Vol.13 No.4 J. of Comput. Sci. & Technol. Jul. 1998

Graph Traversal and Top-Down Evaluation of Logic Queries

Chen Yangjun (陈阳军)

Technical Institute of Changsha, Changsha 410073, P.R. China

Received October 22, 1996; revised May 25, 1997.

Abstract

In this paper, an optimal method to handle cyclic and acyclic data relations in the linear recursive queries is proposed. High efficiency is achieved by integrating graph traversal mechanisms into a top-down evaluation. In such a way, the subsumption checks and the identification of cyclic data can be done very efficiently. First, based on the subsumption checks, the search space can be reduced drastically by avoiding any redundant expansion operation. In fact, in the case of non-cyclic data, the proposed algorithm requires only linear time for evaluating a linear recursive query. On the other hand, in the case of cyclic data, by using the technique for isolating strongly connected components a lot of answers can be generated directly in terms of the intermediate results and the relevant path information instead of evaluating them by performing algebraic operations. Since the cost of generating an answer is much less than that of evaluating an answer by algebraic operations, the time consumption for cyclic data can be reduced by an order of magnitude or more.

Keywords: recursive query, top-down evaluation, RQA/FQI strategy, logic query, graph traversal.

1 Introduction

In recent years, there has been considerable effort directed toward the integration of many aspects of the artificial intelligence field with the database field. An outcome of this effort is deductive databases which can be described simply as advanced database systems augmented with rule processing. A number of strategies for the problem have been developed[1–20]. In this paper, we present an optimal method for handling recursive queries, based on the integration of graph traversal techniques such as the identifications of strongly connected components (SCCs), feedback nodes and linear cycle covers, into a top-down but set-oriented method, the so-called RQA/FQI (*Recursive Query Answering/Frozen Query Iteration*) algorithm. RQA/FQI has been introduced for handling recursive axioms in deductive databases by Nejdl[16]. It is a variant of the QSQR method[18–20] and proves to be complete over all kinds of linear recursion defined by means of function free Horn Clauses[4]. The algorithm consists of two steps. In the first step of it, the search tree is expanded top-down, storing not only the answers already found but also incomplete branches. In the second step, all incomplete branches of the search tree are processed by completing the repeated subqueries iteratively. We show that RQA/FQI can be improved significantly by integrating the algorithm for finding SCCs into its first step to do subsumption checks efficiently, based on which the search space can be reduced by removing redundant expansion operations. In this way, a linear time is achieved for non-cyclic data. Further, we utilize the techniques for identifying SCCs in such a way that a lot of answers can be generated directly in terms of the answers already found and the corresponding path information without performing algebraic operations.

Because the search space is smaller than that of RQA/FQI and the cost of generating an answer is much less than that of evaluating an answer by performing algebraic operations, the refined algorithm improves the efficiency of RQA/FQI non-trivially.

As with the other top-down strategies, a preprocessor is implemented to reorder the body predicates such that the predicates with some of their variables bound to constants (in terms of the query submitted to the system) are before the predicates whose variables have no bindings. In addition, a further step is required. If any two correlated non-recursive predicates are separated by a recursive predicate, we change the position of the one appearing after the recursive predicate such that both of them are before the recursive predicate. (We say that two predicates are correlated if they have at least one shared variable.) This requirement is not only for optimization, but also for the application of the technique for generating answers directly.

As an example, consider the well-known same-generation program below[4]. If the query issued is of the form ?-*sg* (c, y) (where c is a constant), the following two rules are well reordered.

(1) $sg(x, y)$:- $sibling(x, y)$,

(2) $sg(x, y)$:- $parent(x, z), sg(z, w), parent(y, w)$.

These two rules define a popular function-free linear recursion, *sg*, which indicates that x and y are same-generation relatives if they are siblings or their parents are same-generation relatives. Another example is the following rules defining infection risks of a disease (w.r.t. a query of the form ?-$infection\_risk$(c, y)).

(3) $infection\_risk(x, y)$:- $found(x, y), infectious(y)$,

(4) $infection\_risk(x, y)$:- $connected(x, z), infection\_risk(y, z), strong(y)$.

Here, the first rule expresses that there exists an infection risk of disease y in area x if y is found in x and y is infectious. The second rule says that if there exists an infection risk of some strongly infectious disease y in area z and area x is connected with z (by bus, train or flight), then there exists an infection risk of y in x, too.

In the remainder of the paper, we assume that all rules are reordered in this way.

In the next section, we briefly outline the RQA/FQI algorithm[16], based on which our method is developed. In Section 3, we describe the main ideas of the improvements. First, we discuss, in Subsection 3.1, how the algorithm for identifying SCCs can be integrated into RQA/FQI to do a subsumption check. Then, in Subsection 3.2, we show two possibilities of refinements: the reduction of search spaces and the direct generation of answers in terms of the answers already found. In Section 4, we give a complete description of the refined algorithm. In Section 5, we analyze and compare the computational complexity of the refined algorithm. Section 6 is a short conclusion.

2 RQA/FQI and Relevant Definitions

In this section, we briefly describe the RQA/FQI algorithm and some relevant concepts which are necessary for clarifying the main ideas of our refined algorithm. For further details, please refer to the description in [16].

RQA/FQI is a top-down, but set-oriented method. That is, although bindings for arguments are propagated in a tuple-oriented manner, queries over database predicates or database views (database equivalent predicates—described by non-recursive predicates) are processed set-oriented. The newly produced answers are stored in a Prolog database and retrieving these tuples from the Prolog database is done tuple-oriented again. In order to avoid infinite derivations, RQA/FQI distinguishes between two classes of repeatedly appearing subgoals and treats them differently. First, we have the following two definitions.

Definition 2.1. *A substitution θ is a finite set of the form $\{v_1/t_1, \ldots, v_n/t_n\}$, where each v_i is a variable, each t_i is a term (in the absence of function symbols, a term is a constant or a variable) distinct from v_i and the variables $v_1, \ldots, v_n$ are distinct. Each element v_i/t_i is called a binding for v_i. θ is called a ground substitution if t_i are all ground terms. θ is called a variable-pure substitution if t_i are all variables.*

Definition 2.2. *Let s and t be two predicates. We say that s subsumes t if there exists a substitution $\theta = \{v_1/t_1, \ldots, v_n/t_n\}$ such that $s\theta = t$, where $s\theta$ is a new predicate obtained from s by simultaneously replacing each occurrence of the variable v_i in s by term t_i $(i = 1, \ldots, n)$.*

Based on the subsumption concept, the classification of repeatedly appearing subgoals can be defined as follows.

Definition 2.3. *A repeated incomplete query (RIQ) is a query which is subsumed by a previous query which has not yet been answered completely (i.e., subsumed by a query which appeared earlier on the same derivation path as the RIQ).*

For instance, query s(c, v) appearing in the search tree of Example 2.1 is an RIQ since it is subsumed by s(c, x) (see Fig.1).

The RIQs are the only nodes which cannot be expanded during an expansion process (in order to avoid cycles). However, cutting the execution path in the search tree at an RIQ may affect the completeness of any goal relying on this subgoal. If an RIQ is encountered, only the answers already produced can be used in the further expansion of the search tree. (In RQA/FQI, such expansions are postponed to a second step and done iteratively.)

Definition 2.4. *A repeated complete query (RCQ) is a query which is subsumed by a previous query which has already been answered completely (i.e., subsumed by a query which has already appeared but is not on the same derivation path as the RCQ.)*

For example, query s(d, v) appearing in the search tree of Example 2.1 is an RCQ since it is subsumed by s(d, w).

If an RCQ is encountered, all its answers can be taken from the answers already produced.

In order to guarantee the completeness, RQA/FQI introduces an artificial subgoal added in front of each recursive subgoal to record the bindings evaluated so far. After the expansion process, we can then compute the remaining answers in terms of the instantiations of these subgoals in some way. This consideration leads to the following three definitions.

Definition 2.5. *A propagation subgoal (PSG) is a special artificial subgoal added in front of each recursive subgoal in a clause. This can be done manually or by a preprocessor.*

For example, the rules defining same-generation relatives can be transformed by a preprocessor into:

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

$sg(x,y)$:- $parent(x,z), PSG(id,x,y,z), sg(z,w), parent(y,w)$,

where "id" is the identifier of the PSG. The purpose of adding PSGs in a clause is to record and propagate the arguments instantiated so far. Therefore, a PSG for a recursive predicate contains the arguments occurring in the predicate of the head of the clause and the predicates of the body occurring before the recursive subgoal in the original clause (assuming that we proceed from left to right). The rule augmented with a PSG is called the *transformed rule*.

Definition 2.6. *A frozen query (FQ) is a query defined by a newly constructed clause (when an RIQ is encountered during the expansion process) which is constructed by eliminating those predicates appearing before the PSG from a transformed rule.*

For example, if Q :- E, P, S is a clause and P is an RIQ, then Q is a frozen query. It will be stored in the form of FQ (Q :- PSG, P, S), where PSG is a propagation subgoal. Together with a set of different instantiations of the propagation subgoal a frozen query stores the current step of the evaluation for a clause which cannot be evaluated completely in an expansion process.

Definition 2.7. *A critical path (CP) is a path of the search tree which cannot be completely evaluated in the expansion of the search tree (i.e., a path from an RIQ to its subsuming query; for example, the path from "$s(c,v),q(v,w),q(w,x)$" to "$s(c, x)$" shown in Fig.1 is a CP.). Each critical path is represented by a frozen query and a set of different instantiations of the associated PSG and can be used to evaluate the remaining answers in an iteration process (see below).*

Example 2.1. The following program helps to understand the above definitions well.

Axioms: (1) $s(x,y)$:- $r(x,y)$,
(2) $s(x,y)$:- $p(x,z), s(z,w), q(w,y)$.
Facts: p(c, d), p(c, b), p(b, c), p(b, d),
q(e, a), q(a, i), q(i, o),
r(d, e).
Query: ?-s(c, x)

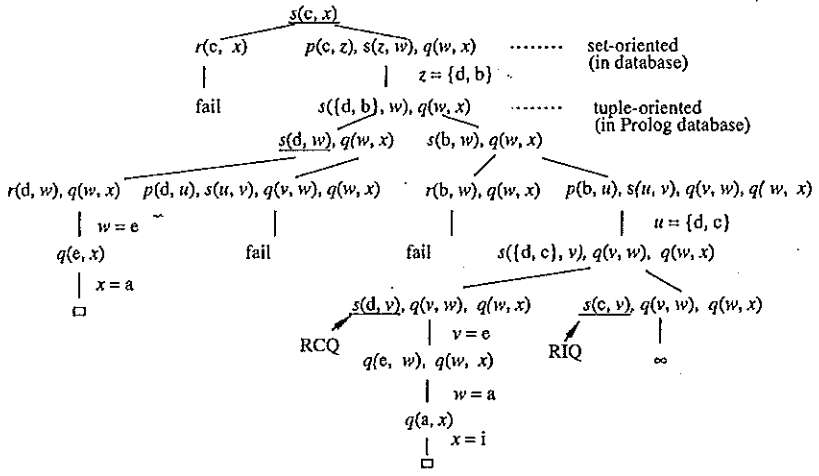

s(c, x)
r(c, x)
p(c, z), s(z, w), q(w, x)
set-oriented (in database)
z = {d, b}
fail
s({d, b}, w), q(w, x)
tuple-oriented (in Prolog database)
s(d, w), q(w, x)
s(b, w), q(w, x)
r(d, w), q(w, x)
p(d, u), s(u, v), q(v, w), q(w, x)
r(b, w), q(w, x)
p(b, u), s(u, v), q(v, w), q(w, x)
w = e
q(e, x)
x = a
fail
fail
u = {d, c}
s({d, c}, v), q(v, w), q(w, x)
s(d, v), q(v, w), q(w, x)
s(c, v), q(v, w), q(w, x)
RCQ
v = e
RIQ
∞
q(e, w), q(w, x)
w = a
q(a, x)
x = i

Fig.1. The search tree when goal ?-s(c, x) is evaluated.

Fig.1 depicts the search tree when the goal ?-s(c, x) is evaluated against the above rules and facts. Note that in the search tree, "∞" represents an infinite branch (i.e. cycles). Based on the above description, RQA/FQI can be summarized as follows:

In the first step of the algorithm, a recursive processing strategy like that of PROLOG is used, expanding the search tree top-down. Answers to recursive queries are stored to be re-used later. The expansion stops whenever RIQs are encountered, or after subqueries are answered completely using basic facts and nonrecursive predicates. The second step uses a different approach. There are still branches (critical paths) in the search tree which are incomplete because of subsumed queries (RIQ), and therefore have been stored as frozen clauses (frozen queries and the set of instantiations of propagation subgoals). These are processed using an efficient variant of Least Fixpoint iteration over these frozen clauses, working bottom-up both from database facts and from facts which have already been found as answers to recursive queries. This step is necessary in order to propagate all answers of the evaluated subsuming goal to its corresponding frozen counterparts (subsumed goals). It corresponds to plugging new answers into subsumed queries and using these answers to expand the search tree further. If new recursive subgoals are found during the iteration step, they are expanded by means of calling the top-down evaluation strategy recursively (in linear recursion, it will not happen).

3 Main Ideas of Refinement

Now we present our refined algorithm. First, we discuss how the algorithm for identifying SCCs can be integrated into RQA/FQI to do a subsumption check. Then, we show two possibilities of refinements: the reduction of search spaces and the direct generation of answers in terms of the answers already found.

3.1 Subsumption Checks

From the above definitions, we see that there are two kinds of subsumption checks which must be handled differently. When an RIQ is encountered, the traversal should be suspended and the corresponding cycle should be recorded explicitly; when an RCQ is encountered, it should be expanded immediately using the answers already found. In addition, as we will see later, the ways in which RIQs and RCQs are used to speed up the evaluation are different. However, distinguishing RCQs from RIQs is not trivial and a more sophisticated technique is needed. To this end, we combine the technique for finding a topological order for a directed graph (digraph for short) with the technique for isolating the strongly connected components of a digraph[21] in such a way that the task can be done in linear time. More concretely, we integrate a labeling technique into RQA/FQI (i.e., during a derivation each node will be labeled in some way) such that RIQs and RCQs can be distinguished by a simple checking of labels associated with them. In fact, a top-down process is a depth-first search process. Therefore, the technique used in Tarjan's algorithm for identifying SCCs can be utilized[22] for identifying RIQs. But for our purpose, this algorithm is extended by combining the idea for finding topological orders with it to facilitate the identification of RCQs. In addition, this combination improves Tarjan's algorithm non-trivially (but not by an order of magnitude), since many stack searches for identifying SCCs can be saved. In the following, we discuss this issue in detail. First, we classify a kind of nodes as *query nodes.*

Definition 3.1. *A node in a search tree is called a query node if it corresponds to a recursive subquery, i.e., if it is of the form: $A_1, \ldots, A_k$, where each A_i is a predicate and especially, A_1 is a recursive predicate with some variables instantiated.*

For example, node "$s(\mathrm{c}, v)$, $q(v, w), q(w, x)$" in the search tree of Example 2.1 is a query node, while "$p(\mathrm{b}, u)$, $s(u, v), q(v, w), q(w, x)$" is not. However, we notice that by the above definition we assume that each node is treated from left to right. But if we use another *computation rule*[22], A_1 should be replaced with "the selected atom in that node". Then, a query node is a node whose "selected atom" is a recursive predicate with some variables instantiated. Based on this concept, we define a query subgraph for a search tree as follows.

Definition 3.2. *A query subgraph for a search tree is a digraph where there are a node for each recursive subquery and an edge from a to b iff there is a path from a to b in the search tree, which contains no other query nodes except a and b.*

For example, the query subgraph of the search tree shown in Fig.1 is as shown in Fig.2.

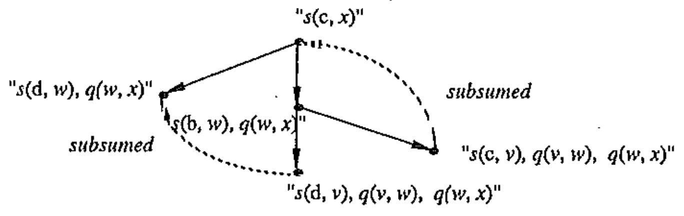


Fig.2. Query subgraph.

The purpose of query subgraphs is to explain the control mechanism used in our method.

In fact, it is sufficient to perform subsumption checks only on those nodes of a search tree, which appear also in its query subgraph (see next subsection). Therefore, we give the following algorithm over a query subgraph instead of a search tree so as to illustrate the key ideas clearly.

We associate each node v of a query subgraph with three integers *dfsnumber*(v), *toplnumber*(v) and *lowlink*(v). *dfsnumber* is used to number the nodes of a query subgraph in the order they are reached during the search. *toplnumber* is used to number the nodes with the property that all descendants of a node having *toplnumber* value m have a lower *toplnumber* value than m, i.e. a topological order numbering. Note that all nodes in an SCC should be taken to be identical in the sense of topological order. Thus, they will have the same *toplnumber* value. In the following algorithm, we use *toplnumber* values to check whether a node is an RCQ. *lowlink* is used to number the nodes in such a way that if two nodes v and w are in the same SCC, then *lowlink*(v) = *lowlink*(w). Therefore, it can be used to identify the "root" of an SCC (a root is a node of an SCC, which is first visited during the traversal). With the help of a stack structure, all SCCs can be feasibly found based on the calculation of *lowlink* values.

Essentially, the algorithm presented below is a modified version of Tarjan's algorithm[21]. The difference between them consists in the use of *toplnumber* in the modified algorithm, which facilitates the identification of an SCC. (In the original algorithm, a stack structure must be searched to do this.) In addition, for our purposes, each RCQ is marked.

```
Procedure graph-algo(v) (*depth-first traversal of a graph rooted at v*)
  begin
    i := 0; j := 1; (*i and j are two global variables, used to calculate
                                  dfsnumber and toplnumber, respectively*)
    toplnumber(v) := 0;
    graph-search(v); (*go into the graph*)
  end
Procedure graph-search(v)
  begin
    i := i + 1; dfsnumber(v) := i; lowlink(v) := i; (*initiate lowlink value;
                                  it may be changed during the search*)
    put v on stack S; (*S is used to store SCCs if any*)
    generate all sons of v;
    for each son w of v do
      begin
        if w is not topologically numbered then
          toplnumber(w) := 0; (*when a node is encountered for the
                                  first time, its toplnumber value is 0*)
      end
    for each son w of v do
      begin
        subsumption checking for w;
        if w is not subsumed by any already visited node then
          begin
            call graph-search(w); (*go deeper into the graph*)
            lowlink(v) := min(lowlink(v), lowlink(w));
            (*the root of a subgraph will have the least lowlink value*)
          end
        else (*w is subsumed by some node*)
          {suppose that w is subsumed by u;
          if dfsnumber(u) < dfsnumber(v) then
            if toplnumber(u) > 0 then
              (*if u is topologically numbered, it cannot be an ancestor node of v*)
              mark w to be an RCQ;
            else (*a cycle is encountered*)
              {mark w to be an RIQ;
               lowlink(v) := min(lowlink(v), dfsnumber(u));}} (*this operation will make
                    all nodes of an SCC have the same lowlink value as the root.*)
    end
  if (lowlink(v) = dfsnumber(v)) then (*v is a root of some SCC*)
    begin
       while w on the stack S satisfies dfsnumber(w) ≥ dfsnumber(v) do
          {delete w from the stack S and put w in current SCC (rooted at v);
            toplnumber(w) := j;} (*topological order numbering*)
       j := j + 1; (*j is used to calculate toplnumber*)
    end
end
```

In the above algorithm, we notice the difference between *lowlink* and *toplnumber*:

1. *lowlink* is numbered top-down as *dfsnumber*; but it will be changed dynamically in such a way that all nodes in an SCC possess the same *lowlink* value. Therefore, it is employed to identify the "root" of an SCC.
2. *toplnumber* is numbered bottom-up. All nodes in an identified SCC will be assigned the same *toplnumber*.

By a simple analysis, we know that this algorithm requires only linear time (see [21]). Fig.3 shows a directed graph, its *dfsnumber*, *lowlink* and *toplnumber* values, and its strongly connected components when the graph is traversed with n_1 being the start node.

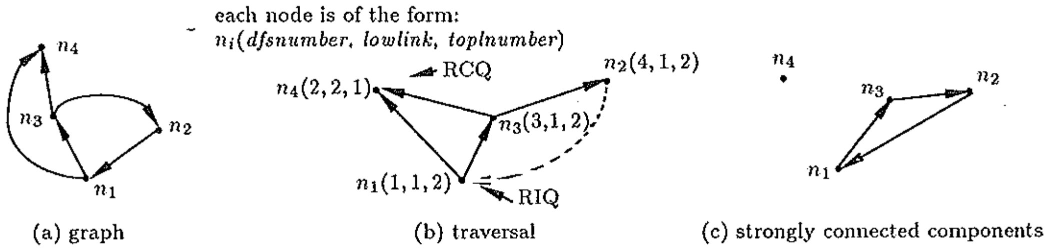


Fig.3. Graph traversal.

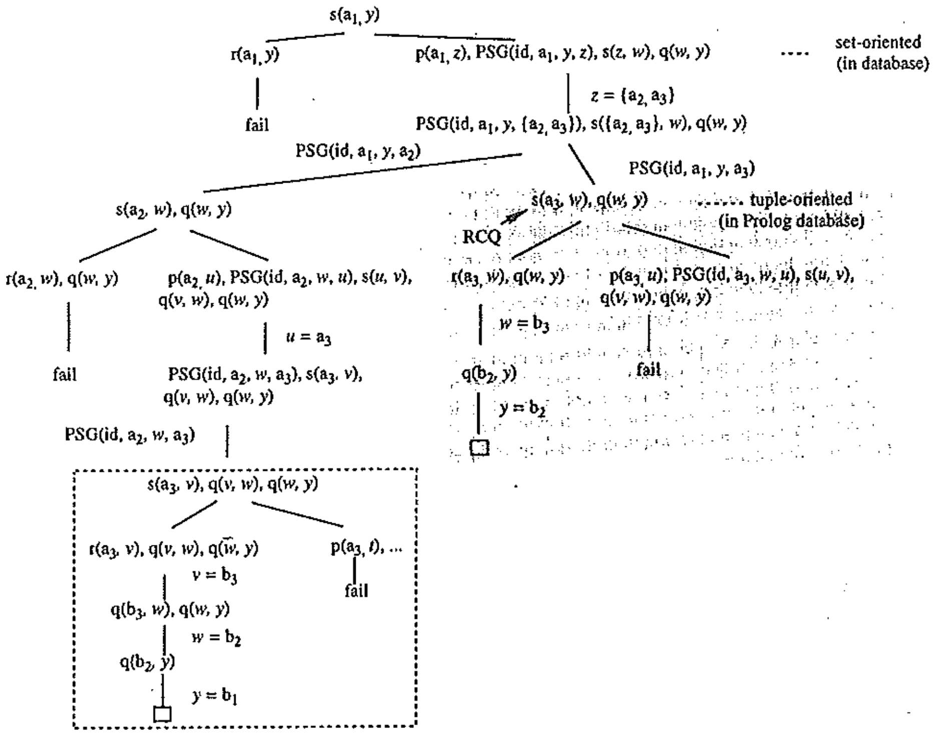
3.2 Reduction of Search Spaces and Direct Generation of Answers

In this subsection, we clarify the main ideas of the improvement by tracing the steps of RQA/FQI for two examples, each being used to show what the search space reduction means and how some answers can be directly generated, respectively. As the first example, consider the following program:

Example 3.1. Axioms: (1) $s(x,y)$:- $r(x,y)$,
(2) $s(x,y)$:- $p(x,z), s(z,w), q(w,y)$.
Facts: $p(\mathrm{a}_1, \mathrm{a}_2)$, $p(\mathrm{a}_1, \mathrm{a}_3)$, $p(\mathrm{a}_2, \mathrm{a}_3)$,
$r(\mathrm{a}_3, \mathrm{b}_3)$,
$q(\mathrm{b}_3, \mathrm{b}_2)$, $q(\mathrm{b}_2, \mathrm{b}_1)$.

The corresponding version of RQA/FQI is:
(1) $s(x,y)$:- $r(x,y)$,
(2) $s(x,y)$:- $p(x,z), PSG(id,x,y,z), s(z,w), q(w,y)$.
Given the query ?-$s(\mathrm{a}_1, y)$, the search tree depicted in Fig.4 will be generated by RQA/FQI.

This is a normal behaviour of the top-down strategy except that the path information is explicitly recorded as a set of instances of the propagation subgoal. (When a propagation subgoal is encountered, we only store it and do nothing else.) Since no cyclic path exists (no RIQs are encountered during the expansion), the second step needn't be executed. Note that the path information is very useful for optimizing the expansion step. For example, during the above process, two paths shown in Fig.5 are recorded. (For exposition, we assume that the propagation subgoal for the initial query ?-$s(a_1, y)$ is $PSG(id, \_, \_, a_1)$.)



s(a1, y)
r(a1, y)
p(a1, z), PSG(id, a1, y, z), s(z, w), q(w, y)
set-oriented (in database)
z = {a2, a3}
fail
PSG(id, a1, y, {a2, a3}), s({a2, a3}, w), q(w, y)
PSG(id, a1, y, a2)
PSG(id, a1, y, a3)
s(a2, w), q(w, y)
s(a3, w), q(w, y)
tuple-oriented (in Prolog database)
RCQ
r(a2, w), q(w, y)
p(a2, u), PSG(id, a2, w, u), s(u, v), q(v, w), q(w, y)
r(a3, w), q(w, y)
p(a3, u), PSG(id, a3, w, u), s(u, v), q(v, w), q(w, y)
u = a3
w = b3
fail
PSG(id, a2, w, a3), s(a3, v), q(v, w), q(w, y)
q(b3, y)
fail
y = b2
PSG(id, a2, w, a3)
s(a3, v), q(v, w), q(w, y)
r(a3, v), q(v, w), q(w, y)
p(a3, t), ...
v = b3
fail
q(b3, w), q(w, y)
w = b2
q(b2, y)
y = b1

Fig.4. Search tree for Example 3.1.

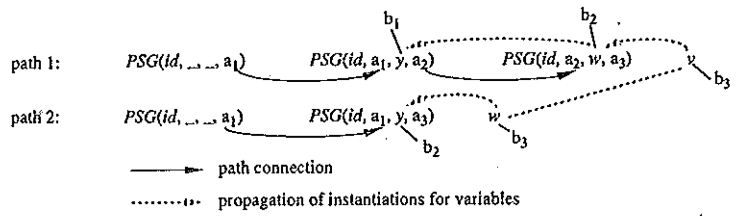


Fig.5. Paths recorded as sets of instances of PSGs.

After the subtree rooted at "$s(a_3,v)$, $q(v,w), q(w,y)$" (enclosed by a dotted rectangle in Fig.4) is expanded, variables v, w and y appearing on the first path are instantiated to b_3, b_2 and b_1, respectively. From this and the fact that the fourth argument of $PSG(id$, a_2, w, $a_3)$ (on the first path) has the same value (a_3) as the fourth argument of $PSG(id$, a_1, y, $a_3)$ (on the second path), we can directly instantiate the variables appearing on the second path instead of expanding the corresponding subtree. That is, we instantiate variables w and y on the second path to b_3 and b_2, respectively, without searching the subtree rooted at "$s(a_3$, $w)$, $q(w,y)$" (marked with a gray rectangle in Fig.4). The reason for this is that since the fourth argument of $PSG(id$, a_2, w, $a_3)$ on the first path is instantiated to the same value as the fourth argument of $PSG(id$, a_1, y, $a_3)$ on the second path (in other words, "$s(a_3,w), q(w,y)$" is subsumed by "$s(a_3,v), q(v,w), q(w,y)$"), variable w on the second path will have the same value as variable v on the first path. Further, in terms of the program property described in the introduction, variable y on the second path will be instantiated to the same value as variable w on the first path, and so on. Fig.5 helps to clarify this claim. (If the first path is shorter than the second, only some of the answers for the second path can be generated and the remaining answers must be evaluated by the standard methods.)

In this way, any subtree rooted at some RCQ can be cut off, since in this case, we can feasibly find the relevant answers in terms of the path information available. Therefore, much time will be saved.

Next, we show another possibility of optimization in the case of cyclic data. In this case, several critical paths will be stored during the first step and will be processed iteratively in the second step. Here, we only explain how the answers can be generated directly in terms of the answers already found. We omit the discussion about the problems w.r.t. linear cycle covers due to space limitation. For illustration, consider the following example.

Example 3.2. Axioms: (1) $s(x,y)$:- $r(x,y)$,
(2) $s(x,y)$:- $p(x,z), s(z,w), q(w,y)$.
Facts: p(c, d), p(c, b), p(b, c), p(b, f), p(f, c),
r(d, e),
q(e, a), q(a, i), q(i, o), q(o, g).

Given the query ?-s(c, y), RQA/FQI first generates a search tree as shown in Fig.7.

During the expansion step, two answers, s(d, e) and s(c, a), are produced. In addition, a set of propagation facts:

PSG(*id*, c, y, d),
PSG(*id*, c, y, b),
PSG(*id*, b, w, f),
PSG(*id*, b, w, c),
PSG(*id*, f, v, c),

is generated and a frozen query: FQ($s(x,y)$:- $PSG(id,x,y,z), s(z,w), q(w,y)$) is constructed. By running the frozen query against the propagation facts and the answers already found (in a bottom-up manner), the following answers will be found:

s(b, i), s(c, o), s(b, g),
s(f, i), s(b, o), s(c, g), s(f, g).

Observe that the set of propagation facts corresponds to two critical paths (cyclic paths, see Fig.6).

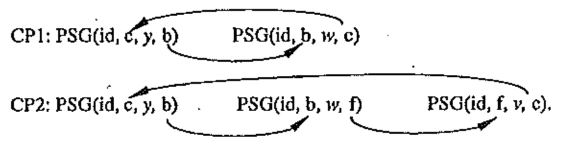


Fig.6. Two critical paths.

In this case, we can further optimize RQA/FQI by elaborating the second step as follows. First, we only evaluate some answers for the first critical path. Then we generate the answers for the second path directly from the associative *PSG*s and the answers already found. For example, the answers for the first path are s(b, i), s(c, o) and s(b, g), and so we can directly generate s(f, i) from s(b, $\underline{\text{i}}$) and the third *PSG* (*PSG*(*id*, $\underline{\text{f}}$, v, c)) on the second critical path instead of evaluating it by performing algebraic operations. Similarly, we can directly generate s(b, o) from s(c, $\underline{\text{o}}$) and the second *PSG* (*PSG*(*id*, $\underline{\text{b}}$, w, f)) on the second critical path, and so on. Fig.8 helps to illustrate this feature.

In general, the new answers generated for one critical path can further be used to generate new answers for the other critical path again. For the above example, we can imagine two circles (C_1 and C_2) with each corresponding to a cyclic path. We run respectively along

C_1 and C_2 in the same direction and generate some answers for C_2 from C_1 at each step. We do this continually until no new answers for C_2 can be generated. Then we generate new answers for C_1 from C_2 in the same way. This process is repeated until no more new answers can be generated. We describe this process formally as shown in Fig.9.

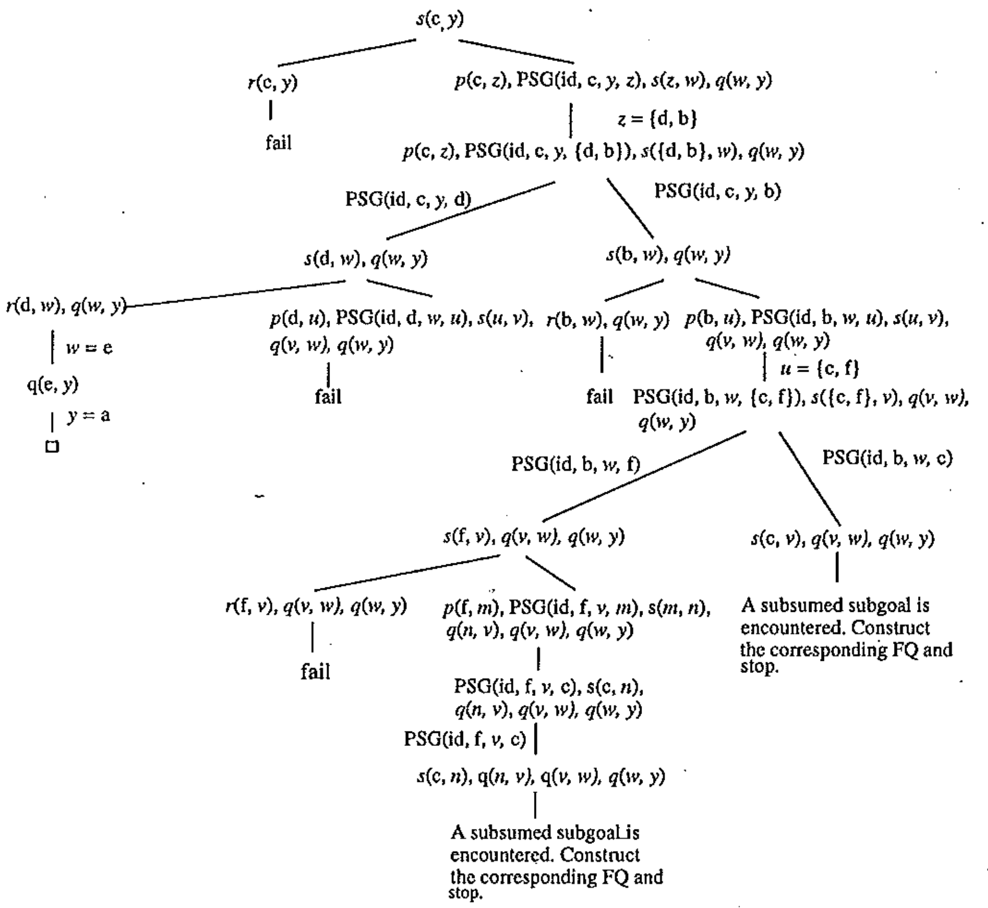

s(c, y)
r(c, y)
fail
p(c, z), PSG(id, c, y, z), s(z, w), q(w, y)
z = {d, b}
p(c, z), PSG(id, c, y, {d, b}), s({d, b}, w), q(w, y)
PSG(id, c, y, d)
PSG(id, c, y, b)
s(d, w), q(w, y)
s(b, w), q(w, y)
r(d, w), q(w, y)
w = e
q(e, y)
y = a
□
p(d, u), PSG(id, d, w, u), s(u, v), q(v, w), q(w, y)
fail
r(b, w), q(w, y)
fail
p(b, u), PSG(id, b, w, u), s(u, v), q(v, w), q(w, y)
u = {c, f}
PSG(id, b, w, {c, f}), s({c, f}, v), q(v, w), q(w, y)
PSG(id, b, w, f)
PSG(id, b, w, c)
s(f, v), q(v, w), q(w, y)
s(c, v), q(v, w), q(w, y)
A subsumed subgoal is encountered. Construct the corresponding FQ and stop.
r(f, v), q(v, w), q(w, y)
fail
p(f, m), PSG(id, f, v, m), s(m, n), q(n, v), q(v, w), q(w, y)
PSG(id, f, v, c), s(c, n), q(n, v), q(v, w), q(w, y)
PSG(id, f, v, c)
s(c, n), q(n, v), q(v, w), q(w, y)
A subsumed subgoalis encountered. Construct the corresponding FQ and stop.

Fig.7. Search tree for Example 3.2.

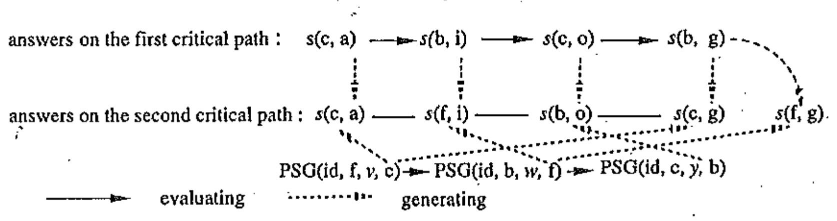


Fig.8. Answers on two critical paths associated with FQ($s(x, y)$:- $PSG(id_1, x, y, z), s(z, w), q(w, y)$).

Let C_1 be the first cycle $\text{psg}_1 \leftarrow \text{psg}_2 \leftarrow \cdots \leftarrow \text{psg}_n \leftarrow \text{psg}_1$. $A_1 = \{a_1, ..., a_n, a_{n+1}, ..., a_{2n}, ..., a_{in}, ..., a_{in+j}\}$ the answer set evaluated along C_1, where i, j are integers and $0 \leq j \leq n$. (It should be noticed that each a_l $(1 \leq l \leq in+j)$ is a subset which is evaluated by running the frozen query with the PSG subgoal being instantiated to psg_λ, where $l = rn + \lambda$ for some integer r.) Let C_2 be the second cycle $\text{psg}'_1 \leftarrow \text{psg}'_2 \leftarrow \cdots \leftarrow \text{psg}'_m \leftarrow \text{psg}'_1$ (see Fig.9 for illustration). In addition, we define

$$A_2 = \{ a_{n+1}, ..., a_{2n}, ..., a_{in}, ..., a_{in+j} \},$$

$A_3 = \{ a_{2n+1}, ..., a_{in}, ..., a_{in+j} \}$,

... ...

$A_i = \{ a_{in}, ..., a_{in+j} \}$.

Then, in terms of A_k $(1 \le k \le i)$ 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 $(1 \le k \le i)$. In practice, each time an A_k needs to be used in the computation, we shrink A_{k-1} by leaving out certain sets (some a_i's)). Without loss of generality, we assume that $n \le m$.

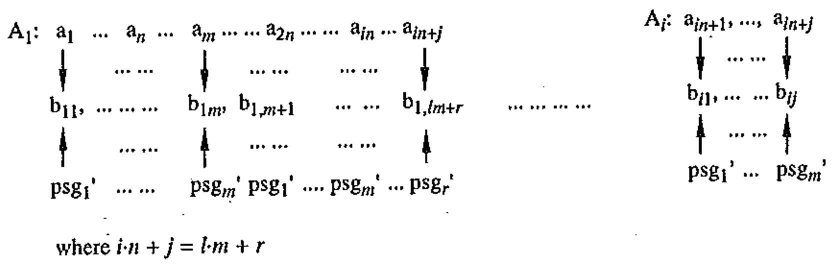

A_1: a_1 ... a_n ... a_m ... ... a_{2n} ... ... a_{in} ... a_{in+j}
b_{11}, ... b_{1m}, $b_{1,m+1}$ ... ... $b_{1,lm+r}$
psg_1' ... ... psg_m' psg_1' .... psg_m' ... psg_r'
A_i: a_{in+1}, ..., a_{in+j}
b_{i1}, ... b_{ij}
psg_1' ... psg_m'
where $i{\cdot}n + j = l{\cdot}m + r$

Fig.9. 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.

$b_1 = b_{11} \cup b_{21} \cup \cdots \cup b_{i1}$,

... ...

$b_{lm+r} = b_{1,lm+r} \cup$ ... ...

(Hereafter, this process is called a *merging operation.*) Then, we construct B_k $(1 \le k \le l-1)$ in the following way.

$B_1 = \{ b_1, ..., b_{lm+r} \}$,

$B_2 = \{ b_{m+1}, ..., b_{lm+r} \}$,

$B_3 = \{ b_{2m+1}, ..., b_{lm+r} \}$,

... ...

$B_l = \{ b_{lm+1}, ..., b_{lm+r} \}$.

(Hereafter, this process is called a *separating operation.*)

In terms of C_1 and B_k $(2 \le k \le l)$, 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. In general, this technique can be applied to any number of cycles if they contain a common node.

Because evaluating answers by performing algebraic operations requires access to the external storage or search of large relations but the "generating" operation happens always in the main memory and requires only access to small data sets (i.e. the answers already evaluated or generated on the other critical path), the generation of answers is much more efficient than the evaluation of answers. Thus, the algorithm is optimal in time complexity in comparison with RQA/FQI.

4 Description of the Refined Algorithm

Based on the analysis above, the new algorithm (for linear recursion) should be composed of four processes:

1. expansion process which acts like that of Prolog, but with "search space reduction" being integrated;
2. generation process for RCQs, by which all answers for repeated complete queries (RCQs) are generated directly in terms of the associated path information and its subsuming query;
3. iteration process, by which some answers for the first critical path are evaluated in a bottom-up fashion;
4. generation process for RIQs, by which all answers for the other critical paths are generated directly from the associated PSGs and the answers already found.

In some cases, there may be several classes of critical paths with all paths in one class corresponding to the same subsuming query. In these cases, we need to execute the third step and fourth step above for each class, since the answers for a path in some class can not be generated from the answers evaluated or generated for other classes. For simplicity, in the following description of the algorithm, we assume that there is only one class of critical paths. However, it is not difficult to replace the corresponding part by a complete version in this algorithm. In addition, in the expansion process, each query node n is associated with three integers $dfsnumber(n)$, $toplnumber(n)$ and $lowlink(n)$ to do subsumption checks. But we do not describe this mechanism in the following algorithm for ease of exposition. In fact, this is no more than what is discussed in Subsection 3.1.

Before the algorithm is applied, a preprocessor has to be executed to add a *PSG* in front of each recursive predicate in a rule and to reorder the predicates in the body of the rule such that the predicates occurring after a recursive predicate are only correlated with the recursive predicate. On the other hand, we employ the normal optimization strategy of reordering subqueries, i.e. we put the predicates with some of their variables bound to constants before the predicates whose variables have no bindings. It is necessary to reorder the predicates in such a way that we can save as many operations over the predicates which are only correlated with the recursive predicate as possible.

In the algorithm, certain set variables are associated with recursive predicate q:

Loc_M_q—	a global set of instances for q, indicating the instantiated arguments of q
Loc_Ans—	a local set of tuples which is used during the expansion to store the answers already found
FQ_set—	a global set of frozen queries
path_set$_j$—	a global set of critical paths for each FQ$_j$
path_set—	the set of path_set$_j$
S_{mn}—	a local set of tuples which is used during the iteration to store the answers evaluated or generated in the n-th step of the m-th critical path

```
Procedure evaluation($q$, Loc_M_$q$)
begin
  for each i in Loc_M_$q$ do
      {expansion($q$, i); (*Note that "generation_for_RCQ" will be called by "expansion".*)
      iteration(Loc_Ans, FQ_set, path_set);
      generation_for_RIQ(Loc_Ans, path_set)};
end

Procedure expansion($q$, i)
(* Input: $q$, i ; Output: Loc_Ans, FQ_set, path_set *)
  for each clause $C_j$ defining $q$ do
    repeat
      begin
        choose the first/next predicate according to a selection function;
        generate the corresponding generalized query $P_k$;
        if $P_k$ is a propagation subgoal
          then insert $P_k$ in the corresponding path in a path_set$_j$ of the path_set;
        if $P_k$ is non-recursive
            then evaluate $P_k$ by standard non-recursive methods and store the results in Loc_Ans;
        if $P_k$ is an RIQ
            then {generate an FQ (incomplete part of current rule + instantiation in the form of a PSG);
                FQ_set := FQ_set ∪ the current FQ} (* If FQ is already stored, only the new PSG has to be
                asserted.*)
        if $P_k$ is an RCQ then call generation_for_RCQ(path$_1$, path$_2$); (*path$_1$ is the path associated with
            the subsuming query; path$_2$ is the path associated with the subsumed query.*)
    end
  until there are no more predicates in the body of $C_j$.

Procedure generation_for_RCQ(path$_1$, path$_2$)
(*Input: path$_1$ (path associated with the subsuming query), path$_2$ (path associated with the subsumed
        query);
  Output: some new answers*)
  begin
    {generating new answers in terms of path$_1$ and path$_2$;
        Loc_Ans := Loc_Ans ∪ new_answers}
  end

Procedure iteration(Loc_Ans, FQ_set, path_set)
(*Input: Loc_ Ans, FQ_set, path_set; Output: set of $S_{mn}$*)
  for each FQ$_j$ in FQ_set do
    (*producing answers for the first critical path which cannot be generated from the answers
        on the other critical path and the associated $PSG$s *)
      for all $m, n$ do $S_{mn} := \phi$; (* $m \geq 1$, $n \geq 0$ *)
      $S_{10}$ := subset of Loc_Ans satisfying ($q$, i);
      let $P$ be the first path in the path_set$_j$;
        repeat
          for $l = 0$ to $L - 1$ do (*$L$ = Length($P$) represents the number of $PSG$s contained in $P$.*)
            begin
              get instantiation for the corresponding $PSG$;
              get answers for $q$ from $S_{1,l\bmod L}$;
              for each predicate $p$ occurring after $q$ do
                if $p$ is a recursive predicate then evaluation($p$, Loc_M_$p$)
                  else evaluate $p$ by standard non-recursive methods;
              $S_{1,l+1\bmod L} := S_{1,l+1\bmod L}$ ∪ new_answers; (*After all predicates occurring after $q$ are
              evaluated, some new answers are produced and stored in $S_{1,l+1\bmod L}$.*)
            end
        until no more answers for the path are evaluated.

Procedure generation_for_RIQ(Loc_Ans, path_set)
(*generating all answers for each critical path*)
  for $m = 2$ to |path_set$_j$| do
    repeat
        for each $k, j \in \{1, 2, ..., m\}$ do
          for $l = 0$ to $L_j - 1$ do (*$L_j$ = Length($P_j$) represents the number of $PSG$s contained in $P_j$.*)
            {generating new answers from $S_{k,l\bmod L_k}$ and the corresponding $PSG$;
              $S_{j,l\bmod L_j} := S_{j,l\bmod L_j}$ ∪ new_answers}
        until no new answers are generated.
```

In the first step of the algorithm, we expand the search tree top-down. During the process, the queries over database predicates or database views are evaluated in a set-oriented way and the answers produced are stored in Loc_Ans. If a propagation subgoal is encountered, it will be inserted into the corresponding critical path which is stored in some path_set$_j$. If an RIQ is encountered, the expansion stops and the corresponding FQ is constructed, which is stored in FQ_set. If an RCQ is encountered, generation_for_RCQ will be called to generate some new answers in terms of the path information associated with both the RCQ and its subsuming query. The second step of the algorithm is separated into two phases. In the first phase, only some of the answers for the first critical path are evaluated with some of the answers already produced (during the expansion) as initial values. In the second phase, all remaining answers for each critical path are generated directly from the corresponding *PSG*'s and the answers already produced or generated on the other critical path.

Example 4.1. Given the rules and facts as in Example 3.1, let ? -$s(\mathrm{a}_1, x)$ be the query. The expansion process of the refined algorithm will produce the following answers:

answers evaluated for the first path: $s(\mathrm{a}_3, \mathrm{b}_3)$, $s(\mathrm{a}_2, \mathrm{b}_2)$, $s(\mathrm{a}_1, \mathrm{b}_1)$;
answers generated for the second path: $s(\mathrm{a}_3, \mathrm{b}_3)$, $s(\mathrm{a}_1, \mathrm{b}_2)$.

The last two answers are generated directly by calling "generation_for_RCQ".

Example 4.2. Given the rules and facts as in Example 3.2, let ?-s(c, x) be the query. The refined algorithm will produce the following answers:

answers evaluated in the first step: s(d, e), s(c, a);
answers evaluated in the first phase of the second step: s(b, i), s(c, o), s(b, g);
answers generated in the second phase of the second step: s(f, i), s(b, o), s(c, g), s(f, g).

5 Comparison with Other Strategies

In order to compare the time complexity of the refined algorithm with RQA/FQI, we consider the following linear recursive program:

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

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. At an abstract level, the expansion phase of RQA/FQI 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". If the indegree and outdegree of each node i in the graph are denoted as *indegree*(i) and *outdegree*(i), respectively, then the cost of the expansion phase of RQA/FQI is:

$$\mathrm{O}(e_p) + \mathrm{O}(e_r) + \mathrm{O}\Big(\sum_{(i,j)\in A} indegree(i) \times outdegree(j)\Big),$$

where A denotes the set of answer tuples. $\mathrm{O}(e_p)$ is the cost for the traversal of the graph for "p". $\mathrm{O}(e_r)$ is the cost for the traversal of the graph for "r". The cost for the traversal of the graph for "q" is $\mathrm{O}(\sum_{(i,j)\in A} indegree(i) \times outdegree(j)) = \mathrm{O}(e_p \cdot e_q)$. Fig.10(a) helps to clarify the result.

From Fig.10(a), we see that crossing the edge (i,j), each edge incident to j will be visited *indegree*(i) times. Similarly, the expansion process of the refined algorithm can be viewed as two processes. However, due to the answer generation for RCQs, the cost is reduced to

$$\mathrm{O}(e_p) + \mathrm{O}(e_r) + \mathrm{O}\Big(\sum_{some(i,j)\in A} outdegree(j)\Big). \text{ (See Fig.10(b) for illustration.)}$$

On one hand, due to the "answer generation", each edge incident to j can be visited at most once. On the other hand, some answer tuples may not be traversed for the same reason. Therefore, its time complexity is less than $\mathrm{O}(n_p \cdot e_q)$. In order to achieve a linear time, we need to do subsumption checks on more nodes, i.e., we check the subsumption not only on query nodes, but also on their son nodes. In other words, the nodes of the form: $q(x_1,x_2), q(x_2,x_3), \ldots, q(x_{i-1},x_i)$ will be checked. Each of them is generated by instantiating a node such as $s(x_0,x_1), q(x_1,x_2), q(x_2,x_3), ..., q(x_{i-1},x_i)$. Consider nodes $n_1 = q(x_1,x_2), q(x_2,x_3), ..., q(x_{i-1},x_i)$ and $n_2 = q(y_1,y_2), q(y_2,y_3), ..., q(y_{j-1},y_j)$. Without loss of generality, assume $i \leq j$. If n_2 is subsumed by n_1, i.e., y_1 will be bound to the same constant as x_1, then $q(x_1, x_2), q(x_2, x_3), ..., q(x_{i-1}, x_i)$ and $q(y_1, y_2), q(y_2, y_3), ..., q(y_{i-1}, y_i)$ can be treated as a commom expression. If the answer set produced by evaluating $q(x_2, x_3), \ldots, q(x_{i-1}, x_i)$ is denoted as S, then we compute the following formula for n_2:

$$S \bowtie q(y_i, y_{i+1}) \bowtie \cdots \bowtie q(y_{j-1}, y_j).$$

In this way, each edge in the graph for "q" will be visited at most once and so no redundant work will be done. Since each edge in the graph for "p" is also visited at most once, the refined algorithm requires only linear time for non-cyclic data.

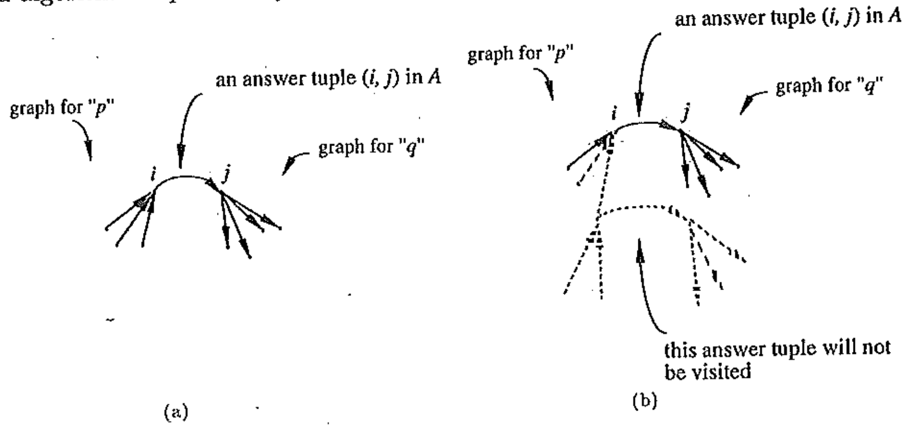

graph for "p"
an answer tuple (i, j) in A
graph for "q"
i
j
(a)
graph for "p"
an answer tuple (i, j) in A
graph for "q"
i
j
this answer tuple will not be visited
(b)

Fig.10. Illustration for time complexity analysis.

In the above analysis, we do not take the cost for "generating answers" into account. In fact, 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 needn't consider it. (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). Now we derive the time complexity of handling cycles. We consider only the case of linear recursion. 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 *PSG*s 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 in 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 second step of RQA/FQI is in the order of

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

where λ is the number of the cycles associated with an FQ 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 fourth process 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 fifth process. Let δ be the cost of generating an answer in the generation process, then the running time for the fifth process of the refined algorithm is

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

Since $\delta \ll C$, the saving on time is significant. If $\delta/C \le n/d^{ml}$, $(\lambda - 1) \cdot (d^{ml} - 1) \cdot \delta$ is less than some constant, and therefore the time complexity of the fourth and fifth processes of the refined algorithm is $O(d^{ml} \cdot C)$. Therefore, the refined algorithm may reduce the worst-case time complexity of RQA/FQI by a factor λ, the number of the cycles, if we do not take the cost of generating an answer into account.

Many other strategies have been proposed to handle this problem, among which the magic set method receives much attention. For the above simple program, the magic set method works in a two-phase manner. In the first phase, the magic set is produced by executing magic rules[3,4]. In the second phase, modified rules are executed using the magic set to restrict the bottom-up computation. Essentially, these two phases correspond to the constant propagation process and the variable instantiation process of a top-down evaluation, respectively. Therefore, the cost of the magic set method is $O(e_p \cdot e_q)$ (see [15] for details). The counting method is an indexed version of the magic set method[1,4,17]. It improves the magic set method by a constant factor. In fact, its time complexity is $O(n_p \cdot e_q)$. In addition, a lot of experiments have been done[6] and they show that QSQR, another top-down strategy[18-20], has the same time complexity as the magic set method. Therefore, our method is also better than QSQR.

6 Conclusion

In this paper, a top-down but set-oriented algorithm for the evaluation of recursive queries has been presented which is much more efficient than RQA/FQI and other well-known strategies such as QSQR, magic set method and counting method. The key idea of the improvements is to distinguish between RCQs and RIQs and to use the path information associated with them elaborately. We may reduce the search space by cutting off any subtree rooted at an RCQ because we can feasibly find the relevant answers in terms of the path information available. Then we minimize the time complexity by separating the second step of RQA/FQI into two phases and generating most answers for each critical path directly from the intermediate results and the associated *PSG*s after some answers for the first critical path are evaluated, without performing algebraic operations. In practice, since performing algebraic operations requires access to the external storage or search of large relations but the "generating" operation happens always in the main memory and requires only access to small data sets, the cost of generating an answer is much less than that of evaluating an answer. Thus, the refined algorithm is optimal. The proof of the correctness of the refined algorithm can be found in [8].

References

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

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

[3] Beeri C, Ramakrishna R. On the power of magic sets. *Int'l J. Logic Programming,* 1991, 10: 255-299.

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

[5] Chen Y. 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.

[6] Chen Y, Härder T. An optimal graph traversal algorithm for evaluating linear binary-chain programs. In *CIKM'94 — The 3rd International Conference on Information and Knowledge Managemen,* Gaithersburg, Maryland, USA: ACM, Nov. 1994, pp.34-41.

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

[8] 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 Kaiserslautern, Germany, Feb. 1995.

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

[10] Chen Y. Magic sets and stratified databases. *Int'l Journal of Intelligent Systems,* March 1997, 12(3): 203-231.

[11] Chen Y. OLDT-based evaluation method for handling recursive queries in deductive databases. accepted by *Science Sinica,* 1997.

[12] Chen Y. Counting and topological order. *Journal of Computer Science and Technology,* 1997, 12(6): 497-509.

[13] Han J. Chain-based evaluation——A bridge linking recursive and nonrecursive query evaluation. In *Proc. 2nd Int'l Workshop on Research Issues on Data Engineering: Transaction and Query Processing,* Los Alamitos, CA, February 1992, pp.132-139.

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

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

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

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

[18] Vieille L. Recursive axioms in deductive databases: The query-subquery approach. In *Proc. First Int'l Conf. Expert Database System,* L. Kerschberg (ed.), Charleston, 1986.

[19] Vieille L. A database complete proof procedure based on SLD resolution. In *Proc. 4th Int'l Conf. Logic Programming ICLP'87,* Melbourne, Australia, May 1987.

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

[21] Tarjan R. Depth-first search and linear graph algorithms. *SIAM J. Comput.,* June 1972, 1(2): 146-140.

[22] Lloyd J W. Foundation of Logic Programming. Springer-Verlag, Berlin, 1987.

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-Zw2ickau, Germany. His research interests include deductive databases, federative databases, constraint satisfaction problem, graph theory and combinatorics. He has about 40 publications in these areas.