On the graph traversal method for evaluating linear binary-chain programs

CHEN Yangjun (陈阳军)

(Technical Institute of Changsha, Changsha 410073, China; Department of Computer Science, Technical University of Chemnitz, 09106 Chemnitz, Germany)

Received December 23, 1997; revised December 23, 1998

Abstract Grahne et al. have presented a graph algorithm for evaluating a subset of recursive queries. This method consists of two phases. In the first phase, the method transforms a linear binary-chain program into a set of equations over expressions containing predicate symbols. In the second phase, a graph is constructed from the equations and the answers are produced by traversing the relevant paths. A new algorithm is described which requires less time than Grahne's. The key idea of the improvement is to reduce the search space that will be traversed when a query is invoked. Further, the evaluation of cyclic data is speeded up by generating most answers directly in terms of the answers already found and the associated "path information" instead of traversing the corresponding paths as usual. In this way, this algorithm achieves a linear time complexity for both acyclic and most of cyclic data.

Keywords: graph traversal method, linear binary-chain programs.

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 the notion of knowledge-based system, which can be described simply as an advanced database system augmented with a mechanism for rule processing. 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[1,10], semi-naive evaluation[11], query/subquery[12], RQA/FQI[13], Henschen-Naqvi[14], and the methods used in compiling recursive queries[8,14—17]. 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 are magic sets[18], counting[18] and their generalized versions[5,19]. In this paper, we discuss a graph method which has been presented for handling a subset of recursive queries, the so-called binary-chain programs, by Grahne et al.[20,21]. (We refer to this method as Grahne's algorithm afterwards.) Binary relations form an important subcase of n-ary relations. This is not only because binary queries are frequently encountered in practical application, but also because any set of relations can be represented as a set of binary relations[13,21]. Therefore, any rule (in the Horn-clause form) can be transformed into a set of binary-chain rules. Compared to the SLD resolution[22] and its different variants[12,13,23], the graph method is advantageous due to the following two benefits: (i) Repeated firing of rules with the same head predicate can be avoided; (ii) instead of maintaining a large "goal node" in each resolution step as done in SLD strategy, a simple structure is used to record nodes encountered during a graph traversal.

Grahne's method works in a two-phase approach. In the first phase, a program is transformed into a set of equations of the form: $r = e_r$, where r is a derived predicate symbol and e_r is an expression whose arguments are predicate symbols and whose operators are chosen from among $\cup$ (union), $\cdot$ (composition), and $*$ (reflexive transitive closure). In the second phase, a directed graph $G(r)$ is constructed from each equation of the form: $r = e_r$ such that $r(x, y)$ is true if and only if $G(r)$ contains a path from a node representing x to a node representing y. This result means that evaluation problems for the predicate r reduce to traversal problems for the graph $G(r)$ or the hierarchy of $G(r)$'s (see below). We show that this method proceeds redundantly in certain cases and can be improved by elaborating its second phase. First, we try to reduce the search space that will be traversed by Grahne's algorithm. We do this by recognizing all similar portions of a graph and manage to produce all the relevant answers by constructing only one of them. The other refinement is concerned with the treatment of cyclic data. In this case, a cycle is stored when it is encountered at the first time. We then suspend the traversal along the corresponding path to avoid duplicate work. However, as many intermediate answers may not be used to produce new answers along a cyclic path, suspending the traversal along the cyclic path may affect the completeness. Therefore, we develop a process to evaluate the remaining answers by iterating on each cyclic path with a different initial value each time. In this iteration process, we further optimize the evaluation by generating most answers for cyclic data directly from the answers already found and the associated path information instead of traversing the relevant subgraphs as usual. In this way, we can decrease the time complexity by one order of magnitude or more. This is because traversing paths requires access to the external storage or search of large relations but the "generating" operations happen always in the main memory and require only access to small data sets (i.e. the answers already found). As a consequence, our algorithm requires only linear time for both cyclic and acyclic data.

This paper is organized as follows. In the next section, we introduce the necessary terminology from refs. [20,21]. In sec. 2, we briefly describe the main idea of Grahne's algorithm. In sec. 3, we give our refined graph traversal algorithm for evaluating linear binary-chain programs. In sec. 4, we compare the computational complexity of our algorithm with the existing strategies. Sec. 5 is a short conclusion.

1 Basic concepts

A rule of the form

$$q(x_1, x_{m+1}) \leftarrow p_1(x_1, x_2), p_2(x_2, x_3), \cdots, p_m(x_m, x_{m+1}),$$

where $m \geqslant 0$ and $x_1, \cdots, x_{m+1}$ are all distinct variables, is called a binary-chain rule. A Datalog program in which the predicates are all binary predicates and the rules in the intensional database are all binary-chain rules is called a binary-chain program.

For a program, we may construct a dependency graph representing a "refer to" relationship between the predicates. This is a directed graph where there is a node for each predicate and an arc from node q to node p if and only if the predicate q occurs in the body of a rule whose head predicate is p. A predicate p depends on a predicate q if there is a path of length greater than or equal to one from q to p. We denote the relation p depends on q by $p \Leftarrow p$, where depends on is the transitive closure of the "refer to" relation. A predicate p is recursive if $p \Leftarrow p$. Two predicates p and q are

mutually recursive if $p \Leftarrow q$ and $q \Leftarrow p$.

A rule in which the head predicate is mutually recursive to one of the body predicates is called a recursive rule. If the body of a recursive rule contains at most one literal whose predicate is mutually recursive to the head predicate, the rule is called a linearly recursive rule. A program that contains at least one such rule is called a linearly recursive program.

A binary-chain rule

$$q(x_1, x_{m+1}) \leftarrow p_1(x_1, x_2),\ p_2(x_2, x_3), \cdots, p_m(x_m,\ x_{m+1})$$

is a right-linear rule if none of the predicates $p_1, \cdots, p_{m-1}$ is mutually recursive to p, and a left-linear rule if none of the predicates $p_2, \cdots, p_m$ is mutually recursive to p. A derived predicate is a regular predicate if its definition is right-linear or left-linear. A binary-chain program is a regular program if all its derived predicates are regular.

In addition, the relations for the predicates appearing left to the recursive predicate is called the left-hand side relations and those right to the recursive predicate is called the right-hand side relations.

2 Grahne's method

In this secion, we briefly describe Grahne's algorithm, which is necessary for introducing our refined method.

2.1 Program transformation

Grahne's method works in a two-phase manner. In the first phase of Grahne's method, any linear binary-chain program is transformed into a system of equations of the form

$$r = E_r$$

with the following properties (see ref.[21]):

(i) For each derived predicate, there is exactly one equation;

(ii) E_r is an expression whose arguments are predicate symbols of the program and whose operators are chosen from among $\cup$ (union), $\cdot$ (composition), and $*$ (reflexive transitive closure);

(iii) in each equation $r = E_r$, the expression E_r does not contain any occurrences of regular derived predicates.

In this way, repeated firing of rules can be avoided since each derived predicate is associated with only one equation no matter how many times it appears in the program. See the following example for illustration.

Example 2.1. Consider the following program:

$$\begin{aligned} &p(x, y):\text{-}b(x, z), q(z, y),\\ &q(x, y):\text{-}c(x, z), p(z, y),\\ &q(x, y):\text{-}d(x, z), r(z, y),\\ &r(x, y):\text{-}a(x, y),\\ &r(x, y):\text{-}e(x, z), q(z, y), \end{aligned}$$

where a, b, c, d and e are base predicates, while p, q and r are derived predicates. This program can be transformed into the following equations by means of the transformation algorithm given in ref.[21]:

$$p = b \cdot (c \cdot b \cup d \cdot e)^* \cdot d \cdot a, \tag{2.1}$$

$$q = (c \cdot b \cup d \cdot e)^* \cdot d \cdot a, \tag{2.2}$$

$$r = a \cup e \cdot (c \cdot b \cup d \cdot e)^* \cdot d \cdot a. \tag{2.3}$$

For further details, please refer to the description in reference [21].

2.2 Description of Grahne's algorithm

The algorithm proposed by Grahne et al. can be described as follows. Let $r = E_r$ be an equation. The algorithm represents the equation as a nondeterministic automaton, denoted by $M(E_r)$, which can be obtained by the standard technique from E_r when we regard E_r as a regular expression over the alphabet consisting of all predicate symbols appearing in E_r. For example, eq. (2.1) given above can be represented as the automaton shown in figure 1. Here q_s, q_f and q_i ($i = 1, 2, \cdots, 7$) represent the initial, final, and intermediate states, respectively.

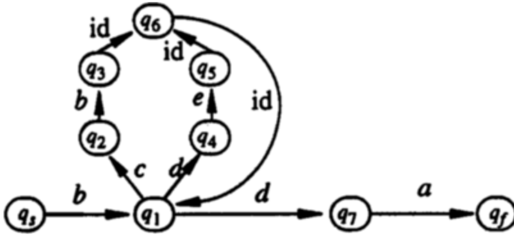


Fig. 1. Automaton for eq. (2.1). Here "id" is interpreted as the identity relation.

If $r = E_r$ is a recursive equation, a hierarchy of automata will be constructed in evaluating answers to the query. The ith level in the hierarchy, denoted by EM(r, i), corresponds to the ith recursive call of r (which appears in E_r) with some of its variables bound to constants. First, EM$(r, 0)$ is the initial state q_s and EM$(r, 1)$ is a copy of $M(E_r)$, and then the interpretation graph of EM$(r, 1)$ is traversed. An interpretation graph of EM(r, i) is a directed graph with a set of nodes (q, u) where q is a state in EM(r, i) and u is a domain element of some base relation labelling a transition leaving q, and with a set of edges of the form, (q, u)-(q', v), where, for some base relation a, $q \xrightarrow{a} q'$ is a transition in EM(r, i) such that $a(u, v)$ is true. (Afterwards, we use the term graph to refer to a directed graph, since we do not discuss undirected ones at all.)

Now we consider the evaluation of a query of the form $r(c, y)$, where c is a constant. The evaluation algorithm will generate a sequence of interpretation graphs of EM$(r, 0) \cup \cdots \cup$ EM(r, i), $i \geqslant 1$. We denote an interpretation graph of EM(r, i) (with a variable bound to d) by $G(r, d, i)$. In general, an EM(r, i) will have several interpretation graphs.

The algorithm starts with $G(r, c, 0)$, which is the graph with a set containing only one node (q_s, c) ((q_s, c) is called the source node) and with no arcs. Here q_s is the initial state of all EM(r, i), $i \geqslant 1$. During the ith iteration of the main loop, $G(r, c', i-1)$ will be extended to $G(r, c'', i)$. Note that here c, c', c'' are different constants to which one of the variables appearing in the predicate of the query is bound. This extension is done by performing a depth-first traversal (i.e. $G(r, c', i-1)$ is traversed using a depth-first search strategy.) When $i = 1$, the traversal starts from the node (q_s, c). All paths not containing arcs labelled with derived predicates are traversed. Whenever a node (q, u) not visited before is entered, all transitions in EM(r, i) leaving q are examined. For any transition $q \xrightarrow{a} q'$ where a is a base predicate and for any term v such that $a(u, v)$ is true and the node (q', v) has not yet been generated, the algorithm generates (q', v) and continues the traversal from this node.

At the end of the iteration, it is examined whether or not any new nodes (q, u) (which are

called extension or continuation points) have been generated, where EM(r, i) contains a transition leaving q and labelled with a derived predicate. If not, the algorithm terminates, and the answers to the query will be $Ans = \{(u, v) \mid$ for some i, (q_s^i, u), $(q_f^i, v) \in G(r, u, i)\}$, where q_s^i and q_f^i are the initial and the final state of EM(r, i), respectively. Otherwise, the algorithm starts a new iteration, the $(i+1)$th. The following example helps for illustration. (See ref.[21] for a detailed description.)

Example 2.2. Consider the following program:

Rules:

$$rp(x, y) :- \text{flat}(x, y), \quad (2.4)$$

$$rp(x, y) :- \text{up}(x, z),\ rp(z, w),\ \text{down}(w, y). \quad (2.5)$$

Facts:

$$\text{up}(a_1, a_2),\ \text{up}(a_1, a_3),\ \text{up}(a_2, a_1),\ \text{up}(a_2, a_3),$$
$$\text{flat}(a_3, b_3),$$
$$\text{down}(b_3, b_2),\ \text{down}(b_2, b_1).$$

The program is a binary-chain program, and the predicate rp is linearly recursive. The equation for rp is

$$rp = \text{flat} \cup \text{up} \cdot rp \cdot \text{down}.$$

The corresponding automaton is shown in figure 2.

Given the query ? -$rp(a_1, y)$, the algorithm proposed by Grahne et al. will traverse the graph shown in fig. 3. The corresponding EM(r, i)'s are shown in figure 4.

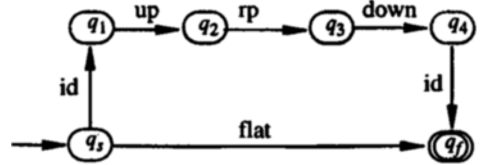


Fig. 2. Automaton $M(e_{rp})$ for expression e_{rp} = flat ∪ up · rp · down. Here q_s is the initial state and q_f is the final state. The symbol "id" is interpreted as the identity relation.

Now we trace the evaluation for a better understanding.

In the beginning, EM(rp,0) is the initial state q_s and $G(rp, a_1, 0)$ contains only node (q_s, a_1). During the first iteration, EM(rp,1) will be established, which is as in fig. 2; and $G(rp, a_1, 1)$ will be traversed, producing two intermediate answers; $rp(a_3, b_3)$, $rp(a_1, b_2)$. The portion enclosed by a broken line in fig. 3 shows $G(rp, a_1, 1)$. Whenever predicate rp is encountered during the traversal, EM(rp,2) will be generated. It is just a copy of EM(rp,1). Then, $G(rp, a_2, 2)$ is traversed, evaluating another group of answers: $rp(a_3, b_3)$, $rp(a_2, b_2)$, $rp(a_1, b_1)$. In a similar way, we construct EM(rp,3) and traverse $G(rp, a_1, 3)$. The process repeats until no new $G(rp, cont, i)$ for some $cont$ and $i > 0$ can be generated or the upper bound on the number of iterations (established manually) is reached. In fact, Grahne's algorithm for this example does not terminate if no such upper bound is established since similar nodes can be met infinitely many times due to cyclic data.

3 Refined algorithm

Now we present our refined algorithm. First, in subsection 3.1, we present the concept of subsumption checks. Then, we discuss several refinements based on this concept in subsection 3.2. Next, in subsection 3.3, the concept of linear cycle covers is proposed, which makes the refinment

Fig. 3. Graph $(rp, a_1, 1) \cup G(rp, a_2, 2) \cup G(rp, a_3, 2) \cup G(rp, a_3, 3)$ with respect to Example 2.2.

Fig. 4. Hierarchy of automata $\mathrm{EM}(rp, 0) \cup \cdots \cup \mathrm{EM}(rp, 3)$ with respect to Example 2.2.

implementable.

3.1. Subsumption checks

As described in sec. 2, Grahne's algorithm has no mechanism to do subsumption checks. Therefore, much redundant work will be done due to repeated accesses to similar portions of a graph. Thus, the first step of the refinement is to develop an efficient algorithm for doing subsumption checks. Then, based on the checks of similarities, we try to eliminate a lot of redundant computations by avoiding the traversal of similar subgraphs and by generating answers directly in terms of "path information" stored explicitly. First, we have the following two definitions.

Definition 3.1. A substitution θ is a finite set of the form $\{v_1/t_1, \cdots, v_n/t_n\}$, 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 $v_1, \cdots, v_n$ are distinct. Each element v_i/t_i is called a binding for v_i. θ is called a ground substitution if the t_i are all ground terms. θ is called a variable-pure substitution if the t_i are all variables.

Definition 3.2. Let s and t be two predicates. We say that s subsumes t if there exists a substitution $\theta = \{v_1/t_1, \cdots, v_n/t_n\}$ 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 $t_i(i = 1, \cdots, n)$.

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

Definition 3.3. A node of the form: (q', c') is subsumed by a node of the form: (q'', c'') if $c' = c''$ and q' and q'' are two different appearances of the same state on two different levels of an automaton hierarchy.

For example, node (q''_s, a_1) of the graph shown in fig. 3 is subsumed by (q_s, a_1). In that graph, q_s and q''_s are the same state, appearing on the first and third levels, respectively.

In addition, a node is usually thought of as being subsumed by itself.

Definition 3.4. A repeated incomplete node (RIN) is a node which is subsumed by a previous node which has appeared earlier on the same path as the RIN.

For example, node (q''_s, a_1) of the graph shown in fig. 3 is an RIN because it is subsumed by (q_s, a_1) which has appeared earlier on the same path.

The RINs are the only nodes which cannot be traversed further during the traversal process. However, cutting off such a path in the graph may affect the completeness, because some answers relying on this node cannot be evaluated. Therefore, a mechanism is needed to evaluate the remaining answers in some way.

Definition 3.5. A repeated complete node (RCN) is a node which is subsumed by a previous node which has appeared earlier but not on the same path as the RCN.

For example, node (q''_s, a_3) of the graph shown in fig. 3 is an RCN because it is subsumed by (q'_s, a_3) which has appeared earlier but not on the same path.

From the above definitions, we see that there are two kinds of subsumption checks which must be handled differently. When an RIN is encountered, the traversal should be suspended and the corresponding cycle should be recorded explicitly, while when an RCN is encountered, it should be expanded immediately using the answers already found. In addition, as we will see later, the ways in which RINs and RCNs are used to speed up the evaluation are different. However, distinguishing RCNs from RINs 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 with the technique for isolating the strongly connected components (SCC) of a directed graph[24] in such a way that the task can be done in linear time.

In what follows, we describe this method in detail.

Note that in fig. 3 the node (q''_s, a_3) is an RCN (subsumed by (q'_s, a_3)) and the node (q''_s, a_1) is an RIN (subsumed by (q_s, a_1)). Because (q''_s, a_1) is subsumed by node (q_s, a_1) that has appeared earlier on the same path, we expect to extend a series of subgraphs similar to the first one from this node, which has already been traversed. Therefore, the algorithm will run infinitely if no control mechanism is provided. (To guarantee both the termination and the completeness, Grahne's method establishes an upper bound on the number of iterations which is sufficiently large to allow all the answers to be found.) Thus, the traversal along a cyclic path has to be cut off to guarantee the termination. However, cutting off a path may affect the completeness. We then have to record

cycles explicitly and evaluate the corresponding answers along the cycles in a subsequent phase. In contrast, each RCN must be handled immediately to get some new answers, which may be re-used in the subsequent traversals.

Below is a graph traversal algorithm which can isolate all cycles of a graph being traversed and at the same time recognize all RCNs of the graph in linear time. Combining this algorithm with techniques described in the next subsection, an optimal strategy for evaluating linear binary-chain programs can be obtained. Its time complexity remains linear.

For convenience, we call the graph shown in fig. 3 the interpretation graph, the partial graph left to the broken vertical line the up-graph and the partial graph right to the broken vertical line the down-graph. In addition, for ease of exposition, we define a character graph for an up-graph (down-graph) as follows.

Definition 3.6. A character graph for an up-graph (down-graph) is a digraph where there is a node for each node of the form $(q_s^i, u)((q_f^i, v))$ in the up-graph (down-graph) and an edge from node a to node b if and only if there is a path from a to b in the up-graph (down-graph), which contains no other nodes of the form $(q_s^i, u)((q_f^i, v))$.

For example, the character graph of the up-graph shown in fig. 3 is as shown in figure 5:

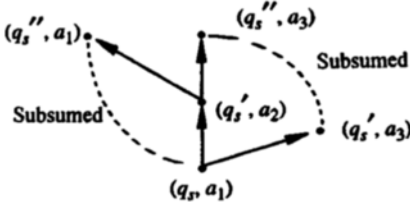


Fig. 5. Character graph.

The purpose of character graphs is to explain the control mechanism used in our method. In fact, it is sufficient to perform subsumption checks only on those nodes of an interpretation graph, which appear also in its character graphs (see next subsection). Therefore, we give the following algorithm over a character graph instead of an interpretation graph so as to illustrate the key ideas clearly.

We associate each node v of a character graph with three integers dfsnumber(v), toplnumber(v) and lowlink(v). dfsnumber is used to number the nodes of a character graph 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. It is used, here, to test whether a node is an RCN or an RIC. lowlink is used to number the nodes in such a way that if two nodes v and w are in the same strongly connected component, then lowlink(v) = lowlink(w). Therefore, it can be used to identify the "root" of a strongly connected component (a root is a node of a strongly connected component, which is first visited during the traversal). With the help of a stack structure, all strongly connected components can be feasibly found based on the calculation of lowlink values.

Essentially, the algorithm presented below is a modified version of Tarjan's algorithm[24]. The difference between them consists in the use of toplnumber in the modified algorithm, which facilitates the identification of a strongly connected component. (In the original algorithm, a stack structure must be searched to do this.) In addition, for our purposes, each RCN is marked.

At last, we notice that the subsumption check can always be done in constant time. Suppose that node u subsumes node v. Then u and v must be of the form (q, c) and (q', c), respectively, where q and q' are two different states and c is a domain element. If the input base relations are stored as graphs and each visited element is marked with the state associated with it when it is ac-

cessed at the first time, we know that any newly generated node (during the graph traversal) with a marked element is a subsumed one.

```
procedure graph-algo(v)(* depth-first traversal of a graph rooted at v *)
  begin
    i := 0; j := 0; (* i and j are two global variables, used to calculate dfsnumber and toplnum-
                    ber, 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 strongly connected components if any *)
    generate all sons of v if they exist;
  for each son w of v do
    begin
      if w is not topologically numbered then
        toplnumber(w): = 0; (* when a node is encountered at the first time, its toplnumber val-
              ue is 0. *)
    end
  for each son w of v do
    begin
        subsumption checking for w;
      if w is not subsumed by any node then
        begin
          call graph-search(w); (* go deeper into the graph *)
          lowlink (v) := min(lowlink(v), dfsnumber(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 (n) > 0 then (* if u is typologically numbered, it cannot be an ancestor
                    node of v. *)
          mark w to be an RCN;
        else (* a cycle is encountered *)
          {mark w to be an RIN;
          lowlink(v) := min (lowlink(v), dfsnumber(u));}} (* this operation will make all
                          nodes of a strongly connected component have the same lowlink value
                          as the root. see ref.[24] *)
```

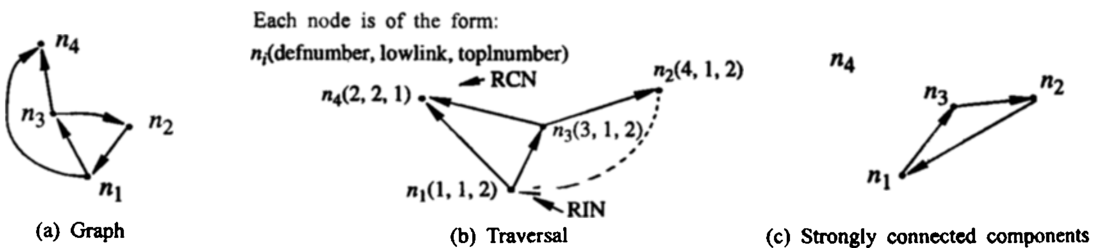
end
if (lowlink(v) = dfsnumber(v)) **then** (* v is a root of some strongly connected component *)
  begin
    while w on the stack S satisfies dfsnumber(w) $\geqslant$ dfsnumber(v) **do**
    {delete w from the stack S and put w in current strongly connected component (rooted at v);
    toplnumber(w) := j; } (* topological order numbering; all the nodes in an SCC have the same toplnumber *)
   j := j + 1; (* j is used to calculate toplnumber *)
  end
end

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

(i) lowlink is used to number the nodes top-down as dfsnumber; but it will be changed dynamically in such a way that all nodes in a SCC possess the same lowlink. Therefore, it is employed to identify the "root" of a SCC.

(ii) toplnumber is used to number the nodes 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 ref. [24]). Fig. 6 shows a directed graph, its defnumber, lowlink and toplnumber values, and its strongly connected components.



(a) Graph (b) Traversal (c) Strongly connected components

Fig. 6. Graph traversal.

3.2 Answer generation

Based on the mechanism for subsumption checks, we have developed three methods for generating answers directly in terms of different "path information": (i) answer generation for RCNs; (ii) answer generation when an up-graph (or a down-graph) contains cycles; and (iii) answer generation when both the up-graph and the down-graph contain cycles.

Due to limitation of space, we discuss here only the second method, from which the other two methods can be derived.

Answer generation when an up-graph (or a down-graph) contains cycles

Example 3.1. Continuing our running program. But suppose that the database contains the following facts:

flat(c_4, c_5),

up(c_3, c_4), up(c_3, c_2), up(c_2, c_3), up(c_2, c_8), up(c_8, c_3),

down(c_5, c_1), down(c_1, c_6), down(c_6, c_7), down(c_7, c_9).

Given the query ?-$rp(c_3, y)$, the algorithm proposed by Grahne et al. will traverse the graph shown in fig. 7. Because the nodes (q''_s, c_3) and (q'''_s, c_3) are subsumed by the node (q_s, c_3), we expect to extend a lot of similar subgraphs that have been traversed earlier.

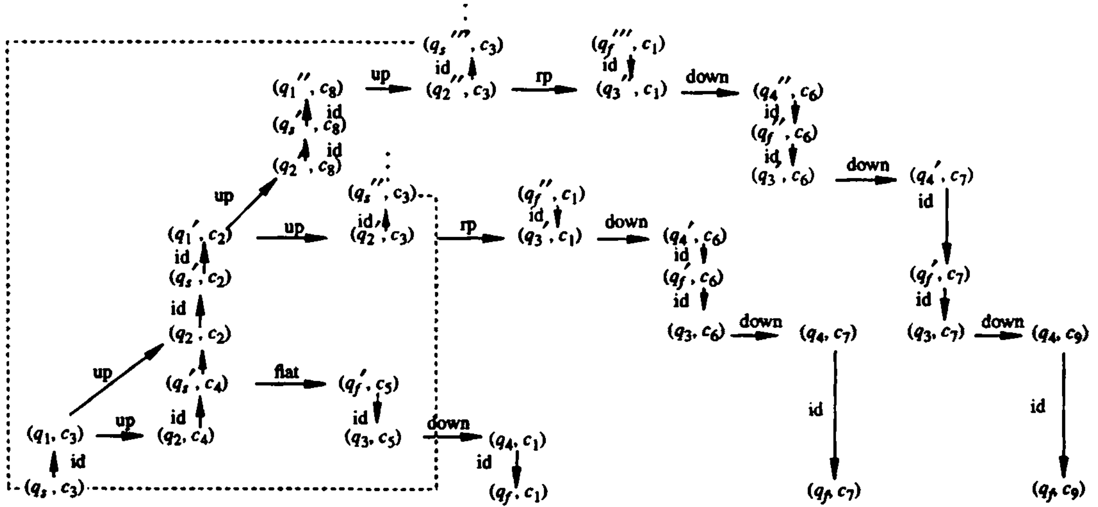


Fig. 7. Graph $G(rp, c_3, 1) \cup \cdots \cup G(rp, c_3, 4)$ with respect to Example 2.2.

Therefore, the traversal along a cycle should be cut off and an iteration process should be used to find the remaining answers. We do this as follows. By performing subsumption checks, two cycles will be recorded explicitly. One of them consists of the starting points: (q''_s, c_3), (q'_s, c_2) and (q_s, c_3). Traversing the corresponding path in the down-graph (the path from q''_f to q_f in the down-graph shown in fig. 7) repeatedly (each time with a newly produced value as the initial value), we will evaluate the following answers: $rp(c_3, c_1)$, $rp(c_2, c_6)$, $rp(c_3, c_7)$ and $rp(c_2, c_9)$. The other cyclic path consists of (q'''_s, c_3), (q''_s, c_8), (q'_s, c_2) and (q_s, c_3). Similarly, traversing the corresponding path in the down-graph (the path from q'''_f of q_f in the down-graph shown in fig. 7) repeatedly, we will produce $rp(c_3, c_1)$, $rp(c_8, c_6)$, $rp(c_2, c_7)$, $rp(c_3, c_9)$ and $rp(c_8, c_9)$. An observation shown that the answers evaluated along the second cycle can be directly generated from the answers produced along the first path. For example, we can directly generate $rp(c_8, c_6)$ from $rp(c_2, \underline{c_6})$ on the first cyclic path and the second node $((q''_s, \underline{c_8}))$ of the second cyclic path and $rp(c_2, \underline{c_7})$ from $rp(c_3, \underline{c_7})$ and the third node $((q'_s, \underline{c_2}))$ and so on, instead of traversing the path again. Fig. 8 helps to illustrate this feature.

Below we describe this method more formally.

Let C_1 be the first cycle $v_1 \leftarrow v_2 \leftarrow \cdots \leftarrow v_n \leftarrow v_1$ and $A_1 = \{a_1, \cdots, a_n, a_{n+1}, \cdots, a_{2n}, \cdots, a_{in}, \cdots, a_{in+j}\}$ the answer set evaluated along C_1, where i, j are integers and $0 \leqslant j \leqslant n$. (It should be noticed that each $a_l (1 \leqslant l \leqslant in + j)$ is a subset which is evaluated when the nodes of the form (q^{λ}_f, c) of the λ th level interpretation graph are encountered, where c stands for a constant and $l = rn + \lambda$ for some integer r.) Let C_2 be the second cycle $w_1 \leftarrow w_2 \leftarrow \cdots \leftarrow w_m \leftarrow w_1$. In addition, we define

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

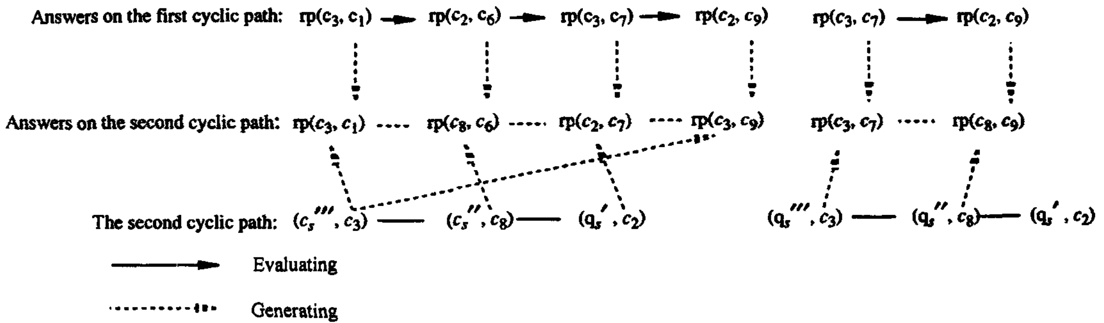


Fig. 8. Illustration of generating answers with respect to Example 3.1.

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

$$\cdots \cdots$$

$$A_i = \{a_{in+1}, \cdots, a_{in+j}\}.$$

Then, in terms of $A_k(1 \leqslant k \leqslant i)$ and C_2, we can generate the first part of answers for C_2 as shown in fig. 9. (Note that we do not need to compute all $A_k(1 \leqslant k \leqslant i)$. In practice, each time an A_k needs to be used in the computation, we shrink A_{k-1} by leaving out certain a_i's). Without loss of generality, we assume that $n \leqslant m$.

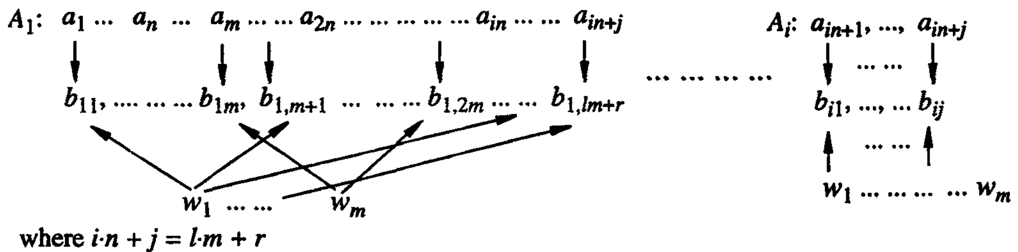


Fig. 9. Illustration of answer generation.

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 as shown in fig. 10(a) (hereafter, this process is called a merging operation.).

Then, we construct $B_k(1 \leqslant k \leqslant l)$ as shown in fig. 10(b) (hereafter, this process is called a separating operation).

In terms of C_1 and $B_k(2 \leqslant k \leqslant l)$, 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.

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

Proposition 3.1. *Let* $\{v_1, v_2, \cdots, v_n\}$ *and* $\{w_1, w_2, \cdots, w_m\}$ *be two cycles having the same starting point. Suppose that each* v_i *is of the form* (q_s^i, c_i) *and each* w_j *is of the form* (q_s^j, d_j). *Let*

$$b_1 = b_{11} \cup b_{21} \cup \dots \cup b_{i1},$$
$$\dots \dots$$
$$b_{lm+r} = b_{1,lm+r} \cup \dots \dots .$$

(a)

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

(b)

Fig. 10. Merging operation and separating operation.

(c_i, h) be an answer. Then, pair (d_j, h) corresponds to an answer if i, j satisfy the equation:

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

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

Proof. The proof of the proposition is similar to Proposition 2.1 and Proposition 2.2 of reference [5].

3.3 Linear cycle covers for a strongly connected graph

Obviously, we have to first enumerate all cycles of an SCC prior to the direct generation of answers in the case of cyclic data. Unfortunately, this cannot always be done in linear time. Using Johnson's algorithm[25], this task requires time $O(n_{scc} + e_{scc}) c_{scc}$, where n_{scc} and e_{scc} are numbers of nodes and edges, respectively and c_{scc} is the number of cycles in the SCC. In many cases, the number of cycles c_{scc} can grow faster with n_{scc} than the exponential $2^{n_{scc}}$. For example, in a complete directed graph (CDG) with n nodes there are exactly

$$\sum_{i=1}^{n-1} \binom{n}{n - i + 1} (n - i)!$$

cycles. In fig. 11, we show a complete directed graph with 4 nodes (4-CDG), which contains 20 cycles.

In addition, the technique for generating answers cannot be applied efficiently to the graph shown in fig. 12(a), since there is no common node among the cycles contained in it and the task of selecting a cycle, along which the answers will be evaluated, becomes difficult. (Remember that for the first cycle, the answers have to be evaluated by the graph traversal).

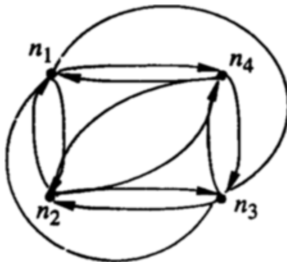


Fig. 11. Complete graph with 4 nodes.

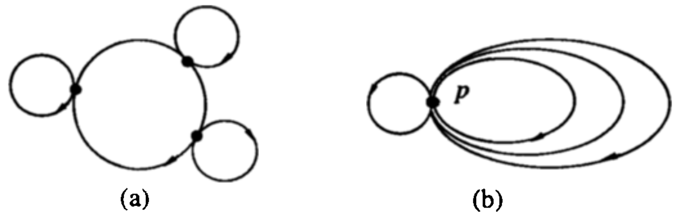


Fig. 12. Cycles without common nodes.

To overcome these difficulties, we define several new concepts and propose a method to make the technique for generating answers useful.

Definition 3.7. Let G be an SCC. A feedback node of G is a node contained in every cycle of G.

For example, node p in the graph shown in fig. 12(b) is a feedback node.

Definition 3.8. A set of cycles C contained in an SCC is called a cycle cover of the SCC, if each edge of the SCC appears at least in one cycle of C. We denote the cardinality of C by $|C|$.

Definition 3.9. A cycle cover w.r.t. an SCC is a linear cycle cover if its cardinality is in the order $O(e_{scc})$, where e_{scc} is the number of the edges of the SCC.

Based on the above definitions, we develop a method to underlie the technique for generating answers. The main idea behind it is to identify first a feedback node and then, using the feedback node as a pivot, to perform a depth-first search to find a linear cycle cover. Using Garey's algorithm[26], one feedback node in an SCC can be found in linear time if any. Thus, we can use Garey's algorithm to check for an SCC whether some feedback node exists. If not, we traverse the SCC in a normal way. Otherwise, if a feedback point exists, we take it as the start node and execute the following algorithm to find a linear cycle cover of the SCC, to which the technique for generating answers can be applied.

Below is an algorithm for finding a linear cycle cover w.r.t. an SCC containing a feedback node. In the algorithm each node n is marked with a boolean value val(n) and in the beginning, all val(n)'s are set to 0. During the depth-first traversal (starting from the feedback node), we set val(n) to 1, when the corresponding node n is visited for the first time. Then, we have a simple property that when a node n with val(n) = 1 is met, at least one cycle through the feedback node and n must already be generated. In terms of this property, when a node n with val(n) = 1 is encountered, a new cycle can immediately be constructed by taking the current path and the path from n to the feedback node (which appears in some already generated cycle) together.

Procedure cycle-cover(*fbn*, SCC)(* *fbn* is a feedback node of SCC. *)

```
1  begin
2    for all node in SCC do
3      val(node) := 0;
4    N := fbn;
5    val(fbn) := 1;
6    search(fbn);
7  end

procedure search(n)
8  begin
9    generate all sons of n;
10    for each son m do
11      if v(m) = 0 then {v(m) := 1; search(m)}
12        else
13          {take one of the paths (from m to N) which have been visited and the current path
             from N to m to form a new cycle; store the new cycle};
14  end
```

In the following, we show that the cycles enumerated by cycle-cover( ) constitute a linear cycle cover.

Proposition 3.2. *Let C be a set of cycles (of an arbitrary SCC) found by cycle-cover( ).*

Then C is a cycle cover for the SCC.

Proof. Assume, to the contrary, that C is not a cycle cover. Then there exists at least one edge in the SCC that does not appear in any cycle in C. Suppose (n_i, n_j) is such an edge. Thus, (n_i, n_j) has not been visited by cycle-cover( ). Obviously, n_i is not visited either. Otherwise, from lines 9—11, we see that (n_i, n_j) will certainly be traversed after n_i is visited, which contradicts the assumption. For the same reason, any edge with n_i being the tail (for an edge (n, m), n is the tail and m is the head of the edge) will not be visited. Consider one of such edges, say (n_k, n_i). Then n_k is also unvisited. In this way, we can find a sequence $n_i, n_k, \cdots, n_l$ with $n_l = N$, which is not visited by cycle-cover( ). But it contradicts the behaviour of the algorithm. Thus, C is a cycle cover.

Proposition 3.3. *The cycle cover found by cycle-cover( ) is linear.*

Proof. We denote the number of the found cycles passing a set of nodes $n_1, n_2, \cdots, n_k$ by numcycles$(n_1, n_2, \cdots, n_k)$. If the indegree and outdegree of each node n_i in an SCC are denoted as in(n_i) and out(n_i), respectively, the number of the cycles found by cycle-cover( ) can be computed as follows:

$$\text{numcycles} = \text{numcycles}(v), \tag{3.1}$$

where v is the start node (a feedback point).

$$\begin{aligned} \text{numcycles}(v) &= \sum_{i=1}^{\text{out}(v)} \text{numcycles}(v, n_i) \\ &= \sum_{i=1}^{\text{out}(v)} \sum_{j=1}^{\text{out}(n_i)} \text{numcycles}(v, n_i, n_{ij}), \end{aligned} \tag{3.2}$$

where each n_i stands for a son of v, while each n_{ij} stands for a son of n_i.

If we use outedges(n_i) to denote the set of edges incident out of n_i and inedges(n_{ij}) the edges incident into n_{ij}, we have $\bigcup_i \text{outedges}(n_i) = \bigcup_{i,j} \text{inedges}(n_{ij})$. This equation can be proved as follows. First, we have $\bigcup_i \text{outedges}(n_i) \subseteq \bigcup_{i,j} \text{inedges}(n_{ij})$ (see fig.13 for illustration). Then we prove $\bigcup_i \text{outedges}(n_i) \supseteq \bigcup_{i,j} \text{inedges}(n_{ij})$. Assume, to the contrary, that there exists a node n such that for some $n_k (n, n_k) \notin \bigcup_i \text{outedges}(n_i)$ but $\in \bigcup_{i,j} \text{inedges}(n_{ij})$. In terms of the property of SCCs, there must be a path connecting n_k and n as shown in fig. 14. Then the cycle composed of this path and the edge (n, n_k) does not contain v, which is a contradiction with the fact that v is a common node of all cycles contained in the SCC.

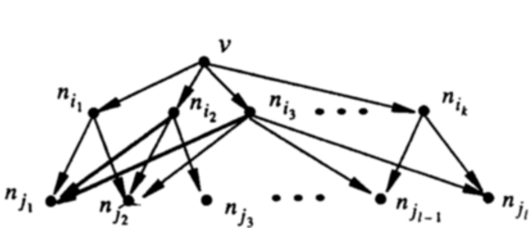


Fig. 13. Illustration for the relationship between indegrees and outdegrees.

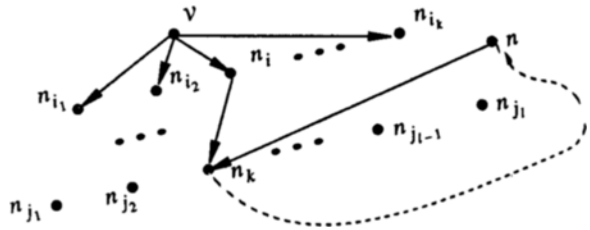


Fig. 14. A cycle which does not contain v.

In terms of the above analysis, (3.2) can be rewritten as follows:

$$\sum_{i=1}^{\text{out}(v)}\sum_{j=1}^{\text{out}(n_i)}\text{numcycles}(v, n_i, n_{ij}) = \sum_{j}\sum_{m=1}^{\text{in}(n_j)}\text{numcycles}(v, n_{i_m}, n_j). \tag{3.3}$$

Note that on the right-hand side of (3.3) each n_{i_m} stands for a son of v. Since in the above algorithm only one of the paths which follows a node n (from n to N) is taken to form a new cycle when n is met once again (see line 13 in cycle-cover(); see also the thick edges shown in fig. 13 for illustration. Along each of them only one path will be considered), we have the following equation.

$$\begin{aligned}\sum_{j}\sum_{m=1}^{\text{in}(n_j)}\text{numcycles}(v, n_{i_m}, n_j) &= \sum_{j}(\text{numcycles}(v, n_i^j, n_j) + \text{in}(n_j) - 1)\\ &= \sum_{j}\text{numcycles}(b, n_i^j, n_j) + \sum_{j}\text{in}(n_j) - j\\ &= \sum_{k}\text{numcycles}(v, n_i^j, n_j^k, n_k) + \sum_{j}\text{in}(n_j) + \sum_{k}\text{in}(n_k) - (j + k)\\ &= \cdots\\ &= \text{in}(v) + \sum_{n_l \neq v}\text{in}(n_l) - (n_{scc} - \text{out}(v))\\ &= o(e_{scc} - n_{scc} + 1).\end{aligned} \tag{3.4}$$

Here n_i^j stands for a father node of n_j, through which n_j is visited for the first time. Therefore, the cycle cover found by cycle-cover() is linear.

In the worst case, the length of a cycle is in the order of $O(n_{scc})$. Then the space complexity of cycle-cover() will be $O(n_{scc} \cdot e_{scc})$. It is not desired for the optimization purpose. In addition, if we generate answers simply along each cycle without any control, some answers may be repeatedly produced many times due to the common part of cycles. Therefore, we do not enumerate all the cycles of a cycle cover using cycle-cover(), but integrate its idea into the process for generating answers.

Another important question is whether the answers produced by traversing an entire SCC is the same as those produced by traversing only one of its cycle covers. In the following, we prove a proposition to give a positive reply to this.

Definition 3.10. Let p_i and p_j be two answers to a recursive query. If p_i can be evaluated on p_j, we say that p_j is a predecessor of p_i, and p_i is a successor of p_j, denoted predecessor(p_i) and successor(p_j) respectively.

Proposition 3.4. *Let A_{scc} and $A_{cycle\text{-}cover}$ be two sets of answers produced by traversing some SCC in the up-graph, which contains at least one feedback node, and by one of its cycle covers, respectively. Then $A_{scc} = A_{cycle\text{-}cover}$.*

Proof. For any $a \in A_{\text{cycle-cover}}$, we have trivially $a \in A_{scc}$. Then $A_{\text{cycle-cover}} \subseteq A_{scc}$. In the following, we prove that $A_{scc} \subseteq A_{\text{cycle-cover}}$. For any $a \in A_{scc}$, there must be a path $v_0 \to v_1 \to v_2 \to \cdots \to v_i$ in the SCC (with v_0 being a feedback node and each $v_j (0 < j \leq i)$ being also a node in the character graph) such that A_{v_0} = predecessor(A_{v_1}), A_{v_1} = predecessor(A_{v_2}), $\cdots$, $A_{v_{i-1}}$ = predecessor(A_{v_i}) and $a \in A_{v_i}$, where $A_{v_j} (0 \leq j \leq i)$ stands for a set of answers produced by traversing path $v_j \to v_{j+1}$ (corresponding to an edge in the character graph) and the corresponding path in the down-graph. In terms of the property of cycle covers, there is a set of cycles C in cycle-cover such that the paths $v_0 \to v_1$, $v_1 \to v_2, \cdots$, and $v_{i-1} \to v_i$ are covered by C. Then, by the first traversal along C, we will get A_{v_0}. By the second traversal along C, we will get A_{v_1} and so on. Obviously, by the ith traversal a-

long C, A_{v_i} will be produced. Therefore, $a \in A_{\text{cycle-cover}}$. Thus, $A_{scc} \subseteq A_{\text{cycle-cover}}$, which completes the proof.

4 Complexity analysis and comparison

In the analysis below, we assume that the rule set is like that of Example 2.1. e and n denote the number of edges and nodes, respectively, in the graph representing the input relations.

At first, we point out that our algorithm uniformly outperforms Grahne's algorithm. On the one hand, the subsumption check is unproblematic, since it requires only linear time. On the other hand, our method will traverse a much smaller graph if there exist RCNs. Suppose that after node i is encountered, e_i edges will be traversed. Then $\sum_i e_i < e$, because any repeated accesses to an edge are avoided by means of removing each subgraph rooted at some RCN. (We generate the corresponding answers in a similar way as discussed in 3.2). In addition, in the case of cyclic data, each cycle is traversed only once. If the length of the longest cycle in up is m and the length of the longest cycle in down is l, then the cost of our algorithm is $O(m + l)$. In sum, the time complexity of our algorithm is $O(n + e)$.

In contrast, the cost of Grahne's algorithm is $O(n^3)$ in the case of cyclic data. This is because $m \cdot l$ iterations of the main loop must be performed for the case mentioned above. In the worst case, every single execution of the main loop may entail accessing $O(n)$ nodes.

Various strategies for processing recursive queries have been proposed. However, all these algorithms require at least $O(e^2)$ or $O(en)$ time[8,11,18,19,27—29].

When the input relations contain no cycles, the cost of Counting is $O(en)$, better than that of Magic Sets which is $O(e^2)$[30]. For cyclic cases, ref.[17] proposed a method which takes $O(ne)$ time. However, this method requires $O(e^2)$ time for preprocessing. Ref.[31] presented an algorithm that runs a counting method until a cycle is detected, then switches over to Magic Sets. This algorithm is also $O(e^2)$ on cyclic data. The method proposed in ref.[27] requires $O(n^3)$ time. But in some cases, the complexity can be reduced to $O(n^2)$. The algorithm proposed in ref.[29] requires $O(ne)$ time.

Note that a little more space is required by our algorithm than Grahne's algorithm. This is because each node, in our method, occupies not only the state and the domain (range) value, but also three integers. If the space complexity of Grahne's algorithm is S, then $4 \cdot S$ space is required by ours.

At last, it should be pointed out that to use our algorithm to treat with rules in the normal Horn-clause form, some more time is needed to make the rule transformation. But this task can be shifted to a pre-process phase, which has no effect on the actual execution of the algorithm.

5 Conclusion

In this paper, a graph traversal algorithm has been presented which is much more efficient than Grahne's algorithm. The key idea of the improvement is to recognize all the similar portions of a graph and to produce all the relevant answers by constructing only one of them. In the case of acyclic data, the algorithm optimizes the evaluation by traversing each path only once and generating the remaining answers directly from the answers already found. In the case of cyclic data, a lot of graph op-

timization techniques are employed to speed up the evaluation, such as the combination of Tarjan's algorithm and the topological numbering, and the algorithm for finding feedback nodes as well as for cycle covers. In this way, most of answers for cyclic paths can also be generated directly by traversing the corresponding subgraphs. Since traversing a path requires access to the external storage or search of large relations but the "generating" operations require only access to small data sets and happen always in main memory, we may suppose that the time complexity of generating answers is $O(1)$, and therefore a linear time is achieved.

References

1 Chang, C., On the evaluation of queries containing derived relations in relational database, in *Advances in Data Base Theory*, Vo.1, New York: Plenum Press, 1981.

2 Chen, Y., Harder, T., Improving RQA/FQI recursive query algorithm, in *Proceedings ISMM-First Int. Conf. on Information and Knowledge Management, Baltimore, Maryland, Nov.* 1992, New York: ACM, 1992, 106—115.

3 Chen, Y., A bottom-up query evaluation method for stratified databases, in *Proceedings of 9th International Conference on Data Engineering, Vienna, Austria, April* 1993, California: IEEE, 1993, 568—575.

4 Chen, Y., Harder, T., On the optimal top-down evaluation of recursive queries, in *Proc. of 5th Int. Conf. on Database and Expert Systems Applications, Greece, Athens, Sept.* 1994 (ed. Karagiannis, D.), Berlin: Springer-Verlag, 1994, 47—56.

5 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.

6 Chen, Y., On the bottom-up evaluation of recursive queries, *Int. J. Intelligent Systems*, 1996, 11(10): 807.

7 Chen, Y., Magic sets and stratified databases, *Int. J. Intelligent Systems*, 1997, 12(3): 203.

8 Han, J., Chain-split evaluation in deductive databases, *IEEE Trans. Knowledge and Data Engineering*, 1995, 7: 261.

9 Ullman, J. D., *Principles of Databases and Knowledge-base Systems*, Rockville: Computer Science Press, 1989.

10 Shapiro, S., Mckay, D., Inference with recursive rules, in *Proceedings of the 1st Annual National Conference on Artificial Intelligence* (ed. Bazler, R.), California: MIT Press, 1980, 151—156.

11 Bancilhon, F., Naive evaluation of recursively defined relations, *On Knowledge Base Management Systems-Integrating Database and AI Systems* (ed. Bancihon, F.), Berlin: Springer-Verlag, 1985, 165—178.

12 Vieille, L., From QSQ to QoSaQ: Global optimization of recursive queries, in *Proc. 2nd Int. Conf. on Expert Database System, Charleston* (ed. Kerschberg, L.), Virginia: Benjamin Cummings, 1988, 743—748.

13 Nejdl, W., Recursive strategies for answering recursive queries-The RQA/FQI strategy, in *Proc. 13th VLDB Conf., Brighton, England* (ed. Stocker, P.M.), California: Morgan Kaufmann, 1987, 43—50.

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

15 Han, J., Zeng, K., Lu, T., Normalization of linear recursion in deductive databases, in *Proc. of the 9th International Conf. on Data Engineering, Vienna, Austria, April* 1993, California: IEEE, 1993, 559—567.

16 Han, J., Chen, S., Graphic representation of linear recursive rules, *International Journal of Intelligent Systems*, 1992, 7: 317.

17 Han, J., Henschen, L. J., The level-cycle merging method, in *Proc. of the 1st International Conf. on Deductive and Object-oriented Databases, Kyoto* (ed. Kim, W.), Berlin: Springer-Verlag, 1989, 113—129.

18 Bancilhon, F., Maier, D., Sagiv, Y. et al., Magic sets and other strange ways to implement logic programs, in *Proc. 5th ACM Symp. Principles of Database Systems, Cambridge, MA, March* 1986 (ed. Zaniolo, C.), California: ACM, 1986, 1—15.

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

20 Grahne, G., Sippo, S., Soisalon-Soininen, E., Efficient evaluation for a subset of recursive queries, in *Proceedings of ACM-PODS*, California: ACM, 1987, 284—293.

21 Grahne, G., Sippo, S., Soisalon-Soininen, E., Efficient evaluation for a subset of recursive queries, *J. Logic Programming*, 1991, 10: 301.

22 Lloyd, J. W., *Foundations of Logic Programming*, Berlin: Springer-Verlag, 1987.

23 Kanamori, T., Kawamura, T., Abstract interpretation based on OLDT resolution, *J. Logic. Programming*, 1993, 15: 1.

24 Tarjan, R., Depth-first search and linear graph algorithm, *SIAM J. Comput.*, 1972, 1(2): 146.

25 Johnson, D. B., Finding all elementary circuits of a directed graph, *SIAM J. Comput.*, 1975, 4(1): 50.

26 Garey, M. R., Tarjan, R. E., A linear-time algorithm for finding all feedback vertices, *Information Processing Letters*, 1978, 7(6): 274.

27 Aly, H., Ozsoyoglu, Z. M., Synchronized counting method, in *Proc. of the 5th International Conf. on Data Engineering, Los Angeles*, California: IEEE, 1989, 366—373.

28 Bancilhon, F., Ramakrishnan, R., An amateur's introduction to recursive query processing strategies, in *Proc.* 1986 *ACM-SIGMOD Conf. Management of Data, Washington, DC, May* 1986, California: ACM, 1986, 16—52.

29 Wu, C., Henschen, L. J., Answering linear recursive queries in cyclic databases, in *Proc. of the* 1988 *International Conf. on Fifth Gen. Computer Systems, Tokyo*, California: ACM, 1988, 16—52.

30 Marchetti-Spaccamela, A., Pelaggi, A., Sacca, D., Worst case complexity analysis of methods for logic query implementation, in *Proc. of ACM-PODS* 87, California: ACM, 1987, 294—301.

31 Sacca, D., Zaniolo, C., 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*, California: ACM, 1986, 16—23.