On the Graph Traversal and Linear Binary-Chain Programs

Yangjun Chen

Abstract—Grahne et al. have presented a graph algorithm for evaluating a subset of recursive queries [14], [15]. 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. Here, we describe a new algorithm 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. Furthermore, we speed up the evaluation of cyclic data 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, our algorithm achieves a linear time complexity for both acyclic and cyclic data.

Index Terms—Deductive database, binary-chain programs, automaton, graph searching, feedback node.

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 the notion of knowledge-based systems, 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 [6], [7], [8], [9], [10], [11], [12], [19], [30]). These strategies include evaluation methods such as naive evaluation [6], [27], seminaive evaluation [2], query/subquery [31], RQA/FQI [25], Henschen-Naqvi [20], and the methods used in compiling recursive queries [16], [17], [18], [19], [20]. 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 [3], counting [3], and their generalized versions [5], [11]. 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. [14], [15]. (We shall subsequently refer to this method as Grahne's algorithm.) 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 [34]. Moreover, it is often the case that the Datalog programs computing many interesting examples of recursive queries fall into the class of "binary-chain" programs [33].

The following is an interesting program defining the recursion $scsg$ (same-country same generation relatives).

$$scsg(x,y) \leftarrow parent(x,x_1), scsg(x_1,y_1), same_country(x_1,y_1), parent(y,y_1).$$
$$scsg(x,y) \leftarrow sibling(x,y).$$
$$same_country(x,y) \leftarrow birth_country(x,v), birth_country(y,v).$$

In addition, compared to the SLD resolution [22] and its different variants [21], [25], [31], the graph method by itself is advantageous due to the following two benefits:

- Repeated firing of rules with the same head predicate can be avoided (see Section 3.1);
- 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 (see Section 3.2).

Grahne et al.'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*). For example, the above program can be transformed into the following equations:

$$scsg = sibling \cup parent \cdot scsg \cdot same_country \cdot parent$$
$$same_country = birth_country \cdot birth_country.$$

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. Here, a pair (a, b) satisfying a predicate $p(x,y)$ (appearing in the equation) is represented as an edge labeled with p in the graph. This result means that evaluation problems for the predicate r reduces to graph 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

- *The author is with the Department of Business Computing, Winnipeg University, 515 Portage Ave., Manitoba, Canada, R3B 2E9. E-mail: ychen@uwinnipeg.ca.*

Manuscript received 20 Sept. 1999; revised 3 Jan. 2001; accepted 11 Apr. 2001.
For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number 110616.

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 an 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 always happen 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. That is, if the input graphs contains n nodes and e edges, our algorithm needs only $O(e)$ time, while the existing methods need $O(ne)$ time or more.

The paper has the following main contributions:

1. Tarjan's algorithm for identifying strongly connected components (SCC) of a directed graph is modified and embedded into a graph traversal algorithm for evaluating binary-chain programs so that subsumption checking can be done in linear time. Using Tarjan's algorithm, we can recognize two kinds of subsumed nodes during a graph traversal. One kind of subsumed nodes (called RINs) implies a similar part of the graph, which needs not be traversed and the corresponding answers can be generated directly from the answers already found and the relevant path information. Another kind of subsumed nodes (called RCNs) shows an infinite expansion. Therefore, special treatment should be done to avoid infinity but without damaging the completeness. In Grahne's algorithm, there is no subsumption checking at all.
2. A new technique for handling cyclic data is developed to minimize time complexity. As mentioned above, an RIN implies an SCC identified during a graph traversal. Therefore, whenever an RIN is encountered, the evaluation should be suspended to avoid nontermination. However, the answers which may be produced if the SCC is traversed will be lost. Thus, they should be found in some way else. This can be done by executing iterations over the SCC until the fix point is reached. This process is also elaborated. We evaluate the answers only along one cycle of the SCC. The answers for the other cycles will be generated in terms of the answers already found. Note that, by "answer generation," we mean that we produce them not by graph traversal but by "analyzing" the intermediate answers and path information available, which is in general much more efficient than normal answer evaluation.

The paper is organized as follows: In Secion 2, we introduce the necessary terminology from [14], [15]. In Section 3, we briefly describe the main idea of Grahne's algorithm. In Section 4, we give our refined graph traversal algorithm for evaluating linear binary-chain programs. In Section 5, we compare the computational complexity of our algorithm with the existing strategies. Section 6 is a short conclusion.

2 Basic Concepts

A *Datalog program* consists of a finite set of *rules* of the form

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

q is called the *head* and the conjunction $p_1, p_2, \ldots, p_m$ is called the *body* of the rule. q is a predicate while each p_i may be either a predicate or a negated predicate. (If p represents a predicate, then $\neg p$ represents a negated one.) When $m = 0$, the rule is of the form

$$q \leftarrow$$

and is known as a *unit clause*.

A predicate $p(t_1, t_2, \ldots, t_n)$, $n \geq 0$, is a *ground predicate* when all of its terms $t_1, t_2, \ldots, t_n$, are constants. A *ground rule* is one defined in which each predicate in the rule is *ground*. A *fact* is a ground unit clause. The definition of a predicate p is the set of rules which have p as the head predicate. A *base predicate* is defined solely by facts. The set of facts in the database is also known as the extensional database. A rule that is not a fact is known as a *derivation rule*. A *derived predicate* is a predicate which is defined solely by derivation rules. The set of derivation rules is also known as the intensional database. As usual, we assume that the base predicate and the derived predicates form two disjoint sets, that is, no base predicate appears in the head of a rule with a nonempty body.

A predicate (or relation) with two argument positions is called a *binary predicate* (or *binary relation*). For a binary predicate p, the set of values assumed by the first argument of p is called the *domain* of p, and the set of values assumed by the second argument of p is called the *range* of p.

A rule of the form

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

where $m \geq 0$ and $x_1, \ldots, x_m + 1$ 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 iff 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 $p \Leftarrow q$, 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

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

is a *right-linear rule* if none of the predicates $p_1, \ldots, p_{m-1}$ is mutually recursive to p, and a *left-linear rule* if none of the predicates $p_2, \ldots, p_m$ 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 predicates is called the *left-hand side relations* and those right to the recursive predicate is called the *right-hand side relations*.

3 GRAHNE'S METHOD

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

3.1 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

$$r = E_r,$$

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 addition, the following constraints hold:

- For each derived predicate, there is exactly one equation;
- 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 original program.

The transformation process of any linear binary-chain program into a system of equations can be summarized as follows:

1. Construct an initial equation for each derived predicate. Let r be a derived predicate and $r \leftarrow e_1, \ldots, r \leftarrow e_n$ be n rules having r in the head. Then, construct an equation of the following form:

$$r = e'_1 \cup \ldots \cup e'_n,$$

   where each $e'_i (i = 1, \ldots, n)$ is a set of predicate symbols connected with "$\cdot$," which appear in e_i.
2. Transform any equation of the form: $r = e_0 \cup r \cdot e_1 \cup \ldots \cup r \cdot e_n$ into $r = e_0 \cup r \cdot (e_1 \cup \ldots \cup e_n)$. (Similarly, any equation of the form: $r = e_0 \cup e_1 \cdot r \cup \ldots \cup e_n \cdot r$ will be transformed into $r = e_0 \cup (e_1 \cup \ldots e_n) \cdot r$.) We notice that an equation like $r = e_1 \cdot r$ is meaningless since there is no initial part of r. Then, it can be removed. Further, an equation like $r = e_0 \cup r$ can be replaced with $r = e_0$ to eliminate any useless part.
3. Replace any equation of the form: $r = e_0 \cup r \cdot e_1$ with $r = e_0 \cdot e_1^*$. (Similarly, replace $r = e_0 \cup e_1 \cdot r$ with $e_1^* \cdot e_0$.). In this way, any regular predicate is removed.
4. Eliminate any equation of the form: $r = e$, where e does not contain any predicate mutually recursive to r. Then, substitute e for every occurrence of r in the right-hand side of all the other equations.
5. Repeat steps 2 to 4 until nothing more can be done.

See the following example for illustration.

Example 3.1. Consider the following program:

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

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 method above:

1. $p = b \cdot (c \cdot b \cup d \cdot e)^* \cdot d \cdot a$,
2. $q = (c \cdot b \cup d \cdot e)^* \cdot d \cdot a$, and
3. $r = a \cup e \cdot (c \cdot b \cup d \cdot e)^* \cdot d \cdot a$.

For further details, please refer to the description in [15].

3.2 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, (1) given above can be represented as the automaton shown in Fig. 1.

Here, q_s, q_f, and $q_i (i = 1, 2, \ldots, 7)$ represent the initial, final, and intermediate states, respectively. The transitions labeled with "id" denote transitions on the empty string. A predicate symbol labeling a transition will be interpreted as the relation it denotes. Then, symbol "id" will be interpreted as the identity relation.

If $r = E_r$ is not recursive, an *interpretation graph* of $M(E_r)$ will be generated, which is a directed graph with a set of nodes (q, u), where q is a state in $M(E_r)$ and u is a domain element of some base relation labeling a transition leaving q, and with a set of edges of the form, $(q, u) - (q', v)$, where, for some base relation a, $q \overset{a}{\rightarrow} q'$ is a transition in $M(E_r)$ such that $a(u, v)$ is *true*. In this way, the query evaluation is reduced to a graph traversal problem. (Afterwards, we use the term *graph* to refer to a directed graph, since we do not discuss undirected ones at all.)

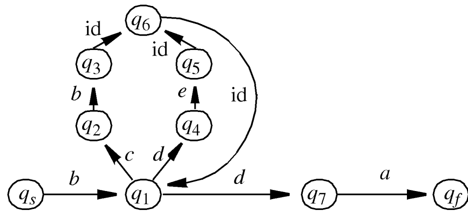


Fig. 1. Automaton for (1). Here, "id" is interpreted as the identity relation.

If $r = E_r$ is a recursive equation, a hierarchy of automata will be constructed in evaluating answers to the process can be described as follows:

1. The *i*th level in the hierarchy, denoted by $\mathrm{EM}(r,i)$, corresponds to the *i*th recursive call of r (which appears in E_r) with some of its variables bound to constants. First, $\mathrm{EM}(r,0)$ is the initial state q_s and $\mathrm{EM}(r,1)$ is a copy of $M(E_r)$.
2. $\mathrm{EM}(r,i)$ is obtained from $\mathrm{EM}(r,i-1)$ by replacing each transition $q\xrightarrow{p}q'$, where p is a derived predicate, with a fresh copy of $M(E_p)$. Concretely, the transition $q\xrightarrow{p}q'$ will be eliminated and two new transitions: $q\xrightarrow{\mathrm{id}}q'_s$ and $q'_f\xrightarrow{\mathrm{id}}q'$ will be added, where q'_s and q'_f are the initial and final states of the fresh copy of $M(E_p)$. (See Fig. 4 for illustration.)
3. For each automaton copy, the corresponding interpretation graph will be traversed.

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 $\mathrm{EM}(r,0) \cup \ldots \cup \mathrm{EM}(r,i), i \geq 1$. We denote an interpretation graph of $\mathrm{EM}(r,i)$ by $G(r,\mathrm{d},i)$, where "d" represents a constant to which the variable appearing in the query is bound. In general, an $\mathrm{EM}(r,i)$ will have several interpretation graphs with each for a different variable binding.

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 edges. Here, q_s is the initial state of all $\mathrm{EM}(r,i), i \geq 1$. During the *i*th iteration of the main loop, $G(r,c',i-1)$ will be extended to $G(r,c'',i)$. Note that here c, c', and 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 edges labeled with derived predicates are traversed. Whenever a node (q,u) not visited before is entered, all transitions in $\mathrm{EM}(r,i)$ leaving q are determined. 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 $\mathrm{EM}(r,i)$ contains a transition leaving q and labeled with a derived predicate. If not, the algorithm terminates and the answers to the query will be

$$Ans = \{(u,v)| \text{ for some } i, (q^i_s,u),(q^i_f,v) \in G(r,u,i)\},$$

where q^i_s and q^i_f are the initial and the final state of $\mathrm{EM}(r,i)$, respectively. Otherwise, the algorithm starts a new iteration, the $(i+1)$th. Since this algorithm is fundamental for this paper, we give its pseudocode below. We briefly discuss the algorithm and explain how the data structures are used to implement the idea described above.

The original algorithm consists of two parts. The first part (named *evaluation-query*) controls the generation of automaton hierarchy. The second part (named *traverse*) traverses interpretation graphs. However, for ease of explanation, we rewrite it as a three-part algorithm consisting of: *evaluation-query*, *traverse*, and *fresh-automaton*. Here, *evaluation-query* and *traverse* correspond to the first and second parts of Grahne's algorithm, respectively. *fresh-automaton* is a subprocedure of *evaluation-query* used to generate a fresh copy of the automaton. In the algorithm, the following data structures are utilized:

- EM: automaton hierarchy,
- G: interpretation graph,
- C: a list used to store the continuation points which are produced whenever a derived predicate is encountered, and
- S: a list to store the starting points which are generated in terms of C whenever a fresh copy of the automaton is made.

The three subprocedures can be specified as follows:

- The procedure *evaluation-query* (p: derived predicate; a: term; **var** Y: set of terms) takes query $p(a,Y)$ as the input and returns the answers as the output, which are stored in Y.
- The procedure *traverse* (q: state; u: term: C: continuation points) takes a node (q,u) as the input and traverses some interpretation graph $G(r,d,i)$ from (q,u). The returned value is a set of continuation points C.
- The procedure *fresh-automaton* (C: continuation points; S: starting points) takes a set of continuation points C as the input and makes a fresh copy of the automaton. The returned value is a set of starting points S.

procedure *evaluation-query*(p: derived predicate; a: term;
var Y: set of terms);
(*evaluate the query $p(a,Y)$ when the equation for p is $p = e^*_p$)
1 **begin**
2 let EM be a copy of $M(e_p)$; (*EM is the current $\mathrm{EM}(p,i)$, the *i*th level of the automaton hierarchy. At the beginning $i=1$.*)
3 $G := \emptyset$; (*G is the current $G(p,a,i)$, the interpretation graph of $\mathrm{EM}(p,i)$.*)
4 $S := \{(q,a)\}$ (*starting point for the 1st traversal*)
5 **repeat** (*main loop*)
6 $C := \emptyset$; (*remove previous continuation points of the form (q^i_s,u)*)
7 **for** all (q,u) in S **do** (*breadth-first search*)
8 **if** (q,u) is not yet in G
9 **then**

10 **begin**
11 insert (q, u) into G;
12 traverse(q, u, C); (*traverse the interpretation graph of $\mathrm{EM}(p, i)$, during which some continuation points will be produced*)
13 **end**
14 $S := \emptyset$; (*remove all starting nodes of the previous level in the hierarchy. The new starting nodes of the current level will be stored in S.*)
15 fresh-automaton(C, S);
16 **until** $C := \emptyset$; (*no new continuation points were generated*)
17 $Y := \{u|(q_f, u) \in G\}$ (*store the instantiations for the variable appearing in the query*)
18 end

procedure *traverse*(q: state; u: term; C: continuation points);
(*depth-first search in an interpretation graph*)
(*computes new nodes of G reachable from (q, u)
1 **begin**
2 **for** all transitions $q \xrightarrow{s} q'$ in EM **do**
3 **if** s is a base predicate **then**
4 **for** all terms v such that $s(u, v)$ is a fact **do**
5 **begin**
6 **if** (q', v) is not yet in G
7 **then**
8 **begin**
9 insert (q', v) into G;
10 traverse(q', v, C);
(*go into the graph*)
11 **end**
12 **end**
13 **else if** $s = \mathrm{id}$ (*handle "identity" symbols *)
14 **then**
15 **begin**
16 **if** (q', u) is not yet in G
17 **then**
18 **begin**
19 insert (q', u) into G;
20 traverse(q', u, C)
21 **end**
22 **end**
23 **else** (*s is a derived predicate*)
24 insert (q', u) into C; (*store continuation points that will be used to form new starting points for the next level traversal in the hierarchy*)
25 **end**

precedure *fresh-automaton*(C: continuation points; S: strating points)
1 **for** all transitions $q \xrightarrow{r} q'$ in EM where r is a derived predicate and (q, u) is in C for some u **do**
2 **begin** (* in this loop, a new automaton in the hierarchy is constructed.*)
3 generate M', a fresh copy of $M(e_p)$;
4 denote by $q_{s'}$ the initial state and by $q_{a'}$ the final state of M';
5 add into EM all states and transitions of M' (without changing the initial and final states of EM);
6 add into EM the transitions $q \xrightarrow{\mathrm{id}} q'_s$ and $q_f \xrightarrow{\mathrm{id}} q'$;
7 $S := S \cup \{(q'_s, u)|(q, u) \in C\}$; (*store new starting nodes in S*)
8 remove the transition $q \xrightarrow{r} q'$ from EM;
9 **end**

Essentially, the algorithm is a combination of the depth-first search and the breadth-first search strategies. That is, all nodes of the form (q^i_s, u) are traversed in a breadth-first fashion (see lines 7-13 in *evaluation-query*), while all nodes in an interpretation graph, say $G(r, c, i)$, is traversed using the depth-first search strategy (see line 12 in *evaluation-query* and the procedure "*traverse*"). All nodes already visited are stored in a set, G. At iteration i, the set G contains the nodes of $G(r, c, i)$. In addition, the procedure "*traverse*" returns a set, C containing those continuation points which are used to form the actual staring points (set S) for the next level traversal. These starting points are stored in set S (see line 7 of *fresh-automaton*). Finally, notice that each automaton EM is generated in the procedure *fresh-automaton*. Since the nodes of the form (q^i_s, u) are traversed in a breadth-first manner, Grahne's algorithm cannot be directly developed to support subsumption checks.

The following example serves as an illustration.

Example 3.2. Consider the following program:

Rules:

1. $rp(x, y) \leftarrow flat(x, y)$,
2. $rp(x, y) \leftarrow up(x, z), rp(z, w), down(w, y)$.

Facts:

$up(a_1, a_2), up(a_1, a_3), up(a_2, a_1), up(a_2, a_3),$
$flat(a_3, b_3),$
$down(b_3, b_2), down(b_2, b_1).$

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

$$rp = flat \cup up \cdot rp \cdot down.$$

The corresponding automaton is shown in Fig. 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 $\mathrm{EM}(r, i)$'s are shown in Fig. 4.

Now, we trace the evaluation for a better understanding. At the very beginning, $\mathrm{EM}(rp, 0)$ is the initial state q_s and $G(rp, a_1, 0)$ contains only one node (q_s, a_1). During the first iteration, $\mathrm{EM}(rp, 1)$ will be established, which is as shown 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,

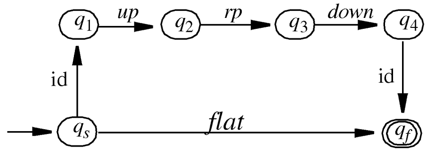


Fig. 2. Automaton $M(e_{rp})$ for expression $e_{rp} = flat \cup up \cdot rp \cdot down$. Here, q_s is the initial state and q_f is the final state. The symbol "id" is intrepreted as the identity relation.

$\mathrm{EM}(rp, 2)$ will be generated. It is just a copy of $\mathrm{EM}(rp, 1)$. Then, $G(rp, a_2, 2)$ will be traversed, producing 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 $\mathrm{EM}(rp, 3)$ and traverse $G(rp, a_1, 3)$, etc. 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.

Observe the dotted line from (q_s'', a_1) to (q_s, a_1), which shows a cycle. One can image that, if we traverse the graph from (q_s'', a_1) continually, we will traverse similar graphs repeatedly without termination. Another observation is that although (q_s'', a_3) and (q_s', a_3) are similar, they do not lead to nontermination. These two kinds of similarities should be distinguished carefully. The following section is mainly devoted to this issue.

4 Optimal Evaluation of Binary-Chain Programs

As discussed in the Introduction, the drawback of Grahne's algorithm is the absence of subsumption checking and no ways are provided to distinguish between cyclic and acyclic data. To remove such insufficiencies, we integrate Tarjan's algorithm for identifying strongly connected components (SCC) of a directed graph with Grahne's by labeling some nodes of interpretation graphs. In the following, we first present the concept of subsumption checks and give a modified version of Tarjan's algorithm which can be easily used for our purpose in Section 4.1. Then, we embed Tarjan's algorithm in Grahne's in Section 4.2. Next, in Section 4.3, we discuss two optimization possibilities for acyclic and cyclic data, respectively. In Section 4.4, we present the concept of linear cycle covers, which underlies the optimization techniques for cyclic data. Finally, Section 4.5 gives two examples to conclude the discussion.

4.1 Subsumption Checks and Tarjan's Algorithm

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 4.1. *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 the* t_i *are all* ground terms. θ *is called a* variable-pure substitution *if the* t_i *are all variables.*

Definition 4.2. *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 nodes can be defined as follows:

Definition 4.3. *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.*

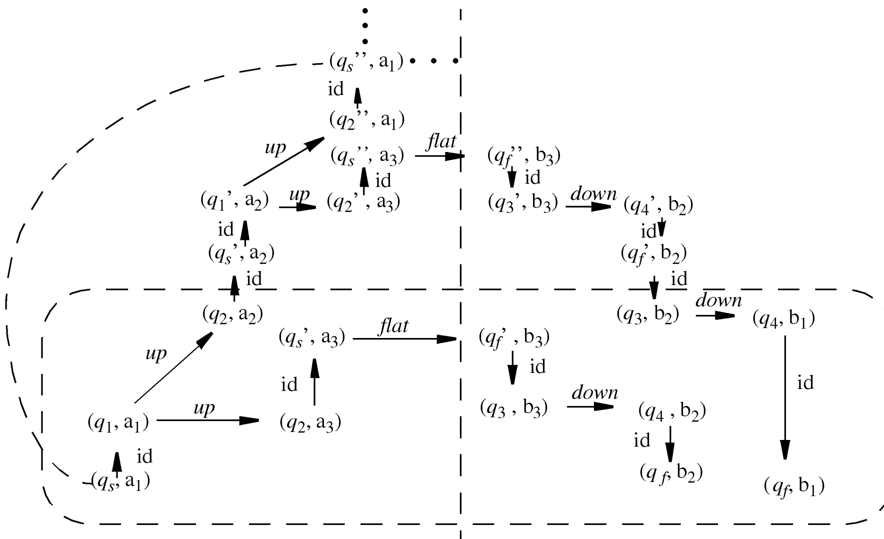


Fig. 3. Graph $(rp, a_1, 1) \cup G(rp, a_2, 2) \cup G(rp, a_3, 2) \cup G(rp, a_3, 3)$ with respect to Example 3.2.

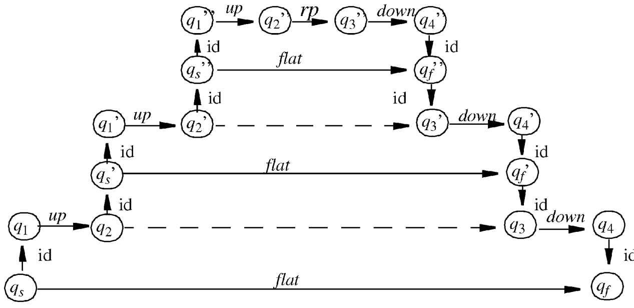


Fig. 4. Hierarchy of automata $\mathrm{EM}(rp,0) \cup \ldots \cup \mathrm{EM}(rp,3)$ with respect to Example 3.2.

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 4.4. *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 can not be evaluated. Therefore, a mechanism is needed to evaluate the remaining answers in some way else.

Definition 4.5. *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 [29] 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. 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 reused 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. An optimal strategy for evaluating linear binary-chain programs can be obtained by combining this algorithm with techniques described in the next section. 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 4.6. *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 iff 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 in Fig. 5.

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 Section 4.2). Therefore, we give the following algorithm over a character graph instead of an interpretation graph so as to illustrate the key ideas more 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 reverse topological order numbering. Here, it is used to test whether a node is an RCN or an RIC. $lowlink$ is dynamically changed and 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.

During a depth-first traversal, we distinguish among four types of edges:

1. *tree edges*. An edge $e: v \to u$ is a tree edge if u is reached from v when it is scanned and u has not been visited before (then at this moment $defnumber(u) = 0$ if we initially assign each $defnumber$ with 0.)

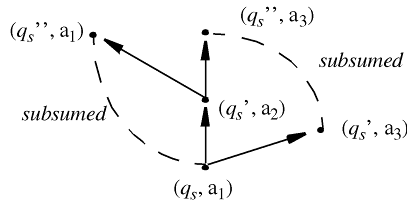


Fig. 5. Character graph.

2. *forward edges*: An edge $e : v \rightarrow u$ is a forward edge if when it is scanned for the first time and $defnumber(u) > defnumber(v) > 0$.
3. *back edge*: An edge $e : v \rightarrow u$ is a back edge if when it is scanned for the first time,

$$defnumber(v) > defnumber(u) > 0$$

and, at the same time, u is an "ancestor" of v.
4. *cross edges*. An edge $e : v \rightarrow u$ is a cross edge if when it is scanned for the first time,

$$defnumber(v) > defnumber(u) > 0$$

but u is not an "ancestor" of v.

Essentially, the algorithm presented below is a modified version of Tarjan's algorithm [29]. The difference between them consists in the use of $toplnumber$ in the modified algorithm, which facilitates the differentiation of back edges from cross edges and then the identification of strongly connected components. (In the original algorithm, a stack structure must be searched to do this.) We will see that in the algorithm below, for an edge $v \rightarrow u$ with $defnumber(v) > defnumber(u)$, if u is topologically numbered ($toplnumber(u) > 0$), then it is a cross edge; otherwise, it is a back edge. Whenever a back edge is visited, all the nodes belonging to an SCC can be simply taken from a stack S.

In addition, for our purposes, each RCN and each RIN are marked.

```
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 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 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 value
                                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(u) > 0 then (*if u is
                 topologically numbered, it can not 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
      end      strongly connected component have the
               same lowlink value as the root. see [29] *)
    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) ≤ 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 $toplnumber$:

1. $lowlink$ 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 an SCC.
2. $toplnumber$ 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 [29]). Fig. 6 shows a directed graph, its $defnumber$, $lowlink$, and $toplnumber$ values, and its strongly connected components.

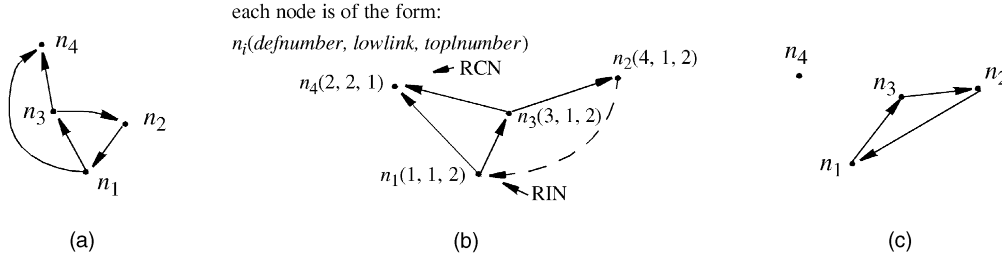


Fig. 6. Graph traversal. (a) Graph, (b) traversal, and (c) strongly connected components.

Now, we make a step-by-step trace of this graph traversal to see how the algorithm works (Table 1).

4.2 Embedding Tarjan's Algorithm into Grahne's

In this section, we discuss the embedding of Tarjan's algorithm in Grahne's. The optimization issues will be discussed in Sections 4.3 and 4.4.

From the discussion conducted in the previous section, we know that any node of the form: $(q_s^i, u)((q_f^i, v))$ should be labeled during a graph traversal using Grahne's algorithm. Whenever such a node is encountered, subsumption checking will be made and different treatments will be performed in terms of its characteristics. To this end, the control manner of Grahne's algorithm has to be changed a lot to facilitate the subsumption checking. That is, both the automaton hierarchy and the interpretation graph will be traversed in a depth-first search way. In the following, we give the detailed description of the algorithm for this task, which consists of four parts: *evaluation* (working for global control), *controlled-traversal* (for generating answers), *son-generation* (for making a fresh copy of the automaton and generating new starting points), and *traversal* (for traversing an interpretation graph from some starting point and generating new continuation points). As with Grahne's algorithm, the following data structures are used in the refined algorithm:

- EM: automaton hierarchy.
- G: interpretation graph.
- C: a list used to store the continuation points which are produced whenever a derived predicate is encountered.
- S: a list to store the starting points which are generated in terms of C whenever a fresh copy of the automaton is made. According to Tarjan's algorithm, S contains the son nodes (newly generated) of a node in the up-graph or in the down-graph during the traversal.

The four subprocedures can be specified as follows:

1. The procedure *evaluation* (r: derived predicate; c: constant; y: free variable) takes query $p(a, y)$ as the input and returns the answers as the output, which are stored in y.
2. The procedure *controlled-traversal* (q: state; u: term) takes a node of the form: (q, u) as the input and generates all the interpretation graphs. It is essentially a depth-first search with Tarjan's technique for identifying SCC used to distinguish RINs from RCNs.
3. The procedure *son-generation* (q: state; u: term; S: starting points) takes a node of the form: (q, u) as the input and makes a fresh copy of the automaton if this has not been generated. The returned value is a set of new starting points S, which can be regarded as the (direct) son nodes of (q, u).
4. The procedure *traverse* (q: state; u: term; C: continuation points) takes a node (q, u) as the input and traverses some interpretation graph $G(r, d, i)$ from (q, u). The returned value is a set of continuation points C.

procedure *evaluation*(r: derived predicate; c: constant; y: free variable)
(*evaluate the query $r(c, y)$*)
1 **begin**
2 $G := \emptyset$; (*G is the current interpretation graph.*)
3 $i := 0$; $j := 0$; $toplnumber((q_s, c)) := 0$; (*$i$ and j are used to calculate $defnumber$ values and $toplnumber$ values.*)
4 call $controlled\text{-}traversal((q_s, c))$; (*$(q_s, c)$ is the source node, corresponding to the issued query*)
5 $Ans := \{(u, v) \mid$ for some i, (q_s^i, u), $(q_s^i, v) \in G(r, u, i)$ and there is a path from (q_s^i, u) to (q_s^i, v) in $G\}$
6 $Y := \{u | (q_f, u)G\}$ (*store the instantiations for the variable appearing in the query*)
7 end

The following three procedures (*controlled-traversal*, *son-generation*, and *traversal*) correspond to the two main iterations of Grahne's algorithm with Tarjan's algorithm embedded. The subsumption checks are performed in *controlled-traversal* whenever a fresh copy of the automaton is generated since only at this moment a node of the form: (q_s^i, u) will be encountered. (For simplicity, the checks over the nodes of the form: (q_f^i, v) are not given. In fact, the subsumption check for the down-graph, which happens when a node of the form (q_f^j, d) is encountered in the procedure *traversal*, works in a similar way.) If an RCN is encountered, no matter whether in the up- or down-graph, procedure, *generation-RCN* will be invoked to generate answers for the subgraph rooted at the RCN. If an RIN is encountered during the traversal of the up-graph, procedure *generation-RIN* will be called to produce answers for cycles, in which *generation-RIN-1* will be called to generate answers directly. Procedure *generation-RIN-2* is performed only for the RINs appearing in the down-graph. These procedures will be discussed in the next two sections in great detail.

procedure *controlled-traversal* (v)
1 **begin**
2 $i := i + 1$; $defnumber(v) := i$; $lowlink(v) := i$; (*initiate

TABLE 1
Trace of a Graph Traversal

Operations:	State of stack S:
at the beginning: *toplnumber*(n_1) := 0.	
call *graph-search*(n_1):	
dfsnumber(n_1) := 1, *lowlink*(n_1) := 1.	
putting n_1 onto S.	$\{n_1\}$
generating the sons of n_1.	
toplnumber(n_4) := 0, *toplnumber*(n_3) := 0.	
(assume that n_4 is first accessed.)	
Since n_4 is not subsumed, recursively call *graph-search*(n_4):	
dfsnumber(n_4) := 2, *lowlink*(n_4) := 2.	
putting n_4 onto S.	$\{n_1, n_4\}$
generating the sons of n_4.	
(since n_4 has no child nodes, the deep search stops.)	
since *lowlink*(n_4) = *dfsnumber*(n_4),	
n_4 will be removed from S as an SCC and topologically numbered:	$\{n_1\}$
toplnumber(n_4) := 1;	
the control returns to the higher level. Then,	
lowlink(n_1) := 1 since min(*lowlink*(n_1), *lowlink*(n_4)) =1.	
Now n_3 will be visited.	
Since n_3 is not subsumed, recursively call *graph-search*(n_3):	
dfsnumber(n_3) := 3, *lowlink*(n_3) := 3.	
putting n_3 onto S.	$\{n_1, n_3\}$
generating the sons of n_3.	
toplnumber(n_4) will not be changed.	
toplnumber(n_2) := 0.	
(Assume that n_4 is first visited.)	
But this time, n_4 is subsumed.	
Since both *dfsnumber*(n_4) < *dfsnumber*(n_3) and *toplnumber*(n_4) > 0 hold,	
(*note that *dfsnumber*(n_4) = 2, *dfsnumber*(n_3) = 3.*)	
n_4 will be marked with RCN.	
Then, n_2 will be visited.	
Since n_2 is not subsumed, recursively call *graph-search*(n_2):	
dfsnumber(n_2) := 4, *lowlink*(n_2) := 4.	
putting n_2 onto S.	$\{n_1, n_3, n_2\}$
generating the sons of n_2.	
toplnumber(n_1) = 0 will not be changed.	
This time, n_1 is subsumed.	
Since *dfsnumber*(n_1) < *dfsnumber*(n_2) and *toplnumber*(n_1) = 0 hold,	
(*note that *dfsnumber*(n_1) = 1, *dfsnumber*(n_2) = 4.*)	
n_1 will be marked with RIN;	
The control returns to the higher level. Then,	
lowlink(n_2) := 1 since min(*lowlink*(n_2), *lowlink*(n_1)) =1;	
The control returns to the higher level. Then,	
lowlink(n_3) := 1 since min(*lowlink*(n_3), *lowlink*(n_2)) =1;	
The control returns to the higher level. Then,	
lowlink(n_1) := 1 since min(*lowlink*(n_1), *lowlink*(n_3)) =1;	
Since *lowlink*(n_1) = *dfsnumber*(n_1) =1, n_1, n_3, and n_2 will be removed from S	
as the current SCC; *toplnumber*(n_1) := 2; *toplnumber*(n_3) := 2; *toplnumber*(n_3) := 2;	
stack S becomes empty. The algorithm terminates.	

lowlink value; it may be changed during the search*)

3 suppose that v is of the form (q, u);

4 put (q, u) on the stack ST; (*ST is used to store cycles if any*)

5 *son-generation*(q, u, S);

6 **for** each w in S **do**
7 **begin**
8 perform subsumption check for w;
9 **if** w is not subsumed by any node **then**
10 call *controlled-traversal*(w); (*go deeper into the graph.*)
11 $lowlink(v) := min(lowlink(v), lowlink(w))$; (*the root of a subgraph will have the least $lowlink$ value.*)
12 **else** (*w is subsumed by some node*)
13 **begin**
14 mark w as a subsumed node; suppose that w is subsumed by a node u;
15 **if** $defnumber(u) < defnumber(v)$ **then**
16 **if** $toplnumber(u) > 0$ **then** (*if u is typologically numbered, it cannot be an ancestor node of v.*)
17 call *generation-RCN*(w, u);
18 **else** (*a cycle is encountered*)
19 $lowlink(v) := min(lowlink(v), dfsnumber(u))$; (*this operation will make all nodes of a strongly connected component have the same $lowlink$ value as the root.*)
20 **end**
21 **end**
22 **if** $(lowlink(v) = dfsnumber(v))$ **then** (*v is the root of a component*)
23 **begin**
24 **while** w on the stack ST satisfies $dfsnumber(w) \geq dfsnumber(v)$ **do**
25 {delete w from the stack ST and put w in current component (rooted at v);
26 $toplnumber(w) := j$;} (*topological order numbering; all the nodes in an SCC have the same $toplnumber$*)
27 $j := j + 1$; (*j is used to calculate $toplnumber$*)
28 **end**
29 **for** each strongly connected component SCC **do** call *generation-RIN*(SCC);
30 **end**

procedure *son-generation* (q: state; u: term; S: starting points)
1 **begin**
2 $C := \emptyset$; (*C is the set of previous continuation points.*)
3 **if** (q, u) is not yet in G **then**
4 **begin**
5 insert (q, u) into G;
6 $traversal(q, u, C)$; (*traverse the interpretation graph of EM(r, i), during which some continuation points will be produced.*)
7 **end**
8 **else** $S := \emptyset$; return;
9 **for** all transitions $q \xrightarrow{a} q'$ in EM where a is a recursive predicate and (q, u) is in C **do**
10 **begin** (*construct a new automaton for the next level traversal and the starting points of the corresponding interpretation graphs.*)
11 generate M' (if it does not exist), a fresh copy of $M(E_r)$;
12 denote the initial state by q'_s and the final state by q'_f;
13 add into EM all states and transitions of M'; (*Note that the initial state and the final state are not changed.*)
14 add into EM the transitions $q \xrightarrow{\text{id}} q'_s$ and $q \xrightarrow{\text{id}} q'$;
15 $S := S \cup \{(q'_s, u) | (q, u) C\}$; $toplnumber((q'_s, u)) := 0$ for each (q'_s, u); (*put new starting points into S.*)
16 remove subsumed continuation points from **C**;
17 remove the transition $q \xrightarrow{a} q'$ from EM; (*This transition is replaced by the newly constructed automaton.*)
18 **end**
19 **end**

procedure *traversal* (q: state; b: term; C: continuation points) (*depth-first search in an interpretation graph*)
1 begin
2 **for** all transitions $q \xrightarrow{a} q'$ in EM **do**
3 **if** a is a base predicate **then**
4 **for** all terms b' such that $a(\mathrm{b}, \mathrm{b}')$ is a fact **do**
5 **begin**
6 **if** (q', b') is not yet in G (*check whether (q', b') has been already accessed.*) **then**
7 **begin**
8 insert (q', b') into G;
9 $traversal(q', \mathrm{b}', C)$; (*go deeper into the graph*)
10 **end**
11 **end**
12 **else if** $a = \text{id}$
13 **then**
14 **begin**
15 **if** q is of the form: q^i_f **then**
16 {perform subsumption checking;
17 **if** RCN **then** {let u be the subsumming node of (q, b); call *generation-RCN* $((q, b), u)$;}
18 **if** RIN **then**
19 {let (q^j_f, b') be the subsumming node of (q, b);
20 let P be a path from (q_s, c) to (q^j_s, b'') in the up-graph, where $(\mathrm{b}'', \mathrm{b}')$ is an answer;
21 calls *generation-RIN-2*$(\mathrm{P}, (q^j_f, b'))$;}}
22 **else if** (q, b') is not yet in G **then**
23 **begin**
24 insert (q', b') into G;
25 $traversal(q', \mathrm{b}', C)$;
26 **end**

27 **end**
28 **else** insert (q, b) into C; (*a is a derived predicate; (q, b) is a continuation point which will be used to form new starting points for the next level traversal.*)
29 **end**

As with Grahne's algorithm, all nodes visited are stored in a set, G. Starting with the graph $G(r, \mathrm{c}, 0)$, which contains only one node (q_s, c) (called the *source* node) and with no edges, all interpretation graphs will be traversed in a depth-first fashion. This is done by calling the procedure *traversal* to extend $G(r, \mathrm{c}', i-1)$ to $G(r, \mathrm{c}'', i)$ (see line 6 of *son-generation*). In *traversal*, the interpretation graphs will be traversed and the newly produced continuation points are stored in the set C (see line 28 of *traversal*), which in turn will be used to form some new starting points for the next level traversal (see line 15 of *son-generation*). In *son-generation*, a copy of the automaton corresponding to a new level will be constructed (see lines 10-18 of *son-generation*). In addition, in terms of the set C, *son-generation* will create a set of new starting points, S, on which the subsumption checks are performed (see lines 6-21 of *controlled-traversal*).

Comparing the above algorithm with Grahne's, we see that the main difference consists in the subsumption checking performed in the former (see lines 6-29 of *controlled-traversal*). This is done in the way as discussed in Section 4.1. Additionally, *controlled-traversal* works in a depth-first manner (see line 10 for going deeper into the graph), which differs a lot from Grahne's, in which a breadth-first search strategy is used and each automaton is generated only once. But, we notice that by a simple examination (see line 9 of *son-generation*), we generate each automaton copy only one time, too. Corresponding to a fresh copy of the automaton, all the interpretation graphs will be searched one after the other using Grahne's method. But in the above algorithm, only one interpretation graph is searched. The remaining interpretation graphs will be searched by backtracking. Within an interpretation graph, the traversal is executed using the depth-first strategy by both methods. Therefore, in the above algorithm, the entire graph generated for evaluating a query is traversed uniformly in the depth-first search manner.

Example 4.1. For illustration, see the up-graph part of Fig. 7. By applying the above algorithm to Example 3.2, we will generate this graph which is the same as that shown in Fig. 3. But, each node of the form: (q_s^i, u) in the graph is labeled with a triple when it is generated as in Tarjan's algorithm. It is also the reason why we change Grahne's algorithm to a pure depth-first search.

Based on such labels, we can distinguish RCNs from RINs. In the following two sections, we discuss how to use RCNs and RINs to optimize the evaluation.

4.3 Search Space Reduction and Answer Generation

Based on the mechanism for subsumption checks, we develop three methods for generating answers directly in terms of different "path informations":

- answer generation for RCNs;
- answer generation when an up-graph (or a down-graph) contains cycles; and
- answer generation when both the up-graph and the down-graph contain cycles.

In the following discussion, six propositions will be established. First, Proposition 4.1 is to show that if an RCN is encountered, some answers can be generated directly using the answers already found. Second, Proposition 4.2 demonstrates that the answers along a cycle can be generated using the answers along another if they have a common point. Next, Proposition 4.3 shows how to generate answers along two cycles which are in the up- and down-graph, respectively. These three propositions reveal the possibility of evaluation optimization. Propositions 4.4 and 4.5 are used to show that a linear cover for an SSC can be found in linear time if it contains a feedback node. Finally, Proposition 4.6 tells us that the answers produced by traversing an SCC can be generated along each cycle in the linear cover for the SCC. In terms of Propositions 4.2 and 4.3, the answers for a cycle can be generated using the answers produced along another one. Thus, Proposition 4.6 hints an efficient method that we produce the answers for an SCC only along one cycle while generating the answers for all the other cycles.

4.3.1 Answer Generation for RCNs

Example 4.2. Consider the query of Example 3.2 again. Its interpretation graph is reproduced in Fig. 8.

From this graph, we can see that the portion enclosed by a broken line is similar to the portion enclosed by a solid line except that the states occurring in the nodes of one portion are "slightly" different from the other. It is due to the fact that (q_f'', a_3) is an RCN and is subsumed by (q_f', a_3). Obviously, we can cut off the portion enclosed by the solid line and directly use the portion enclosed by the broken line to generate the corresponding answers or *vice versa*. In this way, each edge of the interpretation graph can be accessed at most only once in the case of acyclic data since any repeated accesses to an edge are avoided. In this sense, a lot of search space is reduced and a linear time complexity is achieved for acyclic data. With regard to the correctness of the method, we have the following proposition.

Proposition 4.1. *Let n_1 be an RCN and be subsumed by n_2. Then, we can always find two subgraphs of the same height, which have, respectively, n_1 and n_2 as their roots and are similar to each other, i.e., each node in the subgraph rooted at n_1 is subsumed by a node in the subgraph rooted at n_2.*

Proof. The proposition follows from the binary-chaining property that a range value of a predicate depends only on one of the domain values of the predicate, which in turn depends only on some range value of the predicate directly left to its predicate. Assume that $(q_1, a_1) \rightarrow (q_2, a_2) \rightarrow \ldots \rightarrow (q_m, a_m)$ is a path in the graph. Then, each edge is either labeled with "id" or with a predicate symbol. If an edge $(q_i, a_i) \rightarrow (q_{i+1}, a_{i+1})$ is labeled with "id", then $a_i = a_{i+1}$ according to the graph construction. Suppose that the nonid predicate symbols for the path are $p_1 \cdot \ldots \cdot p_{m'} (m' \leq m)$. Then, traversing this path corresponds to an evaluation of the expression: $p_1 \ldots p_{m'}$. Now,

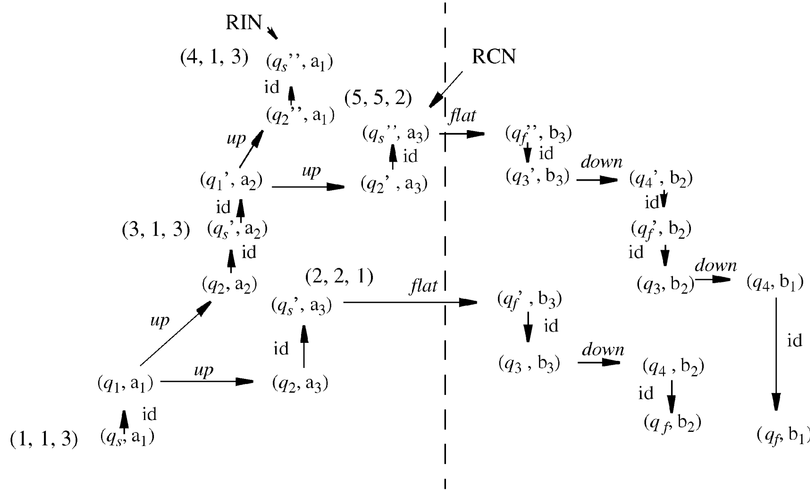


Fig. 7. Labeled graph of Example 3.2.

consider another node $(q_{1'}, a_{1'})$ which is subsumed by (q_1, a_1). Then, $a_1 = a_{1'}$ and q_1 and $q_{1'}$ are two different appearances of the same state on two different levels of the automaton hierarchy. Thus, there must be a path $(q_{1'}, a_{1'}) \rightarrow (q_{2'}, a_{2'}) \rightarrow \dots \rightarrow (q_{n'}, a_{n'})$ labeled with the nonid predicate symbols $r_1 \dots r_{n'} (n'n \leq\leq)$ such that

$$p_1 \cdot \ldots \cdot p_k = r_1 \cdot \ldots \cdot r_k,$$

where $k = min(m', n')$ and $a_2 = a_{2'}, \dots, a_k = a_{k'}$. Therefore, (q_2, a_2) subsumes $(q_{2'}, a_{2'}), \dots,$ and (q_k, a_k) subsumes $(q_{k'}, a_{k'})$. □

The following procedure is an algorithm implementation of the above method for generating answers. In this procedure, the answers for the path from w to the source node (P_2) are generated directly from the answers produced along the path from u to the source node (P_1). If P_1 is shorter than P_2, some answers for P_2 cannot be generated. They must be evaluated by traversing the corresponding subgraph in a usual way.

procedure *generation-RCN* (w: an RCN; u: subsuming counterpart)
begin
let $P_1 = \{(q_{s1}^0, c_0), \dots, (q_{s1}^n, c_n)$ be the path from the source node to u in the up-graph;
let $P_1 = \{(q_{s2}^0, g_0), \dots, (q_{s2}^m, g_m)(c_n = g_m)$ be the path from the source node to w in the up-graph;
let $k = \min\{n, m\}$;
for each path P in the down-graph, which is reachable from P_1 **do**
{let P is of the form: $\{(q_{f1}^0, d_0), \dots, (q_{f1}^n, d_n)\}$
construct new answers: $(g_0, d_0), \dots, (g_k, d_k)$;
if $n < m$ **then**
call *traversal*$(q_{f2}^{m-n+1}, d_{n-m+1})$;} (*$(q_{f2}^{m-n+1}$ represents the final state of $EM(r, m-n+1)$.*)
end

We can give a similar algorithm for the RCNs appearing in the down-graph. But for simplicity, we omit it.

4.3.2 *Answer Generation When an Up-Graph (or a Down-Graph) Contains Cycles*

Example 4.3. Continuing our running program. But, suppose that the database contains the following facts:

$$flat(c_4, c_5)$$
$$up(c_3, c_4), up(c_3, c_2), up(c_2, c_3), up(c_2, c_8), up(c_8, c_3)$$
$$down(c_5, c_1), down(c_1, c_6), down(c_6, c_7), down(c_7, c9).$$

Given the query $? - rp(c_3, y)$, the algorithm proposed by Grahne et al. will traverse the graph shown in Fig. 9. 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.

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. 9)

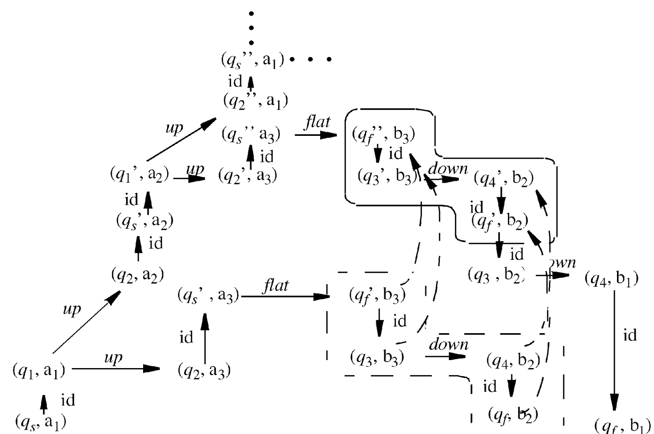


Fig. 8. Graph $(rp, a_1, 1) \cup G(rp, a_2, 2) \cup G(rp, a_3, 2) \cup G(rp, a_3, 3)$ with respect to Example 3.2.

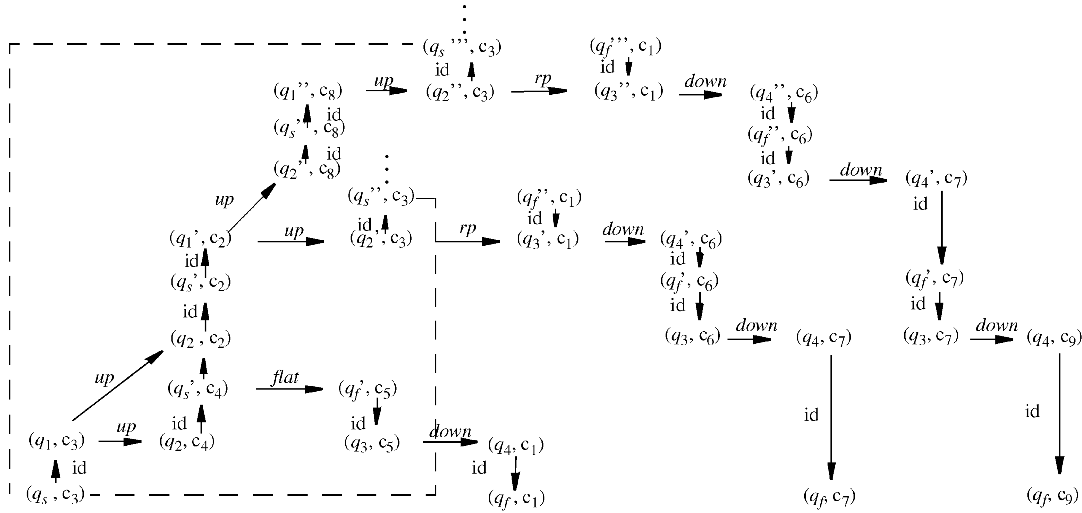


Fig. 9. Graph $G(rp, c_3, 1) \cup \ldots \cup G(rp, c_3, 4)$ with respect to Example 4.2.

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''' to q_f in the down-graph shown in Fig. 9) 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 shows 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 $rp(c_2, c_7)$ 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. 10 helps to illustrates this feature.

Below, we describe this method more formally.

Let C_1 be the first cycle $v_1 \leftarrow v_2 \leftarrow \cdot \leftarrow v_n \leftarrow v_1$ and $A_1 = \{a_1, \ldots, a_n, a_{n+1}, \ldots, a_{2n}, \ldots, a_{in}, \ldots, a_{in+j}\}$ the answer set produced along C_1, where i and j are integers and $0 \le j \le n$. (It should be noticed that each $a_l (1 \le l \le in + j)$ is a subset which is produced when the nodes of the form (q_f^λ, c) of the λth 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

$$A2 = \{a_{n+1}, \ldots, a_{2n}, \ldots, a_{in}, \ldots, a_{in+j}\},$$
$$A3 = \{a_{2n+1}, \ldots, a_{in}, \ldots, a_{in+j}\},$$
$$\ldots \ldots$$
$$A_i = \{a_{in+1}, \ldots, a_{in+j}\}.$$

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 shown in Fig. 11. (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 a_is). Without loss of generality, we assume that $n \le m$.

If $n = m$, no more new answers can be generated after the first 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.

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

(Hereafter, this process is called a *merging operation.*) Then, we construct $B_k (1 \le k \le l)$ as follows:

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

(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.

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

Proposition 4.2. *Let* $\{v_1, v_2, \ldots, v_n\}$ *and* $\{w_1, w_2, \ldots, w_m\}$ *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* (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* $l (0 \le k \le n - 1, 0 \le l \le m - 1)$.

Proof. Without loss of generality, assume that $v_1 = w_1$ is the same starting point of the two cycles and crossing an

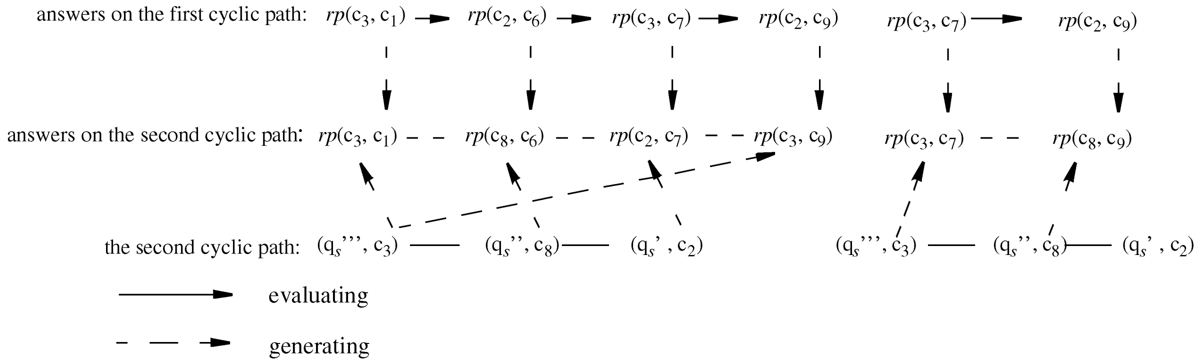


Fig. 10. Illustration of generating answers with respect to Example 4.2.

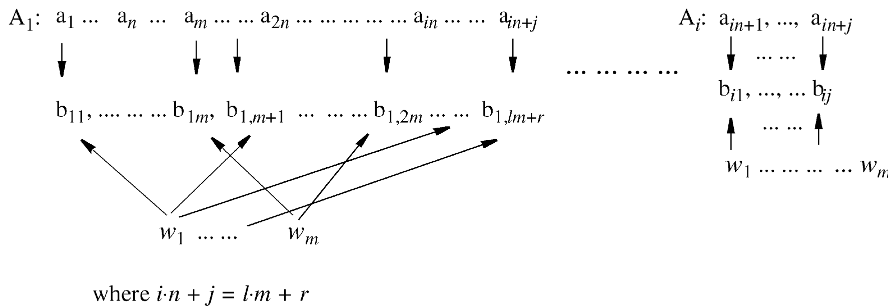


Fig. 11. Illustration of answer generation.

edge (v_1, o) some path in the down-graph will be traversed (see Fig. 12 for illustration).

If there is some l such that the distance from o to o' is $l \cdot n + i$, then (c_i, h) is an anwser. Therefore, if there is some k such that $k \cdot m + j = l \cdot n + i$, (d_j, h) will be an anwser.□

In terms of the above analysis, we have the following algorithm for generating answers.

procedure *generation-RIN-1*(C_1: the first cycle, C_2: the second cycle)

begin
  $u := 1$; $v := 2$;
  repeat
    let $C_u = \{(q^1_{s1}, c_1), \ldots, (q^n_{s1}, c_n)\}$;
    let $A = \{(c_1, a_1), \ldots, (c_n, a_n), (c_1, a_{n+1}), \ldots,$
      $(c_n, a_{2n}), \ldots, (c_n, a_{ln}), \ldots, (c_i, a_{ln+i})\}$
    be the answers evaluated along C_u;
    let $C_v = \{(q^1_{s2}, d_1), \ldots, (q^m_{s2}, c_m)\}$;
    for $s = 0$ to $l - 1$ **do**
      (*through this loop control, the separating operation is made.*)
      for $t = s + 1$ to $ln + i$ **do**
        $\{g := t$ mod m;
        construct new answers of the form: $(d_g, a_t)\}$
    let B be the result;
    do the merging operation over B; let
      $B' = \{b_1, \ldots, b_{km+j}\}$ be the result, where
      $k \cdot m + j = l \cdot n + i$;
    $A := B'/\{b_1, b_2, \ldots, b_m\}$;
    $w := u; u := v; v := w$;
  until no new answers can be generated
end

4.3.3 Answer Generation When Both the Up-Graph and the Down-Graph Contain Cycles

Example 4.4. Continuing our running program with the database represented using Fig. 13.

Given the query $? - rp(a_1, y)$, the algorithm proposed by Grahne et al. will traverse the graph shown in Fig. 14 when $G(rp, a_1, 4)$ is about to be constructed.

This graph contains two cycles. One of them is in its up-graph and the other is in its down-graph.

According to Grahne's method, an upper bound on the number of iterations will be established in this case, which is sufficiently large enough to allow all the answers to be found. In this way, however, too much redundant work may been done because all the similar portions of the graph will be repeatedly traversed.

Consider the following sample of cyclic relations for the rules given in Example 3.2 shown in Fig. 15.

The length of the longest cycle appearing in the relation up is m, and the length of the longest cycle in the relation $down$ is n. In this case, the upper bound established by Grahne's method should be $m \cdot n$. This is because if m and n do not have a common divisor, the entire answers can be produced only when $m \cdot n$ iterations of the main loop are performed. For example, the tuple (c_1, b_1) belongs to the relation denoted by the expression

$$up^{mn-1} \cdot flat \cdot down^{mn-1},$$

but does not belong to the relation denoted by any expression $up^k \cdot flat \cdot down^k$, where $k < m \cdot n - 1$ (see [15]).

To avoid this redundancy, such kind of subsumptions has to be checked. As stated above, if a subsumed node

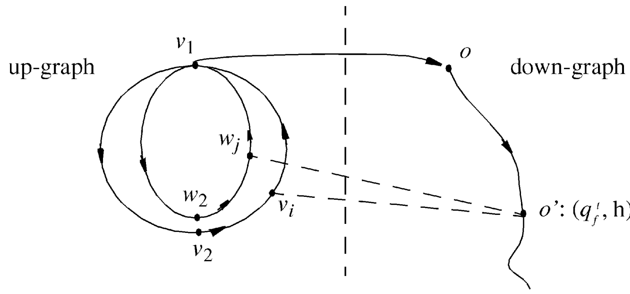


Fig. 12. Illustration for Proposition 4.2.

(RIN) is encountered in the up-graph, the traversal along the corresponding path should be suspended. Similarly, we need an iteration process to produce the remaining answers. During the iteration, if an RIN is encountered in the down-graph, we use the following method to generate the remaining answers. We imagine two circles with each corresponding to a cyclic path of Example 4.3 (see Fig. 16).

Then, we run, respectively, along the two circles in the same direction and, at each step, take one element from each circle to form a pair which corresponds to an answer. We do this continually until no new answers can be produced. For this example, the final set of pairs is:

$$rp(\mathrm{a}_3, \mathrm{b}_2), rp(\mathrm{a}_2, \mathrm{b}_1), rp(\mathrm{a}_1, \mathrm{b}_2),$$
$$rp(\mathrm{a}_3, \mathrm{b}_1), rp(\mathrm{a}_2, \mathrm{b}_2), rp(\mathrm{a}_1, \mathrm{b}_1).$$

The following proposition underlies this idea.

Proposition 4.3. *Let* $(\mathrm{c}_1, \mathrm{c}_2, \ldots, \mathrm{c}_m)$ *be a cycle appearing in the graph representing up,* $(\mathrm{b1}, \mathrm{b}_2, \ldots, \mathrm{b}_n)$ *be a cycle appearing in the graph representing down, and* $rp(\mathrm{c}_m, \mathrm{b}_n)$ *be an answer. Then, pair* $(\mathrm{c}_i, \mathrm{b}_j)$ *corresponds to an answer if* i, j *satisfies the equation:*

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

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

Proof. Consider the relation denoted by the expression

$$up^{k \cdot m - i} \cdot flat \cdot down^{l \cdot n - j}.$$

If $k \cdot m + j = l \cdot n + i$, the relation can evaluated by running the rules given in Example 3.2 repeatedly (in a bottom-up manner) and $rp(\mathrm{c}_i, \mathrm{b}_j)$ belongs to it. Therefore, $(\mathrm{c}_i, \mathrm{b}_j)$ corresponds to an answer. □

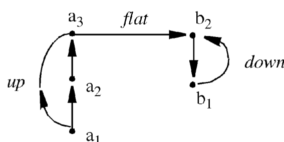
In the following procedure, the answers are generated by pairing the corresponding elements along two cycles. One of them (P_1) is in the up-graph and the other one (P_2) is in the down-graph. If P_1 is not a cycle, the elements on it are taken consecutively until the end of the path, while the elements on P_2 are taken in a circular tour.

procedure *generation-RIN-2*(P_1: a path in the up-graph; v: an RIN in the down-graph)
begin
  let SCC be the strongly connected component containing v;
  for each cycle P_2 in the cycle cover contained in SCC **do**
    {let $\mathrm{P}_1 = \{(q_s^1, \mathrm{c}_1), \ldots, (q_s^n, \mathrm{c}_n)\}$;
    let $\mathrm{P}_2 = \{(q_f^1, \mathrm{c}_1), \ldots, (q_s^m, \mathrm{b}_m)\}$;
    if P_1 is a cycle
    then {**repeat**
        construct all pairs of the form: $(\mathrm{c}_i, \mathrm{b}_j)$ such that $k \cdot m + j = l \cdot n + i$ for some integers k and $l(1 \leq k \leq n, 1 \leq l \leq m)$
      until no new answers can be generated}
    else for $i = 1$ to n **do**
      {construct $(\mathrm{c}_i, \mathrm{b}_j)$ such that $i = \lambda \cdot m + j$ for some $\lambda \geq 0$;}
end

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 requires only access to small data sets (i.e., the answers just found along some cycle), we may suppose that "generating answers" has the time complexity of O(1). (This claim is reasonable because according to the buffer replacement policies such as LRU (*least recently used*) and FIFO (*first-in-first-out*), used in a database management system, the intermediate answers found for some cycle should have not be moved to the external storage when the immediately following generation of answers are performed.) Thus, each cycle in the input relation is traversed at most only once in effect. Therefore, the idea described above requires only $\mathrm{O}(m + n)$ time for the cyclic data shown in Fig. 13.

4.4 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 [23], this task requires time $\mathrm{O}(n_{scc} + e_{scc})c_{scc}$, 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



In the graph, each edge corresponds to a tuple. For example, edge from a_1 to a_2 represents a tuple $up(\mathrm{a}_1, \mathrm{a}_2)$

Fig. 13. Cyclic data.

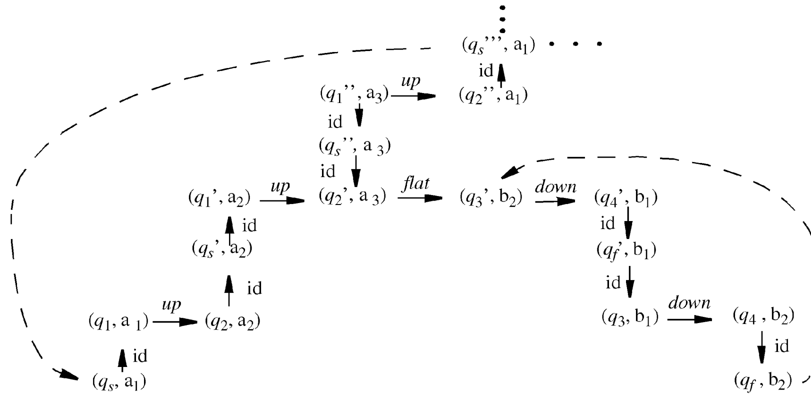


Fig. 14. Graph $(rp, \mathrm{a}_1, 1) \cup G(rp, \mathrm{a}_2, 2) \cup G(rp, \mathrm{a}_3, 3)$ with respect to Example 4.3.

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

cycles. In Fig. 17, we show a complete directed graph with four nodes (4-CDG), which contains 20 cycles.

In addition, the technique for generating answers cannot be applied efficiently to the graph shown in Fig. 18a 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 produced, becomes difficult. (Remember that, for the first cycle, the answers have to be produced by the graph traversal.)

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

Definition 4.7. *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. 18b is a feedback node.

Definition 4.8. *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 4.9. *A cycle cover with regard to 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 [13], 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 node exists, we take it as the start node and proceed to find a linear cycle cover of the SCC, to which the technique for generating answers can be applied. Garey's algorithm will be described in detail in the Appendix, which takes an SCC as the input and finds one or more feedback nodes or reports that such a node does not exist.

Below is an algorithm for finding a linear cycle cover with regard to an SCC containing a feedback node. In the algorithm, each node v is marked with a Boolean value $val(v)$ and at the beginning, all $val(v)$s are set to 0. During the depth-first traversal (starting from the feedback node), we set $val(v)$ to 1, when the corresponding node v is visited for the first time. Then, we have s simple property that when a node v with $val(v) = 1$ is met, at least one cycle through the feedback node and v must already be generated. In terms of this property, when a node v with $val(v) = 1$ is encountered, a new cycle can immediately be constructed by taking the current path and the path from v to the feedback node (which appears in some already generated cycle) together. This algorithm will be embedded into the algorithm for generating answers in such a way that cycles in a cycle cover is enumerated dynamically.

procedure $cycle\text{-}cover(fbn, \mathrm{SCC})$ (*fbn is a feedback node of SCC.*)

1 **begin**
2   **for** all *node* in SCC **do**
3     $val(node) := 0;$
4   $\mathrm{N} := fbn;$
5   $val(fbn) := 1;$
6   $search(fbn);$
7 **end**

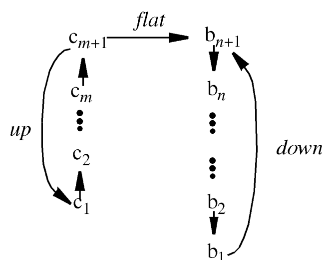


Fig. 15. Cyclic relations for the program given in Example 3.2.

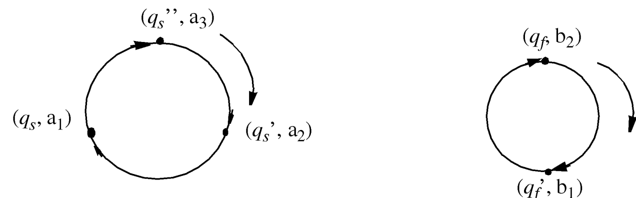


Fig. 16. Illustration of pairing answers for cyclic data.

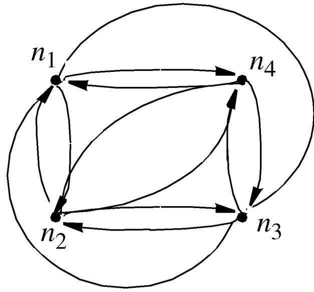


Fig. 17. Complete graph with four nodes.

procedure $search(v)$
8 **begin**
9 generate all sons of v;
10 **for** each son m **do**
11 **if** $val(m) = 0$ **then** $\{val(m) := 1; search(m)\}$
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 4.4. *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 appears 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 has not been 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 (v, u), v is the tail and u 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 $n_i, n_k, \ldots, n_l$ with $n_l = \mathrm{N}$, which is not visited by *cycle-cover*( ). But, this contradicts the behavior of the algorithm. Thus, C is a cycle cover. □

Proposition 4.5. *The cycle cover found by* cycle-cover*( ) is linear.*

Proof. We denote the number of the found cycles passing a set of nodes $n_1, n_2, \ldots, n_k$ by $numcycles(n_1, n_2, \ldots, n_k)$. 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:

$$numcycles = numcycles(v), \tag{1}$$

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

$$\begin{aligned} numcycles(v) &= \sum_{i=1}^{out(v)} numcycles(v, n_i) \\ &= \sum_{i=1}^{out(v)} \sum_{j=1}^{out(n_i)} numcycles(v, n_i, n_{ij}), \end{aligned} \tag{2}$$

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 $inedges(n_{ij})$ to denote the edges incident into n_{ij}, we have

$$\cup_i \, outedges(n_i) = \cup_{i,j} inedges(n_{ij}).$$

This equation can be proved as follows. First, we have

$$\cup_i \, outedges(n_i) \subseteq \cup_{i,j} inedges(n_{ij})$$

(see Fig. 19 for illustration).

Then, we prove $\cup_i outedges(n_i) \supseteq \cup_{i,j} inedges(n_{ij})$. Assume, to the contrary, that there exists a node u such that for some $n_k (n, n_k) \notin \cup_i \, outedges(n_i)$ but $\in \cup_{i,j} inedges(n_{ij})$. In terms of the property of SCCs, there must be a path connecting n_k and n as shown in Fig. 20. Then, the cycle composed of this path and the edge (u, 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.

In terms of the above analysis, (2) can be rewritten as follows:

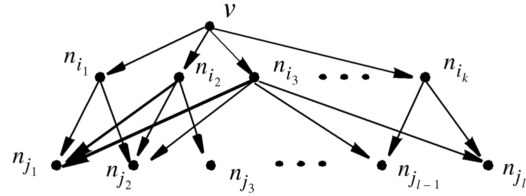


Fig. 19. Illustration for the relationship between indegrees and outdegrees.

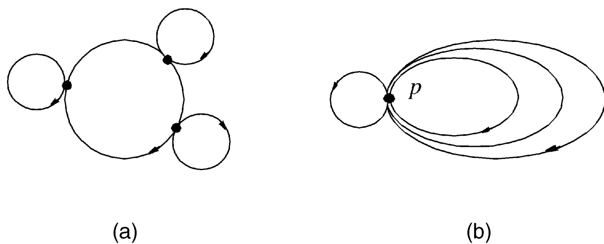


Fig. 18. Cycles without common nodes.

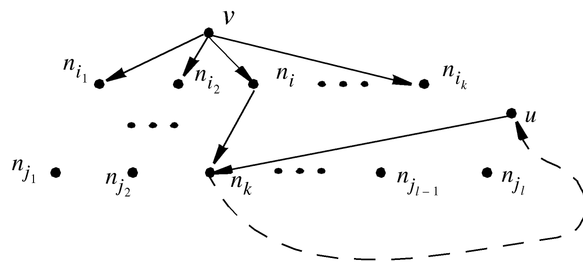


Fig. 20. A cycle which does not contain v.

$$\sum_{i=1}^{out(v)} \sum_{j=1}^{out(n_i)} numcycles(v, n_i, n_{ij}) = \sum_j \sum_{m=1}^{in(n_j)} numcycles(v, n_{i_m}, n_j). \quad (3)$$

Note that in the right-hand side of (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 u (from u 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. 18 for illustration. Along each of them, only one path will be considered), we have the following equation:

$$\begin{aligned}\sum_j \sum_{m=1}^{in(n_j)} numcycles(v, n_{i_m}, n_j) &= \sum_j (numcycles(v, n_i^j, n_j) \\ &\quad + in(n_j) - 1) \\ &= \sum_j numcycles(b, n_i^j, n_j) \\ &\quad + \sum_j in(n_j) - j \\ &= \sum_k numcycles(v, n_i^j, n_j^k, n_k) \\ &\quad + \sum_j in(n_j) \\ &\quad + \sum_k in(n_k) - (j + k) \\ &= \ldots\ \ldots \\ &= in(v) + \sum_{n_l \neq v} in(n_l) \\ &\quad - (n_{scc} - out(v)) \\ &= \mathrm{O}(e_{scc} - n_{scc} + 1).\end{aligned} \quad (4)$$

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 $\mathrm{O}(n_{scc})$. Then, the space complexity of *cycle-cover*() will be $\mathrm{O}(n_{scc} \cdot e_{scc})$. 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 (see procedure "*generation-RIN*" shown in the next section).

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 4.10. *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 4.6. *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 $a \in A_{cycle\text{-}cover}$, we have trivially $a \in A_{scc}$. Then, $A_{cycle\text{-}cover} \subseteq A_{scc}$. In the following, we prove that $A_{scc} \subseteq A_{cycle\text{-}cover}$. For any $a \in A_{scc}$, there must be a path $v_0 \to v_1 \to v_2 \to \ldots \to v_i$ 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 $A_{v_0} = predecessor(A_{v_1})$, $A_{v_1} = predecessor(A_{v_2}), \ldots, A_{v_{i-1}} = predecessor(A_{v_i})$ and $a \in A_{v_j}$, 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 $v_0 \to v_1$, $v_1 \to v_2, \ldots$, 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 *i*th traversal along C, A_{v_i} will be produced. Therefore, $a \in A_{cycle\text{-}cover}$. Thus, $A_{scc} \subseteq A_{cycle\text{-}cover}$, which completes the proof. □

The following two procedures are an algorithm implementation for controlling the direct generation of answers. For a strong component, if all cycles in it have a common node, only partial answers for one cycle is evaluated by traversing the corresponding subgraph. The remaining answers for all cycles are generated directly in terms of the answers already found and the relevant "path information." If the strong component contains no feedback node, it has to be traversed repeatedly until no new answers can be produced. Note that in the case that a feedback node exists, we enumerate a cycle dynamically as *cycle-cover*( ) does and generate answers as discussed in the previous sections.

In procedure "*generation-RIN*," the feedback node will be first found. If it does not exist, the control is switched over to a brute-force evaluation. If it does exist, procedure "*answer-generation*" is called, in which the cycles are enumerated dynamically and both "*generation-RIN-1*" and "*generation-RIN-2*" may be called for generating answers directly.

```
procedure generation-RIN(SCC: a strongly connected
                              component)
   begin
     using Garey's algorithm to find a feedback node if
       any;
     if a feedback node exists then
         {let fsb be the feedback node;
         for all n in SCC do value(n) := 0;
         N := fsb;
         value(fsb) := 1;
         answer-generation(fsb);
     else traverse the SCC in a normal way;
   end

procedure answer-generation(n: a feedback node)
   begin
     for each son m of n do
```

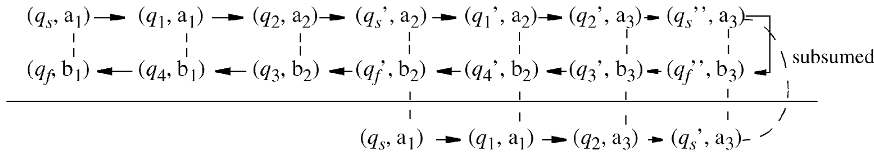


Fig. 21. Answer generation.

if $value(m) = 0$ **then** {$value(m) := 1$;
$answer\text{-}generation(m)$}
else
{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 P;
if P is the first one **then**
{traverse the corresponding down-graph for P;
if an RIN w in the down-graph is encountered
then call *generation-RIN-2*(P, w);}
else
{let P_1 be the first cycle;
call *generation-RIN-1*(P_1, P)}
end

4.5 Sample Trace

In this section, we apply the algorithms discussed above to Examples 3.2 and 4.3 to generate some of answers directly from the path information and the answers already found.

Example 4.5. Consider the rules and facts as in Example 3.2. Let $rp(a_1, y)$ be the query. Algorithm *evaluation* ( ) will produce the answers: $rp(a_3, b_3)$, $rp(a_2, b_2)$, $rp(a_1, b_1)$ by traversing the path shown in the upper part of Fig. 21.

See Fig. 8 for comparison. By this process, two layers of the automaton hierarchy will be generated by executing *controlled-traversal*( ). The path is traversed by performing *traversal*( ). The remaining answers are generated by calling on *generation-RCN*( ), which pairs the corresponding elements from the relevant path and the answers evaluated along it as shown in the lower part of Fig. 19. The answers generated by *generation-RCN*( ) are: $rp(a_3, b_3)$, $rp(a_1, b_2)$.

Example 4.6. Given the rules and facts as in Example 4.4. Let $rp(a_1, y)$ be the query. Algorithm *evaluation* ( ) will produce the answers: $rp(a_3, b_2)$, $rp(a_2, b_1)$, $rp(a_1, b_2)$ by calling on *generation-RIN*( ), in which part of the down-graph will be searched. During this process, another cycle is identified. Then, *generation-RIN-2*( ) will be executed to generate the remaining answers: $rp(a_3, b_2)$, $rp(a_2, b_1)$, $rp(a_1, b_2)$, $rp(a_3, b_1)$, $rp(a_2, b_2)$, $rp(a_1, b_1)$ by running, respectively, along the two cycles as shown in Fig. 22.

5 Complexity Analysis and Comparison

In order to compare the time complexity of our algorithm with Grahne's, we use the following linear recursive program as a benchmark to do an exact time analysis.

$$s(x, y) \leftarrow r(x, y),$$
$$s(x, y) \leftarrow p(x, z), s(z, w), q(w, y).$$

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 graph traversal process of Grahne's algorithm 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" (i.e., the up-graph). The latter corresponds to the traversal of the graphs for "r" and "q" (which corresponds to the down-graph). 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 Grahne's algorithm is:

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

where A denotes the set of answer tuples. $O(e_p)$ is the cost for the traversal of the graph for "p." $O(e_r)$ is the cost for the traversal of the graph for "r." The cost for the traversal of the graph for "q" is

$$O\left(\sum_{(i,j)\,\in\, A} indegree(i) \times outdegree(j)\right) = O(e_p \cdot e_q).$$

The diagram shown in Fig. 23a helps to clarify the result.

From this graph, we see that crossing the edge(i, j), each edge incident to j will be visited indegree (i) times. Similarly, the graph traversal of our algorithm can be viewed as two processes. However, due to the answer generation for RCNs in both the up-graph and down-graph, the cost is reduced to

$$O(e_p) + O(e_r) + O(e_q).$$

See Fig. 23b for illustration. The edges in the down-graph will not be visited more than once because the edges incident to j will be visited only one time through some (i, j) and each (i, j) can be accessed only once due to the subsumption check performed in the up-graph. We also notice that through (i', j) (denoted by a dashed arrow) they will not be visited, either, due to the subsumption check

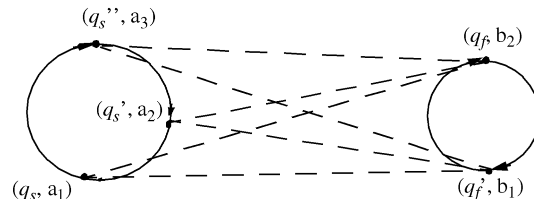


Fig. 22. Generating answers for cyclic data.

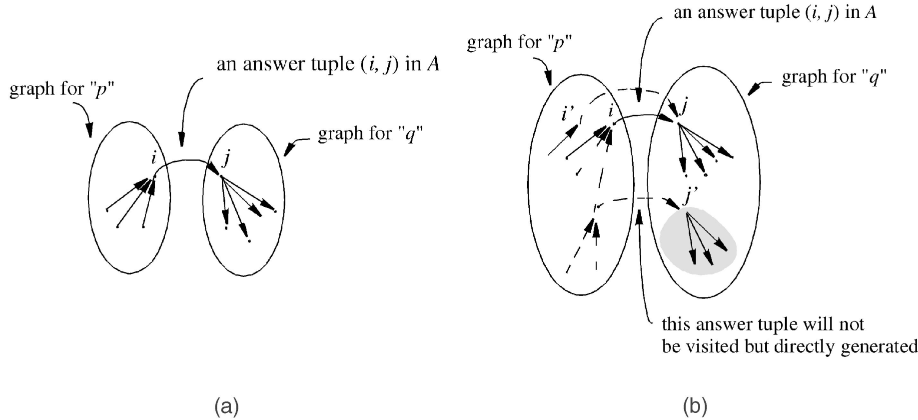


Fig. 23. Illustration for time complexity analysis.

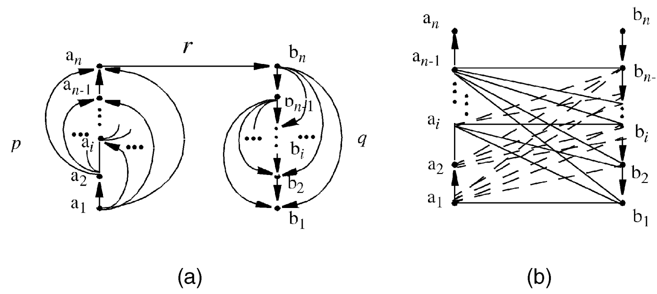


Fig. 24. Graph representation of input relations.

made in the down-graph. In fact, the number of the actually accessed edges of the down-graph should be smaller than $O(e_q)$ since the edges like those incident to j' (marked gray) may not be visited. 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*, *etc.*), 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).

To make the above analysis clear, see the graph shown in Fig. 24a, representing a set of input relations.

Applying our algorithm to the above program against this graph (to evaluate $?\text{-}s(\mathrm{a}_1, x)$), the answer pairs evaluated (using algebraic operations) are of the form: $(\mathrm{a}_i, \mathrm{b}_{i-j})(i = 1, \ldots, n-1; j = 0, \ldots, i-1)$ (see the solid edges connecting a_u and b_v shown in Fig. 23b for illustration) if the first derivation is performed along the path

$$\mathrm{a}_1 \rightarrow \mathrm{a}_2 \ldots \mathrm{a}_n \rightarrow \mathrm{b}_n \rightarrow \ldots \rightarrow \mathrm{b}_1.$$

All the other answer pairs, i.e., of the form:

$$(\mathrm{a}_k, \mathrm{b}_{k+l})(k = 1, \ldots, n-2; l = 1, \ldots, n-k-1)$$

will be directly generated (see the dashed edges shown in Fig. 14b.) By this process, each edge in both the graphs representing p and q is visited only once.

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 nodes in a cycle, which have initial (or final) states) 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 Grahne's algorithm 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 procedure *generation-RIN*( ), $(d^{ml} - 1)/(d - 1)$ answers are evaluated using the assumption above. The remaining answers for each cycle are all generated by executing *answer-generation*( ). Let δ be the cost of generating an answer in the generation process, then the running time for executing *answer-generation*( ) 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 $\delta / C \leq 1/(\lambda \cdot d^{ml}), (\lambda - 1) \cdot (d^{ml} - 1) \cdot \delta$ is less than some constant and, therefore, the time complexity of *generation-RIN*( ) is $O(d^{ml} C)$. Therefore, the refined algorithm may reduce the

TABLE 2
Costs of Methods

	cost			
input graph	Grahne	Counting	Magic sets	refined method (proposed in this paper)
Tree	O(n)	O(n)	O(n)	O(n)
Acyclic	O(en)	O(en)	O(e^2)	O(e)
Cyclic	nontermination	nontermination	O(e^2)	O(e)

worst-case time complexity of Grahne's algorithm by a factor λ, the number of the cycles, if we do not take the cost of generating an answer into account.

Various strategies for processing recursive queries have been proposed. However, all these algorithms require at least $O(e^2)$ or $O(en)$ time [1], [2], [3], [4], [5], [18], [32].

When the input relations contain no cycles, the cost of Counting is $O(en)$, better than that of Magic Sets which is $O(e^2)$ [24]. For cyclic cases, [18] proposed a method which takes $O(ne)$ time. However, this method requires $O(e^2)$ time for preprocessing. Sacca and Zaniolo [26] 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 [1] requires $O(n^3)$ time. But, in some cases, the complexity can be reduced to $O(n^2)$. The algorithm proposed in [32] requires $O(ne)$ time. In terms of the analysis of [35] and the analysis conducted above, the time complexities of several important methods can be summarized in Table 2.

To analyze the space complexity of our algorithm, we first make the following two observations:

1. Since corresponding to each layer of the automaton, there are normally several interpretation graphs. Thus, the graph G dominates the space complexity.
2. Due to the subsumption check, each edge of the input graph can be visited at most once for generating the interpretation graph G. Fig. 25 serves as an illustration.

Assume that Fig. 25a is the graph representing the relation "p." Our algorithm will generate a graph as shown in Fig. 25b. The nodes "b" and "d" both are generated two times. However, due to the subsumption check, when "b" (or "d") is encountered at the second time, the traversal will not be continued from the corresponding nodes. Thus, there are no edges visited more than once. The same analysis applies to the graph representing the relation "q." Therefore, the space complexity of our algorithm is bounded by $O(e)$, where e represents the number of the edges of the input graphs.

6 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 cycle only once and generating the remaining answers directly from the answers already found. In the case of cyclic data, several graph optimization 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 the answers for cyclic paths can also be generated directly instead of traversing the corresponding subgraphs. Since traversing a path requires access to the external storage or search of large relations, but the "generating" operations requires only access to small data sets and happens always in main memory, we may suppose that the time complexity of generating answers is $O(1)$ and, therefore, a linear time is achieved.

APPENDIX

In this appendix, we describe Garey's algorithm and make a sample trace to clarify its main idea.

In Garey's algorithm, the nodes of an SCC are numbered from 1 to n as they are reached during a depth-first search. Based on this numbering, the following two values can be defined:

1. $y = max$ $\{v \mid$ there is a back edge $(u, v)\}$ and
2. $z = max$ $\{w \mid$ for every back edge (u, v), w is an ancestor of u or u itself$\}$.

Then, a feedback node fbn must satisfy $y \leq fbn \leq z$. If $y > z$ or if the subgraph of the SCC obtained by deleting all nodes v satisfying $y \leq v \leq z$ is not acyclic, then no feedback nodes exist.

To determine which of these candidates actually are feedback nodes, each node will be associated with two labels defined as follows:

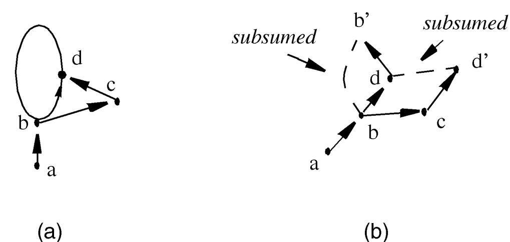


Fig. 25. Illustration for space complexity.

1. $maxi(v) = max\{\{w \mid w \leq z$ and (v, w) is not a back edge$\} \cup \{maxi(w) \mid w > z$ and (v, w) is not a back edge$\}\}$ and
2. $loop(v) = true$ if and only if either v is a proper descendant of z or there is an edge (v, w) which is not a back edge such that $w > z$ and at the same time $loop(w) = true$. Otherwise, $loop(v) = false$.

We notice that the two labels are defined recursively and can be calculated only in a reverse topological order. That is, before the labels for a node v are computed, the labels for v's children must be available. In addition, z has to be known ahead of time. By convention, $\max\{\emptyset\}$ is set to "0."

These two labels can be used to identify feedback nodes due to the following theorem given by Garey and Tarjan [13]. In the theorem, a *feedback set S* is a set such that every cycle of an SCC contains at least one node in S. The reader should not confuse this concept with the set of feedback nodes.

Theorem. *A node* v ∈ SCC *is a feedback node if and only if*

1. $y \leq v \leq z$;
2. *the set* $\{\{x|y \leq x \leq z\}$ *is a feedback set;*
3. $maxi(u) \leq v$ *for* $1 \leq u < v$; *and*
4. $loop(u) = false$ *for* $y \leq u < v$.

In terms of this theorem, the Garey's algorithm can be sketched as follows:

procedure *find-feedback-nodes*(SCC: a strongly connected component)

begin

number the nodes in SCC from 1 to n in depth-first order;

test whether $y \leq z$ and whether SCC becomes acyclic when all nodes v satisfying $y \leq v \leq z$ are deleted. If either test fails, report "SCC contains no feedback nodes";

calculate $maxi(v)$ and $loop(v)$ for each v in a reverse topological order;

initialize $maxitest := max\{maxi(u)|1u < y\}$; (If $y = 1$, $maxitest := 0$;)

initialize set of feedback nodes $S_{fbn} := \emptyset$;

$v := y$; Mark := 0;

repeat

if $maxitest \leq v$, **then** add v to S_{fbn};

$maxitest := max\{maxitest, maxi(v)\}$

if $loop(v) = false$ and $v < z$,

then $v := v + 1$;

else Mark := 1;

until Mark = 1

end

It is not difficult to verify that the algorithm requires only linear time [13]. In the following, we make a sample trace to show how the labels work.

Example. Consider the graph shown in Fig. 26a.

A depth-first numbering with all back edges removed is shown in Fig. 26b.

Then, we have

$$y = 1, z = 3.$$

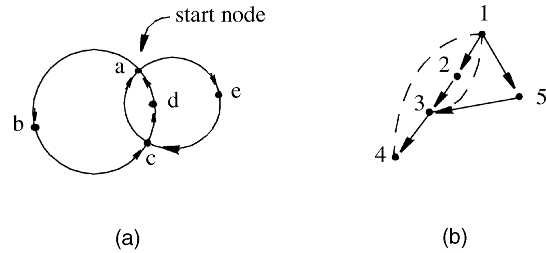


Fig. 26. An SCC.

- $maxi(4) = max\{\emptyset\} = 0$; $loop(4) = true$.
- $maxi(3) = max\{maxi(4)\} = 0$; $loop(3) = true$.
- $maxi(5) = max\{3\} = 3$; $loop(5) = false$.
- $maxi(2) = max\{3\} = 3; loop(2) = false$.
- $maxi(1) = max\{2, maxi(5)\} = 3$; $loop(1) = false$.

In the next step, the repeat-until loop will be executed:

- initial values: $S_{fbn} = \emptyset; maxitest = 0; v = 1$; Mark = 0.
- 1st iteration: $S_{fbn} = \{1\}$; $maxitest = 3$; $v = 2$; Mark = 0.
- 2nd iteration: $S_{fbn} = \{1\}$; $maxitest = 3$; $v = 3$; Mark = 0.
- 3th iteration: $S_{fbn} = \{1, 3\}$; $maxitest = 3$; $v = 4$; Mark = 1.

The loop terminates since Mark =1.

The feedback nodes are those numbered 1 and 3 in Fig. 25b, which correspond to the nodes marked with "a" and "c" in Fig. 25a, respectively.

References

[1] H. Aly and Z.M. Ozsoyoglu, "Synchronized Counting Method," *Proc. Fifth Int'l Conf. Data Eng.*, pp. 316-373, 1989.

[2] F. Bancilhon, "Naive Evaluation of Recursively Defined Relations," *On Knowledge Base Management Systems—Integrating Database and AI Systems,* Springer-Verlag, pp. 165-178, 1985.

[3] F. Bancilhon, D. Maier, Y. Sagiv, and J.D. Ullman, "Magic Sets and Other Strange Ways to Implement Logic Programs," *Proc. Fifth ACM Symp. Principles of Database Systems,* pp. 1-15, 1986.

[4] F. Bancilhon and R. Ramakrishnan, "An Amateur's Introduction to Recursive Query Processing Strategies," *Proc. 1986 ACM-SIGMOD Conf. Management of Data,* pp. 16-52, 1986.

[5] C. Beeri and R. Ramakrishnan, "On the Power of Magic," *J. Logic Programming,* vol. 10, no. 4, pp. 255-299, 1991.

[6] C. Chang, "On the Evaluation of Queries Containing Derived Relations in Relational Database," *Advances in Data Base Theory,* vol. 1, pp. 235-260, 1981.

[7] Y. Chen and T. Härder, "Improving RQA/FQI Recursive Query Algorithm," *Proc. ISMM—First Int'l Conf. Information and Knowledge Management,* pp. 413-422, 1992.

[8] Y. Chen, "A Bottom-Up Query Evaluation Method for Stratified Databases," *Proc. Ninth Int'l Conf. Data Eng.,* pp. 568-575, 1993.

[9] Y. Chen and T. Härder, "On the Optimal Top-Down Evaluation of Recursive Queries," *Proc. Fifth Int'l Conf. Database and Expert Systems Applications,* pp. 47-56, 1994.

[10] Y. Chen, "Processing of Recursive Rules in Knowledge-Based Systems—Algorithms for Handling Recursive Rules and Negative Information and Performance Measurements," PhD thesis, Computer Science Dept., Univ. of Kaiserslautern, Germany, Feb. 1995.

[11] Y. Chen, "On the Bottom-Up Evaluation of Recursive Queries," *Int'l J. Intelligence Systems,* vol. 11, no. 10, pp. 807-832, 1996.

[12] Y. Chen, "Magic Sets and Stratified Databases," *Int'l J. Intelligent Systems,* vol. 12, no. 3, pp. 203-231, 1997.

[13] M.R. Garey and R.E. Tarjan, "A Linear-Time Algorithm for Finding All Feedback Vertices," *Information Processing Letters,* vol. 7, no. 6, pp. 274-276, Oct. 1978.

[14] G. Grahne, S. Sippo, and E. Soisalon-Soininen, "Efficient Evaluation for a Subset of Recursive Queries," *Proc. Sixth ACM SIGACT-SIGMOD-SIGACT Symp. Principles of Database Systems,* pp. 284-293, 1987.

[15] G. Grahne, S. Sippo, and E. Soisalon-Soininen, "Efficient Evaluation for a Subset of Recursive Queries," *J. Logic Programming,* vol. 10, no. 4, pp. 301-332, 1991.

[16] J. Han, K. Zeng, and T. Lu, "Normalization of Linear Recursion in Deductive Databases," *Proc. Ninth Int'l Conf. Data Eng.,* pp. 559-567, 1993.

[17] J. Han and S. Chen, "Graphic Representation of Linear Recursive Rules," *Int'l J. Intelligent Systems,* vol. 7, no. 3, pp. 317-337, 1992.

[18] J. Han and L.J. Henschen, "The Level-Cycle Merging Method," *Proc. First Int'l Conf. Deductive and Object-Oriented Databases,* pp. 113-129, 1989.

[19] J. Han, "Chain-Split Evaluation in Deductive Databases," *IEEE Trans. Knowledge and Data Eng.,* vol. 7, no. 4, pp. 261-273, Aug. 1995.

[20] L.J. Henschen and S. Naqvi, "On Compiling Queries in Recursive First-Order Database," *J. ACM,* vol. 31, no. 1, pp. 47-85, 1984.

[21] T. Kanamori and T. Kawamura, "Abstract Interpretation Based on OLDT Resolution," *J. Logic Programming,* vol. 15, no. 1, pp. 1-30, 1993.

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

[23] D.B. Johnson, "Finding All Elementary Circuits of a Directed Graph," *SIAM J. Computing,* vol. 4, no. 1, Mar. 1975.

[24] A. Marchetti-Spaccamela, A. Pelaggi, and D. Sacca, "Worst Case Complexity Analysis of Methods for Logic Query Implementation," *Proc. Sixth ACM SIGACT-SIGMOD-SIGACT Symp. Principles of Database Systems,* pp. 294-301, 1987.

[25] W. Nejdl, "Recursive Strategies for Answering Recursive Queries—The RQA/FQI Strategy," *Proc. 13th VLDB Conf.,* pp. 43-50, 1987.

[26] D. Sacca and C. Zaniolo, "On the Implementation of a Simple Class of Logic Queries for Databases," *Proc. Fifth ACM SIGMOD-SIGACT Symp. Principles of Database Systems,* pp. 16-23, 1986.

[27] S. Shapiro and D. Mckay, "Inference with Recursive Rules," *Proc. Fifth Ann. Nat'l Conf. Artificial Intelligence,* pp. 151-153, 1980.

[28] G. Schmidt and T. Ströhein, *Relations and Graphs.* Berlin: Springer-Verlag, 1991.

[29] R. Tarjan, "Depth-First Search and Linear Graph Algorithm," *SIAM J. Computing,* vol. 1, no. 2, pp. 146-160, June 1972.

[30] J.D. Ullman, *Principles of Databases and Knowledge-Base Systems.* Computer Science Press, 1989.

[31] L. Vieill, "Recursive Query Processing: The Power of Logic," *Int'l J. Theoretical Computer Science,* vol. 69, no. 1, pp. 1-53, 1989.

[32] C. Wu and L.J. Henschen, "Answering Linear Recursive Queries in Cyclic Databases," *Proc. 1988 Int'l Conf. Fifth Generation Computer Systems,* pp. 727-734, 1988.

[33] J. Ullman and A. Van Gelder, "Parallel Complexity of Logical Query Programs," *Proc. 27th Ann. IEEE Symp. Foundations of Computer Science,* pp. 438-454, 1986.

[34] R. Dechter, "Decomposing a Relation into Tree of Binary Relations," *J. Computer and System Science,* vol. 41, no. 1, pp. 2-24, 1990.

[35] A. Marchetti-Spaccamela, A. Pelaggi, and D. Sacca, "Comparison of Methods for Logic-Query Implementation," *J. Logic Programming,* vol. 10, no. 4, pp. 333-360, 1991.

Yangjun Chen received the BS degree in information system engineering from the Technical Institute of Changsha, China, in 1982, and the diploma and PhD degrees in computer science from the University of Kaiserslautern, Germany, in 1990 and 1995, respectively. From 1995 to 1997, he worked as a research assistant professor at the Technical University of Chemnitz-Zwickau, Germany. After that, he worked as a senior engineer at the German National Research Center of Information Technology (GMD) for more than two years. After a short stay at Alberta University, he joined the Department of Business Computing at the University of Winnipeg, Canada. His research interests include document and web databases, deductive databases, federated databases, constraint satisfaction problems, graph theory, and combinatorics. He has published more than 70 papers in these areas.

▷ **For more information on this or any computing topic, please visit our Digital Library at** http://computer.org/publications/dlib.