On the Bottom–Up Evaluation of Recursive Queries

Yangjun Chen*
Department of Computer Science, University of Kaiserslautern, P.O. Box 3049, 67663 Kaiserslautern, Germany

In this article, we present an optimal bottom–up evaluation method for handling both linear and nonlinear recursion. Based on the well-known magic-set method, we develop a technique: *labeling* to record the cyclic paths during the execution of the first phase of the magic-set method and suspending the computation for the cyclic data in the second phase to avoid the redundant evaluation. Then we postpone this computation to an iteration process (the third phase) which evaluates the remaining answers only along each cyclic path. In this way, we can guarantee the completeness. In addition, for a large class of programs we further optimize our method by elaborating the iteration process and generating most answers for each cyclic path directly from the intermediate results instead of evaluating them by performing algebraic operations (after some of the answers for the first cyclic path are produced). Because the cost of generating an answer is much less than that of evaluating an answer, this optimization is significant.

I. INTRODUCTION

One area in the field of logic programming, known as *deductive database*, is concerned with developing logic-based programming systems which manipulate large quantities of data efficiently. In terms of logic programming, deductive databases are just logic programs without function symbols. An important matter of research in such systems is the efficient evaluation of recursive queries. Various strategies for processing recursive queries have been proposed (see Refs. 1–9). These strategies include evaluation methods such as naive evaluation,[10,11] seminaive evaluation,[12] query/subquery,[13–15] RQA/FQI,[16,17,18] Henschen-naqvi,[19] and the method used in PROLOG implementations. Another class of strategies, called query optimization strategies, are used to transform queries into a form that is more amenable to the existing optimization techniques developed for relational databases. Several examples of this class of approaches

*Author's current address: chen@informatik.tu-chemnitz.de or Dept. of Computer Science, Technical University of Chemnitz-Zwickau, 09107 Chemnitz, Germany.

INTERNATIONAL JOURNAL OF INTELLIGENT SYSTEMS, VOL. 11, 807–832 (1996)
 CCC 0884-8173/96/100807-26

are magic sets,[20] counting,[20] and their generalized versions.[21] In this article, we confine ourselves only to the magic-set method. 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. In the case that the program contains only one linear recursive rule besides the nonrecursive rules (such a program is called *canonical strongly linear* program,[22,23] it may be executed in a two-phase approach. In the first phase, all instantiations of the magic predicates are evaluated. In the second phase, the answers to the original query are computed in terms of the modified rules. With the help of the magic sets, the generation of irrelevant tuples is minimized. We demonstrate that the method may be improved by labeling the magic predicates to keep track of which magic rules are used and which instantiations of magic predicates are generated to reach a certain instantiation of some magic predicate. Thus, all "paths" will be explicitly recorded as the set of labels which appear in the instantiations of the labeled magic predicates. Using the path information we can separate the noncyclic data from cyclic data and suspend the computation for the cyclic data in the second phase. Then we develop an iteration process to compute all cyclic data to guarantee the completeness. In this way, we can remove much redundant work. On the one hand, since there is no cycle, in the second phase we can find an order for the corresponding instantiations of the labeled magic predicates such that in this order each instatiation is used (to restrict the computation) only once. On the other hand, in the third phase we evaluate the fixpoint only for the cyclic data without participation of the noncyclic data. In addition, we may further optimize our method for a large class of programs by separating the iteration process into two steps. In the first step of the iteration process, we compute only some answers for the first cyclic path. In the second step, we generate the remaining answers directly from the answers already found and the corresponding cyclic paths (see below). Because the cost of generating an answer is much less than that of evaluating an answer by performing algebraic operations, this improved algorithm achieves high efficiency.

In general, the idea described above cannot be directly employed to handle more complex linear recursive programs and nonlinear recursive programs. However, we can always partition the rules of a program into several subsets in such a way that for each subset the above idea can be exploited. For example, if two recursive predicates, in a linear recursive program, are not interdependent, i.e., each of them does not appear in the body of any rule defining the other, we can process them independently and apply the optimal technique for each. Otherwise, we first construct the magic rules and the modified rules for the first one which does not appear in the body of any rule defining the other, and then evaluate them using the idea described above. When the second recursive predicate is encountered during the evaluation for the first recursive predicate, we construct the corresponding (transformed) rules for it and evaluate them in the same way. Obviously, a recursive algorithm can be developed for handling any linear recursive program with the optimal idea being integrated.

The latter part of the above analysis applies to the nonlinear recursion if

we think of each different appearance of a recursive predicate (in the body of a clause) as a different predicate except that they have the same modified rules. (In fact, each different appearance corresponds to a different magic rule.) In subsection III, we will discuss this claim in detail.

All optimizations described above are based on a technique: *labeling*, with which all cyclic paths can be recorded during the execution of the first phase and be used in the subsequent iteration process.

In the next section, we briefly describe the magic-set method.[20] In Section III, we present three refined algorithms, based on the labeling technique, for handling canonical strongly linear recursion, linear recursion, and nonlinear recursion, respectively. In Section IV, a further improved algorithm is discussed, which can be employed for a larger class of programs. In Section V, we make a comparison of time complexities between our methods and some existing strategies, which shows that the elaboration of the recursive computation is worthwhile. Section VI is a short conclusion.

II. MAGIC-SET METHOD

In this section, we briefly describe the magic-set method which is necessary for explaining the key idea of our algorithms. The method is based on the idea of the sideways information-passing strategy and improves efficiency by restricting the computation to tuples that are related to the query. Essentially, the magic-set transformation does two things:[21] It creates new magic rules, and it introduces new body literals into the original rules to form modified rules. The modified database enables a bottom-up computation to generate fewer irrelevant tuples. First, we present some notation and preliminary definitions.

A. Basic Definitions

The language of a deductive database consists 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 subscripted). Predicate names will normally be denoted by the letters a, b,and c (possibly subscripted). Predicate names will normally be denoted by the letters p, q, r, and s (possibly subscripted). 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

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

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*.

B. Sideways Information-Passing Strategy

A sideways information-passing strategy (SIPS) is an inherent component of any query evaluation strategy. Informally, for a rule of a program, a SIPS represents a decision about the order in which the predicates of the rule will be evaluated, and the variables for which the values are passed from predicates to other predicates during evaluation. Intuitively, a SIPS describes how bindings passed to a rule's head by unification are used to evaluate the predicates in the rule's body. Thus, a SIPS describes how we evaluate a rule when a given set of head arguments are bound to constants. Consider, for example, the familiar *ancestor* predicate where *ancestor*(x, y) is *true* if y is an ancestor of x, and where *parent* is a base predicate, such that *parent*(x, y) is *true* if y is a parent of x:

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

The query $\leftarrow$ *ancestor*(*john*, y) retrieves all the ancestors of *john*. By unification, the variable x in the second rule is bound to *john*. We can evaluate *parent* (x, z) using this binding, and obtain a set of bindings for z. These are passed to *ancestor* to generate subgoals, which in this case have the same binding pattern. The values for z can then be said to be passed sideways from *parent* to *ancestor*.

Generalizing from this example, we may say that the basic step of sideways information passing is to evaluate a set of predicates (possibly with some arguments bound to constants), and to use the results to bind variables appearing in another predicate. It is important to stress that SIPS do not say how this information is passed. Indeed, there may be more than one way to pass the information for given SIPS. For example, SIPS do not specify whether the information is passed on a tuple-at-a-time basis, or as a set of tuples. SIPS describe only the flow of information with respect to a rule with a given set of

head arguments bound to constants, which will be generated when a top–down strategy like Prolog evaluates the rule. In general, SIPS are associated with a rule according to the query form. Different query forms, such as $ancestor^{bf}(x, y)$ and $ancestor^{fb}(x, y)$ (see below), usually have different SIPS for the same set of defining rules. The choice of one SIPS over another is guided by factors such as the current and expected size of different relations and the indexing mechanism employed.

The following definition of SIPS is borrowed from Ref. 24. It is the refining of the definition given by Ref. 21. In a given rule, two literals are called *connected* if they share a common argument. This is extended in the obvious way to connection through a chain of predicates, where each adjacent pair shares an argument.

DEFINITION 2.1. *Let $B(r)$ denote the set of body literals for a rule r, and let p^a be a special literal, denoting the head literal restricted to its bound arguments. An SIPS for a rule r is a labeled bipartite graph $G(V_1, V_2)$, where V_1 is the set of subsets of $B(r) \cup \{p^a\}$ and $V_2 = B(r)$, and which satisfies the following two conditions:*

(1) *Each arc is of the form $N \rightarrow_\chi p$, where $N \in V_1$, $p \in V_2$. The label χ stands for a nonempty set of variables which satisfies the following conditions:*
   (i) *each variable in χ appears in a member of N and in p;*
   (ii) *each literal in N is connected to p;*
   (iii) *each variable appearing in N appears in a positive literal in N, or in a bound argument position of p^a in N.*

(2) *There exists a total order of $B(r) \cup \{p^a\}$ in which:*
   (i) *p^a precedes all members of $B(r)$;*
   (ii) *any literal which is not in the graph follows every literal that is in the graph; and*
   (iii) *for every arc $N \rightarrow_\chi p$, if the literal $p' \in N$, then p' precedes p.*

Note that this definition is different from the definition given by Ref. 21 in that it preserves the allowedness under the magic-set transformation.[24]

Example 2.1. Consider the above program defining *ancestor*

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

Let the query be $\leftarrow ancestor(john, y)$ and $ancestor_1$ be a special predicate, denoting $ancestor(x, y)$ restricted to its first bound argument. An arc for the first rule might be

$$\{ancestor_1(x)\} \rightarrow_{\{x\}} parent(x, y).$$

The SIPS for the second rule is

$$\{ancestor_1(x)\} \rightarrow_{\{x\}} parent(x, y),$$

$$\{ancestor_1(x), parent(x, z)\} \rightarrow_{\{z\}} ancestor(z, y).$$

C. The Adorned Program

In terms of the SIPS for a program, we can adorn the program. This is done by annotating predicates with a character string, which is called *adornment*. 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 (SIPS). 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.

Example 2.2. The following is the adorned rule set corresponding to the familiar *ancestor* predicate for the SIPS of example 2.1:

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

D. The Magic-Set Algorithm

Now we consider the magic-set transformation of a program. Magic-set algorithms are program transformations that take an initial adorned program and query and return a modified program which gives the same answers for a particular query as the initial program. Using the bottom–up method, the transformed program generates fewer irrelevant tuples than the initial program. There have been several magic-set algorithms reported in the literature.[20,21,24]

A common trait among these algorithms is that, based on the adornment of the head and body literals, some new positive literals are introduced into the body of rules, and new rules are added to the program which define these literals. The new literals are called magic literals and are related to the existing literals of the program as follows. For a positive adorned predicate p^a with l bound argument positions where $l > 0$, define the magic predicate of p^a to be the predicate whose name is the predicate name of p^a prefixed with "*Magic–*" and whose arity is l. The new rules defining the magic predicates are called magic rules.

The following is a magic-set algorithm.[24] The input of the algorithm consists of the adorned query, $q^a(\mathbf{v})$, the adorned program, $\mathbf{P}^a$, and the corresponding set of SIPS, $\mathbf{S}^a$. The output of the algorithm is the modified version of the adorned program plus the magic rules, $\mathbf{P}^{am}$, and the seed, $magic\text{–}q^a(\mathbf{v}^b)$, where $\mathbf{v}^b$ is the vector of arguments which are bound in the adornment a of q. In the algorithm, the following definitions are used:

bodyLit(N) denotes the conjunction of body literals of a rule r^a in N, where N is the tail of an arc $\mathrm{N} \rightarrow_\chi p$ in the SIPS for r^a.
$magic(r^a(\mathbf{v}))$ returns a literal $magic\text{–}r^a(\mathbf{v}^b)$.

function *magic-set-transformation* $(q^a(\bar{v}), \mathbf{P}^a, \mathbf{S}^a)$
  $\mathbf{P}^{am} := \varnothing$
  for each rule r^a in $\mathbf{P}^a$ of the form $p \leftarrow p_1, p_2, \ldots, p_m$ **do**
    add the rule $p \leftarrow magic(p), p_1, p_2, \ldots, p_m$ to $\mathbf{P}^{am}$
    for each arc $N \rightarrow_\chi r$ in the SIPS associated with r^a **do**
      if p is in N **then**
        add the rule $magic(r) \leftarrow magic(p), bodyLit(N)$ to $\mathbf{P}^{am}$
    else add the rule $magic(r) \leftarrow bodyLit(N)$ to $\mathbf{P}^{am}$
return $(magic\text{-}q^a(\mathbf{v}), \mathbf{P}^{am})$

Example 2.3. 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 magic rules and modified rules are:

$$ancestor^{bf}(x, y) \leftarrow magic\text{-}ancestor^{bf}(x), parent(x, y),$$

$$ancestor^{bf}(x, y) \leftarrow magic\text{-}ancestor^{bf}(x), parent(x, z), ancestor^{bf}(z, y),$$

$$magic\text{-}ancestor^{bf}(z) \leftarrow magic\text{-}ancestor^{bf}(x), parent(x, z),$$

$$magic\text{-}ancestor^{bf}(john).$$

III. REFINED ALGORITHMS

In this section, we describe our refined algorithms. For ease of exposition, we first present the query graph concept in subsection III-A, which may facilitate the clarification of the key of the refinement. In subsection III-B and III-C, we describe our refined algorithms for the cases of canonical strongly linear and linear recursions, respectively. In subsection III-F, we present another version for the case of nonlinear recursion.

A. Query Graph

We can associate a directed graph to a (recursive) query form with respect of a canonical strongly linear program which contains only one linear recursive rule besides the nonrecursive rules. The nodes of the graph correspond to tuples of constants; in particular, the source node corresponds to the tuple of constraints in the query goal. The other nodes (and incoming arcs) are obtained by retrieving tuples from database **D** via a goal composed by a conjunction of base predicates, using restrictions from the tuples corresponding to previously generated nodes. Essentially, such a graph is introduced to describe the constant

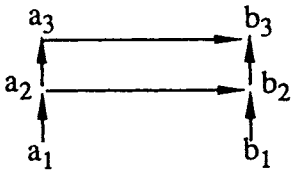

Figure 1. A simple graph G.

propagation and the construction of derivation trees which happens in a top–down evaluation. Before the formal definition, let us consider the following example:

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

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

$$p(a_1, a_2), p(a_2, a_3)\ q(b_3, b_2), q(b_2, b_1), r(a_3, b_3).$$

With respect to the query $?\text{-}s(a_1, y)$, this program can be graphically interpreted as follows. We define a directed graph $G = (V, A)$, where the set of nodes is the set of constraints appearing in the ground terms (facts) and the set of arcs A consists of the union of the free following disjoint sets:

$$A_1 = \{(x, y) \mid p(\text{x, y}) \text{ is } \textit{true} \text{ in the program}\};$$

$$A_2 = \{(x, y) \mid r(\text{x, y}) \textit{ is true} \text{ in the program}\};$$

$$A_3 = \{(x, y) \mid q(\text{x, y}) \text{ is } \textit{true} \text{ in the program}\}.$$

The graph G is shown in Figure 1. In fact, the query graph is the generalization of such a graph. 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) 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) 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 some 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_{j+1} is the adornment of $q(\mathbf{v}_1)$ and $\mathbf{c}'_j = \mathbf{v}_1(a_{j+1})\theta$.

(c) If $[a_i, \mathbf{c}_j]$ is in N_u and there exists both a nonrecursive rule in $\mathbf{P}$, say $q(\mathbf{v})$ $:\text{-}p_1, \ldots, p_n$, 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) 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 clause, $q(\mathbf{v}_0)$, 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, $G_u = (N_u, A_u)$, $G_f = (N_f, A_f)$, and $G_d = (N_d, A_d)$, 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.

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

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

The query graph w.r.t. query ?-s(c, y) is shown in Figure 2. 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.[25]

THEOREM 3.1. *Let G_q be the query graph of the query form 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* $[a_i^-, \mathbf{c}]$ *in G_q, if $q(\mathbf{b}, \mathbf{c})$ is an answer to the query* ?-$q(\mathbf{b}, \mathbf{y})$.

Proof. See the appendix of Ref. 25. ∎

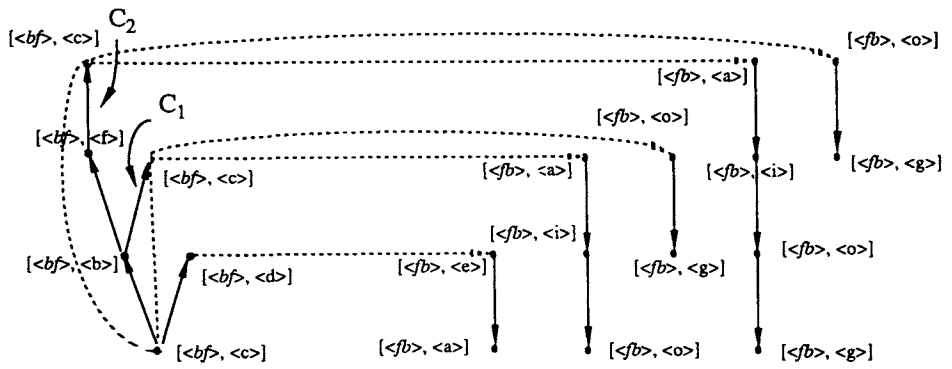

Figure 2. Query graph of Example 3.1.

In Figure 2 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: [⟨*bf*⟩, ⟨c⟩] → [⟨*bf*⟩, ⟨b⟩] → [⟨*bf*⟩, ⟨c⟩], we have two paths in A_d: [⟨*fb*⟩, ⟨a⟩] → [⟨*fb*⟩, ⟨i⟩] → [⟨*fb*⟩, ⟨o⟩] and [⟨*fb*⟩, ⟨o⟩] → [⟨*fb*⟩, ⟨g⟩] with each corresponding to a traversal along the cyclic path C_1. Similarly, for the other cyclic path C_2 in A_u: [⟨*bf*⟩, ⟨c⟩] → [⟨*bf*⟩, ⟨*b*⟩] → [⟨*bf*⟩, ⟨f⟩] → [⟨*bf*⟩, ⟨c⟩], we have [⟨*fb*⟩, ⟨a⟩] → [⟨*fb*, ⟨i⟩] → [⟨*fb*⟩, ⟨o⟩] → [⟨*fb*⟩, ⟨g⟩] and [⟨*fb*⟩, ⟨o⟩] → [⟨fb⟩, ⟨g⟩] in A_d. In addition, from the graph shown in Figure 2, 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 {[⟨*fb*⟩, ⟨a⟩], [⟨*fb*⟩, ⟨i⟩], [⟨*fb*⟩, ⟨o⟩], [⟨*fb*⟩, ⟨g⟩]}. The set of nodes (answer tuples) of the paths (in A_d) corresponding to C_2 (in A_u) is also {[⟨*fb*⟩, ⟨a⟩], [⟨*fb*⟩, ⟨i⟩], [⟨*fb*⟩, ⟨o⟩], [⟨*fb*⟩, ⟨g⟩]}. 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 nonrecursive 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* **P** *be a canonical strongly linear program with the property that for the recursive rule r in* **P** *and any query form the unbound variables of the head predicate appear neither in the nonrecursive predicates with some variables bound to constants nor in the nonrecursive predicates which are directly or indirectly (but not via recursive predicates) correlated with the nonrecursive predicates having bound arguments. Let* G_q *be the query graph associated with the query form q with respect to* **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.

Proof. Let $q(\mathbf{x}, \mathbf{y})$:- $p_1, \ldots, q(\mathbf{s}, \mathbf{t}), \ldots, p_n$ be a recursive rule (with the rule number r) in **P**, where **x**, **v**, **s**, and **t** are all variable tuples, and **Z** be the set of all answer tuples with respect to the query ?-$q(\mathbf{c}, \mathbf{y})$. We define the function $f(\mathbf{v}, \mathbf{z})(\mathbf{z} \in \mathbf{Z})$ as follows. If there exists a substitution θ for the variables appearing in r such that $\mathbf{v}_x\theta = \mathbf{v}$, where $\mathbf{v}_x$ are all variables which appear simultaneously in the predicates before $q(\mathbf{s}, \mathbf{t})$ and the predicates after it (but contain no variables in **t** and **y**), $\mathbf{t}\theta = \mathbf{z}$, $\mathbf{y}\theta = \mathbf{z}'$, and every p_i $(i = 1, \ldots, n)$ is *true*, then $f(\mathbf{v}, \mathbf{z}) = \mathbf{z}'$. In general, f is a multivalue 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 **P**, since all predicates occuring after a recursive predicate are only correlated with the recursive predicate 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 ith call are completely determined by the answer tuples for the recursive predicate of $(i + 1)$th call. 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 **z** is an answer tuple whose corresponding bound tuple is the start node of the cyclic path. Because P_i and P_j have the same start node, they have also the same **z**. Thus, the set of answer tuples corresponding to is the same as that corresponding to P_j. ∎

The theorem is important to the optimization algorithm described in Section IV.

Finally, we note that essentially the magic-set method works in two phases. The first phase consists of determining all node 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 $(n_{i'}, n_{j'})$ such that $(n_i, n_{j'})$ is in

A_u and $(n_j, n_{j'})$ is in A_d. The so-obtained pair, in turn, is used to derived other pairs. The magic set is used to make this derivation more efficient.

B. Refined Algorithm for Canonical Strongly Linear Recursion

Below is the refinement of the magic-set method for a simple case that in a program there is exactly one linear recursive rule besides the nonrecursive rules (such a program is called a canonical strongly linear program.[22,23] 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 (of the magic predicate) already produced, but also the cyclic and noncyclic paths of G_u explicitly. 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 evaluates 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) is 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 nonindex fields. Before the use of the argument is explained, we first define some relevant concepts.

DEFINITION 3.1. *Let magic_*$q^a(y)$:- *magic_*$p^a(z_1)$, $p_1(z_1, z_2)$, . . . , $p_n(z_n, y)$ *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* 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 3.2. *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, . . . , ith, . . . immediate descendent node by* 0.1, 0.2, ... *The jth 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 descending 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 descent node. We use them only for identifying the instantiations of a magic predicate.

DEFINITION 3.3. *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, . . . , m].

DEFINITION 3.4. *Let mr: magic_q^a(y) :- . . . . . be a magic rule. The indexed version of magic_q^a(y) is of the form: ind_magic_q^a(ind,* **v**), *where ind is a two-element list: (mrs, label).*

As 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 → 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.

Now consider the adorned rule:

$$r_i: q^a(\chi) :- p_1^{a_1}(v), \ldots, p_n^{a_n}(v_n).$$

We generate a magic rule defining *ind_magic_$p_j^{a_j}$* if the SIPS associated with r_i contains an arc N → p_j. The head of the magic rule is *ind_nagic_$p_j^{a_j}$*((*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*), χ^b) to the body. Otherwise, we add the built-in predicates (*mrs* = " ") and (*label* = 0) to the body. Using the pair (*mrs, label*), we cannot only label each instantiation of a magic predicate, but can also record each path. An instantiation with index (mr_1 . . . .mr_i .mr_{i+1}, i_1 . . . .i_i.i_{i+1}) will have an immediate precedent instantiation with index (mr_1 . . . .mr_i .mr_{i+1}, i_1 . . . .i_i.i_{i+1}) will have an immediate precedent instantiation with index (mr_1 . . . .mr_i, i_1 . . . .i_i).

The indexed magic rule corresponding to N → p_j with respect to r_i is:

$$mr_{i1}: ind\_magic\_p_i^a((mrs.mr_{i1}, label.num_{v_i^b}), v_i^b) \\ :- ind\_magic\_q^a((mrs, label), \chi^b), \ldots, p_{i-1}^{a_{i-1}}(v_{i-1}).$$

The rules are modified in the same way as the magic-set method.[21]

Example 3.2. Continuing our running example, we present below the rewritten rules, with respect to the query ?-s(c, y), produced by the above method.

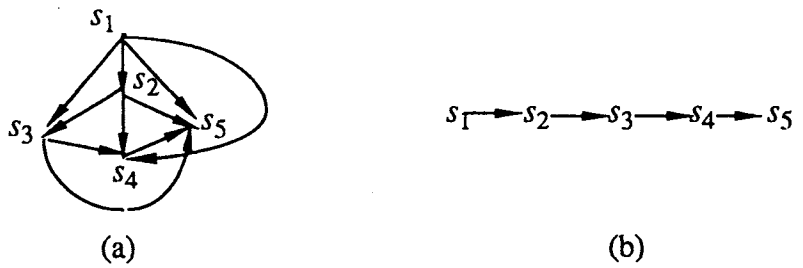

Figure 3. Magic graph and topological order.

$s(x, y)$:- $magic\_s^{bf}(x)$, $r(x, y)$,

$s(x, y)$:- $magic\_s^{bf}(x)$, $p(x, z)$, $s(z, w)$, $q(w, y)$,

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. For example, assume that $s_1 = ind\_magic\_s^{bf}(l_1, v_1)$ and $s_2 = ind\_magic\_s^{bf}(l_2, v_2)$ are two indexed magic predicates evaluated in the first phase, where l_1 and l_2 are indexes and v_1 and v_2 are vectors of values. If v_1 is equal to v_2 and s_1 is on the same path as s_2 (by checking the indexes l_1 and l_2 we can know that whether s_1 and s_2 are on the same path), then path from s_1 to s_2 is cyclic. In the first phase, we use the seminaive method to compute the indexed magic predicates, thereby checking whether a cycle appears. If it is the case, we store the cycle. In the second phase, we evaluate the modified rules with a naive or seminaive strategy in terms of the set of the instantiations of the magic predicates. Because we have eliminated the cyclic points of each cyclic path such that the resultant set contains no cyclic data, we need not check the fixpoint in the second phase if we arrange the remaining set of the instantiations of the magic predicates carefully. For example, if $s_1, s_2, \ldots, s_5$ are five instantiations of the indexed magic predicate produced in the first phase for a canonical strongly linear recursive program and the arcs between them are shown in Figure 3(a). [We obtain the arcs by checking the label of each $s_i (i = 1, \ldots, 5)$. Then we can find a topological order for them with the property that all descendants of a node s_i are before s_i in the order, shown in Fig. 3(b)]. In this order the modified rules are evaluated only in one scan, i.e., only one iteration is required. In this way, all answers for these noncyclic paths are produced, but no redundancy will be done. For large data sets, this improvement is significant. We guarantee the completeness by using an iteration process over all cyclic paths. The following is the refined magic-set algorithm.

Algorithm *refined-magic-set-method-1*
phase 1. (*computing the magic sets*)
$S_{mg} := \{seed\}$, $S_{\Delta\text{-}mg} := S_{mg}$;
while $S_{\Delta\text{-}mg} \neq \emptyset$ do
$new\text{-}S_{mg}$:= new_answers evaluated in terms of magic rules and $S_{\Delta\text{-}mg}$;
$S_{\Delta\text{-}mg} := new\text{-}S_{mg} - S_{mg}$;
$S_{mg} := S_{mg} \cup S_{\Delta\text{-}mg}$
end

phase 2. (*computing answers along the acyclic paths*)
$S_a := \emptyset$;
S_{mg}' := the resultant set obtained by removing all cyclic points from S_{mg};
repeat
$new\text{-}S_a$:= new_answers evaluated in terms of modified rules, S_{mg}', S_a, and **D**;
$S_a := S_a \cup new\text{-}S_a$
until no new answers are produced

phase 3. (*computing answers along the cyclic paths*)
$S_{cyclic\text{-}paths}$:= cyclic_paths and the paths, of which each is from the start node of a cyclic path to the source node;
repeat
for each $path \in S_{cyclic\text{-}paths}$ **do**
$S_a := S_a \cup$ new_answers evaluated in terms of modified rules, *path*, S_a and **D**
until no new answers are produced

The algorithm works in three phases. The first phase consists of determining all instantiations of the magic predicates (magic set). For efficiency, this phase is implemented using the seminaive approach. As a consequence, $S_{\Delta\text{-}mg}$ is introduced to store the new instantiations (of the magic predicate) produced within every loop. At the end (of the first phase), the evaluated magic set is stored in S_{mg}. $new\text{-}S_{mg}$ 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 (instantiations of the magic predicate) on the acyclic paths. Then, in the topological order, all modified rules are evaluated only in one scan. 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 each cyclic path is carried out. Here, $S_{cyclic\text{-}paths}$ contains all cyclic paths and $S_{remaining\text{-}paths}$ contains those paths, each of which is from the source node to the start node of a cyclic path.

PROPOSITION 3.1. *The above algorithm complete over the domain of function-free Horn clauses.*

Proof. Note that if we do not remove the cyclic points of each cyclic path from the set of the instantiations of magic predicates, the second phase of the algorithm will produce all answers.[21] Due to the absence of the cyclic points for each cyclic path, all answer tuples of the form: $f(\mathbf{v}_{n+1}, f(\mathbf{v}_n, \ldots f(\mathbf{v}_1, \mathbf{z}) \ldots))$, where f is the function defined in the proof of Theorem 3.2, $\mathbf{z}$ is an answer tuple whose corresponding bound tuple is the start node of a cyclic path, and $\mathbf{v}_i (i = 1, \ldots, n + 1)$ is a constant tuple bound to the variables which simultaneously appear before and after the recursive predicate, and all answers relying on them will not be produced by the second phase of the algorithm. In the third phase of the algorithm, we will evaluate all answer tuples of the form: $f(\mathbf{v}_{n+1}, f(\mathbf{v}_n, \ldots f(\mathbf{v}_1, \mathbf{z}) \ldots))$ along each cyclic path. Since $S_{cyclic\text{-}paths}$ contains also all paths, each of which is from the start node of a cyclic path to the source node, the answers relying on these answer tuples will be evaluated in the third phase. Therefore, the above algorithm is complete over the domain of canonical strongly linear recursive programs. ■

Example 3.3. Here we trace the steps of the above algorithm for our running example.

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 may record all paths during the execution:

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

The last two paths are cyclic.

phase 2:

D
$S_a = \varnothing$
$S_{mg}' = \{magic\_s^{bf}(d), magic\_s^{bf}(b), magic\_s^{bf}(f)\}$

step 1: $S_a = \{s(d, e)\}$

phase 3:

D
$S_a = \{s(d, e)\}$

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

step 1:
for *loop 1*: S_a = {s(c, a), s(b, i), s(c, o)}
for *loop 2*: S_a = {s(c, a), s(b, i), s(c, o), s(f, i), s(b, o), s(c, g)}

step 2:
for *loop 1*: S_a = {s(c, a), s(b, i), s(c, o), s(b, g), s(f, i), s(b, o), s(c, g)}
for *loop 2*: S_a = {s(c, a), s(b, i), s(c, o), s(b, g), s(f, i), s(b, o), s(c, g), s(f, g)}

From the example above, we can see that no redundant answers are computed. In fact, we can further improve the efficiency of the algorithm for a large class of programs with the property described in Theorem 3.2. In the next section, we describe the optimization method.

C. Refined Algorithm for Linear Recursion

Now we consider linear recursive programs which contain several recursive predicates. We differentiate between two cases.

(1) Each two recursive predicates are not interdependent, that is, a predicate does not appear in the body of a clause defining another recursive predicate.
(2) There are at least two recursive predicates which are interdependent.

In the first case, we can handle each recursive predicate consecutively using the technique described in subsection III-B. For example, consider the following program:

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

In this program, s_1 and s_2 are not interdependent. With respect to the query ?- s(c, y) (where c is a constant), we first construct the magic rules and modified rules for the first clause of the program as follows:

$ind\_magic\_s^{bf}$((, 0), c),
$s(x, y)$:- $magic\_s^{bf}(x)$, $s_1(x, z)$, $s_2(z, y)$.

In the evaluation of the transformed program above, $s_1(x, z)$ and $s_2(z, y)$ (with the first argument bound to a constant, respectively) will be evaluated consecutively. For the evaluation of $s_1(x, z)$, the corresponding indexed magic rules and the modified rules will be constructed:

$ind\_magic\_s_1^{bf}((\ ,0), c)$,
$mr: ind\_magic\_s_1^{bf}((mrs.mr, label.num_z), z) :- ind\_magic\_s_1^{bf}((mrs, label), x), p_1(x, z)$,
$s_1(x, y) :- magic\_s_1^{bf}(x), p_1(x, z), s_1(z, w), q_1(w, y)$.

Then we compute them in a three-phase approach. That is, in the first phase all instantiations of the magic predicate are produced by computing the corresponding magic rule; in the second phase some answers are evaluated for the noncyclic paths which are arranged in a topological order; in the third phase the least fix point is evaluated for each cyclic path.

After $s_1(x, z)$ (with the first argument bound to c) is evaluated, the first argument of $s_2(z, y)$ will be instantiated to a constant (or a set of constants). Obviously, we can evaluate it in the same way as $s_1(x, z)$.

In the second case, the technique described in subsection III-A can also be used. However, we need to arrange the computation in a different way. As an example, consider the following example:

$s_1(x, y) :- r_1(x, y)$,
$s_1(x, y) :- p_1(x, z), s_1(z, w), q_1(w, y_1), s_1(y_1, y)$,
$s_2(x, y) :- r_2(x, y)$,
$s_2(x, y) :- p_2(x, z), s_2(z, w), q_2(w, y)$.

With respect to the query $?\text{-}s_1(c, y)$, where c is a constant, we first construct the magic rules and modified rules in terms of the first two clauses as below:

$ind\_magic\_s_1^{bf}((\ ,0), c)$,
$mr: ind\_magic\_s_1^{bf}((mrs.mr, label.num_z)\ , z) :- ind\_magic\_s_1^{bf}((mrs, label), x), p_1(x, z)$,
$s_1(x, y) :- magic\_s_1^{bf}(x), r_1(x, y)$,
$s_1(x, y) :- magic\_s_1^{bf}(x), p_1(x, z), s_1(z, w), q_1(w, y_1), s_1(y_1, y)$.

Then we execute the above rules in a three-phase approach. During the evaluation of each phase, when $s_2(x, y)$ is first met, we need to construct the magic rules and modified rules in terms of the clauses defining $s_2(x, y)$ in the way above. Similarly, we can use the three-phase approach to evaluate the corresponding rules each time $s_2(x, y)$ is encountered.

In terms of the above analysis, we have the following algorithm for linear recursive programs.

Algorithm *evaluation-for-linear(query)*
construct the indexed magic rules *mag-rules* and the modified rules *mod-rules*
in terms of the clauses defining *query* if they do not exist;
construct the corresponding *seed*;
for each $r \in$ *mag-rules* **do**
evaluate r in the bottom-up manner (in terms of the database **D**)
(if a derived predicate q in the body of r is encountered,
call *evaluation-for-linear(q)*);
determine the topological order for the noncyclic paths;

assume ($magic\_s_1$, $magic\_s_2$, . . . , $magic\_s_m$) is the corresponding topological order.

for i = 1 to m **do**
  evaluate each modified rule r in terms of $magic\_s_i$ and **D**
  (if a derived predicate q in the body of r is encountered,
  call *evaluation-for-linear*(q));
  $S_a := S_a \cup$ new answers produced in this step;
for each path $P \in S_{cyclic\text{-}paths}$ **do**
  repeat
    evaluate each modified rule r along P
    (if a derived predicate q in the body of r is encountered,
    call *evaluation-for-linear*(q));
    $S_a := S_a \cup$ new answers produced in this step;
  until no new answers are produced
for each path $P \in S_{remaining\text{-}paths}$ **do**
    evaluate each modified rule r along P
    (if a derived predicate q in the body of r is encountered,
    call *evaluation-for-linear*(q));
    $S_a := S_a \cup$ new answers produced in this step.

D. Refined Algorithm for Nonlinear Recursion

In this subsection, we present the refined method for handling nonlinear recursive queries. For ease of representation, we assume that for the following nonlinear recursive rule:

$$q(x, y) \text{ :- } p_1(x, z), q(z, z_1), p_2(z_1), \ldots, q(z_{i-1}, z_i), \ldots, q(z_{n-1}, z_n), p_{n+1}(z_n)$$

a set of magic rules will be constructed by using the magic-set algorithm:

$m\text{-}r_1$: $magic\_q^{bf}(z)$:- $magic\_q^{bf}(x)$, $p_1(x, z)$,
$m\text{-}r_2$: $magic\_q^{bf}(z_1)$:- $magic\_q^{bf}(x)$, $p_1(x, z)$, $q(z, z_1)$, $p_2(z_1)$,
. . . . . .
$m\text{-}r_{i:}$ $magic\_q^{bf}(z_{i-1})$:- $magic\_q^{bf}(x)$, . . . , $p_i(z_{i-1})$,
. . . . . .
$m\text{-}r_n$: $magic\_q^{bf}(z_{n-1})$:- $magic\_q^{bf}(x)$, . . . , $p_n(z_{n-1})$.

Here each ith magic rule corresponds to the ith appearance of the recursive predicate. For example, for the Horn-clause

$$s(x, y) \text{ :- } p(x, u), s(u, v), q(v, w), s(w, z), t(z, y),$$

the following magic rules will be constructed (if the query is of the form: ?- s(c, y)):

$magic\_s^{bf}(u)$:- $magic\_s^{bf}(x)$, $p(x, u)$,
$magic\_s^{bf}(w)$:- $magic\_s^{bf}(x)$, $p(x, u)$, , $s(u, v)$, $q(v, w)$.

Roughly speaking, the algorithm works in a two-steps approach. In the first step all instantiations of the magic predicate are evaluated. In the second step, all answers are evaluated in terms of the instantiations and the facts available. Note that in order to instantiate the magic predicate defined by the nth magic rule, all those answers will have to be evaluated first, which satisfy all ith appearances of the recursive predicate ($i = 1, 2, \ldots, n - 1$) in the rule. Similarly, in order to instantiate the magic predicate defined by the $(n - 1)$th magic rule, all answers which satisfy all jth appearances of the recursive predicate ($j = 1, 2, \ldots, n - 2$) in the corresponding rule will have to be evaluated beforehand, and so on. Based on the analysis, we give the algorithm for handling nonlinear recursion as follows.

procedure evaluation(m-r_i, *init*) {m-r_i: the ith magic rule; *init*: *Seed*}
  if m-$r_i = \varnothing$ **then** return {*init*}
step 1: **repeat**
    temp := *init*;
    execute evaluation(m-r_{i-1}, *init*);
    evaluate some instantiations in terms of m-r_i and the facts available;
    init := new_instantiations - *temp*;
  until no new instantiations are produced

step 2:
  phase 1. (*computing answers along the acyclic paths*)
    $S_a := \varnothing$;
    S_{mg}' := the resultant set obtained by removing all cyclic points from S_{mg};
    repeat
      new-S_a := new_answers evaluated in terms of modified rules, S_{mg}', S_a, and **D**; (If a recursive predicate is encountered, call evaluation() with the corresponding m-r and initial values)
      $S_a := S_a \cup$ *new-*S_a
    until no new answers are produced
  phase 2. (*computing answers along the cyclic paths*)
    $S_{cyclic\text{-}paths}$:= cyclic_paths and the paths, of which each is from the start node of acyclic path to the source node;
    repeat
      for each *path* $\in S_{cyclic\text{-}paths}$ **do**
      $S_a := S_a \cup$ new_answers evaluated in terms of modified rules, *path*, S_a and **D**
        (If a recursive predicate is encountered, call evaluation() with the corresponding m-r and initial values.)
    until no new answers are produced

As with the linear recursion algorithm, the second step consists of two phases. In the first phase, all answers along each acyclic path are evaluated. In the second phase, we further handle each cyclic path to compute the remaining

answers. An important difference is that in order to evaluate new answers the recursive call to evaluation() will be performed when a recursive predicate is encountered.

IV. FURTHER IMPROVEMENT

As described in Theorem 3.2, the set of nodes on the paths in A_d corresponding to a cyclic path in A_u is the same as the set of nodes on the paths in A_d corresponding to the other cyclic path in A_u. This feature may be employed to optimize the evaluation of queries. We further separate the third phase into two steps. In the first step, we compute some answers for the first cyclic path. In the second step, we generate the remaining answers directly according to the answers already found and the corresponding cyclic paths. Because the cost of generating an answer is much less than that of evaluating an answer by performing algebraic operations, we may achieve high efficiency in this way. For example, in the example above we may directly generate s(f, i) for the second cyclic path in terms of $magic\_s^{bf}(f)$ and s(b, *i*) (which has been produced along the first cyclic path) instead of evaluating it by performing algebraic operations. Similarly, we may also generate s(b, o) for the second cyclic path in terms of $magic\_s^{bf}(\underline{b})$ and s(c, $\underline{o}$) (which has also been produced along the first cyclic path). In general, evaluating answers by performing algebraic operations requires access to the external storage or search of large relations. But the "generating" operations happen always in main memory and require only access to small data sets (i.e., the answers already evaluated or generated on the other cyclic paths). Therefore, the generation of answers is much more efficient than the evaluation of answers.

Algorithm *refined-magic-set-method-2*
phase 1. (*computing the magic sets*)
$S_{mg} := \{seed\}$, $S_{\Delta\text{-}mg} := S_{mg}$;
while $S_{\Delta\text{-}mg} \neq \emptyset$ **do**
$new\text{-}S_{mg}$:= new_answers evaluated in terms of magic rules and $S_{\Delta\text{-}mg}$;
$S_{\Delta\text{-}mg} := new\text{-}S_{mg} - S_{mg}$;
$S_{mg} := S_{mg} \cup S_{\Delta\text{-}mg}$
end

phase 2. (*computing answers along the acyclic paths*)
$S_a := \emptyset$
S_{mg}' := the resultant set obtained by removing all cyclic points from S_{mg};
repeat
$new\text{-}S_a$:= new_answers evaluated in terms of modified rules, S_{mg}', S_a, and **D**;
$S_a := S_a \cup new\text{-}S_a$
until no new answers are produced

phase 3. (*computing answers along the first cyclic path*)

$S_{first\text{-}cyclic\text{-}paths}$:= the first cyclic_path
repeat
S_a := S_a ∪ new_answers evaluated in terms of modified rules, $S_{first\text{-}cyclic\text{-}paths}$, S_a, and **D**
until no new answers are produced

phase 4. (*generating the remaining answers along the cyclic paths*)

$S_{cyclic\text{-}paths}$:= cyclic_paths and the paths, of which each is from the start node ofa cyclic path to the source node;
repeat
for each $path_i$ and $path_j \in S_{cyclic\text{-}paths}$ **do**
$S_a := S_a \cup$ new_answers generated in terms of $path_j$ and the answers evaluated or generated for $path_i$
until no new answers are produced
begin
generating all answers to the initial goal in terms of S_a and paths, of which each is from the start node of a cyclic path to the source node
end

Example 4.1. Given the rules and facts as in the running example. Let q(c, x) be the query. The refined algorithm will produce the following answers:

answers evaluated in phase 2: s(d, e);
answers evaluated in phase 3: s(c, a), s(b, i), s(c, o), s(b, g);
answers generated in phase 4: s(f, i), s(b, o), s(c, g), s(f, g).

Obviously, the algorithm above can be used to improve the efficiency for handling nonlinear recursion.

V. TIME COMPLEXITY

Now we consider the time complexity of the refined algorithms. We first consider the case of linear recursion. To simplify the description of the results of the analysis, we assume that each cyclic path has the same length (by "length" we mean the number of instantiations of a magic predicate on a cyclic path) and along each cyclic path the number of new answers got by the refined algorithm from an initial value or an evaluated answer in one step is e. Therefore, if each cyclic path has the length m and the number of iteration over a cyclic path is l, then the time complexity of the third phase of *refined-magic-set-method-1* is on the order of

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

where n is the number of the cyclic paths and C represents the cost of evaluating an answer in the iteration step. In the worst case, C is the elapsed time of a

read access to the external storage, i.e., each evaluation in the step requires an I/O.

In the third phase of *refined-magic-set-method-2*, $(e^{ml} - 1)/(e - 1)$ answers are evaluated on the assumption above. The remaining answers for each cyclic path are all generated in the fourth phase. In comparison with the cost of evaluating an answer (by algebraic operations; *join*, *selection*, *projection*, ...), the cost of generating an answer is very little such that we cannot take it into account (In practice, the time complexity of a computation mainly depends on the number of accesses to the external storage which in turn depends on the number of the relations participating in the computation and their cardinalities). Let δ be the cost of generating an answer in the generation phase, then the running time for the third and fourth phase of *refined-magic-set-method-2* is

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

Since $\delta \ll C$, the saving on time is significant. If $\delta/C \leq n/e^{ml}$, $(n - 1) \cdot (e^{ml} - 1) \cdot \delta$ is less than some constant, and therefore the time complexity of the third and fourth phase of *refined-magic-set-method-2* is $O(e^{ml} C)$. Therefore, *refined-magic-set-method-2* may reduce the worst-case time complexity of *refined-magic-set-method-1* by a factor n, the number of the cyclic paths, if we do not take account of the cost of generating an answer.

Below we further consider the case of double recursion (i.e., the body of a recursive rule contains two recursive predicates) and suppose that the revised algorithm described in subsection III-C is employed. For ease of representation, assume that each cyclic path has the same length. Let n_1 be the number of the cyclic paths associated with the first appearance of the recursive predicate and l_1 be the length of such a cyclic path during the execution. Let n_2 be the number of the cyclic paths associated with the second appearance of the recursive predicate and l_2 be the length of such a cyclic path (l_2 is the number of instantiations evaluated only in terms of m-r_2), then we have the following recurrence relation with respect to evaluation() (which was given in subsection III-C):

$$\begin{cases} C_1^k = E_2^k + A_2^k + n_2 \cdot l_2 \cdot C_2^k \\ C_2^k = A_1^{k+1} + n_1 \cdot l_1 \cdot C_1^{k+1} \end{cases}$$

where C_1^k represents the amount of time spent in kth call to evaluation(m-r_2).

E_2^k represents the amount of time spent in the execution of the first step by the kth recursive call,
A_2^k represents the amount of time spent in the execution of the first phase of the second step by the kth recursive call,
A_1^{k+1} represents the amount of time spent in the computation over all acyclic paths associated with the first appearance of the recursive predicate by $(k + 1)$th recursive call, and
C_2^k represents the amount of time spent in an iteration step during the execution of the second phase of the second step by the kth recursive call.

The first equation is self-explanatory. The second equation reflects the fact that in the worst case each iterative step along a cyclic path associated with the second appearance of the recursive predicate includes all computations over the acyclic and cyclic paths (associated with the first appearance of the recursive predicate) which are generated by an iterative computation over m-r_1 with the corresponding instantiation as an initial value.

For simplicity we assume that

$$A_1 = A_1^k = A_1^{k+1} (k = 2, \ldots, \gamma - 1),$$

$$A_2 = A_2^k = A_2^{k+1} (k = 1, \ldots, \gamma - 1) \text{ and}$$

$$E_2 = E_2^k = E_2^{k+1} (k = 1, \ldots, \gamma - 1),$$

where γ is the number of times of recursive calls.

Then we will have

$$C = C_1^1 = (E_2 + A_2) \frac{(n_1 \cdot l_1)^\gamma \cdot (n_2 \cdot l_2)^{\gamma-1}}{(n_1 \cdot l_1) \cdot (n_2 \cdot l_2) - 1} + n_2 \cdot l_2 \cdot A_1 \frac{(n_1 \cdot l_1)^{\gamma-1} \cdot (n_2 \cdot l_2)^{\gamma-1} - 1}{(n_1 \cdot l_1) \cdot (n_2 \cdot l_2) - 1}.$$

If we use the optimal method described Section N, we will have the following recurrence relation

$$\begin{cases} C_1^k = E_2^k + A_2^k + l_2 \cdot C_2^k \\ C_2^k = A_1^{k+1} + l_1 \cdot C_1^{k+1} \end{cases}$$

The reason for this is that for each n_1 cyclic paths associated with the first appearance of the recursive predicate only one path is evaluated by the optimal method (the remaining answers for the other $n_1 - 1$ cyclic paths are generated) and for each n_2 cyclic paths associated with the second appearance of the recursive predicate only one path is evaluated as well (the remaining answers for the other $n_2 - 1$ cyclic paths are generated). Therefore, n_1 and n_2 do not appear in the above recurrence relation.

Using the above assumption, we will have

$$C = C_1^1 = (E_2 + A_2) \frac{l_1^\gamma \cdot l_2^\gamma - 1}{l_1 \cdot l_2 - 1} + l_2 \cdot A_1 \frac{l_1^{\gamma-1} \cdot l_2^{\gamma-1} - 1}{l_1 \cdot l_2 - 1}$$

Therefore, the optimal algorithm will reduce the worst-case time complexity of the algorithm described in subsection III-C by a factor $(n_1 \cdot n_2)^\gamma$ (in the case of double recursion).

VI. CONCLUSION

In this article, 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 recording the cyclic paths during the execution of the first phase of the magic-set algorithm and forbidding the computation for the cyclic data in the second phase to avoid the redundant evaluation. We

guarantee the completeness of the method by using an iteration process which evaluates all answers along each cyclic path. Further, we optimize the evaluation for a large class of programs by separating the iteration process into two steps. In the first step of the iteration process, we compute only some answers for the first cyclic path. In the second step, we generate the remaining answers directly according to the answers already found and the corresponding cyclic paths.

All optimizations reported in this article are based on a technique called *labeling*, which enables us to record all cyclic paths during the execution of the first phase and use them in the iteration process.

We have given a brief comparison of the time complexity which shows that in the case of linear recursion *refined-magic-set-method-2* may reduce the worst-case time complexity of *refined-magic-set-method-1* by a factor n, the number of the cyclic paths, if we do not take account of the cost of generating an answer. In the case of double recursion, the optimal method may reduce the worst-case time complexity of evaluation() by a factor $(n_1 \cdot n_2)^\gamma$, where n_1 is the number of the cyclic paths associated with the first appearance of the recursive predicate, n_2 is the number of the cyclic paths associated with the second appearance of the recursive predicate and γ is the number of times of recursive calls which happen during the execution of evaluation(). In practice, performing algebraic operations requires access to the external storage or search of large relations. In contrast, the "generating" operation happens always in the main memory and requires only access to small data sets. Hence, we may suppose that "generating answers" has the time complexity of $O(1)$, and therefore the cost of generating an answer is much less than that of evaluating an answer. Therefore, the further improved algorithm (*refined-magic-set-method-2*) achieves high efficiency.

References

1. F. Bancilhon and R. Ramakrishnan, "An amateur's introduction to recursive query processing strategies," *Proc. 1986 ACM-SIGMOD Conf. Management of Data*, Washington, DC, May 1986, pp. 16–52.
2. Y. Chen and T. Härder, "Improving RQA/FQI recursive query algorithm," in *Proceedings ISMM—First Int. Conf. on Information and Knowledge Management*, Baltimore, MD, Nov. 1992.
3. Y. Chen, "A bottom-up query evaluation method for stratified databases," in *Proceedings of 9th International Conference on Data Engineering*, Vienna, Austria, April 1993, pp. 568–575.
4. Y. Chen and T. Härder, "On the optimal top-down evaluation of recursive queries," in *Proc. of 5th Int. Conf. on Database and Expert Systems Applications*, Athens, Greece, Sept. 1994, pp. 47–56.
5. Y. Chen and T. Härder, "An optimal graph traversal algorithm for evaluating linear binary-chain programs," in *Proc. CIKM'94—the 3rd Int. Conf. on Information and Knowledge Management*, Gaithersburg, MD, Nov. 1994, pp. 34–41.
6. Y. Chen and T. Härder, *Graph Traversal and Linear Binary-Chain Programs, ZRI-Report 4/94*, University of Kaiserslautern, 1994.
7. Y. Chen, "Processing of recursive rules in knowledge-based systems—Algorithms for handling recursive rules and negative information and performance measure-

ments," Ph.D. Thesis, Computer Science Department, University of Kaiserslautern, Germany, Feb. 1995.
8. G. Grahne, S. Sippo, and E. Soisalon-Soininen, "Efficient evaluation for a subset of recursive queries," *J. Logic Programming*, **10**, 301–332 (1991).
9. J.D. Ullman, "Implementation of logical query languages for databases," *ACM Trans. Database Systems*, **10**, 3 (1985).
10. C. Chang, "On the evaluation of queries containing derived relations in relational database," in *Advances in Data Base Theory*, Vol. 1, Plenum, New York, 1981.
11. S. Shapiro and D. McKay, "Inference with recursive rules," in *Proceedings of the 1st Annual National Conference on Artificial Intelligence*, 1980.
12. F. Bancilhon, "Naive evaluation of recursively defined relations," *On Knowledge Base Management Systems–Integrating Database and AI Systems*, Springer-Verlag, Berlin, 1985.
13. L. Vieille, "Recursive axioms in deductive databases: The query–subquery approach," *Proc. First Int. Conf. on Expert Database System*, L. Kerschberg, Ed., Charleston, 1986.
14. L. Vieille, "A database complete proof procedure based on SLD resolution," *Proc. 4th Int. Conference on Logic Programming ICLP'87*, Melbourne, Australia, May 1987.
15. L. Vieille, "From QSQ to QoSaQ: Global optimization of recursive queries," *Proc. 2nd Int. Conf. on Expert Database System*, L. Kerschberg, Ed., Charleston, 1988.
16. W. Nejdl and E. J. Neuhold, *The PROLOG-DB System: Integrating Prolog and Relational Databases, Technical Report*, TU Vienna, June 1986.
17. W. Nejdl and G. Fleischanderl, *QSQR Revisit—Incompleteness, Causes and Improvements*, Technical Report, TU Vienna, December 1986.
18. W. Nejdl, "Recursive strategies for answering recursive queries—The RQA/FQI strategy," *Proc. 13th VLDB Conf.*, Brighton 1987, pp. 43–50.
19. L.J. Henschen and S. Naqvi, "On compiling queries in recursive first-order database," *J. ACM*, **31**(1), 1984, pp. 47–85.
20. F. Bancilhon, D. Maier, Y. Sagiv, and J.D. Ullman, "Magic sets and other strange ways to implement logic programs," *Proc. 5th ACM Symp. Principles of Database Systems*, Cambridge, MA, March 1986, pp. 1–15.
21. C. Beeri and R. Ramakrishnan, "On the power of magic," in *Proceedings of the 6th ACM SIGACT–SIGART Symposium on Principles of Database Systems*, 1987.
22. D. Sacca and C. Zaniolo, "On the implementation of a simple class of logic queries for databases." in *Proceedings of the 5th ACM SIGMOD–SIGACT Symposium on Principles of Database Systems*, 1986, pp. 16–23.
23. D. Sacca and C. Zaniolo, "Implementation of recursive queries for a data language based on pure horn logic," in *Proceedings of the 4th International Conf. on Logic Programming*, Univ. of Melbourne, 1987, pp. 104–135.
24. G. Balbin, S. Port, K. Ramamohanarao, and K. Meenakshi, "Efficient bottom–up computation of queries on stratified databases," J. Logic Programming, November, 295–344 (1991).
25. M.-S. Alberto, A. Pelaggi, and D. Sacca, "Comparison of methods for logic-query implementation," *J. Logic Programming*, **10**, 333–360 (1991).