Counting and Topological Order

Chen Yangjun* (陈阳军)

Technical Institute of Changsha, Changsha 410073

Received September 6, 1996; revised January 12, 1997.

Abstract

The counting method is a simple and efficient method for processing linear recursive datalog queries. Its time complexity is bounded by $O(n \cdot e)$, where n and e denote the numbers of nodes and edges, respectively, in the graph representing the input relations. In this paper, the concepts of *heritage appearance function* and *heritage selection function* are introduced, and an evaluation algorithm based on the computation of such functions in topological order is developed. This new algorithm requires only linear time in the case of non-cyclic data.

Keywords: Recursive query, counting, topological order, heritage function.

1 Introduction

The evaluation of recursive queries expressed as sets of Horn Clauses over a database has been studied in the last several years. An important matter of research in such systems is the efficient evaluation of recursive queries (queries against a recursive program). Various strategies for processing recursive queries have been proposed[1−18]. In this paper, we confine ourselves to the counting method[4,17] for linear recursion and try to improve its performance in the case of non-cyclic data. This method seeks to perform a compile-time transformation of the database, based on the query form, into an equivalent form which enables a bottom-up computation to focus on relevant tuples. As with the magic set method[3,4], the transformed programs consist of two rule sets: *counting rules* and *modified rules*. Thus, the computation can be done in a two-phase approach. In the first phase, we produce a counting set by evaluating the counting rules. In the second phase, we produce all answers by evaluating modified rules with the counting set being used to restrict the computation. According to the graphic analysis performed in [15,16], the worst-case time complexity of this method is $O(n \cdot e)$, better than magic sets. Here, we introduce two new concepts: *heritage appearance function* and *heritage selection function*, and transform many algebraic operations into simple computations of such functions (i.e., some Boolean operations) in topological order. In this way, high efficiency can be obtained not only due to the simplicity of Boolean operations, but also due to

*Current address: Dept. of Computer Science, Technical University Chemnitz-Zwickau, 09107 Chemnitz, Germany

the elimination of much redundancy by using binary sequence (string of 1's and 0's) property.

The paper is organized as follows. In the next section, we introduce the concepts of linear recursive queries and query graph and describe the counting method in a graphical formalism. In Section 3, we define the heritage appearance function and heritage selection function, and present an evaluation algorithm which reduces the costs of both the first and second phases of the counting method to $O(e)$. In Sections 4 and 5, we prove the correctness of the refined algorithm and compare the time complexity of the refined algorithm with other well-known strategies. Section 6 is a short conclusion.

2 Basic Concepts

In this section, we define some concepts and terminologies which are necessary for introducing our efficient method.

2.1 Some Terminologies from Graph Algorithm Theory

In this paper, we use the term graph to refer to the directed graph, since we don't discuss the undirected one at all. We assume that a graph G is specified as follows: for each node v_i in the graph, there is a set of *successors* $\mathrm{adj}(v_i) = \{ v_j \mid (v_i, v_j)$ is an edge of $G\}$. Without loss of generality, we assume that G has no self-loops, i.e., for all nodes v_i, $v_i \notin \mathrm{adj}(v_i)$. For an edge (v_i, v_j), node v_i is called the *source* or *tail* and node v_j is called the *destination* or *head* of the edge. We denote the transitive closure of a graph G by G^*. The successors of v_i in G^* are the *descendents* of v_i in G. The *strongly connected component* (or *strong component*) of node v_i is defined as $V_i = \{v_i\} \cup \{v_j \mid (v_i, v_j) \in G^*$ and $(v_j, v_i) \in G^*\}$. The component V_i is not trivial if $V_i \neq \{v_i\}$. As will be seen, our algorithm is based on depth-first traversal of graphs, so we review now some relevant definitions. Depth-first traversal induces a spanning tree on a graph based on the order in which nodes are visited. If we assume that the main routine in depth-first traversal is visit(v_i) for a node i, then there is an edge (v_i, v_j) in the spanning tree, if there is a call to visit(v_j) during the execution of the call visit(v_i). An edge (v_i, v_j) in the graph G is called a *tree edge*, if it belongs to the spanning tree. An edge (v_i, v_j) in the graph G but not in the spanning tree is called a *forward edge*, a *back edge*, or a *cross edge*, if in the spanning tree, v_j is a descendant of v_i, v_j is an ancestor of v_i, or v_j is not related to v_i with an ancestor-descendant relationship, respectively. For every strong component, its node r on which visit(r) is first called is the root of the strong component.

2.2 Query Graph and Counting

We assume that the reader is familiar with the deductive database terminology. In our representation, we will use the concept of query graphs and will associate a

directed graph to a query with respect to a program of the form:

$$\begin{array}{lllll} (1) & rp(x,y) & :- & flat(x,y) & \\ (2) & rp(x,y) & :- & up(x,z), & rp(z,w), \quad down(w,y). \end{array}$$

A query graph basically consists of three parts: up-part (UP), flat-part (FP), and down-part (DP). The UP is that relation part which is reachable from the constants in the query. The FP is that part which can be reached using the non-recursive rule, and the DP can be reached using the recursive rule. For example, if the *up*, *flat*, and *down* predicates in the above program are defined as:

$$\begin{array}{ll} up = & \{ (a_1,a_2), (a_1,a_3), (a_1,a_4), (a_1,a_5), (a_2,a_3), \\ & (a_2,a_4), (a_2,a_5), (a_3,a_4), (a_3,a_5), (a_4,a_5)\} \\ flat = & \{ (a_5,b_5)\} \\ down = & \{ (b_5,b_4), (b_4,b_3), (b_3,b_2), (b_2,b_1) \} \end{array}$$

then the query graph representing the query $? - rp(a_1, y)$ will be as shown in Fig.1, where the edges going up represent tuples in UP, the broken edges represent tuples in FP, and the edges going down represent tuples in DP.

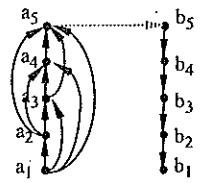

Fig.1. Graph representing input relations.

We assume only linear rules that can be reduced to a form like the above example and will use the notation $G_u = \langle N_u, E_u \rangle$, $G_f = \langle N_f, E_f \rangle$ and $G_d = \langle N_d, E_d \rangle$ to represent the subgraphs induced from UP, FP, and DP, respectively. In Fig.1, node $a_1 \in N_u$ represents the constant in the query $rp(a_1, y)$ and we call such a node the *source node*. Given a subgraph G (i.e., G can be G_u, G_f, or G_d) and a subset X of G, we denote by $\mathrm{adj}G(X)$ ($\mathrm{adj}^{-1}G(X)$) the set of all nodes v_j such that the edge (v_i, v_j) ((v_j, v_i)) is in G and v_i is in X. In other words, $\mathrm{adj}G(X)$ is the set of all nodes that are adjacent to some node in X, whereas $\mathrm{adj}^{-1}G(X)$ is the set of all nodes having at least one adjacent node in X. It is easy to see that these two sets of nodes can be computed using the rules stated above, which can be also expressed as relation-algebra expressions.

As an example, consider the above query graph. Say,

$$X = \{a_4, a_5\}.$$

Then

$$\mathrm{adj}G_u(X) = \{a_5\},$$
$$\mathrm{adj}^{-1}G_u(X) = \{a_1, a_2, a_3, a_4\},$$

and

$$\mathrm{adj}G_f(X) = \{b_5\}.$$

Since we are going to refer frequently to the counting method, we feel that it will be convenient to describe it in a graphical formalism[16]. Fig.2 represents the basic idea of the counting method (essentially, it corresponds to the implementation of the supplementary counting method[4].) It works as follows. Let $U_i (i \geq 0)$ contain

all nodes v in G_u that have distance i from the source node s. In the first phase, the method computes U_i. (Notice that, in general, such sets are not disjoint and are called the counting set. This process corresponds to the evaluation of the counting rules.) Suppose that U_g contains the nodes with the greatest distance (thus g) from s. In the second phase, we start computing the set D_g of all nodes in G_f that are adjacent to some nodes of U_g in G_f. Then we compute D_{g-1} as the set of all nodes in G_d that are adjacent to some node of U_{g-1} in G_f and that are adjacent to some node of D_g in G_d. We continue until we compute D_0, which contains all the answers (answer nodes) of the query. (This process corresponds to the evaluation of the modified rules.) If the graph G_u is cyclic, this version of the counting method is not safe.

$U_0 := \{s\}$; $i := 0$;
while $U_i \neq \emptyset$ **do**
begin
    $U_{i+1} := \text{adj}G_u(U_i)$; $i := i + 1$;
end
$D_{i-1} := \text{adj}G_f(U_{i-1})$;
for $j := i$ - 1 **downto** 1 **do**
    $D_{j-1} := \text{adj}G_f(U_{j-1}) \cup \text{adj}G_d(D_j)$;
Answer $:= D_0$.

Fig.2. Counting method.

From the description of the algorithm, we can see that in the case of acyclic data the first loop can be performed $O(|N_u|)$ times and every iteration has a cost of $O(|E_u|)$. Therefore, the total cost of the first phase is $O(|N_u| \cdot |E_u|)$. Similarly, the second phase has a total cost of $O(|N_u| \cdot |E_d|)$. Hence, the cost of the counting method for acyclic queries is $O(n \cdot e)$, where n and e denote the numbers of nodes and edges, respectively, in the graph representing the input relations.

The purpose of this paper is to reduce the time complexity of both first and second phases using a new method based on the notions of heritage appearance function and heritage selection function. In the following, we discuss these two functions and the corresponding computation methods in Section 3.

3 Optimal Algorithm for Non-Cyclic Data

In this section, we describe our efficient algorithm for non-cyclic data. Similar to the counting method, our algorithm works in a two-phase manner. In the first phase, we compute the appearance function sequence for each node of G_u in $O(e)$ time. Then, in the second phase, we extract the answers in terms of such appearance function sequence in some way. As will be seen later, the time requirement of the second phase is also bounded by $O(e)$.

3.1 The First Phase of the Algorithm

Here we describe the first phase of the counting method. First, we present the concept of appearance function which was introduced in [1] to describe the possible distances of a node v in G_u from a source node s.

Definition 1. *The appearance function $A_{v,s}(i)$ of a node v with respect to a source node s is a binary-valued function.*

For any integer $i \geq 0$:

$$A_{v,s}(i) = \begin{cases} 1, & \text{if there exists a path from } s \text{ to } v \text{ of length } i, \\ 0, & \text{otherwise.} \end{cases}$$

Then the sequence $A_{v,s} = A_{v,s}(0), A_{v,s}(1), A_{v,s}(2), \ldots$ corresponds to the different appearances of v with different distances from s. For example, for the query graph shown in Fig.1, we have

A_{a_5,a_1}= 01111, A_{a_4,a_1}= 01110, A_{a_3,a_1}= 01100, A_{a_2,a_1}= 01000, A_{a_1,a_1}= 10000.

An observation shows that in the case of non-cyclic data, if the height of G_u is h (we define the height of an acyclic graph to be the number of the nodes on the longest path in the graph), then the length of each $A_{v,s}$ is bounded by h and $h \leq |N_u|$. Thus, the number of 1's appearing in all appearance function sequences with respect to G_u (denoted as N_{1-bit}) is bounded by $h \cdot |N_u|$. If we can find a method to generate each "1" only once, the time complexity of the first phase will be reduced to $O(h \cdot |N_u|)$. In fact, we can find an algorithm which can generate all such sequences in linear time.

In order to generate each "1" only once, we introduce another concept, so called heritage appearance function for the nodes of G_u, which can be defined as follows.

Definition 2. *The heritage appearance function $HA^V_{v',s}(i)$ of a node v' with respect to a node v (and a source node s) is a binary-valued function.*

$$HA^v_{v',s}(i) = \begin{cases} 1, & \text{if } A_{v,s}(i-1) = 1 \text{ and } (v,v') \in E_u, \\ 0, & \text{otherwise.} \end{cases}$$

Then $HA^v_{v',s} = HA^v_{v',s}(1), HA^v_{v',s}(2), \ldots$ can be easily computed by shifting $A_{v,s}$ right 1 bit and filling the emptied position with 0. Further, we will use $HA_{v,s}$ to denote the sequence

$$HA^{v_1}_{v,s} \cup HA^{v_2}_{v,s} \cup \cdots \cup HA^{v_j}_{v,s},$$

where $v_1, v_2, \ldots, v_j$ are the direct precedents of v in G_u and $\cup$ is used to denote OR operation on the corresponding positions of two binary sequences respectively.

This definition hints an efficient computation method, in which all appearance function sequences can be generated in linear time. First, we have the following proposition.

Proposition 1. *Let s be the source node of a G_u. Then, for all nodes v of G_u except s, we have*

$$A_{v,s} = HA_{v,s} = HA_{v,s}^{v_1} \cup HA_{v,s}^{v_2} \cup \cdots \cup HA_{v,s}^{v_j},$$

where v_1, v_2, ..., v_j *are the direct precedents of* v *in* G_u.

Proof. It follows directly from the definitions of $A_{v,s}$ and $HA_{v,s}$.

Based on the above proposition, we propose an algorithm which works in a two-step manner and can generate all appearance function sequences for the nodes of G_u in linear time. In the first step, we produce a directed graph corresponding to G_u. In the second step, we first find a topological order (for G_u) with the property that all precedents of a node n_i are before n_i in the order. Then we compute the appearance function sequence for each node in such an order as follows. At the beginning, the appearance function sequence of the source node $A_{s,s}$ is initialized to "100 ... 0". Then, the second node in the topological order can be obtained by shifting $A_{s,s}$ right 1 bit, filling the emptied position with 0. The ith node can be computed using the equation given in Proposition 1. Since the appearance function sequences of all direct precedents of a node have already been generated before the node is encountered, we can evaluate the appearance function sequence for each node of G_u in a topological order without difficulty. More importantly, in this way, each $A_{v,s}$ can be generated by only performing d_v shifting operations and d_v logical OR operations, where d_v represents the indegree of v. That is, to compute the appearance function sequence for a node v, we need only to shift each v_i ($i = 1$, ..., d_v) right 1 bit and then to add them together, where v_i ($i = 1$, ..., d_v) are the direct precedents of v. This process can be summarized as follows. (In the following algorithm, the operation $ShiftRight(A_{v,s})$ shifts $A_{v,s}$ right 1 bit, filling the emptied position with 0.)

procedure *first-phase*
begin
(1) generate G_u;
(2) find a topological order for G_u;
let $\{ v_1, v_2, \ldots, v_n \}$ be the corresponding topological order
and $v_1 = s$ be the source node;
$A_{v_1,s}$:= "10... 0";
for $i = 2$ to n **do**
let $v_{i_1}, \ldots, v_{i_j} \in \{v_1, v_2, \ldots, v_{i-1}\}$ be the direct precedents of v_i
for $j = 1$ to l **do**
$HA_{v_i,s}^{v_{i_j}} := ShiftRight(A_{v_{i_j},s})$;
$A_{v_i,s} := HA_{v_i,s}^{v_{i_1}} \cup \ldots \cup HA_{v_i,s}^{v_{i_l}}$;
end

Obviously, the time complexity of the above algorithm is $O(|E_u|)$. On the one hand, G_u can be generated in $O(|E_u|)$ time and the topological order for it can also be found in $O(|E_u|)$ (see [19]). On the other hand, the cost of generating an $A_{v,s}$ is bounded by $O(d_v)$ and then the total cost of generating all appearance function sequences is $\sum_{v \in \{v_1, \cdots, v_n\}} O(d_v) = O(|E_u|)$. In practice, the entire time spent in doing the shifting operations and the OR operations for a node can be taken to be $O(1)$. Therefore, the total cost of generating all appearance function sequences should be $O(|N_u|)$.

3.2 The Second Phase of the Algorithm

In terms of the appearance function sequences, the answers can be extracted by determining the distances for the nodes of G_d (the graph induced from the down-part DP). First, we define the heritage selection functions. Then we discuss how such functions can be used to find the correct answers in linear time.

In [1], a concept, *selection function*, is introduced to provide a termination condition for the counting method in the case of cyclic data. Based on this notion, we propose a new concept, *heritage selection function*, for each node $e \in G_d$ to select those nodes which belong to the answer set and are reachable from e. In this way, any repeated access to an edge in G_d can be replaced by a simple Boolean computation. In what follows, we first give the definition of the selection function. Then we define the notion of the heritage selection function and derive a refined algorithm based on the computation of the function for each node of G_d in a topological order.

In terms of the definition of the appearance functions, we know that if $v \in N_u$, $w \in N_d$, s is the source node, and $(v, e) \in E_f$, then w is in the answer set iff

$$A_{v,s}(i) = A_{w,e}(i) = 1 \quad \text{for some } i \tag{1}$$

Since the flat relation is many-to-many in general, we define the following function to represent the union of all the appearance functions of those nodes $v \in N_u$ such that $(v, e) \in E_f$.

Definition 3. *The selection function $S_e(i)$ of a node e in G_d is a binary-valued function. For any integer $i \geq 0$:*

$$S_e(i) = \begin{cases} 1, & \text{if } A_{v,s}(i)=1 \text{ for some } v \in N_u \text{ and } (v, e) \in E_f, \\ 0, & \text{otherwise.} \end{cases}$$

Then the sequence $S_e = S_e(0), S_e(1), S_e(2), \ldots$ indicates the distances from e with the property that if a node in G_d possesses one of such distances from e, it should be in the answer set. This property is called the *selection* property. For example, if we have (u, e), $(v, e) \in E_f$ and $A_{u,s} = 01010$, $A_{v,s} = 00101$, where s represents the source node in G_u, then $S_e = A_{u,s} \cup A_{v,s} = 01111$, which indicates that the nodes (in G_d) with the distance one, two, three or four from e are answer nodes.

From the definition of the selection function, one can rewrite condition (1) as follows: A node $w \in N_d$ is in the answer set iff for some $e \in N_f \cap N_d$,

$$S_e(i) = A_{w,e}(i) = 1 \quad \text{for some } i \tag{2}$$

Now consider a sequence $S_{e'}$ obtained by shifting S_e left one bit and filling the emptied position with 0. Such a sequence has an important meaning that if there exists a node v such that $(e, v) \in E_d$, then $S_{e'}$ indicates the distances of some nodes from v which belong to the answer set if they are in G_d. This property, together with the fact that the answer node can be checked dynamically in terms of selection

function sequences, provides a possibility to optimize the evaluation of the second phase.

Definition 4. *The heritage selection function $H^e_{e'}(i)$ of a node e' (in G_d) with respect to a node e is a binary-valued function. For any integer $i \geq 0$, if e has no precedents in G_d, then*

$$H^e_{e'}(i) = \begin{cases} 1, & \text{if } S_e(i+1) = 1, \\ 0, & \text{otherwise.} \end{cases}$$

In other cases,

$$H^e_{e'}(i) = \begin{cases} 1, & \text{if } S_e(i+1) = 1 \text{ or } H^{e_1}_e(i+1) = 1 \text{ for some } e_1 \text{ such that } (e_1, e) \in N_d, \\ 0, & \text{otherwise.} \end{cases}$$

Then the sequence $H^e_{e'} = H^e_{e'}(0), H^e_{e'}(1), H^e_{e'}(2), \ldots$ indicates the distances from e', which have the selection property.

We will use H_e to denote the sequence $H^{e_1}_e \cup H^{e_2}_e \cup \ldots \cup H^{e_l}_e$, where e_1, e_2, ..., e_l are the direct precedents of e in G_d. Intuitively, H_e can be used to propagate control information to extract answer subsets correctly.

Definition 5. *The generalized selection function $GH_e(i)$ is a binary-valued function. For any integer $i \geq 0$:*

$$GH_e(i) = \begin{cases} 1, & \text{if } S_e(i) = 1 \text{ or } H^{e_1}_e(i) = 1 \text{ for some } e_1 \text{ such that } (e_1, e) \in N_d, \\ 0, & \text{otherwise.} \end{cases}$$

In terms of the above definitions, we have the following proposition.

Proposition 2. *If (e, e') is an edge in E_d and $GH_e(k) = 1$ for some $k \geq 1$, then we have*

$$GH_{e'}(k-1) = 1$$

Proof. It follows directly from the definitions of $S_e(k)$ and $GH_e(k)$.

From the above discussion, we know that GH_e indicates the distances of all those nodes from e, which belong to the answer set if they are in G_d. For each $e \in N_f \cap N_d$, we can produce its S_e by evaluating the (modified) non-recursive rule with the corresponding appearance function sequences being propagated (see Definition 3). Obviously, this can be done in linear time (see [1] for a more detailed description). Similar to the treatment of the first phase, we then generate a graph corresponding to G_d and compute all generalized selection function sequences in a topological order, thereby generating the answers dynamically in terms of the generalized selection function sequences computed so far. In the following algorithm, the operation $ShiftLeft(GH_e)$ returns a binary sequence obtained by shifting GH_e left 1 bit and filling the emptied position with 0.

procedure *second-phase*
begin

(1) generate S_e for each $e \in N_f \cap N_d$;
(2) generate G_d;
(3) find a topological order for G_d;
let $\{ e_1, e_2, \ldots, e_n \}$ be the corresponding topological order
for $i = 1$ to n **do**
let $e_{i_1}, \ldots, e_{i_l} \in \{e_1, e_2, \ldots, e_{i-1}\}$ be the direct precedents of e_i
for $j = 1$ to l **do**
$H_j := ShiftLeft(GH_{e_{i_j}})$;
$GH_{e_i} := S_{e_i} \cup H_1 \cup \ldots \cup H_j$;
* **if** $GH_{e_i}(0) = 1$ **then**
insert e_i into the answer set;
end

Note the statement marked with $*$. By executing this statement, each node is checked whether it belongs to the answer set in terms of its generalized selection function sequence. That is, if the 0th position of the generalized selection function sequence of a node e is "1", then e should be in the answer set (and then e is called an answer node). It is because in this case either (s, e) is in E_f, where s is the source node, or there is a path of length $2i+1, i > 0$, from the source node s to e such that the first i edges are in E_u, the $(i+1)$th edge is in E_f, and the last i edges are in E_d. (see Proposition 4 given in the next section.)

Example 1. Continue with our running program. Suppose that the facts in the database can be represented as a graph shown in Fig.3.

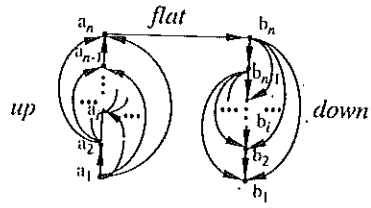

Fig.3. Graph representing input relations.

Given the query $?-rp(a_1, y)$, the algorithm *first-phase* first generates G_u, which corresponds to the *up* part of the graph shown in Fig.3. Then the topological order for it can be found in linear time:

$$a_1 \longrightarrow a_2 \longrightarrow \cdots \longrightarrow a_{n-1} \longrightarrow a_n.$$

In this order, the algorithm will produce the following appearance function sequences:

A_{a_1,a_1} = "10 ... 00",
$A_{a_2,a_1} = ShiftRight(A_{a_1,a_1})$ = "010... 00",
$A_{a_3,a_1} = ShiftRight(A_{a_1,a_1}) \cup ShiftRight(A_{a_2,a_1})$ = "0110...00",
...
$A_{a_n,a_1} = ShiftRight(A_{a_1,a_1}) \cup ShiftRight(A_{a_2,a_1}) \cup \cdots \cup ShiftRight(A_{a_{n-1},a_1})$ = "011 ... 1".

For this example, the algorithm *second − phase* will produce only one selection function sequence:

S_{b_n} = "011 ... 1".

The selection function sequence for each S_{b_j} $(j = 1, \ldots, n-1)$ can be taken to be "00 ... 0". Then, *second-phase* will generate a graph which corresponds to the *down* part of the graph shown in Fig.3. The topological order for it is:

$$b_n \longrightarrow b_{n-1} \longrightarrow \cdots \longrightarrow b_2 \longrightarrow b_1.$$

In this order, the generalized selection function sequences will be generated and in terms of them the answer set will be produced dynamically:

GH:	Answer set:
$GH_{b_n} = S_{b_n}$ = "011 ... 1"	$\emptyset$
$GH_{b_{n-1}} = S_{b_{n-1}} \cup ShiftLeft(GH_{b_n})$ = "11 ... 10"	$\{b_{n-1}\}$
$GH_{b_{n-2}} = S_{b_{n-2}} \cup ShiftLeft(GH_{b_{n-1}}) \cup ShiftLeft(GH_{b_n})$ = "11...10"	$\{b_{n-1}, b_{n-2}\}$
...	
$GH_{b_1} = S_{b_1} \cup ShiftLeft(GH_{b_2}) \cup \ldots \cup ShiftLeft(GH_{b_n})$ = "11...10"	$\{b_{n-1}, b_{n-2}, \ldots, b_1\}$.

4 Correctness of the Refined Algorithm

In this section, we prove the correctness of our algorithm in the case of non-cyclic data. To this end, we first show that the first phase of our algorithm will generate all appearance function sequences correctly. Then we prove that the second phase can produce all answer nodes.

Proposition 3. *In the case of non-cyclic data, the algorithm first-phase will generate all appearance function sequences for the nodes of G_u.*

Proof. We prove the proposition by induction over the generated appearance function sequences on the number of the nodes of G_u.

Basis: If G_u contains only one node (the source node), its appearance function sequence is "10...0" and thus is correct.

Induction step: Suppose that the proposition holds for all graphs containing k nodes $(k \leq n-1)$ and that G_u contains n nodes. Let $v_1 \longrightarrow v_2 \longrightarrow \cdots \longrightarrow v_n$ be a topological order for G_u. Then the subgraph of G_u generated by $\{v_1, v_2, \ldots, v_{n-1}\}$ contains $n-1$ nodes. By the induction hypothesis, the appearance function sequence for each $v_i \in \{v_1, v_2, \ldots, v_{n-1}\}$ can be correctly generated. Let $v_{i(1)}, \ldots, v_{i(l)}$ be the direct precedents of v_n in G_u. Then $A_{v_n,s} = ShiftRight(A_{v_{i(1)},s}) \cup \ldots \cup ShiftRight(A_{v_{i(l)},s})$, which is exactly evaluated by the algorithm *first-phase*. Since each $A_{i(j),s}$ $(j = 1, \ldots, l)$ is correctly evaluated, $A_{v_n,s}$ is correctly evaluated. This completes the proof.

In order to prove the second phase of our algorithm, we first give the following proposition.

Proposition 4. *If $e \in N_d$ belongs to the answer set, then there exists a path of length $2i+1$, $i \geq 0$, from the source node s to e such that the first i edges are in E_u,*

the $(i+1)$th edge is in E_f, and the last i edges are in E_d (such a path is called an answer path.)

Proof. See the appendix of [16].

Based on this proposition, we can immediately prove the correctness of the algorithm *second-phase*.

Proposition 5. *In the case of non-cyclic data, the algorithm second-phase will produce all answer nodes.*

Proof. First, we claim that the algorithm *second-phase* can correctly produce all generalized selection function sequences for the nodes of G_d. This can be proved in a similar way as Proposition 3 is manifested. Then, we clarify that for each e in the "answer set" produced by *second-phase* there exists an answer path. Consider the statement marked with $*$ in *second-phase*, by which we insert a node into the answer set if the 0th position of its generalized selection function sequence is "1". We distinguish between two cases in which the 0th position of e's GH is "1". The first case is that there is an edge (s, e) in E_f, where s is the source node, then e is in the answer set and the length of the associated path is $2 \cdot 0 + 1 = 1$. The second case is that among the direct precedents of e there is at least one node e_1, the 1st position of its GH is "1". If $S_{e_1}(1) = 1$, then there is an answer path of length $2 \cdot 1 + 1 = 3$. Otherwise, consider the direct precedents of e_1. Similarly, among these nodes there must exist at least another node e_2, the 2nd position of its GH is "1". If $S_{e_2}(2) = 1$, then there is such an answer path of length $2 \cdot 2 + 1 = 5$. Otherwise, we further consider its direct precedents. In this way, we can always find a sequence $e_1, e_2, \ldots, e_i$ connecting e_i and e with $S_{e_i}(i) = 1$. Thus, there is a path (of length $2i + 1$, $i \geq 0$), which connects the source node and e. Therefore, in terms of Proposition 4, we know that the answer set is correctly evaluated.

5 Comparison with Other Strategies

In the analysis below, we consider only the following abstract linear recursive program:

$$s(x, y) \text{:-} r(x, y)$$
$$s(x, y) \text{:-} p(x, z), s(z, w), q(w, y).$$

Assume that the graph representing the relation for "r" contains n_r nodes and e_r edges, the graph for "p" contains n_p nodes and e_p edges, and the graph for "q" contains n_q nodes and e_q edges.

As stated above, in the case of non-cyclic data, the first phase and second phase of the refined algorithm require $O(n_p + e_p)$ time and $O(n_q + e_q)$ time, respectively. Therefore, it is a linear time algorithm. In contrast, the counting method requires $O(n_p \cdot e_p)$ time in the first phase and $O(n_p \cdot e_q)$ time in the second phase. The magic-set method is another bottom-up algorithm which works also in two phases. The first phase of it consists of determining all nodes in N_u (N_u is then called the magic set). In the second phase, the method computes all possible pairs of nodes (i, j) satisfying the following conditions:

(1) i is in N_u, j is in N_d,
(2) there is another pair (i', j') which is in E_f or is produced in previous steps, and
(3) $(i, i') \in E_u$ and $(j, j') \in E_d$.

Therefore, if we use the semi-naive approach, the cost of the first phase of the magic-set method is $O(e_p)$. The cost of the second phase is $\sum_{(i,j)\in A} indegree(i) \times outdegree(j) = O(e_p \cdot e_q)$, where A denotes the set of answer tuples. The graph shown in Fig.4 helps to clarify this result.

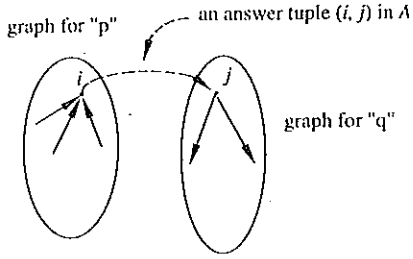

Fig.4. Illustration for time complexity analysis.

From this graph, we see that crossing an answer tuple, say (i, j), each edge incident to j will be visited $indegree(i)$ times by the magic set method. Since the number of answer tuples is bounded by $(n_p \cdot n_q)$, the cost of the magic set method is $O(e_p \cdot e_q)$[15,16]. In recent years, there has been considerable effort directed toward the extension of the counting method for dealing with cyclic relations, such as the level-cycle merging method proposed in [13, 14], the synchronized counting method[1] and the method proposed by Haddad and Naughton[12]. All those methods try to reduce the time complexity to $O(n \cdot e)$ in the case of cyclic data. But no progress has been made in the direction of decreasing the time complexity of the counting method itself. In addition, a lot of experiments have been done[5] and show that QSQR, a well-known top-down strategy[18], has the same time complexity as the magic set method. At an abstract level, the expansion phase of QSQR can be viewed as two processes: a constant propagation process and a variable instantiation process. The former corresponds to the traversal of the graph for "p". The latter corresponds to the traversal of the graphs for "r" and "q". Therefore, the analysis for the magic set method applies to QSQR.

6 Conclusion

In this paper, two new concepts, heritage appearance function and heritage selection function, have been introduced and an efficient algorithm for evaluating recursive queries has been developed. Based on the computation of such functions in topological order, this algorithm reduces the cost of the counting method significantly and can be used to treat non-cyclic relations. The algorithm is efficient not only due to the simplicity of Boolean operations, but also due to the elimination of much redundancy by using binary sequence property. In the case of non-cyclic data, the algorithm requires only $O(n + e)$ time, where n and e denote the numbers of nodes and edges, respectively, in the graph representing the input relations.

References

[1] Aly H, Ozsoyoglu Z M. Synchronized counting method. In *Proc. the 5th Int'l Conf. on Data Engineering*, Los Angeles, 1989.

[2] Balbin G S, Port K Ramamohanarao, Meenakshi K. Efficient bottom-up computation of queries on stratified databases. *J. Logic Programming*, November 1991, pp.295–344.

[3] Bancilhon F, Maier D, Sagiv Y, Ullman J D. Magic sets and other strange ways to implement logic programs. In *Proc. 5th ACM Symp. Principles of Database Systems*, Cambridge, MA, March 1986, pp.1–15.

[4] Beeri C, Ramakrishnan R. On the power of magic. *International Journal of Logic Programming*, 1991, 10: 255–299.

[5] Ceri S, Gottlob G, Tanca L. Logic Programming and Databases. Springer-Verlag, Berlin, 1990.

[6] Chen Y, Haerder T. Improving RQA/FQI recursive query algorithm. In *Proc. ISMM — First Int'l Conf. on Information and Knowledge Management*, Baltimore, Maryland, USA, ACM, Nov. 1992.

[7] Chen Y. A bottom-up query evaluation method for stratified databases. In *Proc. 9th Int'l Conf. on Data Engineering*, Vienna, Austria: IEEE, April 1993, pp.568-575.

[8] Chen Y, Haerder T. On the optimal top-down evaluation of recursive queries. In *Proc. 5th Int'l Conf. on Database and Expert Systems Applications*, Greece, Athens: Springer-Verlag, Sept. 1994, pp.47-56.

[9] Chen Y, Haerder T. An optimal graph traversal algorithm for evaluating linear binary-chain programs. In *Proc. CIKM'94 — The 3rd Int'l Conf. on Information and Knowledge Management*, Gaithersburg, Maryland: ACM, Nov. 1994, pp.34-41.

[10] Chen Y. Magic sets and stratified databases. *International Journal of Intelligent Systems*, 1997, 12: 203-231.

[11] Chen Y. Magic sets revisited. *Journal of Computer Science and Technology*, 1997, 12(4): 346-365.

[12] Haddad R W, Naughton J F. Counting method for cyclic relations. In *Proc. the 7th ACM SIGMOD-SIGACT Symp. on Principles of Database Systems*, 1986, pp.16-23.

[13] Han J, Henschen L J. The level-cycle merging method. In *Proc. the 1st Int'l Conf. on Deductive and Object-Oriented Databases*, Kyoto, 1989.

[14] Wu C, Henschen L J. Answering linear recursive queries in cyclic databases. In *Proc. the 1988 Int'l Conf. on Fifth Generation Computer Systems*, Tokyo, 1988.

[15] Marchetti-Spaccamela A, Pelaggi A, Sacca D. Worst case complexity analysis of methods for logic query implementation. In *Proc. ACM-PODS*, 1987.

[16] Marchetti-Spaccamela A, Pelaggi A, Sacca D. Comparison of methods for logic-query implementation. *J. Logic Programming*, 1991, 10: 333-360.

[17] Sacca D, Zaniolo C. Magic counting method. In *Proc. ACM-SIGMOD*, May 1987.

[18] Vieille L. From QSQ to QoSaQ: Global optimization of recursive queries. In *Proc. 2nd Int'l Conf. on Expert Database System*, Kerschberg L (ed.), Charleston, 1988.

[19] Knuth D E. The Art of Computer Programming. Addison-Wesley Series in Computer Science and Information Processing, 1968, pp.257-265.

For the biography of **Chen Yangjun** please refer to p.365 No.4, Vol.12 of this Journal.