Magic Sets Revisited

Chen Yangjun* (陈阳军)

Technical Institute of Changsha, Changsha 410073

Received April 10, 1995; revised July 19, 1996.

Abstract

This paper distinguishes among three kinds of linear recursions: canonical strongly linear recursion (CSLR), non-interdependent linear recursion (NILR) and interdependent linear recurstion (ILR) and presents an optimal algorithm for each. First, for the CSLRs, the magic-set method is refined in such a way that queries can be evaluated efficiently. Then, for the NILRs and ILRs, the concept of query dependency graphs is introduced to partition the rules of a program into a set of CSLRs and the computation is elaborated so that the oplimization for CSLRs can also be applied.

Keywords: Deductive database, recursive query, magic set, query graph, bottom-up evaluation.

1 Introduction

The evaluation of recursive queries expressed as sets of Horn Clauses over a database has been studied in the last several years. A lot of strategies for processing recursive queries (queries against a recursive program) have been proposed (see [1–8]) such as naive method[9–16], seminaive method[17], magic sets and counting[16,18,19] as well as top-down evaluations[7,20]. In addition, several graph traversal algorithms are used to deal with the difficulties of cyclic data[14,15]. In this paper, we develop a new bottom-up evaluation algorithm which requires less time complexity than the magic set method. First, we try to optimize the evaluation of CSLRs by refining magic sets in such a way that not only the constant propagation which happens in a top-down evaluation, but also the derivation tree constructed by a top-down strategy, will be simulated through the magic transformation of original programs. Then we define the concept of query dependency graphs and develop a control mechanism to decompose a linear recursive program so that it can be treated as a set of CSLRs glued together in some way. In this sense, the optimization described above is extended to handle any complicated (linear recursive) program.

In the next section, we briefly describe the magic-set method[19]. In Section 3, we introduce two graph concepts, i.e., the query graph and the query dependency graph which may be used to clarify the key idea of the refined algorithm. In Section 4, the optimization for CSLRs is described in detail. First, in Subsection 4.1, we

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

present our algorithm. Then, in Subsection 4.2, we compare its time complexity with some well-known strategies. In Section 5, we further consider NILRs and ILRs. Finally, a short conclusion is set forth in Section 6.

2 Basic Definitions and Magic-Set Method

In this section, we first present some notation and preliminary definitions and then briefly describe the magic-set method which is necessary for explaining the key idea of our algorithms.

2.1 Basic Definitions

The language of a deductive database cousists of the variables, constants, and predicate names in the database. We adopt some informal notational conventions for them. Variables will normally be denoted by the letters u, v, x, y and z (possibly with subscript). Constants will normally be denoted by the lellers a, b and c (possibly with subscript). Predicate names will normally be denoted by the letters p, q, r and s (possibly with subscript). In the absence of function symbols, a term is either a constant or a variable. Occasionally, it will be convenient not to apply these conventions rigorously. In such a case, possible confusion will be avoided by the context.

An atom is an n-ary predicate, $p(t_1, t_2, \ldots, t_n)$, $n \geq 0$, where p is a predicate name and $t_1, t_2, \ldots, t_n$ are terms. A literal is an atom or the negation of an atom. A positive literal is just an atom. A negative literal is the negation of an atom.

A *rule* is a first-order formula of the form

$$q \leftarrow p_1, p_2, \ldots, p_m, \quad m \geq 0$$

q is called the *head* and the conjunction $p_1, p_2, \ldots, p_m$ is called the *body* of the rule. Each p_i is a body literal. When $m = 0$, the rule is of the form

$$q \leftarrow$$

and is known as a *unit clause.*

An atom $p(t_1, t_2, \ldots, t_n)$, $n \geq 0$, is *ground* when all of its terms $t_1, t_2, \ldots, t_n$ are constants. A *ground rule* is one in which each atom in the rule is ground. A *fact* is a ground unit clause. The definition of a predicate p is the set of rules which have p as the head predicate. A *base predicate* is defined solely by facts. The set of facts in the database is also known as the *extensional database.* A rule that is not a fact is known as a derivation rule. A *derived predicate* is a predicate which is defined solely by derivation rules. A derived (base) literal is one whose predicate is derived (base). The set of derivation rules is also known as the *intensional database* or *program.*

In addition, in this paper we use the term *graph* to refer to a directed graph, since we do not discuss undirected ones at all. A graph is said to be *strongly connected* if for every two nodes a and b there exists a directed path from a to b. A topological order for an acyclic graph is an order with the property that all descendants of a node precede the node in the order.

2.2 Magic-Set Method

Now, we briefly describe the magic set method, based on which our algorithm is developed. The purpose of the magic-set method is the optimization of programs with particular *adornment* of the goal predicates[1,18,19]. (An adornment for an m-ary predicate $p(t_1, t_2, \ldots, t_m)$ is a string of length m made up of the letters b and f, where b stands for *bound* and f stands for *free*. We obtain an adornment for a predicate as follows. During a computation, each argument t_i, $1 \leq i \leq m$, of the literal $p(t_1, t_2, \ldots, t_m)$ is expected to be bound or free, depending on the information flow. If t_i is expected to be bound (free), it acquires a $b(f)$ annotation, and so the length of the adornment string is m. Note that the adornment is attached to the predicate and becomes part of it.) This method is based on the idea of the *sideways information-passing strategy* (SIPS). Intuitively, given a certain rule and a predicate (subgoal) in the rule body with some bound arguments, one can solve this subgoal, and so obtain bindings for uninstantiated variables in other argument positions. These bindings can be transferred to other subgoals in the rule body, and they, in their turn, transmit bindings to other variables.

This is the normal behaviour of top-down evaluation methods. In order to simulate the binding passing strategy of top-down methods, the magic-set method introduces constraint into the program, by means of additional subgoals (called magic predicates) added to the body of the original rules, and additional rules (called magic rules) defining these goals added to the program. The additional rules constrain the program variables to satisfy other predicates also. Thus, during bottom-up computation, the variables assume only some values instead of all possible ones. In most cases, this makes the new programs more efficient.

Example 2.1. Consider the (adorned) query $\leftarrow ancestor^{bf}(John, y)$ to the program

$ancestor(x, y) \leftarrow parent(x, y)$,
$ancestor(x, y) \leftarrow parent(x, z), ancestor(z, y)$.

The corresponding adorned program is

$ancestor^{bf}(x, y) \leftarrow parent(x, y)$,
$ancestor^{bf}(x, y) \leftarrow parent(x, z), ancestor^{bf}(z, y)$.

The modified rules (1) and (2), and magic rules (3) and (4) are:

(1) $ancestor^{bf}(x, y) \leftarrow magic\_ancestor^{bf}(x), parent(x, y)$,
(2) $ancestor^{bf}(x, y) \leftarrow magic\_ancestor^{bf}(x), parent(x, z), ancestor^{bf}(z, y)$,
(3) $magic\_ancestor^{bf}(z) \leftarrow magic\_ancestor^{bf}(x), parent(x, z)$,
(4) $magic\_ancestor^{bf}(John)$.

For further details please refer to the description in [1,18,19].

3 Query Graph and Query Dependency Graph

In this section, we introduce two graph concepts: *query graph* and *query dependency graph*, which help to illustrate the main idea of the optimization. While the query graph is used to demonstrate the behaviour of the evaluation of CSLR queries, the query dependency graph is utilized to specify program decomposition

strategies w.r.t. NILRs and ILRs. We define these two graphs in Subsections 3.1 and 3.2, respectively.

3.1 Query Graph

We can associate a directed graph to a query with respect to a canonical strongly linear recursive program. The nodes of the graph correspond to tuples of constants; in particular, the source node corresponds to the tuple of constants in the query goal. The other nodes (and incoming arcs) are obtained by retrieving tuples from database $\mathbf{D}$ via a goal composed by a conjunction of base predicates, using restrictions from the tuples corresponding to previously generated nodes. In order to formally define such a graph, we require additional definitions.

Let $\mathbf{v}$ be a list of arguments, and let a be an adornment. Then $\mathbf{v}(a)$ stands for the ordered list of the arguments of $\mathbf{v}$ whose elements correspond to the b annotations of a. Let q be a recursive query. We define the query graph of q to be a directed graph $G_q = \langle N_u \cup N_d, A_u \cup A_f \cup A_d \rangle$ having nodes of the form $[a, \mathbf{c}]$, where a is an adornment and $\mathbf{c}$ is a tuple of constants. The number of b's in a is the same as the number of components of $\mathbf{c}$. The query graph is constructed as follows:

(a) (source node)

The node $[a_1, \mathbf{c}_1]$ is in N_u (source node), where a_1 is the adornment of the query goal and $\mathbf{c}_1$ is the list of constants in the query goal.

(b) (subgraph $G_u = \langle N_u, A_u \rangle$)

If $[a_i, \mathbf{c}_j]$ is in N_u and there exists a substitution θ for the set of variables in the predicates of the body occurring before the recursive subgoal, $q(\mathbf{v}_1)$, and the predicate of the head of the recursive rule, $q(\mathbf{v}_0)$, such that $\mathbf{v}_0(a_i)\theta = \mathbf{c}_j$ and every predicate occurring before $q(\mathbf{v}_1)$ is *true* if θ is applied to them, then the node $[a_{i+1}, \mathbf{c}'_j]$ is in N_u and the arc $([a_i, \mathbf{c}_j], [a_{i+1}, \mathbf{c}'_j])$ is in A_u, where a_{i+1} is the adornment of $q(\mathbf{v}_1)$ and $\mathbf{c}'_j = \mathbf{v}_1(a_{i+1})\theta$.

(c) (subgraph $G_f = \langle N_f, A_f \rangle$, $N_f \subset N_u \cup N_d$)

If $[a, \mathbf{c}_j]$ is in N_u and there exist both a nonrecursive rule in $\mathbf{P}$, say

$q(\mathbf{v})$:-$p_1, \ldots, p_n$,

and a substitution θ for the set of variables in $p_1, \ldots, p_n$ and in $\mathbf{v}$ such that $\mathbf{v}(a_i)\theta = \mathbf{c}_j$ and every $p_i\theta (i = 1, \ldots, n)$ is in $\mathbf{D}$, then the node $[a_i^-, \mathbf{b}]$ is in N_d and the arc $([a_i, \mathbf{c}_j], [a_i^-, \mathbf{b}])$ is in A_f, where a_i^- is an adornment obtained by replacing each b and f in a_i with f and b, respectively, and $\mathbf{b} = \mathbf{v}(a_i^-)\theta$.

(d) (subgraph $G_d = \langle N_d, A_d \rangle$)

If $[a_i^-, \mathbf{b}]$ is in N_d, there is a_j such that $[a_i, \mathbf{c}]$ is the subsequent node of $[a_j, \mathbf{c}']$, and there exists a substitution θ for the set of variables in the recursive predicate, $q(\mathbf{v}_1)$, in the predicates occurring after it, and in the predicate of the head of the recursive rule, $q(\mathbf{v}_0)$, such that $\mathbf{v}_1(a_i)\theta = \mathbf{b}$ and every predicate occurring after $q(\mathbf{v}_1)$ is *true* if θ is applied to them, then the node $[a_j^-, \mathbf{b}']$ is in N_d and the arc $([a_i^-, \mathbf{b}], [a_j^-, \mathbf{b}'])$ is in A_d, where $\mathbf{b}' = \mathbf{v}_0(a_j^-)\theta$. $[a_j, \mathbf{c}']$ and $[a_j^-, \mathbf{b}']$ are called *bound tuple* and *answer tuple*, respectively.

We note that the query graph G_q is composed of three subgraphs, $G_u = \langle N_u, A_u \rangle$, $G_f = \langle N_f, A_f \rangle$, and $G_d = \langle N_d, A_d \rangle$, that are induced by A_u, A_f and A_d, respectively. Note that A_u, A_f and A_d are disjoint. It is easy to see that $N_f \subset N_u \cup N_d$ is bipartite, since every arc in A_f goes from a node in N_u to a node in N_d.

Example 3.1. Consider the following program.

$s(x, y)$:-$r(x, y)$,

$s(x,y)$:-$p(x,z), s(z,w), q(w,y)$,
where p, q and r are base predicates. Suppose that the facts in **D** are those stored in the following database relations:

p	
c	d
c	b
b	c
b	f
f	c

q	
e	a
a	i
i	o
o	g

r	
d	e

Given the query ?:-$s(c,y)$, the corresponding query graph is as shown in Fig.1. The dotted arcs are in A_f, the solid arcs going up are in A_u, and those going down are in A_d. One could expect that any answer to a query will correspond to a node that is reachable from the source node through a path having i arcs from A_u, one arc from A_f, and i arcs from A_d, where $i \geq 0$. In fact, we have the following theorem for a simple class of queries[21].

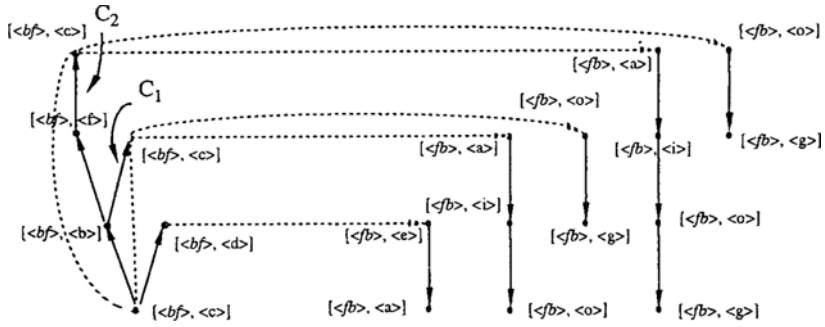


Fig.1. Query graph of Example 3.1.

Theorem 3.1. *Let G_q be the query graph of the query q with respect to the program* **P**. *If* **P** *contains exactly one recursive rule, and this rule is linear, then there exists an answer path from the source node $[a, \mathbf{b}]$ to the node $[a^-, \mathbf{c}]$ in G_q, if $q(\mathbf{b}, \mathbf{c})$ is an answer to the query ?-$q(\mathbf{b}, \mathbf{y})$.*

Proof. See the Aappendix of [21]. □

In the figure above there are two cyclic paths in A_u: C_1 and C_2. (We connect the nodes representing the same bound tuple in A_u with dotted lines and we call these nodes *cyclic points*.) When a new answer tuple corresponding to a cyclic point is evaluated, we add a dotted arc from the cyclic point to the node representing the new answer tuple and subsequently construct a new path leaving this node in A_d according to part (d) above. Therefore, corresponding to each iteration along a cyclic path in A_u we have a separate path in A_d. For example, for the cyclic path C_1 in A_u: $[\langle bf\rangle, \langle c\rangle] \rightarrow [\langle bf\rangle, \langle b\rangle] \rightarrow [\langle bf\rangle, \langle c\rangle]$, we have two paths in A_d: $[\langle fb\rangle, \langle a\rangle] \rightarrow [\langle fb\rangle, \langle i\rangle] \rightarrow [\langle fb\rangle, \langle o\rangle]$ and $[\langle fb\rangle, \langle o\rangle] \rightarrow [\langle fb\rangle, \langle g\rangle]$ with each corresponding to a traversal along the cyclic path C_1. Similarly, for the other cyclic path C_2 in A_u: $[\langle bf\rangle, \langle c\rangle] \rightarrow [\langle bf\rangle, \langle b\rangle] \rightarrow [\langle bf\rangle, \langle f\rangle] \rightarrow [\langle bf\rangle, \langle c\rangle]$, we have $[\langle fb\rangle, \langle a\rangle] \rightarrow [\langle fb\rangle, \langle i\rangle] \rightarrow [\langle fb\rangle, \langle o\rangle] \rightarrow [\langle fb\rangle, \langle g\rangle]$ and $[\langle fb\rangle, \langle o\rangle] \rightarrow [\langle fb\rangle, \langle g\rangle]$ in A_d. In addition, from the graph shown in Fig.1, we see that the set of nodes of some paths (in A_d) corresponding to a cyclic path in A_u is the same as that of some other paths (in A_d) corresponding to

another cyclic path in A_u. For example, the set of nodes (answer tuples) of the paths (in A_d) corresponding to C_1 (in A_u) is $\{[\langle fb\rangle, \langle a\rangle], [\langle fb\rangle, \langle i\rangle], [\langle fb\rangle, \langle o\rangle], [\langle fb\rangle, \langle g\rangle]\}$. The set of nodes (answer tuples) of the paths (in A_d) corresponding to C_2 (in A_u) is also $\{[\langle fb\rangle, \langle a\rangle], [\langle fb\rangle, \langle i\rangle], [\langle fb\rangle, \langle o\rangle], [\langle fb\rangle, \langle g\rangle]\}$. In fact, this interesting property exists for all linear recursive programs, if a preprocessor is used to reorder the body predicates of a recursive rule in the following way. If any two non-recursive predicates which are directly or indirectly (but not via the recursive predicate) correlated are separated by the recursive predicate, we shift the latter such that both of them are before the recursive predicate. A program is called *well ordered* if it is reordered in this way. An example of well ordered programs is the same-generation problem:

$sg(x, y)$:-$flat(x, y)$,

$sg(x, y)$:-$up(x, z), sg(z, w), down(w, y)$.

Note that this reorder is consistent with the normal optimization strategy of reordering subqueries, which makes the predicates with some of their variables bound to constants appear before the predicates whose variables have no bindings. However, a well reordered rule will have an extra property that the predicates appearing after the recursive predicate are not correlated with any predicate appearing before the recursive predicate. We have the following theorem.

Theorem 3.2. *Let* $\mathbf{P}$ *be a well ordered program. Let* G_q *be the query graph associated with the query* q *with respect to* $\mathbf{P}$*, and* P_i *and* P_j *be two cyclic paths starting from the same node in* G_u *of* G_q*. Then the set of answer tuples corresponding to* P_i *is the same as that corresponding to* P_j*. (Note that answer tuples are not to be confused with answers.)*

Proof. Let $q(\mathbf{x}, \mathbf{y})$:-$p_1, \ldots, q(\mathbf{s}, \mathbf{t}), \ldots, p_n$ be a linear recursive rule (with the rule number r) in $\mathbf{P}$, where $\mathbf{x}$, $\mathbf{y}$, $\mathbf{s}$ and $\mathbf{t}$ are all variable tuples, and $\mathbf{Z}$ be the set of all answer tuples with respect to the query ?-$q(\mathbf{c}, \mathbf{y})$. Let $\mathbf{V}$ be the set of all variables appearing in r, $\mathbf{V}_x = \mathbf{V}/(\mathbf{t} \cup \mathbf{y})$ and $\mathbf{IV}_x$ be those constant tuples, to which $\mathbf{V}_x$ will be bound. We define the function $f(\mathbf{v}, \mathbf{z})$ ($\mathbf{v} \in \mathbf{IV}_x, \mathbf{z} \in \mathbf{Z}$) as follows. If there exists a substitution θ for the variables appearing in r such that $\mathbf{V}_x\theta = \mathbf{v}$, $\mathbf{t}\theta = \mathbf{z}$, $\mathbf{y}\theta = \mathbf{z}'$, and every p_i ($i = 1, \ldots, n$) is *true*, then $f(\mathbf{v}, \mathbf{z}) = \mathbf{z}'$. In general, f is a multi-value function, since we may find several substitutions θ_j ($j = 1, \ldots, m$) such that $\mathbf{V}_x\theta_j = \mathbf{v}$, $\mathbf{t}\theta_j = \mathbf{z}$, $\mathbf{y}\theta_j = \mathbf{z}'_j$, and every $p_i (i = 1, \ldots, n)$ is *true*. For any recursive rule in $\mathbf{P}$, since all predicates occurring after a recursive predicate are only correlated with the recursive predicate (due to the fact that $\mathbf{P}$ is well ordered) and the bindings for their variables are completely determined by the answer tuples of the recursive predicate produced so far, the answer tuples for the recursive predicate of the i-th call are completely determined by the answer tuples for the recursive predicate of $(i+1)$-th call. That is, in G_d, the answer tuple corresponding to a node is completely determined by the answer tuple corresponding to its direct precedent node. Therefore, each answer tuple evaluated along a cyclic path is in a set of answer tuples of the form: $f(\_, f(\ldots f(\_, \mathbf{z})\ldots))$, where "_" means "do not care" and $\mathbf{z}$ is an answer tuple whose corresponding bound tuple is the starting point of the cyclic path. Because P_i and P_j have the same starting point, they have also the same $\mathbf{z}$. Thus, the set of answer tuples corresponding to P_i is the same as that corresponding

to P_j. □

The theorem manifests a new possibility to speed up the evaluation (see Fig.4 for illustration).

Also, we note that in the case of linear recursion the magic-set method may work in two phases. The first phase consists of determining all nodes in N_u, i.e., evaluating all instantiations of the magic predicates. In the second phase, the magic-set method computes all possible pairs of nodes (n_i, n_j) such that n_i is in N_u, n_j is in N_d, and there is an answer path from n_i to n_j. In practical implementation, we start from an arc (n_i, n_j) in A_f and compute all pairs $(h_{i'}, n_{j'})$ such that $(n_i, n_{i'})$ is in A_u and $(n_j, n_{j'})$ is in A_d. The so-obtained pair, in turn, is used to derive other pairs. The magic set is used to make this derivation more efficient.

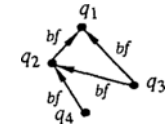
3.2 Query Dependency Graph

In order to specify the program decomposition strategy with respect to NILRs and ILRs, we define a graph, called a query dependency graph, for each linear recursive program as follows. The nodes of the graph represent the recursive predicates appearing in the program. An edge $p \rightarrow q$ connects p and q iff p appears in the body of some rule defining q and is labelled with an adornment that is expected for p in terms of the current query submitted to the system. For example, for the program shown in Fig.2(a), we have a query dependency graph as shown in Fig.2(b).

?-$q_1(c, y)$

(1) $q_1(x, y)$:-$p_1(x, z), r_1(z, y)$,
(2) $q_1(x, y)$:-$p_1(x, z), q_1(z, w), q_2(w, u), q_3(u, y)$,
(3) $q_2(x, y)$:-$p_1(x, z), r_2(z, y)$,
(4) $q_2(x, y)$:-$p_2(x, z), q_3(z, w), q_2(w, u), q_4(u, y)$,
(5) $q_3(x, y)$:-$r_3(x, y)$,
(6) $q_3(x, y)$:-$p_3(x, z), r_3(z, w), q_3(w, y)$,
(7) $q_4(x, y)$:-$p_4(x, y)$,
(8) $q_4(x, y)$:-$p_4(x, z), q_4(z, y)$.

(a)



(b)

Fig.2. Illustration of query dependency graph.

Obviously, the query dependency graph for an NILR is just a node set, containing no edges at all; and the query dependency graph for an ILR is a graph containing no cycles.

In fact, a query dependency graph corresponds exactly to a composition of the corresponding program. That is, each node p in the graph can be thought of as a CSLR defining p with the other recursive predicates (appearing in it) handled as non-recursive ones. These CSLRs are associated with each other with the adornments labeling edges. More importantly, in terms of such a graph and SIPSs, each CSLR can be compiled (as discussed in Subsection 4.1) independently and only once.

Of course, there may be many different methods for partitioning a program. For our purpose, however, decomposing a program into a set of CSLRs is desirable so that the optimization ideas proposed in Subsection 4.1 can be directly employed for a decomposed program.

4 Refined Algorithm for CSLRs

In this section, we discuss only the optimization for CSLRs. First, we present the optimal algorithm in Subsection 4.1. Then, we analyse the time complexity of the proposed algorithm and compare it with other existing methods in Subsection 4.2.

4.1 Algorithm

Below is the refinement of the magic-set method. The key idea of the improvement is that in the first phase of the magic-set method we instantiate all magic predicates, storing not only the instantiations already produced, but recording the cyclic and non-cyclic paths of G_u as well. In the second phase, we suspend the evaluation along each cyclic path of G_d to avoid the redundant computation. In order to guarantee the completeness, an iteration process is developed. The process iterates over some instantiations of the magic predicate which represent the cyclic paths and produces the remaining answers in terms of the answers already found.

In order to record the paths of a query graph, we have to define indexed magic predicates such that in the first phase not only the magic predicates are instantiated, but the paths are also constructed automatically. We construct indexed magic predicates as follows. Let $magic\_p^a(y)$ be a magic predicate. We prefix it with "*ind_*". The indexed version has one new argument. This argument is used for constructing labels and pointers, and we assume that it is the first argument. Note that the adornment a refers only to the non-index fields. Before the use of the argument is explained, we first define some relevant concepts.

Definition 4.1. *Let* $magic\_q^a(y)$*:-*$magic\_p^a(z_1), p_1(z_1, z_2), \ldots, p_n(z_n, y)$ *be a magic rule (with the rule number mr). Let* Y *be the set of instantiations for* $magic\_q^a(y)$*. We assign each* $y \in Y$ *a different number, denoted by* num_y*, and call the number the ordinal number of* y*. In addition, we use* num_Y *to represent the set of all* num_y*'s, where* $y \in Y$*. Without loss of generality, we assume that the ordinal numbers of* Y *are* $\{1, \ldots, m\}$*, where* m *is the cardinality of* Y*.*

Definition 4.2. *A label* l *is either* 0 *or a finite sequence of ordinal numbers* $0.i_1.i_2. \ldots .i_k$*. Beginning with a seed, denoted by* 0*, we denote its first, second, ..., i-th, ... immediate descendent node by* $0.1, 0.2, \ldots$*. The* j*-th direct descendent node of* $0.i$ *is denoted by* $0.i.j$*, and in general* $0.i_1.i_2. \ldots .i_k$ *denotes the* i_k*-th immediate descendent node of the ... of the* i_2*-th immediate descendent node of the* i_1*-th immediate descendent node of the seed.*

Since we deal chiefly with the case in which all immediate descendent nodes appear simultaneously, there is no obvious physical interpretation for the meaning of "first", "second", ... immediate descendent node. We use them only for identifying the instantiations of a magic predicate.

Definition 4.3. *Assume we have* m *magic rules, numbered* mr_1 *to* mr_m*. Then the sequence of magic rules mrs used in a derivation can be represented by a sequence of* mr_{i_k}*'s, each* i_j *in the range* $[1, \ldots, m]$*.*

Definition 4.4. *Let mr: magic_$q^a(y)$:-$\cdots$ be a magic rule. The indexed version of magic_$q^a(y)$ is of the form: ind_magic_$q^a(ind, \mathbf{v})$, where ind is a two-element list: $(mrs, label)$.*

With the magic method, we rewrite a program as follows. For each rule r in $\mathbf{P}^a$:

1. For each occurrence of an adorned predicate p^a in the body of the adorned rule, we generate a magic rule defining the indexed magic predicate *ind_magic_*p^a if the SIPS associated with r contains an arc $N \rightarrow p$.

2. The rule is modified by the addition of magic predicates to its body.

3. We create a *seed* for the indexed magic predicates from the query, in the form of a fact, with *mrs* and *label* being " " and 0, respectively. For example, from the query ?-$p(c, y)$, we may create the seed of the form: *ind_magic_*$p^a((, 0), c)$.

Note that we modify the rules by adding magic predicates rather than indexed magic predicates. It is because we use indexed magic predicates only in the execution of the first phase to record the cyclic paths of G_u. In the second phase, we perform the bottom-up computation as does the magic-set method except that we suspend the computation along the corresponding paths of G_d by eliminating the cyclic points of each cyclic path from the set of different instantiations of the magic predicates. (It should be apparent that some further optimization of the magic-transformed program is possible using common subexpression elimination. For example, the body of the second rule in Example 2.1 is reevaluated in the body of the fourth rule. An algorithm to remove such redundancies which uses supplementary magic sets was introduced in [22] and generalized in [19]. For simplicity, we do not present the supplementary version of our method in this paper. However, in practice all programs should be further optimized in this way.)

Now consider the adorned rule:

$$r_i : q^a(\mathcal{X}) \text{:-} p_1^{a_1}(v_1), \ldots, p_n^{a_n}(v_n).$$

We generate a magic rule defining *ind_magic_*$p_j^{a_j}$ if the SIPS associated with r_i contains an arc $N \rightarrow p_j$. The head of the magic rule is *ind_magic_*$p_j^{a_j}((mrs.mr_{i1}, label.num_{v_j^b}), v_j^b)$. If p_i, $i < j$, is in N, we add $p_i^{a_i}(v_i)$ to the body of the magic rule. If q^a is in N, we add *ind_magic_*$q^a((mrs, label), \mathcal{X}^b)$ to the body. Otherwise, we add the built-in predicates (mrs =" ") and ($label$ =0) to the body. Using the pair $(mrs, label)$, we can not only label each instantiation of a magic predicate, but also record each path. An instantiation with index $(mr_1 \ldots .mr_i.mr_{i+1}, i_1 \ldots .i_i.i_{i+1})$ will have an immediate precedent instantiation with index $(mr_1 \ldots .mr_i, i_1 \ldots .i_i)$.

The indexed magic rule corresponding to $N \rightarrow p_j$ with respect to r_i is:

mr_{i1} : *ind_magic_*$p_i^a((mrs.mr_{i1}, label._{num_{v_i^b}})$

:-*ind_magic_*$q^a((mrs, label), \mathcal{X}^b), \ldots, p_{i-1}^{a_{i-1}}(v_{i-1})$.

The rules are modified in the same way as the magic-set method[19].

Example 4.1. Continuing our running example, we present below the rewritten rules, with respect to the query ?-$s(c, y)$, produced by the above method.

$s(x, y)$:-*magic_*$s^{bf}(x), r(x, y)$,

$s(x, y)$:-*magic_*$s^{bf}(x), p(x, z), s(z, w), q(w, y)$,

mr : *ind_magic_*$s^{bf}((mrs.mr, label.num_z), z)$:-*ind_magic_*$s^{bf}((mrs, label), x), p(x, z)$,

*ind_magic_*$s^{bf}((, 0), c)$.

With the index argument, we can record the paths, especially, the cyclic paths. In the first phase, we use the semi-naive method to compute the indexed magic predicates, thereby checking whether a cycle appears. If it is the case, we store the cycle. (Note that this can be done using Tarjan's algorithm for identifying strongly connected components (subgraphs) of a digraph. This algorithm requires only linear time[23].) In the second phase, we evaluate the modified rules as follows. First, we find a topological order for the remaining acyclic digraph obtained by eliminating cyclic points. (Note that a topological order for an acyclic digraph can be found in linear time[6].) Then in the topological order, we evaluate the modified rules with each magic fact being used only once. That is, at each iteration step, the magic predicate appearing in the body of a modified rule is instantiated to exactly one magic fact. In the third phase, we classify the cycles into several groups. Each group corresponds to a strongly connected component and thus each cycle in a group has the same starting node. We evaluate the modified rules only along one cycle for a group. In the fourth phase, we generate the remaining answers for the cyclic paths directly in terms of the answers already found by using the property described in Theorem 3.2. (For ease of exposition, in the following description of our algorithm we consider a simple situation that there is only one group of cycles in the data. In addition, since the generation of answers in terms of the answers already found is a complex process, we postpone the discussion of this problem to the end of this section.)

Algorithm. refined-magic-set-method (query)

begin

Phase 1. (Computing the magic sets)

1 $S_{mg} := \{seed\}, S_{\Delta-mg} := S_{mg}$;
2 **While** $S_{\Delta-mg} \neq \emptyset$ **do** {
3 $new - S_{mg}$:=new_answers evaluated in terms of labelled magic rules, $S_{\Delta-mg}$ and **D**;
(*D is the database.*)
4 $S_{\Delta-mg} := new - S_{mg} - S_{mg}$;
(*The operation "difference" is performed with labels being not considered.*)
5 $S_{mg} := S_{mg} \cup new - S_{mg}$;
(*Note that instead of $S_{\Delta-mg}$ we add $new - S_{mg}$ to S_{mg} to maintain path information.*)

Phase 2. (Computing answers along the cyclic paths)

6 $S_a := \emptyset$;
7 S'_{mg} :=the resultant set obtained by removing all cyclic points from S_{mg};
8 let $S_{topo} = \{s_1, s_2, \ldots, s_l\}$ be the topological order for S'_{mg};
9 **for** $i = 1$ to l **do**
10 $\{new - S_a$:=new_answers evaluated in terms of modified rules, s_i, S_a and **D**;
11 $S_a := S_a \cup new - S_a;\}$

Phase 3. (Computing answers along the first cyclic path)

12 $S_{\text{first-cyclic-paths}}$:=the first cyclic_path;
13 Suppose $S_{\text{first-cyclic-paths}} = \{s_1 \rightarrow s_2 \rightarrow \cdots \rightarrow s_n \rightarrow s_1\}$;
14 **repeat**
15 **for** $j = n$ to 1 **do**
16 $S_a := S_a \cup$ new_answers evaluated in terms of modified rules, s_j, S_a and **D**;
(*Note that the computation is done in the coounter direction of the cycle.*)
17 **until** no new answers are produced

Phase 4. (Generating the remaining answers along the cyclic paths)
18 $S_{\text{cyclic-paths}}$:=cyclic_paths;
19 **repeat**
20 **for** each $path_i$ and $path_j \in S_{\text{cyclic-paths}}$ **do**
21 $S_a := S_a \cup$ new_answers generated in terms of $path_j$ and the answers evaluated or generated for $path_i$;
22 **until** no new answers are produced
23 **begin**
24 evaluating all answers to the initial goal in terms of modified rules, S_a and paths,
25 each of which is from the start node of a cyclic path to the source node;
26 **end**
end

The algorithm works in four phases. The first phase consists of determining all instantiations of the labelled magic predicates (labelled magic set). For efficiency, this phase is implemented using the seminaive approach. (Note that we use here the seminaive approach with a bit modification. First, we do not take the labels into account when checking the equivalence of two elements. Second, we insert, at each step, all produced labelled magic elements (instead of only new ones) into S_{mg} to preserve the complete path information.) As a consequence, $S_{\Delta-mg}$ is introduced to store the new instantiations (of the labelled magic predicate) produced within every loop (and to control the termination of the first phase). At the end (of the first phase), the evaluated labelled magic set is stored in S_{mg}. $new - S_{mg}$ is used to store the temporary results. In the second phase, all answers along the acyclic paths are evaluated. First, a topological order is determined for all nodes of the acyclic graph obtained by eliminating cyclic points. Then, in the topological order, all modified rules are evaluated only in one scan. (In this way, the time complexity can be reduced non-trivially; see Subsection 4.2 for illustration.) At the end, all answers produced in this phase are stored in S_a. In the third phase, the computation of the least fixpoint for the first cyclic path is carried out. Here, $S_{first-cyclic-paths}$ contains the first cyclic path. In the fourth phase, the remaining answers are generated directly in terms of the answers for the first cycle and the corresponding path information. We use $S_{cyclic-paths}$ to store all cycles. After this step, we further evaluate some answers for those paths, each of which is from the start node of a cyclic path to the source node (*seed*). (In the above algorithm, neither the separation of cycles from a graph nor the finding of a topological order is described. The relevant linear algorithms can be found in [6,23].)

Example 4.2. Continuing our running program with the database given in Example 3.1.

Here we trace the steps of the above algorithm for the query ?-$s(c, y)$ against those database relations.

Phase 1. *seed* : $ind\_magic\_s^{bf}((,0), c)$;
Step 1: $ind\_magic\_s^{bf}((mr, 0.1), d), ind\_magic\_s^{bf}((mr, 0.2), b)$;
Step 2: $ind\_magic\_s^{bf}((mr.mr, 0.2.1), c), ind\_magic\_s^{bf}((mr.mr, 0.2.2), f)$;
Step 3: $ind\_magic\_s^{bf}((mr.mr.mr, 0.2.2.1), c)$.

With the help of the index argument, we can record a graph as shown in Fig.3(a) during the execution:

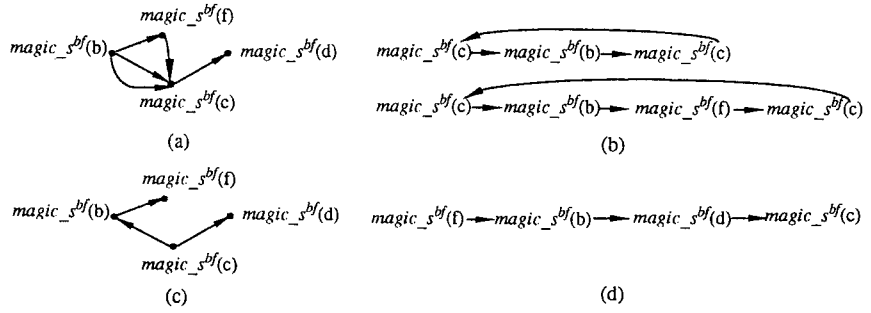


Fig.3. The treatment of magic graphs.

Using Tarjan's algorithm, we can check all cyclic points and identify all cycles in linear time. For this example, we have two cycles as shown in Fig.3(b). The acyclic graph obtained by eliminating cyclic points is shown in Fig.3(c) and a topological order for it is shown in Fig.3(d).

Phase 2. **D**; $S_a = \emptyset$;

$S'_{mg} = \{magic\_s^{bf}(c), magic\_s^{bf}(d), magic\_s^{bf}(f), magic\_s^{bf}(b)\}$;

$S_{topo} = \{magic\_s^{bf}(f), magic\_s^{bf}(b), magic\_s^{bf}(d), magic\_s^{bf}(c)\}$;

Step 1. $S_a = \emptyset$;

Step 2. $S_a = \emptyset$;

Step 3. $S_a = \{s^{bf}(d,e)\}$;

Step 4. $S_a = \{s^{bf}(d,e), s^{bf}(c,a)\}$.

Phase 3. **D**; $S_a = \{s^{bf}(d,e), s^{bf}(c,a)\}$;

$S_{first-cyclic-paths} = \{magic\_s^{bf}(c) \leftarrow magic\_s^{bf}(b) \leftarrow magic\_s^{bf}(c)\}$;

Step 1. (along the first cyclic path)

for loop 1: $S_a = \{s^{bf}(d,e), s^{bf}(c,a), s^{bf}(b,i), s^{bf}(c,o)\}$;

for loop 2: $S_a = \{s^{bf}(d,e), s^{bf}(c,a), s^{bf}(b,i), s^{bf}(c,o), s^{bf}(b,g)\}$.

Phase 4. **D**; $S_a = \{s^{bf}(d,e), s^{bf}(c,a), s^{bf}(b,i), s^{bf}(c,o), s^{bf}(b,g)\}$;

$S_{cyclic-paths} = \{magic\_s^{bf}(c) \leftarrow magic\_s^{bf}(b) \leftarrow magic\_s^{bf}(c),$

$magic\_s^{bf}(c) \leftarrow magic\_s^{bf}(b) \leftarrow magic\_s^{bf}(f) \leftarrow magic\_s^{bf}(c)\}$;

Step 1. (generate answers for the second cyclic path)

for loop 1: $S_a = \{s^{bf}(d,e), s^{bf}(c,a), s^{bf}(b,i), s^{bf}(c,o), s^{bf}(b,g),$

$s^{bf}(f,i), s^{bf}(b,o), s^{bf}(c,g), s^{bf}(f,g)\}$;

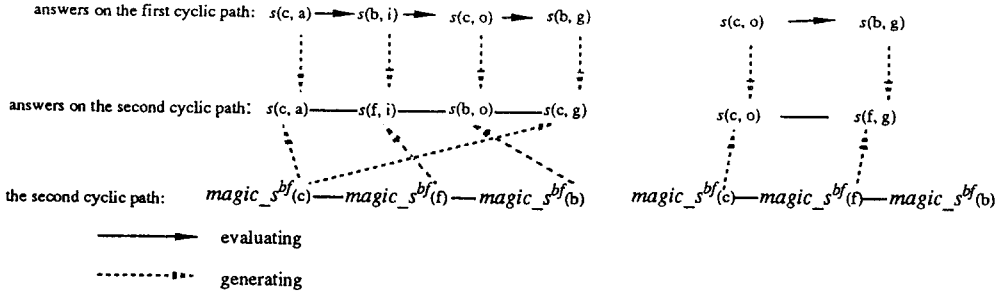


Fig.4. Illustration of generating answers with respect to Example 3.1.

Note that in Step 1 of Phase 4 we directly generate $s(f,i)$ for the second cycle in terms of $magic\_s^{bf}(f)$ and $(s(b,i)$ (which has been produced along the first cycle) instead of producing it by performing algebraic operations. Similarly, we generate

$s(b, o)$ for the second cycle in terms of $magic\_s^{bf}(\underline{b})$ and $s(c, \underline{o})$ (which has also been produced along the first cycle) and so on. Fig.4 helps to illustrate this feature.

Let C_1 be the first cycle $magic_1 \leftarrow magic_2 \leftarrow \cdots \leftarrow magic_n \leftarrow magic_1$ and $A_1 = \{a_1, \ldots, a_n, a_{n+1}, \ldots, a_{2n}, \ldots, a_{in}, \ldots, a_{in+j}\}$ the answer set evaluated along C_1, where i is an integer and j is an integer with $0 \leq j \leq n$. (It should be noticed that each a_l $(1 \leq l \leq in + j)$ itself is a subset which is evaluated by running the modified rules with the magic predicate being instantiated to $magic_\lambda$, where $l = k \cdot n + \lambda$ for some integer k.) Let C_2 be the second cycle $magic'_1 \leftarrow magic'_2 \leftarrow \cdots \leftarrow magic'_m \leftarrow magic'_1$. In addition, we define

$A_2 = \{a_{n+1}, \ldots, a_{2n}, \ldots, a_{in}, \ldots, a_{in+j}\}$,
$A_3 = \{a_{2n+1}, \ldots, a_{in}, \ldots, a_{in+j}\}$,
$\cdots$
$A_i = \{a_{in+1}, \ldots, a_{in+j}\}$.

Then, in terms of A_k $(1 \leq k \leq i)$ and C_2, we can generate the first part of answers for C_2 as follows. (Note that we do not necessarily compute all A_k $(1 \leq k \leq i)$. In practice, each time an A_k needs to be used in the computation, we shrink A_{k-1} by leaving out some a_i's). (Without loss of generality, assume that $n \leq m$.)

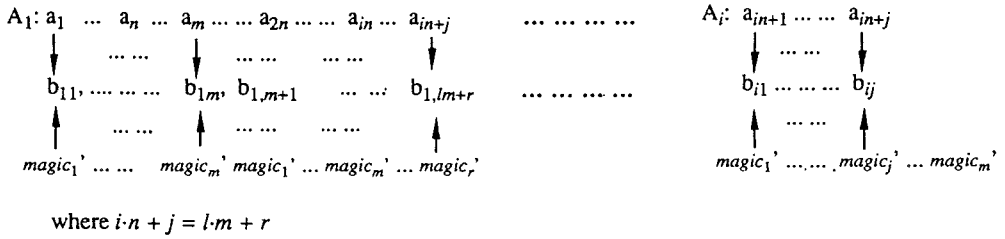


Fig.5. Illustration of answer generation.

If $n = m$, no more new answers can be generated after this step. Otherwise, in terms of C_1 and the newly generated answers for C_2, we can further generate some new answers for C_1 in the same way. To this end, we first merge the newly generated answers as shown in Fig.6(a).

$$b_1 = b_{11} \cup b_{21} \cup \cdots \cup b_{i1},$$
$$b_2 = b_{12} \cup b_{22} \cup \cdots \cup b_{i2},$$
$$\cdots$$
$$b_{lm+r} = b_{1,lm+r} \cup \cdots.$$

(a)

$$B_1 = \{b_1, \ldots, b_{lm+r}\},$$
$$B_2 = \{b_{m+1}, \ldots, b_{lm+r}\},$$
$$B_3 = \{b_{2m+1}, \ldots, b_{lm+r}\},$$
$$\cdots$$
$$B_l = \{b_{lm+1}, \ldots, b_{lm+r}\}.$$

(b)

Fig.6. Merging newly generated answers.

Then, we construct B_k $(1 \leq k \leq l)$ as shown in Fig.6(b).

In terms of C_1 and B_k $(2 \leq k \leq l)$, some new answers for C_1 can be generated as described above. Note that B_1 will not be used in this step. It is because no new answers can be generated in terms of it, i.e., using it, only the same answer set as A_1 can be generated. In the next step, some new answers for C_2 can be generated in terms of C_2 and the newly generated answers for C_1 again. This process continues until no more new answers can be generated.

The correctness of this method is based on the following proposition.

Proposition 4.1. *Let* $C_1 = \{v_1 \leftarrow v_2 \leftarrow \cdots \leftarrow v_n \leftarrow v_1\}$ *and* $C_2 = \{w_1 \leftarrow w_2 \leftarrow \cdots \leftarrow w_m \leftarrow w_1\}$ *be two cycles having the same starting point with respect to a magic rule of the form* $magic\_s^a(y)$:-$\cdots$. *Suppose that each* v_i *is of the form* $magic\_s^{bf}(c_i)$ *and each* w_j *is of the form* $magic\_s^{bf}(d_j)$. *Let* $s(c_i, h)$ *be a fact evaluated along* C_1. *Then, pair* (d_j, h) *corresponds to an* s *fact if* i, j *satisfy the equation:*

$$k \cdot m + j = l \cdot n + i$$

for some integers k *and* l $(0 \leq k \leq n-1,\ 0 \leq l \leq m-1)$.

Proof. If $s(c_i, h)$ is a fact evaluable along C_1, then h can be represented as $f(\_, f(\ldots f(\_, \mathbf{z})\ldots))$, where f is the function defined in the proof of Theorem 3.2, and $\mathbf{z}$ is an answer tuple whose corresponding bound tuple is the starting node of C_1. Assume that h can be obtained by applying f to $\mathbf{z}(l \cdot n + i)$ times for some integer l. We note that C_2 has the same starting node as C_1. Therefore, each answer tuple reachable by evaluating the modified rules along C_2 can also be represented as $f(\_, f(\ldots f(\_, \mathbf{z})\ldots))$. If $k \cdot m + j = l \cdot n + i$ for some integer k, then $s(d_j, h)$ can be obtained by iterating over C_2 k times and then evaluating the modified rules j times. □

4.2 Time Complexity

In order to show the refinement of our method over the magic-set algorithm, we use the following program as a benchmark to do an exact analysis of time complexities.

$s(x, y)$:-$r(x, y)$,
$s(x, y)$:-$p(x, z), s(z, w), q(w, y)$.

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. In the following, we give an abstract analysis of time complexity, which applies to the best implementation of the different methods considered. That is, the results derived for the magic sets, the counting and our method are actual for their supplementary versions (see [19]).

The magic set method works in a two-phase manner. In the first phase, the magic set is produced by executing magic rules. The cost of this phase is $O(e_p)$, since at most $O(e_p)$ edges will be visited. In the second phase, both the modified rules are evaluated. The corresponding cost is $O(e_r) + O(\sum_{(i,j)\in A} indegree(i) \times outdegree(j)) = O(e_r) + O(e_p \cdot e_q)$, where A denotes the set of answer tuples. The graph shown in Fig.7(a) helps to clarify this result.

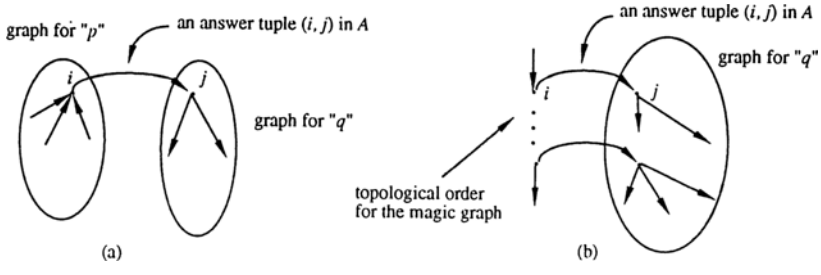


Fig.7. Illustration for time complexity analysis.

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 bound by $(n_p \cdot n_q)$, the cost of the magic set method is $O(e_p \cdot e_q)$[8,24]. The counting method is an indexed version of the magic set method. In this method, each magic predicate is augmented with a new argument to record, during the first phase, information about "distance" which is then used to control the computation of the second phase. It improves the magic set method by a constant factor[8]. In the worst case, its time complexity is $O(n_p \cdot e_q)$[21,24–27]. In addition, a lot of experiments have been done[28] and show that QSQR, a well-known top-down strategy[20], 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.

In the case of non-cyclic data, if the intermediate answers are not stored in a graph structure, the cost of the refined algorithm is:

$$O(e_p) + O(e_r) + \sum_{(i,j)\in A} outdegree(j) = O(e_p) + O(e_r) + O(n_p \cdot e_q) = O(n_p \cdot e_q)$$

(see Fig.7(b) for illustration). From this graph, we see that crossing an answer tuple, say (i, j), each edge incident to j will be visited only once. In addition, separating cycles from a digraph and finding a topological order for an acyclic digraph are rather easy, because both can be done in $O(n_p + e_p)$ time (see [6,23]). Now we derive the time complexity of handling cycles. To simplify the description of the results of the analysis, we assume that each cycle has the same length (by "length" we mean the number of magic facts contained in a cycle) and along each cycle the number of new answers got by the algorithm from an initial value or an evaluated answer at one step is d. Thus, if each cycle has the length m and the number of iterations over a cycle is l, then the time complexity of the magic set method for handling cycles is in the order of

$$\lambda \cdot \sum_{i=1}^{m \cdot l} d^{i-1} \cdot C = \lambda \cdot \frac{d^{ml}-1}{d-1} \cdot C$$

where λ is the number of the cycles in one group and C represents the cost of evaluating an answer at an iteration step. In the worst case, C is the elapsed time of a read access to the external storage, i.e. each evaluation (of an answer) requires an I/O. In the third phase of the refined algorithm, $(d^{ml} - 1)/(d - 1)$ answers are evaluated using the assumption above. The remaining answers for each cycle are all generated in the fourth phase. Let δ be the cost of generating an answer in terms of the answers already produced, then the running time for the fourth phase of the refined algoriflun is

$$\frac{1}{d-1}[(d^{ml} - 1) \cdot C + (\lambda - 1) \cdot (d^{ml} - 1) \cdot \delta]$$

Since $\delta \ll C$, the saving of time is significant. If $\delta/C \leq 1/\lambda d^{ml}$, $(\lambda - 1) \cdot (d^{ml} - 1) \cdot \delta$ is less than some constant, and thus the time complexity of the third and fourth phases of the refined algorithm is $O(d^{ml}C)$. Therefore, the refined algorithm may reduce the worst-case time complexity of the magic set method for handling cycles by a factor λ, the number of the cycles if we do not take the cost of generating

answers into account.

Finally, we point out that the usage of labels will not increase space complexity because we use labels only "locally". That is, at each step (during the first process), we insert the newly produced elements into the digraph with the aid of their associated labels and then discard such labels after the next step is completed.

5 Refined Algorithms for NILRs and ILRs

In general, the idea described above can not be directly employed to handle more complex linear recursive programs. However, we can always partition the rules of a program into several CSLRs so that the optimization idea discussed in the previous section can be directly exploited. In the following, we address this problem in detail.

First, we consider no-interdependence linear recursions (NILRs) which are an important class of programs and may appear in the so-called *disjunctive databases*. As an example, consider the following abstract program.

$q(x,y)$:-$s_1(x,y) \vee s_2(x,y)$,
$s_1(x,y)$:-$r_1(x,y)$,
$s_1(x,y)$:-$p_1(x,z), s_1(z,w), q_1(w,y)$,
$s_2(x,y)$:-$r_2(x,y)$,
$s_2(x,y)$:-$p_2(x,z), s_2(z,w), q_2(w,y)$.

To evaluate a query like ?-$q(c,y)$, both ?-$s_1(c,y)$ and ?-$s_2(c,y)$ will be computed and each corresponds to a CSLR query. Therefore, this program can be simply partitioned into two CSLRs which are not interdependent at all and can be compiled independently in the same way as shown in Subsection 4.1. Thus, we can apply the optimization technique to them.

Algorithm. evaluation-for-NILR (query)

```
begin
  for each NILR q_i involved in the evaluation of query do
      call refined-magic-set-method(query);
end
```

For the ILRs, the technique described in Subsection 4.1 can also be used. However, we need to arrange the computation in a different way. First, we introduce the concept of *variable-seeds*. A variable-seed is a predicate of the form $magic\_q^a(\mathbf{x}^b)$, where q is a recursive predicate, a is an adornment and $\mathbf{x}^b$ is a vector of variables which are expected to be bound during the constant propagation. Notice the difference between the variable-seed and the seed (denoted by $magic\_q^a(\mathbf{v}^b)$) introduced in Subsection 2.4, which is a ground predicate. Then, using the depth-first (or breadth-first) search algorithm, we develop a new magic set transformation algorithm as follows. In the algorithm, the following definitions are employed:

$partitionP(q, \mathbf{P}^a)$ retums part of $\mathbf{P}^a$, which corresponds to the rules (in $\mathbf{P}$) defining q. Remember that $\mathbf{P}^a$ represents the adorned program of $\mathbf{P}$.

$partitionS(q, \mathbf{S}^a)$ returns part of $\mathbf{S}^a$, which is associated only with the rules (in $\mathbf{P}$) defining q.

Further, we modify the function $magic(r^a(\mathbf{v}))$ (which is defined in Subsection 2.4) a bit so that the returned value of it may be a vector of variables that are expected to be bound during the constant propagation.

Algorithm. new-set-transformation $(q^a(\mathbf{v}), \mathbf{P}^a, S^a, T)$ (*T is the query dependency graph of $\mathbf{P}$ w.r.t. ?-$q(\mathbf{v})$.*)

begin
1 call *magic-set-transformation*$(q^a(\mathbf{v}),$ *partitionP*$(q, \mathbf{P}^a),$ *partitionS*$(q, \mathbf{S}^a))$;
2 label the resulting magic rules as discussed in Subsection 4.1;
3 push all the child nodes of q into *stack*;
4 **while** *stack* is not empty **do**
5 {s:= pop(*stack*);
6 let *ad* be the adornment labeling the corresponding edge;
7 call *magic-set-transformation*$(s^{ad}(\mathbf{x}),$ *partitionP*$(s, \mathbf{P}^a),$ *partitionS*$(s, \mathbf{S}^a))$;
8 label the resulting magic rules as discussed in Subsection 4.1;
9 push all the child nodes of s into *stack*;}
end

Obviously, the output of *magic-set-transformation*$(q^a(\mathbf{v}), \mathbf{P}^a, \mathbf{S}^a)$ is quite different from that of the above algorithm, in which for each recursive predicate there exist a seed or a variable-seed, a labelled magic rule set (as discussed in Subsection 4.1) and a modified rule set. As an example, consider the program shown in Fig.2(a) again. In respect of the query ?-$s_1(c, y)$, where c is a constant, the algorithm first constructs the following magic rules and modified rules in terms of the first two clauses of the program (by executing lines 1–3):

$$ind\_magic\_q_1^{bf}((\ ,0), c),$$
$$mr: ind\_magic\_q_1^{bf}((mrs.mr, label.num_z), z) \text{:-} ind\_magic\_q_1^{bf}((mrs, label), x), p_1(x, z),$$
$$q_1(x, y) \text{:-} magic\_q_1^{bf}(x), p_1(x, z), r_1(z, y),$$
$$q_1(x, y) \text{:-} magic\_s_1^{bf}(x), p_1(x, z), q_1(z, w), q_2(w, u), q_3(u, y).$$

Then, by executing the main loop (lines 4–9), the rest of the program will be transformed into the following form:

$$ind\_magic\_q_2^{bf}((\ ,0), x),$$
$$mr: ind\_magic\_q_2^{bf}((mrs.mr, label.num_w), w) \text{:-} ind\_magic\_q_1^{bf}((mrs, label), x), p_2(x, z), q_3(z, w),$$
$$q_2(x, y) \text{:-} magic\_q_1^{bf}(x), p_2(x, z), r_2(z, y),$$
$$q_2(x, y) \text{:-} magic\_q_2^{bf}(x), p_2(x, z), q_3(z, w), q_2(w, u), q_4(u, y),$$

$$ind\_magic\_q_3^{bf}((\ ,0), x),$$
$$mr: ind\_magic\_q_3^{bf}((mrs.mr, label.num_z), z) \text{:-} ind\_magic\_q_3^{bf}((mrs, label), x), p_3(x, w), r_3(w, z),$$
$$q_3(x, y) \text{:-} magic\_q_3^{bf}(x), r_3(x, y),$$
$$q_3(x, y) \text{:-} magic\_q_3^{bf}(x), p_3(x, z), r_3(z, w), q_3(w, u),$$

$$ind\_magic\_q_4^{bf}((\ ,0), x),$$
$$mr: ind\_magic\_q_4^{bf}((mrs.mr, label.num_z), z) \text{:-} ind\_magic\_q_4^{bf}((mrs, label), x), p_1(x, z),$$
$$q_4(x, y) \text{:-} magic\_q_4^{bf}(x), p_4(x, y),$$
$$q_4(x, y) \text{:-} magic\_s_1^{bf}(x), p_4(x, z), q_4(z, y).$$

In order to evaluate a transformed program like the one above, we present the following recursive algorithm *evaluation-for-ILR,* which works in a similar fashion to *refined-magic-set-method* given in Subsection 4.1. During the computation, however, recursive calls may be executed. It happens when a recursive predicate, say q, is encountered. In this case, we replace the variable vector of the q's variable-seed by

the constant vector with which the bound variables of q will be instantiated. Then, we switch the control to compute q's corresponding magic program, which can be done by a recursive call of *evaluation-for-ILR*. Obviously, a four-phase approach can be employed as *refined-magic-set-method* does. In *evaluation-for-ILR*, the following definition is utilized:

$seedForm(magic\_q^a(\mathbf{x}), \mathbf{v})$ returns a seed w.r.t. a certain recursive predicate q by substituting constant vector $\mathbf{v}$ for $\mathbf{x}$. Note that $\mathbf{v}$ represents a constant vector obtained by the constant propagation and is available when q is encountered during the computation.

Algorithm. evaluation-for-ILR(query)

```
   begin
1    S_mg := {seed}, S_Δ−mg := S_mg; (*computing the magic sets*)
2    while S_Δ−mg ≠ ∅ do {
3      new − S_mg := new_answers evaluated in terms of labelled magic rules, S_Δ−mg and D;
4        (if a recursive predicate q is encountered during the evaluation
5        then {seed := seedForm(magic_q^a(x), v); call evaluation-for-ILR(q);})
6      S_Δ−mg := new − S_mg − S_mg;
7      S_mg := S_mg ∪ new − S_mg;}
8      S_a := ∅; (*computing answers along the acyclic paths*)
9      S'_mg:= the resultant set obtained by removing all cyclic points from S_mg;
10     let S_topo = {s_1, s_2, ..., s_l} be the topological order for S'_mg;
11     for i = 1 to l do
12       {new − S_a := new_answers evaluated in terms of modified rules, s_i, S_a and D;
13         (if a recursive predicate q is encountered during the evaluation
14         then {seed := seedForm(magic_q^a(x), v); call evaluation-for-ILR(q);})
15       S_a := S_a ∪ new − S_a;}
16     S_first−cyclic−paths := the first cyclic_path; (*computing anwsers along the first cyclic path*)
17     Suppose S_first−cyclic−path = {s_1 → s_2 → ··· → s_n → s_1};
18     repeat
19       for j = n to 1 do
20         S_a := S_a∪ new_answers evaluated in terms of modified rules, s_j, S_a and D;
21           (if a recursive predicate q is encountered,
22           then {seed := seedForm(magic_q^a(x), v); call evaluation-for-ILR(q);})
23     until no new answers are produced
24     S_cyclic_paths := cyclic_paths; (*generating the remaining answers along the cyclic paths*)
25     repeat
26       for each path_i and path_j ∈ S_cyclic−paths do
27         S_a := S_a∪ new_answers generated in terms of path_j and the answers evaluated or
           generated for path_i;
28     until no new answers are produced
29     begin
30       evaluating all answers to the initial goal in terms of modified rules, S_a and paths,
31       each of which is from the start node of a cyclic path to the source node;
32     end
   end
```

Note lines 4–5, 13–14 and 21–22, by which a recursive call may be executed if any recursive predicate q is encountered during the evaluation. In this case, the control is switched to the computation of q's magic program.

6 Conclusion

In this paper, a bottom-up method for the evaluation of recursive queries has been presented which is more efficient than the magic-set algorithm. The key idea of the improvement is to record the graph with respect to a magic set during the execution of the first phase of the magic-set method and separate the cycles from the graph to handle the cyclic and non-cyclic data differently. First, we find a topological order for the remaining (non-cyclic) graph and execute the modified rules in this order with only one magic fact being used at each iteration step. Then, we evaluate the least fixpoint for the cyclic paths with the exclusion of noncyclic data. In this way, we can guarantee the completeness without redundant work. Furthermore, using the similarities between the cycles in a group, most of answers can be directly generated in terms of the intermediate results and the corresponding path information. Finally, based on a strategy decomposing a complex linear recursive program into several SCLRs, the optimization for SCLRs can also be employed for evaluating any linear recursion.

References

[1] Bancilhon F, Ramakrishnan R. An Amateur's introduction to recursive query processing strategies. In *Proc. 1986 ACM-SIGMOD Conf. on Management of Data,* Washington, D.C., May 1986, pp.16-52.

[2] Han J, Zeng K, Lu T. Normalization of linear recursion in deductive databases. In *Proc. of the 9th Int'l Conf. on Data Engineering,* Vienna, Austria, April 1993, pp.559-567.

[3] Han J, Chen S. Graphic representation of linear recursive rules. *International Journal of Intelligence System,* 1992, 7: 317-337.

[4] Henschen L J, Naqvi S. On compiling queries in recursive first-order database. *J. ACM,* 1984, 31(1): 47-85.

[5] Ioannidis Y, Wong E. Towards an algebraic theory of recursion. *Journal of the Association for Coinpuling Machinery,* 1991, 38(2): 329-381.

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

[7] Nejdl W. Recursive strategies for answering recursive queries — The RQA/FQI strategy. In *Proc. 13th VLDB Conf.,* Brighton, 1987, pp.43-50.

[8] Marchetti-Spaccamela A, Pelaggi A, Sacca D. Worst case complexity analysis of methods for logic query implementation. In *Proc. ACM-PODS 87,*

[9] Chang C. On the Evaluation of Queries Containing Derived Relations in Relational Database. Advances in Data Base Theory, Vol.1, Plenum, 1981.

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

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

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

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

[14] Chen Y, Härder T. Graph traversal and linear binary-chain programs. ZRI-Report 4/94, University of Kaiserslautem, 1994.

[15] Chen Y. Processing of recursive rules in knowledge-based systems — Algorithms for handling recursive rules and negative information and performance measurements. Ph.D. Thesis, Computer Science Department, University of Kaiserslautem, Germany, Feb. 1995.

[16] Sacca D, Zaniolo C. Implementation of recursire queries for a data language based on pure Horn logic. In *Proc. the 4th Int'l Conf. on Logic Programming,* Univ. of Melbourne, 1987, pp.104-135.

[17] Bancilhon F. Naive Evaluation of Recursively Defined Relations. On Knowledge Base Management Systems-Integrating Database and AI Systems. Springer-Verlag, 1985.

[18] BancilhonF, 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.

[19] Beeri C, Ramakrishnan R. On the power of magic. *J. Logic Programming,* 1991, 10: 255-299.

[20] Ullman J D. Implementation of logical query languages for databases. *ACM Trans. Database Systems,* 1985, 10(3).

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

[22] Sacca D, Zaniolo C. On the implementation of a simple class of logic queries for databases. In *Proc. the 5th ACM SIGMOD-SIGACT Symp. on Principles of Database Systems,* 1986, pp.16-23.

[23] Shapiro S, McKay. Inference with recursive rules. In *Proc. of the 1st Annual National Conference on Artificial Intelligence,* 1980.

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

[25] 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.

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

[27] 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.

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

[29] Balbin G S, Port K R, Meenakshi. Efficient bottom-up computation of queries on stratified databases. *J. Logic Programming,* 1991, 10: 295-344.

[30] Bancilhon F, Ramakrishnan R. Performance evaluation of data intensive logic programs. In *Preprints of Workshop on Foundations of Deductive Databases and Logic Programming,* August 1986.

[31] Tarjan R. Depth-first search and linear graph algorithms. *SIAM J. Compt.,* 1972, 1(2).

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

Chen Yangjun received his B.S. degree in information system engineering from the Technical Institute of Changsha, China, in 1982, and his Diploma and Ph.D. degrees in computer science from the University of Kaiserslautern, Germany, in 1990 and 1995, respectively. Dr. Chen is currently an Assistant Professor of the Technical University of Chemnitz-Zwickau, Germany. His research interests include deductive databases, federative databases, constraint satisfaction problem, graph theory and combinatorics. He has about 30 publications in these areas.