------------------------------------------------------------------|
|           | if $\tilde{t}'$ is not an instance of any tuple from $\Gamma$ then |
|           | delete from $\Gamma$ all tuples that are instances of $\tilde{t}'$; |
|           | add $\tilde{t}'$ to $\Gamma$; |

We present below properties of Algorithm 1. Due to the lack of the space, we refer the reader to Cao (2016) for their proofs.
Soundness: After a run of Algorithm 1 on a query \((P, q(\bar{x}))\) and an extensional instance \(I\), for every intensional predicate \(p\) of \(P\), every tuple \(\bar{t} \in \text{tuples}(ans,p)\) is a correct answer in the sense that \(P \cup I \models \forall \bar{t}(p(\bar{t}))\).

Completeness: After a run of Algorithm 1 (using parameter \(l\)) on a query \((P, q(\bar{x}))\) and an extensional instance \(I\), for every SLD-refutation of \(P \cup I \cup \{\leftarrow q(\bar{x})\}\) that uses the leftmost selection function, does not contain any goal with term-depth greater than \(l\) and has a computed answer \(\theta\) with term-depth not greater than \(l\), there exists \(\bar{s} \in \text{tuples}(ans,q)\) such that \(\bar{x}\theta\) is an instance of a variant of \(\bar{s}\).

\begin{procedure}
\textbf{transfer}(D, u, v)
\begin{algorithmic}[1]
\Function{transfer}{$D, u, v$}
\State \textbf{Global data:} a Horn knowledge base \((P, I)\), a QSQ-net \(N = (V, E, T, C)\) of \(P\), and a term-depth bound \(l\).
\State \textbf{Input:} data \(D\) to transfer through the edge \((u, v) \in E\).
\If{$D = \emptyset$} \State return; \EndIf
\If{$u$ is \textit{input.p}} \State $\Gamma := \emptyset$; \EndIf
\ForEach{$\bar{t} \in D$} \If{$p(\bar{t})$ and \textit{atom}(\(\bar{v}\)) are unifiable by an mgu $\gamma$} \State add-subquery($\Gamma_{T}(\bar{\gamma} \cdot \gamma_{\text{post.vars}(\bar{v})}, \Gamma, \text{succ}(\bar{v}))$); \EndIf
\State transfer($\Gamma, v, \text{succ}(\bar{v}))$; \EndIf
\Else \If{$u$ is \textit{ans.p}} \State \textit{unprocessed.tuples}(\(v\)) := \textit{unprocessed.tuples}(\(v\)) \cup \{D\}; \EndIf
\Else \If{$v$ is \textit{input.p} or \textit{ans.p}} \EndIf
\EndIf
\EndFor
\State \textbf{else} \If{$\bar{t}$ is \textit{input.p} or \textit{ans.p}} \State \textbf{foreach} \(\bar{t} \in D\) \Do \If{$\bar{t}$ is not an instance of any tuple from \textit{tuples}(\(v\))} \State \textbf{foreach} \(\bar{t}'' \in \textit{tuples}(v)\) \Do \If{$\bar{t}''$ is not an instance of \(\bar{t}\)} \State delete \(\bar{t}''\) from \textit{tuples}(\(v\)); \EndIf
\State \EndIf
\Else \If{$v$ is \textit{input.p}} \State add \(\bar{t}\) to \textit{tuples}(\(v\)); \EndIf
\Else \If{$v$ is \textit{filter.i} and \textit{kind}(\(v\)) = \textit{extensional} and \textit{T}(\(v\)) = \textit{false}} \State let \(p = \text{pred}(\bar{v})\) and set \(\Gamma := \emptyset\); \EndIf
\State \textbf{foreach} $(\bar{t}, \delta) \in D$ \Do \If{term-depth(\text{atom}(\(v\))\(\delta\)) \leq l} \State \textbf{foreach} $\bar{t} \in f(\bar{v})$ \Do \If{\text{atom}(\(v\))\(\delta\) is unifiable with a fresh variant of \textit{p}(\(\bar{t}\)) by an mgu \(\gamma\)} \State add-subquery($\Gamma_{T}(\bar{\gamma} \cdot \gamma_{\text{post.vars}(\bar{v})}, \Gamma, \text{succ}(\bar{v}))$); \EndIf
\State transfer($\Gamma, v, \text{succ}(\bar{v}))$; \EndIf
\Else \If{term-depth(\text{atom}(\(v\))\(\delta\)) \leq l} \EndIf
\EndIf
\Else \If{$v$ is \textit{filter.i} and \textit{kind}(\(v\)) = \textit{extensional} and \textit{T}(\(v\)) = \textit{true} or \textit{kind}(\(v\)) = \textit{intensional}} \EndIf
\State \textbf{foreach} $(\bar{t}, \delta) \in D$ \Do \If{term-depth(\text{atom}(\(v\))\(\delta\)) \leq l} \State \textbf{if no subquery in subqueries}(\(v\)) is more general than $(\bar{t}, \delta)$ \Do \State delete from subqueries(\(v\)) all subqueries less general than $(\bar{t}, \delta)$; \EndIf
\Else \If{\textit{kind}(\(v\)) = \textit{intensional}} \State delete from \textit{unprocessed.subqueries}(\(v\)) all subqueries less general than $(\bar{t}, \delta)$; \EndIf
\Else \If{$\text{term-depth}(\text{atom}(\(v\))\(\delta\)) \leq l$} \State delete from \textit{unprocessed.subqueries}(\(v\)) all subqueries less general than $(\bar{t}, \delta)$; \EndIf
\Else \EndIf
\State add $(\bar{t}, \delta)$ to \textit{unprocessed.subqueries}(\(v\)); \EndIf
\Else \textbf{else} \If{$v$ is of the form \textit{post.filter.i}} \State $\Gamma := \{\bar{t} \mid (\bar{t}, e) \in D\}$; \EndIf
\State transfer($\Gamma, v, \text{succ}(\bar{v}))$; \EndIf
\EndIf
\State \EndIf
\EndIf
\EndIf
\EndFor
\EndIf
\EndFor
\EndIf
\EndIf
\EndIf
\EndIf
\EndIf
\EndIf
\EndIf
\EndIf
\EndIf
\EndIf
\EndIf
\EndIf
\EndIf
\EndIf
\EndIf
\EndIf
\EndIf
\EndIf
\EndIf
\EndProcedure
Function active-edge\( (u, v) \)

Global data: a QSQ-net \( N = (V, E, T, C) \).
Input: an edge \((u, v) \in E\).
Output: true if there is data to transfer through the edge \((u, v)\), and false otherwise.

1. if \( u \) is pre_filter\(_i\), or post_filter\(_i\) then return false;
2. else if \( u \) is input_p or ans_p then return unprocessed\((u, v) \neq \emptyset\);
3. else if \( u \) is filter\(_{i,j}\) and kind\((u) = \textit{extensional} \) then
4. \hspace{1em} return \( T(u) = \text{true} \wedge \text{unprocessed\_subqueries}(u) \neq \emptyset\);
5. else if \( u \) is of the form filter\(_{i,j} \) and kind\((u) = \textit{intensional} \) then
6. \hspace{1em} let \( p = \text{pred}(u)\);
7. \hspace{1em} if \( v = \text{input}_p \) then return unprocessed\_subqueries\(_2\)(u) \( \neq \emptyset\);
8. \hspace{1em} else return unprocessed\_subqueries\(_2\)(u) \( \neq \emptyset \) or unprocessed\_tuples\(_2\)(u) \( \neq \emptyset\);

Procedure fire\((u, v)\)

Global data: a Horn knowledge base \((P, I)\), a QSQ-net \( N = (V, E, T, C) \) of \( P\), and a term-depth bound \( l\).
Input: an edge \((u, v) \in E\) such that active-edge\((u, v)\) holds.

1. if \( u \) is input_p or ans_p then
2. \hspace{1em} transfer\((\text{unprocessed}(u, v), u, v)\);
3. \hspace{1em} \text{unprocessed}(u, v) := \emptyset;
4. \hspace{1em} else if \( u \) is filter\(_{i,j}\) and kind\((u) = \textit{extensional} \) and \( T(u) = \text{true} \) then
5. \hspace{2em} let \( p = \text{pred}(u)\) and set \( \Gamma := \emptyset\);
6. \hspace{2em} foreach \((l, \delta) \in \text{unprocessed\_subqueries}(u)\) do
7. \hspace{3em} foreach \( l' \subseteq I(p)\) do
8. \hspace{4em} if \( \text{atom}(u)\delta \) is unifiable with a fresh variant of \( p(l') \) by an mgu \( \gamma \) then
9. \hspace{5em} \text{add-subquery}(l\gamma, (\delta\gamma)|_{\text{post\_vars}(u)}, \Gamma, v);
10. \text{unprocessed\_subqueries}(u) := \emptyset;
11. \text{transfer}(\Gamma, u, v);
12. \hspace{1em} else if \( u \) is filter\(_{i,j}\) and kind\((u) = \textit{intensional} \) then
13. \hspace{2em} let \( p = \text{pred}(u)\) and set \( \Gamma := \emptyset\);
14. \hspace{2em} if \( v = \text{input}_p \) then
15. \hspace{3em} foreach \((l, \delta) \in \text{unprocessed\_subqueries\(_2\)}(u)\) do
16. \hspace{4em} let \( p(l') = \text{atom}(u)\delta\), \text{add-tuple}(l', \Gamma);
17. \hspace{4em} \text{unprocessed\_subqueries\(_2\)}(u) := \emptyset;
18. \hspace{3em} else
19. \hspace{4em} foreach \((l, \delta) \in \text{unprocessed\_subqueries\(_2\)}(u)\) do
20. \hspace{5em} foreach \( l' \subseteq \text{tuples\(_2\)}(\text{ans\_p})\) do
21. \hspace{6em} if \( \text{atom}(u)\delta \) is unifiable with a fresh variant of \( p(l') \) by an mgu \( \gamma \) then
22. \hspace{7em} \text{add-subquery}(l\gamma, (\delta\gamma)|_{\text{post\_vars}(u)}, \Gamma, v);
23. \hspace{5em} \text{unprocessed\_subqueries}(u) := \emptyset;
24. \hspace{3em} \text{foreach } l \in \text{unprocessed\_tuples\(_2\)}(u) do
25. \hspace{4em} \text{foreach } (l', \delta) \in \text{subqueries\(_2\)}(u) do
26. \hspace{5em} if \( \text{atom}(u)\delta \) is unifiable with a fresh variant of \( p(l) \) by an mgu \( \gamma \) then
27. \hspace{6em} \text{add-subquery}(l\gamma, (\delta\gamma)|_{\text{post\_vars}(u)}, \Gamma, v);
28. \hspace{5em} \text{unprocessed\_tuples\(_2\)}(u) := \emptyset;
29. \hspace{3em} \text{transfer}(\Gamma, u, v);
Together with the completeness of SLD-resolution (Clark, 1979), this property makes a relationship between correct answers for \( P \cup I \cup \{ \leftarrow q(x) \} \) and the answers computed by Algorithm 1 for the query \((P, q(x))\) on the extensional instance \(I\).

For queries and extensional instances without function symbols, we take term-depth bound \( l=0 \) and obtain the following completeness result, which immediately follows from the above property:

After a run of Algorithm 1 using \( l=0 \) on a query \((P, q(x))\) and an extensional instance \(I\) that do not contain function symbols, for every computed answer \( \theta \) of an SLD-refutation of \( P \cup I \cup \{ \leftarrow q(x) \} \) that uses the leftmost selection function, there exists \( \bar{t} \) such that \( \bar{t} \in \text{tuples}(\text{ans},q) \).

**Data complexity:** For a fixed query and a fixed bound \( l \) on term-depth, Algorithm 1 runs in polynomial time in the size of the extensional instance.

### 4. Preliminary experiments

In Cao (2016), we presented three control strategies DAR (Disk Access Reduction), DFS (Depth-First Strategy), IDFS (Improved Depth-First Strategy) and implemented QSQN together with these strategies to obtain the corresponding evaluation methods QSQN-DAR, QSQN-DFS and QSQN-IDFS. The intention of DAR is to reduce the number of accesses to the secondary storage. Because our current implementation of the DAR control strategy is not advanced enough and the implemented QSQN-DAR method is not more efficient than the implemented QSQN-IDFS method, for comparison with the Magic-Sets and QSQR methods we used QSQN-IDFS. We compared the QSQN-IDFS, Magic-Sets and QSQR evaluation methods with respect to:

- the number of read/write operations on relations,
- the maximum number of tuples/subqueries kept in the computer memory,
- the number of accesses to the secondary storage when the memory is limited.

Our experiments consider different kinds of logic programs, including non-recursive, tail recursive, non-tail recursive as well as logic programs with or without function symbols. We used typical examples from well-known articles related to deductive databases. We also provided new examples. Due to the lack of the space, we refer the reader to Cao (2016) for more details on control strategies, experimental settings, test cases and experimental results. We report below only one test.

For the Datalog database and the query given in Example 1.1 with \( m=n=100 \), the QSQN-IDFS method reads data from relations 361 times, writes data to relations 154 times and keeps maximally in the memory 204 tuples, while the corresponding numbers of the Magic-Sets method are 721, 301 and 10,105, respectively, and the corresponding numbers of the QSQR method are 410, 358 and 356, respectively. When the number of tuples kept in the memory is restricted to 5052 (about 50% of the mentioned number 10,105), the Magic-Sets method needs to write relations to the secondary storage 29 times and read them from the secondary storage 60 times (using a certain unloading
strategy), while the QSQN-IDFS method reads data from the secondary storage only once and does not need to write data to the secondary storage. When the number of tuples kept in the memory is restricted to 2021 (i.e. 20% of the mentioned number 10,105), the Magic-Sets method fails to evaluate the query, while the QSQN-IDFS method does not. This test shows that, when the positive logic program defining intensional predicates is specified using the Prolog programming style, the QSQN-IDFS and QSQR methods (which use depth-first search) are usually more efficient than the Magic-Sets method (which uses the breadth-first search).

As can be seen in Cao (2016, Tables 6.1-6.3), the QSQR method is often worse than the QSQN-IDFS and Magic-Sets methods w.r.t. the number of accesses to the secondary storage. As discussed by Madalińska-Bugaj and Nguyen (2012), QSQR uses iterative deepening search and clears input relations at the beginning of each iteration of the main loop, thus it allows redundant recomputations. In addition, the formulation of QSQR in Madalińska-Bugaj and Nguyen (2012) is at a logical level and uses the same relation for the whole sequence of supplements. This requires more relation loading/unloading when the recursive sequence of depth is high and no more memory is available.

5. Conclusions

We have provided the first framework for developing algorithms for evaluating queries to Horn knowledge bases with the properties that: the approach is goal-directed; each sub-query is processed only once and each supplement tuple, if desired, is transferred only once; operations are done set-at-a-time; and any control strategy can be used. Our framework is an adaptation and a generalization of the QSQ approach of Datalog for Horn knowledge bases. One of the key differences is that we do not use adornments and annotations, but use substitutions instead. This is natural for the case with function symbols and without the range-restrictedness condition. When restricting to Datalog queries, it groups operations on the same relation together regardless of adornments and allows to reduce the number of accesses to the secondary storage although ‘joins’ would be more complicated.

Our framework forms a generic evaluation method called QSQN. This method is designed so that the query processing is divided into appropriate steps which can be delayed to maximize adjustability and allow various control strategies. In comparison with the most well-known evaluation methods, the generic QSQN evaluation method does not do redundant recomputations as the QSQR evaluation method and is more adjustable and thus has essential advantages over the Magic-Sets evaluation method. The QSQN method is sound and complete, and has polynomial time data complexity when the term-depth bound is fixed. Notice the significance of this: it states that one can develop and use any control strategy for QSQN and the resulting evaluation method is always guaranteed to be sound and complete. Our proofs (Cao, 2016) are important in the context that, without proofs, the methods proposed in Vieille (1986), Abiteboul et al. (1995) and Madalińska-Bugaj and Nguyen (2008) were wrongly claimed to be complete.

Our experiments presented in Cao (2016) show that the QSQN-IDFS evaluation method is more efficient than the QSQR evaluation method and as competitive as the Magic-Sets evaluation method. In the case when the order of program clauses and the order of atoms in the bodies of program clauses are essential as in Prolog programming, the QSQN-IDFS
evaluation method usually outperforms the Magic-Sets method. As QSQN-IDFS is just an instance of the generic QSQN evaluation method, we conclude that this generic method is useful.

QSQ-nets are a more intuitive representation than the description of the QSQ approach of Datalog given in Abiteboul et al. (1995). Our notion of QSQ-net makes a connection to flow networks and is intuitive for developing efficient evaluation algorithms. For example, we have incorporated tail-recursion elimination into QSQ-nets (Cao, 2016; Cao & Nguyen, 2015) to obtain the QSQN-TRE method, as well as stratified negation into QSQ-nets (Cao, 2016) to obtain the QSQN-STR method for evaluating queries to stratified knowledge bases.

Notes
1. By ‘adjustability’ we mean easiness in adopting advanced control strategies.
2. when \( T(v) = \text{false} \) for all nodes \( v \) of the form \( \text{filter}_{i,j} \) with \( \text{kind}(v) = \text{extensional} \)

Disclosure statement
No potential conflict of interest was reported by the authors.

Funding
This work was supported by the Polish National Science Centre (NCN) [grant number 2011/02/A/HS1/00395].

Notes on contributors

Son Thanh Cao is a lecturer at Faculty of Information Technology, Vinh University. He obtained the Ph.D. degree in Computer Science in 2016 from the University of Warsaw. His research interests include logic programming, deductive databases and semantic web.

Linh Anh Nguyen is an associate professor of Computer Science at the Institute of Informatics, University of Warsaw. Since 2014 he has been cooperating with Faculty of Information Technology, Ton Duc Thang University in doing research. He has published more than 90 papers in international scientific journals and proceedings of international conferences/workshops.

References
Abiteboul, S., Hull, R., & Vianu, V. (1995). *Foundations of databases*. Boston, MA: Addison Wesley.
Bancilhon, F., Maier, D., Sagiv, Y., & Ullman, J. (1986). *Magic sets and other strange ways to implement logic programs*. Paper presented at the proceedings of PODS’1986, Cambridge, Massachusetts, pp. 1–15. ACM.
Beeri, C., & Ramakrishnan, R. (1991). On the power of magic. *Journal of Logic Programming*, 10, 255–299.
Cao, S. (2016). *Methods for evaluating queries to Horn knowledge bases in first-order logic* (Ph.D. dissertation). University of Warsaw. Retrieved from http://mimuw.edu.pl/~sonct/stc-thesis.pdf
Cao, S., & Nguyen, L. (2015). *An empirical approach to query-subquery nets with tail-recursion elimination*. Paper presented at proceedings II of ADBIS’2014, advances in intelligent systems and computing, Vol. 312, Ohrid, Macedonia, pp. 109–120. Springer.
Cao, S. T., Nguyen, L. A., & Szalas, A. (2014). WORL: A nonmonotonic rule language for the semantic web. *Vietnam Journal of Computer Science*, 1(1), 57–69.

Clark, K. (1979). *Predicate logic as a computational formalism* (Research report DOC 79/59). Department of Computing, Imperial College.

Eiter, T., Ianni, G., Lukasiewicz, T., & Schindlauer, R. (2011). Well-founded semantics for description logic programs in the semantic web. *ACM Transactions on Computational Logic*, 12(2), 1–41.

Huang, S., Green, T., & Loo, B. (2011). *Datalog and emerging applications: An interactive tutorial*. Paper presented at proceedings of SIGMOD’2011, Athens, Greece, pp. 1213–1216. ACM.

Lloyd, J. (1987). *Foundations of logic programming* (2nd ed.). Berlin: Springer.

Madalińska-Bugaj, E., & Nguyen, L. (2008). Generalizing the QSQR evaluation method for Horn knowledge bases. In N.-T. Nguyen & R. Katarzyniak (Eds.), *New challenges in applied intelligence technologies* (Vol. 134, pp. 145–154). Berlin: Springer.

Madalińska-Bugaj, E., & Nguyen, L. (2012). A generalized QSQR evaluation method for Horn knowledge bases. *ACM Transactions on Computational Logic*, 13(4), 32:1–32:28.

Nguyen, L., & Cao, S. (2012). *Query-subquery nets*. Paper presented at proceedings of ICCCI’2012, Ho Chi Minh City, Vietnam, pp. 239–248. Springer-Verlag.

Ramakrishnan, R., & Ullman, J.D. (1995). A survey of deductive database systems. *Journal of Logic Programming*, 23(2), 125–149.

Rohmer, J., Lescouer, R., & Kerisit, J.-M. (1986). The Alexander method – a technique for the processing of recursive axioms in deductive databases. *New Generation Computing*, 4(3), 273–285.

Ruckhaus, E., Ruiz, E., & Vidal, M.-E. (2008). Query evaluation and optimization in the semantic web. *Theory and Practice Logic Programming*, 8(3), 393–409.

Vieille, L. (1986). *Recursive axioms in deductive databases: the query/subquery approach*. Paper presented at proceedings of expert database systems, Charleston, South Carolina, pp. 253–267. Benjamin Cummings.

Vieille, L. (1989). Recursive query processing: The power of logic. *Theoretical Computer Science*, 69(1), 1–53.

Zhou, N.-F., & Sato, T. (2003). *Efficient fixpoint computation in linear tabling*. Paper presented at proceedings of PPDP’2003, Uppsala, Sweden, pp. 275–283. ACM.