COMPARISON OF METHODS FOR LOGIC-QUERY IMPLEMENTATION*

ALBERTO MARCHETTI-SPACCAMELA,
ANTONELLA PELAGGI, AND DOMENICO SACCÀ

▷ A logic query Q is a triple $\langle G, \mathrm{LP}, D\rangle$, where G is the query goal, LP is a logic program without function symbols, and D is a set of facts, possibly stored as tuples of a relational database. The answers of Q are all facts that can be inferred from $\mathrm{LP} \cup D$ and unify with G. A logic query is *bound* if some argument of the query goal is a constant; it is *canonical strongly linear* (a *CSL query*) if LP contains exactly one recursive rule and this rule is linear, i.e., only one recursive predicate occurs in its body. In this paper, the problem of finding the answers of a bound CSL query is studied with the aim of comparing for efficiency some well-known methods for implementing logic queries: the eager method, the counting method, and the magic-set method. It is shown that the above methods can be expressed as algorithms for finding particular paths in a directed graph associated to the query. Within this graphical formalism, a worst-case complexity analysis of the three methods is performed. It turns out that the counting method has the best upper bound for noncyclic queries. On the other hand, since the counting method is not safe if queries are cyclic, the method is extended to safely implement this kind of queries as well. ◁

1. INTRODUCTION

We assume that the reader is familiar with the basic concepts of *logic programming* [6] and *relational databases* [11, 13], and with the *Logic query language* as described in [12, 13]. A (*logic*) *query* is expressed as a triple $\langle G, \mathrm{LP}, D\rangle$, where G is the

Address correspondence to Professor Domenico Saccà, Dipartamento di Sistemi, Università della Calabria, 87036 Rende Italy.

Accepted July 1990.

*This work was partially supported by the Commission of the European Communities within the project 2424 KIWIS of the program ESPRIT; the second author was working at this project as member of the team of the University of L'Aquila, and she is now with SIP (Rome, Italy). The work of the first author was also supported by a grant of M.P.I. within the project "Progetto e Analisi di Algoritmi". The work of the third author was also supported by a grant of M.P.I. within the project "Metodi formali e strumenti per basi di dati evolute" and by a grant of CNR within the project "LOGIDATA+" of the program "Sistemi Informatici e Calcolo Parallelo".

THE JOURNAL OF LOGIC PROGRAMMING

655 Avenue of the Americas, New York, NY 10010 0743-1066/91/$3.50

query goal to be solved using the rules of the *logic program* LP (without function symbols) and the facts in D possibly stored as tuples of a relational database. We focus on a subclass of queries introduced in [7] and called *canonical strongly linear queries* (*CSL queries*). The logic program of a CSL query contains exactly one recursive rule, and this rule is linear, i.e., only one recursive predicate occurs in its body. In this paper, we study the problem of finding the answer to a bound CSL query (i.e., a CSL query having some constants in the goal) with the aim of comparing for efficiency some well-known methods for implementing logic queries. To this end, we show that the problem of finding the answer of a bound CSL query Q can be formulated in terms of graphs. In particular, it is possible to associate to the query Q a *query graph*, that is, a directed graph having three different kinds of arcs, denoted by A_u, A_f, and A_d. All nodes in the query graph are reachable from a source node (say a), which corresponds to the initial bindings of the query. A node (say b) corresponds to a fact in the answer of the query if there is a (possibly cyclic) directed path from a to b having k arcs from A_u, one arc from A_f, and k arcs from A_d, where k is any nonnegative integer.

Within the above formalism, we present a graphical interpretation of three methods: the *eager* method, which was introduced in [7] and is very similar to the method of [4]; the *counting* method [1, 7, 8, 3]; and the *magic-set* method [1, 7, 8, 3]. We perform a worst-case complexity analysis of the three methods. It turns out that the counting method has the best upper bound for noncyclic queries. We note that the above mentioned methods (together with other methods) were evaluated on common examples of queries in [2]. The results of our paper confirm the results of [2] and provide a formal ground for their generalization. Finally, we extend the counting method to deal with cyclic queries.

The paper is organized as follows. In Section 2, we present an intuition of our graphical interpretation of logic queries using a simple example. Then, in Section 3, we introduce CSL queries and their properties, and we generalize the graphical interpretation to show that the problem of finding the answers of any CSL query can be formulated in terms of simple graph concepts. The proofs of some results of this section are reported in the appendix. In Section 4, we describe the three mentioned methods in terms of the graph formalism, and in Section 5 we supply the complexity analysis of these methods. Finally, we extend the counting method to cope with cyclic CSL queries in Section 6, and we give the conclusions and discuss further work in Section 7.

2. AN EXAMPLE

Consider the following logic program:

Program P1.

r_0: $g(X, Y) :- up(X, W), down(Z, Y), g(W, Z).$
r_1: $g(X, Y) :- flat(X, Y).$
r_2: $up(a, a_1).$
r_3: $up(a_1, a_2).$
r_4: $up(a, a_3).$
r_5: $up(a_4, a_2).$
r_6: $up(a_5, a_4).$

r_7: $flat(a_2, b_1)$.
r_8: $flat(a_1, b_1)$.
r_0: $down(b_1, b_2)$.
r_{10}: $down(b_2, b_3)$.
r_{11}: $down(b_1, b_3)$.

This logic program is a syntactic variation of the well-known same-generation example. In fact, the predicates *up*, *flat*, and *down* correspond to the relations *Child*, *Sibling*, and *Parent*, respectively, and $g(X, Y)$ is true if X and Y are of the same generation. In fact, two human beings x and y are of the same generation if either x is sibling of y (see rule r_1) or there exist w and z such that x is child of w, z is parent of y, and w and z are of the same generation (see rule r_0). In other words, the fact $g(x, y)$ is inferred from P1 if either the fact $flat(x, y)$ is in P1 or there is an integer i, $i > 0$, and constants $w_1, w_2, \ldots, w_i, z_1, z_2, \ldots, z_i$ such that all the facts

$$up(x, w_1); up(w_1, w_2); \ldots; up(w_{i-1}, w_i), flat(w_i, z_i);$$
$$down(z_i, z_{i-1}); \ldots; down(z_2, z_1); down(z_1, y)$$

are in P1.

As an example, consider the two constants a and b_3. The fact $g(a, b_3)$ can be inferred from P1 in two different ways:

(1) $up(a, a_1); flat(a_1, b_1); down(b_1, b_3)$, or

(2) $up(a, a_1); up(a_1, a_2); flat(a_2, b_1); down(b_1, b_2); down(b_2, b_3)$.

The logic program P1 can be graphically interpreted as follows. We define a directed graph $G = (N, A)$, where the set of nodes is the Herbrand universe and the set of arcs A consists of the union of the three following disjoint sets:

(1) $A_u = \{(x, y) \mid up(x, y)$ is in P1$\}$;

(2) $A_f = \{(x, y) \mid flat(x, y)$ is in P1$\}$;

(3) $A_d = \{(x, y) \mid down(x, y)$ is in P1$\}$.

The graph G is shown in Figure 1 (arcs in A_u and in A_d are represented by solid arrows going up and down, respectively, and arcs in A_f are represented by dotted arrows).

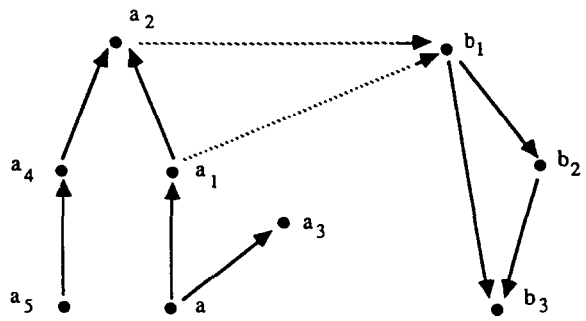

FIGURE 1. Graph G.

Using the graphical representation, it is easy to see that a fact $g(x, y)$ can be inferred by P1 if and only if there is a path from x to y of length $2i + 1$, where $i \geq 0$, such that

(1) the first i arcs belong to A_u,

(2) the $(i + 1)$th belongs to A_f, and

(3) the last i arcs belong to A_d

(this kind of path is called an *answer path*).

As an example, we consider again the pair of constants a, b_3. There are two answer paths from a to b_3, namely,

(1) $\langle (a, a_1), (a_1, b_1), (b_1, b_3) \rangle$, and

(2) $\langle (a, a_1), (a_1, a_2), (a_2, b_1), (b_1, b_2), (b_2, b_3) \rangle$.

These two paths correspond to the two derivations of the fact $g(a, b_3)$, shown before.

It follows that the problem of answering a query on the logic program P1 coincides with the problem of finding answer paths in the graph of Figure 1. Moreover, methods to implement logic queries differ in the strategy for determining such paths. As an example, consider the query ? $g(a, Y)$. A first strategy is to start from the node a, to consider one arc at the time, and to find all possible answer paths leaving a; this means that the same arcs can be taken into account several times and, if the graph is cyclic, termination is not guaranteed. This is the implementation strategy used by PROLOG [6].

A different strategy is to add new dotted arcs to the graph of Figure 1 as follows. If there are a dotted arc from w to z, a solid arc going up from x to w, and a solid arc going down from z to y, then a dotted arc from x to y is added. As soon as no new arc can be added, the query answers are determined by taking the target nodes of dotted arcs leaving a. This is the strategy used by database-oriented implementations of logic queries [12], and it can be expressed as the least fixpoint of the following function over database relations:

$$f(\mathbf{g}) = \textbf{\textit{flat}} \cup \pi_{1,6}((\textbf{\textit{up}} \bowtie_{2=1} \mathbf{g}) \bowtie_{4=1} \textbf{\textit{down}}).$$

The function f is defined by a join-project-union algebra expression [11] having as operands the constants ***flat, up,*** and ***down*** and the variable $\mathbf{g}$, where ***flat***, ***up***, and ***down*** are database relations corresponding to the predicates with symbols *flat*, *up*, and *down*, respectively, and $\mathbf{g}$ is an unknown database relation. After computing $\mathbf{g}$, the answers are given by the second components of those tuples of $\mathbf{g}$ whose first component is equal to a; thus

$$\pi_2(\sigma_{1=a}\mathbf{g}).$$

It is easy to see that the second strategy terminates even when the graph is cyclic; however, the binding a of the query is not used to restrict the fixpoint computation. To overcome this limitation, a number of methods have been introduced that allow an efficient database-oriented implementation of bound logic queries. Also these methods can be characterized by their strategies for finding answer paths, as will be shown in the next sections, where a similar graph formalism is used to describe properties of a larger class of logic programs.

3. GRAPHICAL REPRESENTATION OF LOGIC QUERIES

3.1. Logic Queries

A logic program is a set of *rules* (Horn clauses). We assume that no function symbols occur in logic programs; thus we confine ourselves to so-called *Datalog* programs. We often denote predicates with capital letters such as $P, Q, \ldots$, and we assume that the symbols of such predicates are the corresponding lowercase letters $p, q, \ldots$. In other words, if P denotes a predicate, then we assume that this predicate has the form $p(\boldsymbol{x})$, where $\boldsymbol{x}$ is a list of arguments and p is the predicate symbol.

A predicate without variables is *ground*. A rule with ground head predicate and empty body is a *fact*.

Let L be a logic program and S be the set of all predicate symbols occurring in L. The *dependency graph* [13] of L is the directed graph $\mathrm{DG}_L = (S, A)$ such that there is an arc (p, q) in A if and only if there is a rule in L where q is the head predicate symbol and p is a predicate symbol occurring in its body [13]. A predicate symbol is *recursive* if it is on one or more cycles of DG_L; predicates are then classified as recursive or nonrecursive according to their symbols. A rule r (say, with head predicate symbol p) is *recursive* if there is a predicate symbol q in the body of r such that p and q belong to the same strong component in DG_L. Given a logic program L and a predicate symbol q, a rule r in L *defines* q if the head predicate symbol of r is q.

Given a logic program L and a predicate G, we denote by LP_G the set of all rules in L defining g and all predicate symbols belonging to the same strong component as g (recall that g is the predicate symbol of G). Moreover, we denote by D the set of all rules in $L - LP_G$ defining all predicate symbols q such that there is a path from q to g in DG_L. A *logic query* is a triple $\langle G, \mathrm{LP}_G, D \rangle$; the predicate G is called the *query goal*. From now on, we shall refer to LP_G simply as LP. Furthermore, since the rules in D can be solved independently of those in LP, without loss of generality we shall assume that D is a (possibly infinite) set of facts. All predicates that are defined in LP are called *query predicates*, whereas all predicates that are defined in D are called *datum predicates*. In addition, all datum predicates that are defined by a finite number of facts in D are called *database predicates*.[1] The *answers* of Q are all facts that both can be inferred from $\mathrm{LP} \cup D$ and unify with G.

Example 1. Consider the same-generation example of the previous section (Program P1). A possible logic query is $\langle g(a, Y), \mathrm{LP}, D \rangle$, where $\mathrm{LP} = \{r_0, r_1\}$ and $D = \{r_2, \ldots, r_{11}\}$. The answers are $g(a, b_2)$ and $g(a, b_3)$.

Two queries are *equivalent* if they have the same answers.

[1] The facts defining database predicates can be though of as tuples of a relational database. On the other hand, comparison predicates are examples of datum predicates that are defined by an infinite number of facts in D.

3.2. CSL Queries

A query $Q = \langle G, \mathrm{LP}, D \rangle$ is

(1) *recursive* if LP contains at least one recursive rule (it follows that query predicates are recursive);

(2) *linear* if it is recursive and every recursive rule in LP contains exactly one query predicate (i.e., recursive predicate) in its body;

(3) *strongly linear* if it is linear and there are no two recursive rules in LP with the same head predicate symbol.

It turns out that strongly linear queries have at most one cycle of recursion; thus they are the simplest recursive queries.

A query $Q = \langle G, \mathrm{LP}, D \rangle$ is *canonical strongly linear* (a *CSL query*) if it is linear and the logic program LP contains exactly one recursive rule [7]. It is easy to see that any strongly linear query can be transformed into a CSL query by unfolding the logic program LP so that all rules in LP have g as head predicate symbol. Therefore, all results for CSL queries hold for strongly linear queries as well.

Example 2. The query $Q = \langle g(a, Y), \mathrm{LP}_1, D \rangle$, where LP_1 is the logic program

r_0: $g(X, Y) :- b(X, W), g(W, Z), g(U, V), c(V, Y).$
r_1: $g(Z, W) :- d(Z, W).$

and D is a set of facts defining the predicates symbols b, c, and d, is a recursive nonlinear query.

If we replace LP_1 with the following logic program LP_2:

r_0: $g(X, Y) :- g(X, Z), c(Z, Y).$
r_1: $g(Z, W) :- b(Z, Y), g(Y, W).$
r_2: $g(U, V) :- d(U, V).$

then the query $\langle g(a, Y), \mathrm{LP}_2, D \rangle$ is linear but not strongly linear. On the other side, given the following logic program LP_3:

r_0: $g(X, Y) :- p(X, Z), c(Z, Y).$
r_1: $p(Z, W) :- b(Z, Y), g(Y, W).$
r_2: $p(U, V) :- d(U, V).$

the query $\langle g(a, Y), \mathrm{LP}_3, D \rangle$ is strongly linear and can be reduced to a CSL query by modifying LP_3 into the following program LP_4:

r_0: $g(X, Y) :- b(X, W), g(W, Z), c(Z, Y).$
r_1: $g(U, V) :- d(U, Z), c(Z, V).$

It is easy to see that $\langle g(a, Y), \mathrm{LP}_3, D \rangle$ and $\langle g(a, Y), \mathrm{LP}_4, D \rangle$ have the same answers. Finally, the query in Example 1 is a CSL query.

Given a CSL query $Q = \langle g(\mathbf{x}), \mathrm{LP}, D \rangle$, where $\mathbf{x}$ is a list of arguments, the logic program LP has the following structure:

r_0: $g(\mathbf{x}_0) :- C_0, g(\mathbf{y}).$
r_1: $g(\mathbf{x}_1) :- C_1.$
$\vdots$
r_{lp}: $g(\mathbf{x}_{\mathrm{lp}}) :- C_{\mathrm{lp}}.$

where $\mathbf{x}_0, \ldots, \mathbf{x}_n$, $\mathbf{y}$ are lists of arguments, $lp = |LP| - 1$, and $C_0, \ldots, C_n$ are conjunctions of database predicates that are defined in D. For simplicity but without real restriction, we require that all the variables in $\mathbf{x}$ be distinct and that $\mathbf{x}_0$ and $\mathbf{y}$ only contain variables. In addition, without loss of generality, we suppose that all constants appearing in LP also appear in D, so that the Herbrand's universe is given by the set of all constants occurring in D. From now on, we assume that every logic program has the above structure, i.e., r_0 is the recursive rule, g is the recursive predicate symbol, and so on.

3.3. *Bound CSL Queries*

In this paper, we are interested in CSL queries where some of the arguments in the query goal are constants (*bound*) and this initial binding may be propagated top-down through the recursive rule using database predicates. (Note that datum predicates that are not defined by a finite set of facts cannot effectively propagate bindings.) For instance, the constant a in the query goal of Example 1 binds the first argument of the recursive predicate. This binding is propagated by the database predicate *up* of the rule r_0, giving two additional bindings a_1, a_3 for the first argument of the recursive predicate. In turn, again rule r_0 propagates the binding a_1 into a_2. At this point, no further propagation is possible. The so-computed bindings can be now be used to restrict the actual computation of the answer. In general, the binding is propagated in a quite complex way (or it is not propagated at all). For instance, a binding on the first argument of the recursive predicate can generate a binding on the second argument, which, in turn, adds bindings for the first argument, and so on. We now provide a precise characterization of how bindings are propagated via database predicates.

Given a predicate P, we denote a list of arguments in P by a tuple S of ordered position indices. In general, we shall use this notation to indicate bound arguments in P. For example, given the predicate $g(X, Y, Z, W)$, if $S = \{1, 3\}$ then the arguments X and Z are bound. We note that a different notation has been proposed in [12], where bound arguments are denoted by a string of b (bound) and f (free), called *adornment*; in the above example, the adornment is "*bfbf*". Clearly, the two notations are equivalent.

Suppose that the S-arguments of the predicate $g(\mathbf{x}_0)$ in r_0 are bound.[2] Then the *set of variables bound in* r_0 *by* S, denoted by B_S, is defined as follows:

(1) every variable appearing in any bound argument in S is in B_S;

(2) if a variable occurring in a bound argument in S appears in a database predicate P of C_0, then all the other variables in P are in B_S.

Obviously, if S is empty then B_S is empty as well. If all variables of a datum predicate are in B_S, then the predicate is said to be *bound* by S.

Let T^S be the index tuple denoting all arguments y in $\mathbf{y}$ such that y is a variable in B_S. The database predicates in C_0 propagate bindings from the bound arguments S in the head of r_0 to the T^S-arguments of the recursive predicate in

[2] We often blur the difference between arguments and indices denoting them.

the body; we say that the T^S-arguments in the body are bound by S. Note that T^S may be empty, i.e., the binding is not propagated.

The *binding graph* of the CSL query $Q = \langle g(\mathbf{x}), \mathrm{LP}, D\rangle$ is the directed graph $B_Q = \langle N, A\rangle$ having nodes of the form S, where S is an index tuple for the arguments of g. The binding graph B_Q is constructed as follows:

(1) if X denotes the index tuple of all constant arguments in the goal $g(\mathbf{x})$, then $X \in N$ (*source node*);

(2) if there exists a node S in N, then T^S and (S, T^S) are in N and A, respectively, where T^S denotes the arguments of the recursive predicate in the body of r_0 that are bound by the arguments S in the head.

Let G be a directed graph with nodes $S_1, S_2, \ldots, S_k$. G is a *single-cycle* graph if it is composed of an initial (possibly empty) acyclic path followed by a cycle; thus the arcs of G are $(S_1, S_2), (S_2, S_3), \ldots, (S_{j-1}, S_j), (S_j, S_{j+1}), \ldots, (S_{k-1}, S_k), (S_k, S_j)$. If $j = 1$, then the initial path is empty and the graph is a cycle.

Example 3. Consider the query $\langle g(a, b, Z, W), \mathrm{LP}, D\rangle$, where D contains a number of facts defining the database predicates a, b, c, d, e, f, and where LP is

$$r_0\colon g(X,Y,Z,W) :- a(X,U), b(Y,U,\hat{Z}), c(Z,\hat{X},\hat{Y}), d(W), e(\hat{W}), g(\hat{X},\hat{Y},\hat{Z},\hat{W}).$$

$$r_1\colon g(X,Y,Z,W) :- f(X,Y,Z,W).$$

Suppose that $S = \{1,2\}$; then the database predicates that are bound by S are $a(X,U)$ and $b(Y,U,\hat{Z})$. Moreover, $T^S = \{3\}$; thus the argument $\hat{Z}$ of $g(\hat{X},\hat{Y},\hat{Z},\hat{W})$ is bound by S.

The binding graph of the query is shown in Figure 2(a). If we now replace the query goal with $g(a,Y,Z,W)$ and $g(X,Y,Z,a)$, the corresponding binding graphs are those shown in Figure 2(b) and (c), respectively. Notice that the three binding graphs are single-cycle.

Next we present some properties of CSL queries whose proofs directly derive from the previous definitions.

Fact 1. Let $Q = \langle G, \mathrm{LP}, D\rangle$ be a query. Then

(a) *the recognition of whether Q is a CSL query can be done in time linear in the size of* LP;

(b) *if Q is a CSL query, then the binding graph B_Q is a single-cycle graph and can be constructed in time $O(2^n|\mathrm{LP}|)$, where $|\mathrm{LP}|$ is the size of* LP *and n is the largest arity of the predicates in* LP.

By Fact 1(b), the binding graph B_Q can be represented as $\langle S_1, S_2, \ldots, S_j, S_{j+1}, \ldots, S_k, S_j\rangle$, where S_i, $1 \le i \le k$, are the nodes of B_Q; S_1 is the source node; the arcs (S_i, S_{i+1}), $1 \le i \le j-1$, form the initial path; and the arcs $(S_j, S_{j+1}), \ldots, (S_{k-1}, S_k), (S_k, S_j)$ form a cycle.

The CSL query Q is *bound* if no node in the binding graph B_Q is an empty list. The first two queries of Example 3 [see the binding graphs in Figure 2(a) and (b)] are bound, whereas the third query [see Figure 2(c)] is not.

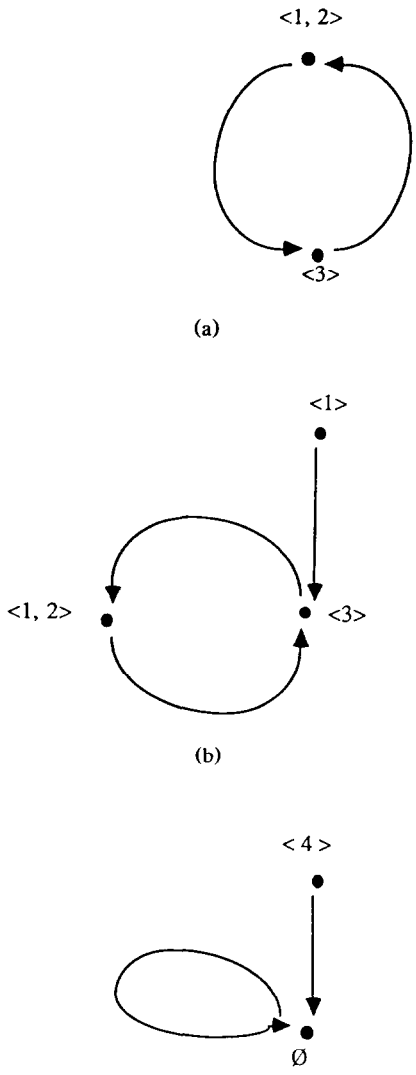

FIGURE 2. Binding Graphs.

Let Q be a bound query and B_Q be its binding graph, represented by $\langle S_1, S_2, \ldots, S_k, S_j \rangle$. For each S_i, $1 \leq i \leq k$, we denote (1) by S_i^- the indices that are not in S_i, (2) by L^{S_i} the conjunction of all predicates in C_0 that are bound by S_i, and (3) by $L^{S_i^-}$ the conjunction of all predicates in C_0 that are not in L^{S_i}. The initial bindings of the query goal on the arguments denoted by S_1 are propagated to the arguments denoted by S_2 through the database predicates in L^{S_1}, then from S_2 to S_3 through the database predicates in L^{S_2}, and so on. Eventually, the bindings are propagated back from S_k to S_j via L^{S_k}. If for some i, $1 \leq i \leq k$, L^{S_i} is empty, then the binding is directly propagated; thus every argument y in T^{S_i} is a variable appearing in some S_i-argument of the head predicate.

Example 4. Consider the query $\langle g(a, b, Z, W), \mathrm{LP}, D\rangle$ of Example 3. We have $L^{S_1} = a(X, U), b(Y, U, \hat{Z})$ and $L^{S_2} = c(Z, \hat{X}, \hat{Y})$, where $S_1 = \langle 1, 2\rangle$ and $S_2 = \langle 3\rangle$. In addition, $S_1^- = \langle 3, 4\rangle$ and $L^{S_1^-}$ is equal to

$$c(Z, \hat{X}, \hat{Y}), d(W), e(\hat{W}).$$

Finally, $S_2^- = \langle 1, 2, 4\rangle$ and $L^{S_2^-}$ is equal to

$$a(X, U), b(Y, U, \hat{Z}), d(W), e(\hat{W}).$$

3.4. *Query Graph*

We now associate a directed graph to a bound CSL query to provide an interpretation of it. The nodes of the graph correspond to tuples of constants; in particular, the source node corresponds to the tuple of constants in the query goal. The other nodes (and incoming arcs) are obtained by retrieving tuples from the database D via a goal composed by a conjunction of database predicates, using restrictions from the tuples corresponding to previously generated nodes. In order to formally define such a graph, we require additional definitions.

Let $\mathbf{z}$ be a list of arguments, and let S be an index tuple denoting some arguments of $\mathbf{z}$. Then $\mathbf{z}(S)$ stands for the ordered list of the arguments of $\mathbf{z}$ that are denoted by S.

A (*ground*) *substitution* σ for a set of variables $\mathbf{X}$ is a mapping from $\mathbf{X}$ to the Herbrand universe. Let $\mathbf{z}$ be a list of arguments whose variables are in $\mathbf{X}$. Then $\mathbf{z}\sigma$ denotes the list of arguments obtained from $\mathbf{z}$ by simultaneously replacing each occurrence of the variable X (for every X in $\mathbf{X}$) with the constant $\sigma(X)$. Furthermore, if L is a conjunction of predicates whose variables are in $\mathbf{X}$, then $L\sigma$ denotes the conjunction obtained from L by replacing the argument list $\mathbf{z}$ of every predicate in L with $\mathbf{z}\sigma$. It turns out that every predicate in $L\sigma$ is ground.

Let $Q = \langle G, \mathrm{LP}, D\rangle$ be a bound CSL query whose binding graph is represented by $\langle S_1, \ldots, S_k, S_j\rangle$. For each S_i, $1 \le i \le k$, we denote by S_{i+} the subsequent node in the above list (i.e., S_{i+1} if $i < k$, or S_j if $i = k$).

Example 5. If $\mathbf{z} = \langle X, Y, Z, W\rangle$ and $U = \langle 1, 2, 4\rangle$, then $\mathbf{z}(U) = \langle X, Y, W\rangle$. Let $\sigma = \{(X, a), (Y, b), (Z, c), (W, a)\}$ and $L = p(X, d, Y), q(Y, W)$. Then $\mathbf{z}\sigma = \langle a, b, c, a\rangle$ and $L\sigma = p(a, d, b), q(b, a)$. Finally, given the binding graph $\langle S_1, S_2, S_3, S_2\rangle$, we have $S_{1+} = S_2$, $S_{2+} = S_3$, and $S_{3+} = S_2$.

Let Q be a query and $\langle S_1, S_2, \ldots, S_k, S_j\rangle$ be its binding graph. The *query graph* of Q is the directed graph $G_Q = \langle N_u \cup N_d, A_u \cup A_f \cup A_d\rangle$ having nodes of the form $[S, t]$, where S is an index tuple and t is a tuple of constants, both having the same number of components. The query graph G_Q is constructed as follows:

(a) The node $[S_1, \mathbf{a}]$ is in N_u (*source node*), where $\mathbf{a}$ is the list of constants in the query goal.[3]

(b) If $[S_i, \mathbf{a}_1]$ is in N_u and there exists a substitution σ for the set of variables in L^{S_i} and $\mathbf{x}_0(S_i)$ such that $\mathbf{x}_0(S_i)\sigma = \mathbf{a}_1$ and every predicate in $L^{S_i}\sigma$ is in D, then the node $[S_{i+}, \mathbf{a}_2]$ is in N_u and the arc $([S_i, \mathbf{a}_1], [S_{i+}, \mathbf{a}_2])$ is in A_u,

[3]Here and in what follows, constants are listed in the order in which they appear in the predicate.

where $\mathbf{a}_2 = \mathbf{y}(S_{i+})\sigma$. [Note that every variable in $\mathbf{y}(S_{i+})$ also occurs in L^{S_i} or in $\mathbf{x}_0(S_i)$.]

(c) If $[S_i, \mathbf{a}_1]$ is in N_u and there exists both a nonrecursive rule in LP, say

$$r_q\colon g(\mathbf{x}_q) :- C_q,$$

and a substitution σ for the set of variables in C_q and in $\mathbf{x}_q$ such that $\mathbf{x}_q(S_i)\sigma = \mathbf{a}_1$ and every predicate in $C_q\sigma$ is in D, then the node $[S_i^-, \mathbf{b}_1]$ is in N_d and the arc $([S_i, \mathbf{a}_1], [S_i^-, \mathbf{b}_1])$ is in A_f, where $\mathbf{b}_1 = \mathbf{x}_q(S_i^-)\sigma$.

(d) If $[S_q^-, \mathbf{b}_2]$ is in N_d, there is S_i such that $S_{i+} = S_q$, and there exists a substitution σ for the set of variables in $L^{S_i^-}$, in $\mathbf{y}(S_q^-)$, and in $\mathbf{x}_0(S_i^-)$ such that $\mathbf{y}(S_q^-)\sigma = \mathbf{b}_2$ and every predicate in $L^{S_i^-}\sigma$ is in D, then the node $[S_i^-, \mathbf{b}_1]$ is in N_d and the arc $([S_1^-, \mathbf{b}_2], [S_i^-, \mathbf{b}_1])$ is in A_d, where $\mathbf{b}_1 = \mathbf{x}_0(S_i^-)\sigma$.

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

Example 6. Consider the query $\langle g(a, b, Z, W), \mathrm{LP}, D\rangle$ of Example 3. Suppose that the facts in D are those stored in the following database relations:

a

a	e_1
a_1	e_2
a_2	e_3
a_3	e_3

b

b	e_1	c_1
b	e_1	c_2
b_1	e_2	c_3
b_2	e_3	f_1

c

c_1	a_1	b_1
c_2	a_1	b_1
c_4	h_1	g_1
c_5	a_2	b_2

d

m

e

l_1
l_2
m

f

h_1	g_1	c_1	l_1
a_1	b_1	f_1	l_2

The query graph is shown in Figure 3. The dotted arcs are in A_f, the solid arcs going up are in A_u, and those going down are in A_d. The three subgraphs G_u, G_f, and G_d are outlined in the figure.

Notice that the graph of Figure 1 resembles the query graph of the query $\langle g(a, Y), \mathrm{LP}, D\rangle$, where LP consists of the two rules r_0 and r_1 defining the same-generation predicate and D is the set of facts $r_2, \ldots, r_{12}$. As a consequence, one could expect that any answer of a bound CSL query will correspond to a node that is reachable from the source node through a path having i arcs from A_u, one arc from A_f, and i arcs from A_d, where $i \geq 0$. In fact, this property is confirmed by the next theorem.

Before stating the theorem, we note that, in finding query answers, we are actually interested only in those arguments that are unbound in the query goal; we

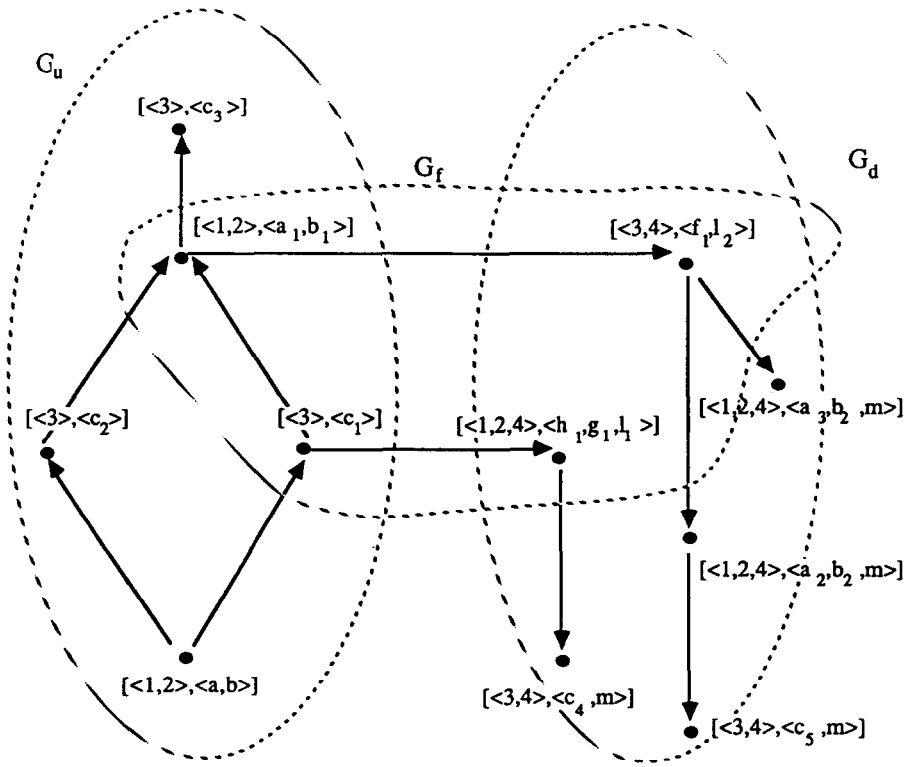

FIGURE 3. Query graph.

call these arguments an *answer tuple*. More formally, an answer tuple of Q is a tuple $\mathbf{b}$ of constants such that there exists an answer $g(\mathbf{c})$ for which $\mathbf{c}(S_1^-) = \mathbf{b}$, where S_1^- denotes the unbound arguments of the query goal. In the following, whenever no confusion arises, we refer to an answer tuple simply as an answer. Furthermore, given the query graph, we define an *answer path* as a (possibly cyclic) path of length $2i + 1$, $i \geq 0$, from a node in N_u to a node in N_d such that the first i arcs are in A_u, the $(i + 1)$th arc is in A_f, and the last i arcs are in A_d.

Theorem 1. Let $Q = \langle G, \mathrm{LP}, D \rangle$ be a bound CSL query, and G_Q be the query graph of Q. If a tuple $\mathbf{b}$ is an answer of Q, then there exists an answer path from the source node $[S_1, \mathbf{a}]$ to the node $[S_1^-, \mathbf{b}]$ in G_Q.

PROOF. In the appendix. □

The question now is whether the reverse condition of Theorem 1 holds, that is, whenever there exists an answer path from the source node $[S_1, \mathbf{a}]$ to a node $[S_1^-, \mathbf{b}]$ in G_Q, then the tuple $\mathbf{b}$ is an answer of Q. Unfortunately, this is not the case, as shown in the next example.

Example 7. Consider the query $\langle g(X, a), \mathrm{LP}, D \rangle$, where LP is

$$g(X,Y) :- p_1(X, \hat{Y}, Y), g(\hat{X}, \hat{Y}), p_2(\hat{X}).$$
$$g(X,Y) :- p_3(X, Y).$$

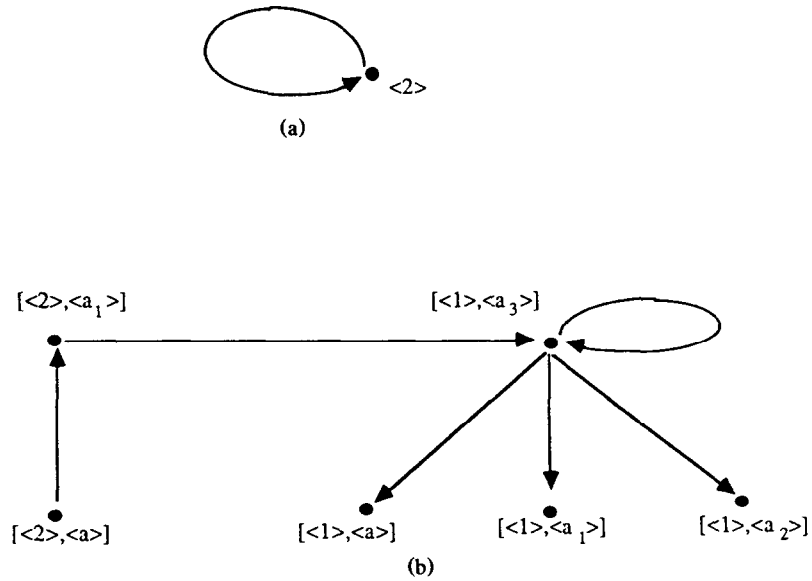

FIGURE 4. Graphs for the query of Example 7: (a) binding graph, (b) query graph.

and D consists of the following facts:

$p_1(a_2, a_1, a)$.

$p_2(a_3)$.

$p_3(a_3, a_1)$.

The binding graph is shown in Figure 4(a). The query graph is constructed as follows. The source node is $[\langle 2\rangle, \langle a\rangle]$, and the Herbrand universe is $\{a, a_1, a_2, a_3\}$. Consider the substitution $\sigma = \{(X, a_2), (Y, a), (\hat{Y}, a_1)\}$ for the variables in $L^{\langle 1\rangle} = p_1(X, \hat{Y}, Y)$ and in $\mathbf{x}_0(\langle 2\rangle) = \langle Y\rangle$. Since $p_1(a_2, a_1, a)$ is in D, by the definition of query graph [part (b)] the node $[\langle 2\rangle, \langle a_1\rangle]$ is in N_u and the arc $([\langle 2\rangle, \langle a\rangle], [\langle 2\rangle, \langle a_1\rangle])$ is in A_u. It is easy to see that no other node is in N_u and no other arc is in A_u. Consider now the substitution $\sigma = \{(X, a_3), (Y, a_1)\}$ for the variables in $C_1 = p_3(X, Y)$ and in $\mathbf{x}_1 = \langle X, Y\rangle$. Since $[\langle 2\rangle, \langle a_1\rangle]$ is in N_u and $p_3(a_3, a_1)$ is in D, by the definition of query graph [part (c)] the node $[\langle 1\rangle, \langle a_3\rangle]$ is in N_d and $([\langle 2\rangle, \langle a_1\rangle], [\langle 1\rangle, \langle a_3\rangle])$ is in A_f (and not other arc is in A_f). Finally, consider the following four different substitutions for the variables in $L^{\langle 1\rangle} = p_2(\hat{X})$, in $\mathbf{y}(\langle 1\rangle) = \amalg[\hat{X}\rangle$ and in $\mathbf{x}_0(\langle 1\rangle) = \langle X\rangle$ (note that $S_1^- = \langle 1\rangle$):

$\sigma_1 = \{(X, a), (\hat{X}, a_3)\}$;

$\sigma_2 = \{(X, a_1), (\hat{X}, a_3)\}$;

$\sigma_1 = \{(X, a_2), (\hat{X}, a_3)\}$;

$\sigma_1 = \{(X, a_3), (\hat{X}, a_3)\}$.

Since the fact $p_2(a_3)$ is in D, by the definition of query graph [part (d)] the nodes $[\langle 1\rangle, \langle a\rangle], [\langle 1\rangle, \langle a_1\rangle], [\langle 1\rangle, \langle a_2\rangle]$ are in N_d (besides the node $[\langle 1\rangle, \langle a_3\rangle]$

that is already in N_d), and the arcs $([\langle 1\rangle, \langle a_3\rangle], [\langle 1\rangle, \langle a_3\rangle])$, $([\langle 1\rangle, \langle a_3\rangle], [\langle 1\rangle, \langle a\rangle])$, $([\langle 1\rangle, \langle a_3\rangle], [\langle 1\rangle, \langle a_1\rangle]$ and $([\langle 1\rangle, \langle a_3\rangle], [\langle 1\rangle, \langle a_2\rangle])$ are in A_d. The query graph is shown in Figure 4(b). Obviously, the only answer of the query is $g(a_2, a)$. Therefore, the existence of answer paths from $[\langle 2\rangle, \langle a\rangle]$ to $[\langle 1\rangle, \langle a\rangle]$, $[\langle 1\rangle, \langle a_1\rangle]$, and $[\langle 1\rangle, \langle a_3\rangle]$ does not guarantee that $g(a, a)$, $g(a_1, a)$, and $g(a_3, a)$ are in the answer as well.

3.5. *1-Bound CSL Query*

We now introduce a subclass of bound CSL queries for which the reverse of Theorem 1 holds. Let $Q = \langle G, \mathrm{LP}, D\rangle$ be a bound CSL query and B_Q be its binding graph. Q is 1-*bound* if for each node S in B_Q,

$$\mathbf{X} \cap B_S = \varnothing,$$

where B_S is the set of all variables bound by S, and $\mathbf{X}$ is the set of all variables that are in L^{S^-} or in the arguments of the head predicate that are not denoted by S. In other words, it is required that no bound variable occur in unbound datum predicates or in unbound arguments of the head predicate.[4]

Example 8. The query of Example 7 is not 1-bound, because the head argument X, denoted by $S_1^- = \langle 1\rangle$, is in B_S, where $S_1 = \langle 2\rangle$ is a node of the binding graph [see Figure 4(a)]. On the other hand, the query $\langle g(a, b, Z, W), \mathrm{LP}, D\rangle$ of Example 3 and the query of Example 1 are 1-bound. To see another example of a query that is not 1-bound, consider the following logic program LP_1:

r_0: $g(X,Y) :- a(X,Z), g(Z,W), b(W,Y), Z > Y.$
r_1: $g(X,Y) :- a(X,Y).$

The CSL query $\langle g(a, Y), \mathrm{LP}_1, D\rangle$, where D is any set of facts, is not 1-bound, since the variable Z is both in L^{S_1} and in $L^{S_1^-}$ [note that $S_1 = \langle 1\rangle$, $S_1^- = \langle 2\rangle$, $L^{\langle 1\rangle}$ is $a(X,Z)$, and $L^{\langle 2\rangle}$ is $b(W,Y), Z > Y$].

Theorem 2. Let $Q = \langle G, \mathrm{LP}, D\rangle$ be a 1-bound CSL query and G_Q be the query graph of Q. If there exists an answer path from the source node $[S_1, \mathbf{a}]$ to a node $[U, \mathbf{b}]$ in N_d, then the tuple $\mathbf{b}$ is an answer tuple of Q.

PROOF. In the appendix. □

It is easy to see that 1-bound CSL queries constitute the largest class of CSL queries for which Theorem 2 holds. In other words, given a bound query $Q = \langle G, \mathrm{LP}, D\rangle$, if Q is not 1-bound, then there exists a query $\hat{Q} = \langle G, \mathrm{LP}, \hat{D}\rangle$ such that there is an answer path from the source node $[S_1, \mathbf{a}]$ to a node $[S_1^-, \mathbf{b}]$ in the query graph of $\hat{Q}$, and $\mathbf{b}$ is not an answer tuple of $\hat{Q}$. We can therefore say that the class of 1-bound CSL-queries is the maximum generalization of the well-known same-generation query.

We conclude the section by observing that recognition of a 1-bound CSL-query can be easily done while constructing the binding graph.

[4] This condition corresponds to the condition for the counting method to be applied in a simple way, without introducing the so-called supplementary counting rules (see [8]).

Fact 2. Let $Q = \langle G, \mathrm{LP}, D \rangle$ be a CSL query. If there is a bound on the arity of the recursive predicate, then the recognition of whether Q is 1-bound can be done in time linear in the size of LP.

4. GRAPH ALGORITHMS FOR LOGIC-QUERY IMPLEMENTATION

Many methods have been introduced to implement logic queries, based on the fixpoint computation of relational-algebra expressions. We now show that such methods can be expressed in terms of graph algorithms using the results of the previous section. In particular, since answering a 1-bound CSL query coincides with finding answer paths, the various methods can be characterized by the strategy used to single out such paths. As a consequence, the graph interpretation provides a unifying ground to compare methods for efficiency.

Let us consider a 1-bound CSL query Q and its query graph $G_Q = (N, A)$. Recall that G_A is composed of three subgraphs: $G_u = (N_u, A_u)$, $G_f = (N_f, A_f)$, and $G_d = (N_d, A_d)$. Furthermore, $N = N_u \cup N_d$, N_u and N_d are disjoint, $N_f \subset N_u \cup N_d$, and G_f is bipartite, since every arc in A_f goes from a node in N_u to a node in N_d. Finally, A_u, A_f, and A_d are disjoint. We denote the source node of G_Q by a.

We are now ready to present a uniform graph description of three methods for implementing logic queries, namely, the eager method, the counting method, and the magic-set method. In the description of the methods, given a subgraph G (i.e., G can be G_u, G_f, or G_d) and a subset X of nodes of G, we denote by $\mathrm{adj}\, G(X)$ [$\mathrm{adj}^{-1} G(X)$] the set of all nodes j such that the arc (i, j) [(j, i)] is in G and i is in X. In other words, $\mathrm{adj}\, G(X)$ is the set of all nodes that are adjacent to some node in X, whereas $\mathrm{adj}^{-1} G(X)$ is the set of all nodes having at least one adjacent node in X. It is easy to see that these two sets of nodes can be computed using the rules stated in the definition of the query graph, which can be also expressed as relational-algebra expressions.

Example 9. Consider the query $\langle g(a, b, Z, W), \mathrm{LP}, D \rangle$ of Example 3 and the query graph in Figure 3. Say that

$$X = \{[\langle 3 \rangle, \langle c_1 \rangle], [\langle 3 \rangle, \langle c_2 \rangle]\}.$$

Then

$$\mathrm{adj}\, G_u(X) = \{[\langle 1,2 \rangle, \langle a_1, b_1 \rangle]\},$$

$$\mathrm{adj}^{-1} G_u(X) = \{[\langle 1,2 \rangle, \langle a, b \rangle]\},$$

and

$$\mathrm{adj}\, G_f(X) = \{[\langle 1,2,4 \rangle, \langle h_1, g_1, l_1 \rangle]\}.$$

We recall that $L^{\langle 1,2 \rangle}$ is the conjunction $a(X, U), b(Y, U, \hat{Z})$ and that $L^{\langle 3 \rangle}$ is the predicate $c(Z, \hat{X}, \hat{Y})$. It is easy to see that the relation-algebra expression

$$\pi_{2,3}(\sigma_{(1=c_1) \text{ or } (1=c_2)} c)$$

computes $\mathrm{adj}\, G_u(X)$.

```
U := {a};
Answer := adj G_f(U); {INITIALIZATION }
v := 0;
While U ≠ ∅do
begin
    U := adj G_u(U); v := v + 1; {UP}
    D := adj G_f(U); {FLAT}
    t := v;
    While D ≠ ∅ and t > 0
    do begin
        D := adj G_d(D); {DOWN}
        t := t - 1
    end;
    Answer := D ∪ Answer
end
```

FIGURE 5. Eager method.

4.1. *The Eager Method*

The eager method is described in [7] and is similar to the method of [4]. The method works as follows. Let U be a variable whose type is the power set of N_u, and let v be a counter associated to U. At the beginning, U only contains the source node a. Then all nodes j in G_d such that there is an arc (a, j) in G_f are answers of the query. At the second step, U contains all nodes in G_u which have distance 1 from a. Let D be the set of all nodes j in G_d that are adjacent to some node of U in G_f, i.e., $D := \text{adj}\, G_f(U)$. Then all nodes in G_d that have distance 1 from some node in D are answers of the query. At the generic $(v + 1)$th step, U contains all nodes in G_u which have distance v from a. Let D be the set of all nodes j in G_d that are adjacent to some node of U in G_f. Again all nodes in G_d that have distance v from some node in D are answers of the query. The method terminates as soon as U becomes empty. Obviously, if the graph G_u is cyclic, the method is not safe. The method is presented in Figure 5.

4.2. *The Counting Method*

The counting method is described in [1, 7, 8, 3]. The method works as follows. Let U_v $(v \geq 0)$ contain all nodes j in G_u that have distance v from a (such sets are called *counting sets*). In the first phase, the method computes all nonempty U_v (notice that, in general, such sets are not disjoint). Suppose the U_s contains the nodes with the greatest distance (thus s) from a. In the second phase, we start computing the set D_s of all nodes j in G_f that are adjacent to some node of U_s in G_f. Then we compute D_{s-1} as the set of all nodes j in G_d that are adjacent to some node of U_{s-1} in G_f or that are adjacent to some node of D_s in G_d. We continue until we compute D_0, which contains all the answers of the query. As for the eager method, if the graph G_u is cyclic, the counting method is not safe. The method is presented in Figure 6.

$U_0 := \{a\}$; $v := 0$;
{1st PHASE: UP}
While $U_v \neq \emptyset$ **do**
begin
    $U_{v+1} := \text{adj}\, G_u(U_v)$; $v := v + 1$;
end;
$D_{v-1} := \text{adj}\, G_f(U_{v-1})$;
{2nd PHASE: FLAT and DOWN}
For $t := v - 1$ **downto** 1 **do**
    $D_{t-1} := \text{adj}\, G_d(D_t) \cup \text{adj}\, G_f(U_{t-1})$;
Answer $:= D_0$;

FIGURE 6. Counting method.

4.3. The Magic-Set Method

The magic-set method is described in [1, 9, 3]. The method works in two phases as follows. The first phase consists of determining all nodes in N_u (N_u is called the *magic set*). In the second phase, the method computes all possible pairs of nodes (i, j) such that i is in N_u, j is in N_d, and there is an answer path from i to j. To this end, we start from an arc (i, j) in A_f and compute all pairs $(\hat{i}, \hat{j})$ such that $(i, \hat{i})$ is in A_u and $(j, \hat{j})$ is in A_d. The so-obtained pair, in turn, is used to derive other pairs. The magic set is used to make this derivation more efficient. In fact, since the arcs of the query graph are actually computed by means of retrieving tuples from the database, it may happen that some pair $\hat{i}, \hat{j}$ is computed even though the node $\hat{i}$ is not in N_u (see, for instance, the node a_4 in Figure 1). The magic set forbids the use of $\hat{i}$ for deriving further arcs not in the query graph. Since a node in the magic set needs to be determined only once, independently of the number of its different distances from the source node, it turns out that the magic-set method is always safe. The algorithm is presented in Figure 7. Notice that, for efficiency,

$\hat{N}_u := \{a\}$; $N_u = \hat{N}_u$;
{COMPUTING THE MAGIC SET}
While $\hat{N}_u \neq \emptyset$ **do**
begin
    $\hat{N}_u := \text{adj}\, G_u(N_u) - N_u$;
    $N_u := \hat{N}_u \cup N_u$
end;
{COMPUTING THE ANSWER PATHS}
$\hat{E} := \{(i, j) \mid (i, j) \in A_f \text{ and } i \in N_u\}$; (a)
$sE = \hat{E}$;
While $\hat{E} \neq \emptyset$ **do**
begin
    $\hat{E} := (\{i, j) \mid (i, \hat{i}) \in A_u, (\hat{i}, \hat{j}) \in E, (\hat{j}, j) \in A_d\} - E$; (b)
    $E = E \cup \hat{E}$
end;
Answer $:= \{j \mid (a, j) \in E\}$.

FIGURE 7. Magic-set method.

both phases of the method are implemented using a differential approach (also called "seminaive"); as a consequence, two sets $\hat{N}_u$ and $\hat{E}$ are introduced to store, respectively, the new nodes and arcs that are generated at each step.

We note that the computation of additional pairs $\hat{E}$ in step (b) can be carried out by determining $\text{adj}^{-1} G_u(X)$ and $\text{adj}\, G_d(Y)$, where X and Y are the sets of source and target nodes, respectively, of the pairs in $\hat{E}$. We stress that such adjacent nodes are in general found using relational-algebra operations on the database [7]. This means that a pair $(i, \hat{i})$ can be retrieved that is not in A_u; recognizing whether such a pair is in A_u is done by checking whether i is in N_u. To avoid this situation, it is possible to compute new pairs by considering those arcs in G_u whose target nodes are the source nodes of arcs in $\hat{E}$.

5. COMPLEXITY ANALYSIS

In order to perform our analysis we distinguish different kinds of queries depending on the structure of the query graph G_Q.

Definition. Let Q be a 1-bound CSL query, and let G_Q be its query graph. Let G_u, G_f, G_d be the subgraphs of G_Q as defined before.

(i) Q is a *tree* if for each i in G_Q there is exactly one path from a to i;

(ii) Q is *regular* if G_Q is layered, i.e., for each i in G_Q, all paths from a to i have the same length;

(iii) Q is *acyclic* if G_u is acyclic;

(iv) Q is *cyclic* if it is not acyclic.

We are now ready to supply the worst-case complexity analysis of the three methods with respect to the different types of queries. To this end, we denote the numbers of nodes of G_Q, G_u, G_f, and G_d by n, n_u, n_f, and n_d, respectively. Accordingly, the numbers of arcs are m, m_u, m_f, and m_d. Moreover, all operations have unit costs except the union and difference (which have a cost linear in the size of sets involved), the adjacency operators $\text{adj}\, G(U)$ and $\text{adj}^{-1} G(U)$ [whose cost is $O(s)$, where s is the sum of all the outdegrees and the indegrees of nodes in U], and the operations for constructing pairs in statements (a) and (b) of the magic-set method, whose complexity will be explained later. We point out that computing a node of $\text{adj}\, G(U)$ or of $\text{adj}^{-1} G(U)$ is not an elementary operation, since it requires some complex retrievals from the database. However, since this operation appears in all methods, we may assume that it is the dominant cost unit. We observe that the magic set computes, at step (b), $\text{adj}\, G_d(U)$ and $\text{adj}^{-1} G_u(U)$ at the same time; so one could argue that the actual cost of such computations is not their sum, since the whole $\text{adj}\, G_d(U)$ (or part of it) can be determined from the computation of $\text{adj}^{-1} G_u(U)$ or vice versa. But this is not the case, since, by the definition of query graph, such computations involve different database relations (in fact, for any node S in the binding graph, L^S and L^{S^-} are disjoint).

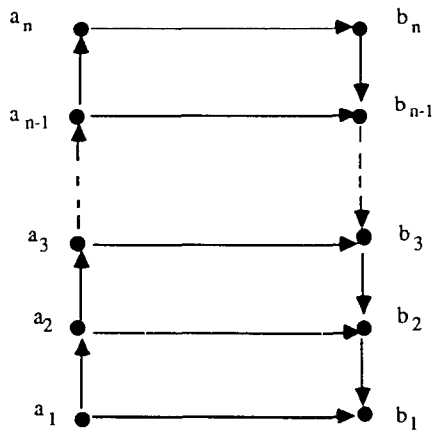

FIGURE 8.

5.1. *The Eager Method*

It is easy to see that in the case of a tree query the Eager method performs $O(m)$ operations.

In the case of a regular query the outer loop can be executed $O(n_u)$ times in the worst case. In turn, the overall cost of the operations within the inner loop is $O(m_d)$, since the query graph is layered, and thus no arc in G_d is handled twice.

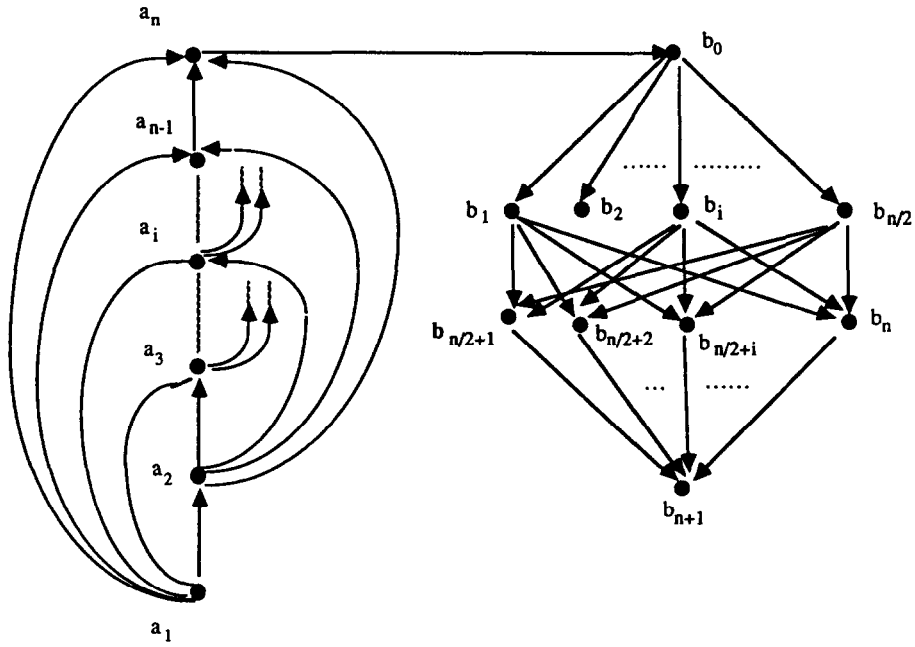

FIGURE 9.

Hence $O(nm)$ is an upper bound on the cost of the Eager method in the case of a regular query. The graph of Figure 8 shows that the above bound is tight.

In the case of an acyclic query, the outer loop can be performed $O(n_u)$ times. On the other hand, for every iteration of the outer loop (say for a given $\bar{v}$), the inner loop can be performed $O(\bar{v})$ times. In turn, every single execution of the inner loop may entail assessing $O(m_d)$ arcs. In sum, $O(n^2m)$ is an upper bound on the cost of the Eager method in the case of an acyclic query graph. Again, this bound is tight, as shown in Figure 9.

We recall that the eager method may loop forever in case of cyclic query graphs. Later in this paper, we shall show that it is possible to extend the method to handle cyclicity also.

5.2. *The Counting Method*

It is easy to see that in the case of tree and regular queries the counting method performs $O(m)$ operations.

In the case of an acyclic query, the first loop can be performed $O(n_u)$ times and every iteration has cost $O(m_u)$. Therefore, the total cost of the first loop is $O(n_u m_u)$. Similarly, the second loop has a total cost of $O(n_u m_d)$. Hence, the cost of the counting method for acyclic queries is $O(nm)$. This bound is actual, as shown in Figure 10.

Again, we recall that the counting method is not safe when the query is cyclic, although the method can be extended to deal with cyclicity, as shown later in the paper.

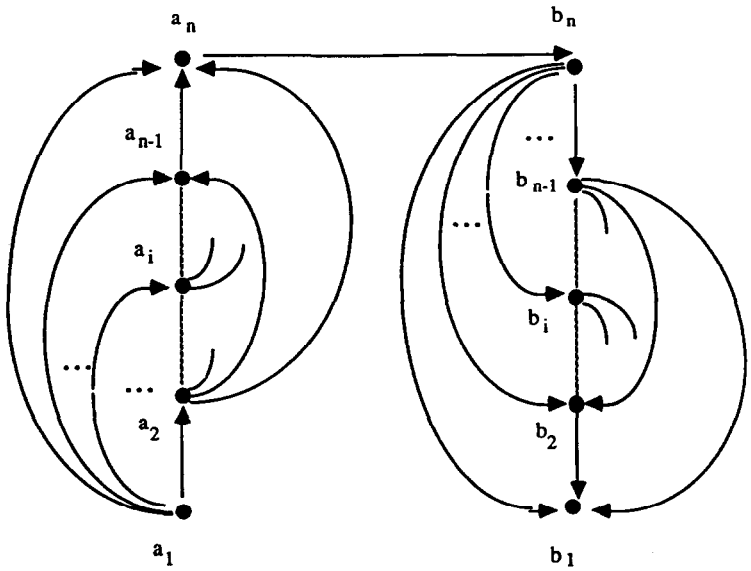

FIGURE 10.

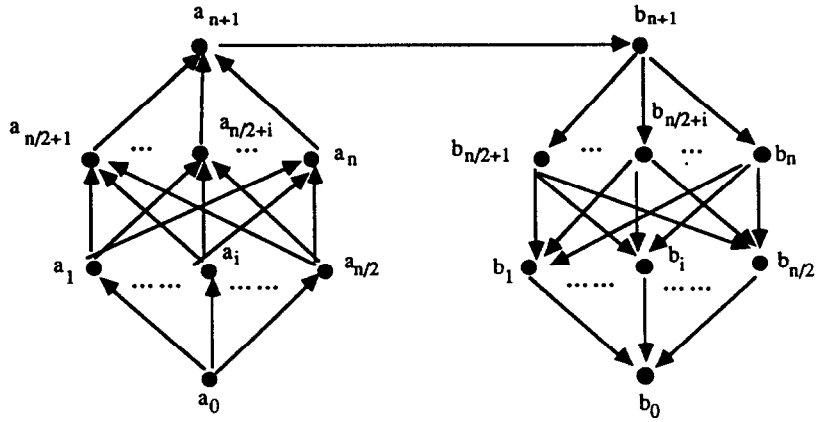

FIGURE 11.

5.3. The Magic-Set Method

We first note that the cost of the first loop as well as statement (a) does not depend on the type of query graph. Obviously, the cost of the first loop is $O(m_u)$, since we only need to perform a breadth-first search starting from the source node. On the other hand, it is easy to see that the cost of statement (a) is $O(m_f)$. The cost of the second loop does depend on the type of query graph and is analyzed next. Observe that a fundamental component of this cost is the analysis of the set operation in statement (b).

If the query is a tree, then the second loop can obviously be performed in $O(m_u + m_d)$ time, since each arc is considered once. In all other cases, for each pair (i, j) in $\hat{E}$, the cost of statement (b) is proportional to $|\text{adj}^{-1}\{i\}| \times |\text{adj}\{j\}|$. Hence, an upper bound of the total cost of the third loop is

$$\sum_{(i,j)\in E} \text{indegree}(i) \times \text{outdegree}(j) = O(m_d m_u).$$

Figure 11 shows that the above bound is tight for the case of regular queries and thus for acyclic and cyclic queries.

5.4. Comparison of Methods

We summarize the complexity analysis reported in Sections 5.1–5.3 in the next proposition.

Proposition 1. The costs of the three methods for the different kinds of queries are as shown in Table 1.

It turns out that the counting method gives better performance than the other two methods for all cases but cyclic queries. But we recall that the results are asymptotic and based on worst-case analysis; therefore, as shown in [1], there are cases where magic set works better than counting. However, we next show that this

TABLE 1 Costs of methods.

Query	Cost		
	Eager	Counting	Magic set
Tree	$\Theta(m)$	$\Theta(m)$	$\Theta(m)$
Regular	$\Theta(mn)$	$\Theta(m)$	$\Theta(m^2)$
Acyclic	$\Theta(mn^2)$	$\Theta(mn)$	$\Theta(m^2)$
Cyclic	Nonterminating	Nonterminating	$\Theta(m^2)$

cannot happen for tree or regular queries; in addition, we show that counting works better than eager for every query.

Proposition 2. Let Q be a query. Then

(a) *the cost of the counting method for Q is $O(E)$, where E is the cost of the eager method for Q;*

(b) *if Q is regular, then the cost of the counting method for Q is $O(M)$, where M is the cost of the magic-set method for Q.*

PROOF. (a): Let x be a node of the query graph, and $d(x)$ be the number of different lengths of paths in the query graph from the source node to x. If $x \in N_u$, then both the eager and the counting method process node x exactly $d(x)$ times. Hence the cost of both methods is the same. On the other side, if $x \in N_d$, then the costs of processing x by the eager and the counting methods are $\Omega(d(x))$ and $O(d(x))$ respectively.

(b): The magic-set method considers each arc of the query graph at least once. Therefore, part (b) follows by observing that, for regular query graphs, the counting method has cost linear in the number of arcs of the query graph. □

6. EXTENDING THE COUNTING AND EAGER METHODS

In this section, we present and analyze modified versions of both the counting method and the eager method, which are safe also for cyclic queries. The main idea is to set an upper bound on the value of the counter v that denotes the depth of the recursion (see the algorithms in Sections 4.1 and 4.2). In fact, the next results shows that there exists a value t, polynomially bound on the number of nodes in the query graph, such that the answers to the query can be found when the value of the counter u is less than t. Since this value can be determined while the algorithm is running, it follows that the iteration can be eventually stopped, so that the two methods can be made safe.

Theorem 3. Let G_Q be a query graph of a 1-bound query Q. If there is an answer path from the source node a to a node b, then there exists an answer path from a to b with length less or equal to $2n_u n_d + 1$.

PROOF. Consider any answer path from a to b. We represent this path as a sequence of nodes $\langle a_1, a_2, \ldots, a_k, b_k, \ldots, b_2, b_1 \rangle$, where $a_1 = a$, $b_1 = b$, the arc (a_k, b_k) is in A_f, and for each i, $1 \le i \le k-1$, the arcs (a_i, a_{i+1}) and (b_{i+1}, b_i) are

in A_u and in A_d, respectively. Suppose that $k > n_u n_d$ so that the answer path has length greater than $2n_u n_d + 1$. In order to prove the theorem, it is sufficient to show that there exists another answer path from a to b with length less than $2k + 1$. In fact, if this is the case, then we can prove the theorem by repeatedly deriving shorter and shorter answer paths from a to b until we get an answer path with length less than or equal to $2n_u n_d + 1$.

We construct an answer path from a to b with length less than $2k + 1$ as follows. First of all, we observe that, since the number of all pairs (c, d) with $c \in N_u$ and $d \in N_d$ is $n_u n_d$ and $k > n_u n_d$, there must exist two indices i, j, $1 \le i < j \le k$, such that $a_i = a_j$ and $b_i = b_j$. Consider now the following sequence of nodes:

$$\langle a_1, \ldots, a_{i-1}, a_i, a_{j+1}, \ldots, a_k, b_k, \ldots, b_{j+1}, b_i, b_{i-1}, \ldots, b_1 \rangle.$$

We have that, by assumption, (a_k, b_k) is in A_f and for each h, $1 \le h \le i - 1$ and $j + 1 \le h \le k - 1$, (a_h, a_{h+1}) and (b_{h+1}, b_h) are in A_u and in A_d, respectively. On the other hand, (a_i, a_{j+1}) is in A_u and (b_{j+1}, b_i) is in A_d, since $a_i = a_j$ and $b_i = b_j$ by construction, and (a_j, a_{j+1}) is in A_u and (b_{j+1}, b_j) is in A_d by assumption. Therefore, the above sequence represents an answer path from $a = a_1$ to $b = b_1$ with length less than $2k + 1$. This concludes the proof. □

Using Theorem 1, we can modify the algorithm of the counting method, shown in Figure 6, as follows. We compute the set U_u of the nodes in G_u with distance u from the source node, with $u \ge 0$. Then we compute the set D_u of the nodes that are adjacent to the nodes in U_u in G_f. Further, we compute the nodes that have distance u from the nodes in D_u. In doing so, we compute $D_{u-1}, \ldots, D_0$. However, since the same set of nodes D_v, $u > v \ge 0$, was already used for the previous step k, $v \le k < u$, the nodes of D_v previously exploited are not used any more. In this extension, the counting method can be considered as an efficient implementation of the eager method. The algorithm stops when u becomes greater than $n_u n_d$, where n_u and n_d are the numbers of nodes of G_u and G_d that have been currently

```
Answer := ∅;
U_0 := {a}; u := 0;
N_u := ∅; N_d := ∅;
Repeat
    N_u := N_u ∪ U_u;
    D_u := adj G_f(U_u);
    D̂_u := D_u;
    For i := u downto 1 do
    begin
        D̂_{i-1} := adj G_d(D̂_i) − D_{i-1};
        D_{i-1} := D_{i-1} ∪ D̂_{i-1};
        N_d := N_d ∪ D̂_{i-1};
    end;
    u := u + 1;
    U_u := adj G_u(U_{u-1})
until (U_u = ∅) or (u > |N_u| × |N_d|)
answer := D_0
```

FIGURE 12. Modified counting method.

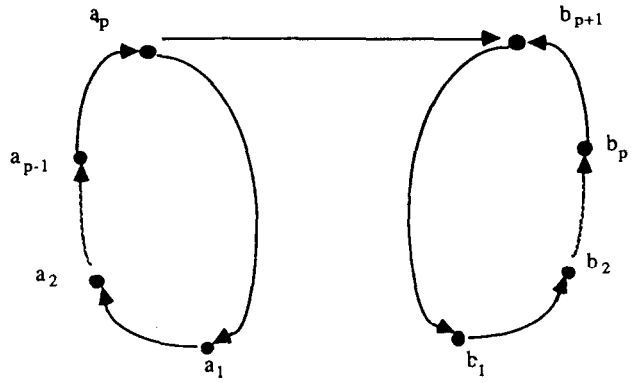

FIGURE 13. Query graph.

retrieved. Therefore, the modified counting method shown in Figure 12 is safe also for cyclic queries.

It is easy to see that $O(mn^2)$ is an upper bound on the cost of the modified counting method for cyclic queries. In order to prove that it is a tight upper bound, it is sufficient to show that it is not possible to change the termination condition $u > |N_u| \times |N_d|$ with a lower increasing function. To this end, consider the query graph of Figure 13, where G_u and G_d are two cycles, of length p and $p+1$, respectively. Clearly the node b_1 is in the answer, $|N_u| = p$, and $|N_d| = p+1$. Let $2d+1$ be the length of the shortest answer path from a_1 to b_1. We have that there are integers i and j such that $d = ip = j(p+1)$; hence $j = p(i-j)$. Since $i > j$, we have $j \geq p$. This implies $d \geq p(p+1) = |N_u| \times |N_d|$. Therefore, the algorithm works in $\Theta(mn^2)$.

We note that a different extension of the counting method for cyclic queries has been proposed in [5] that runs in $O(mn)$.

7. CONCLUSION

In this paper, we have introduced a simple class of logic queries, called 1-bound CSL queries, and we have shown that answering a 1-bound CSL query corresponds to finding particular paths in a graph associated to the query. Within this framework, three methods for implementing logic queries, namely, the eager method, the counting method, and the magic-set method, have been expressed in terms of graph algorithms. Therefore, using this simple computation model, it has been possible to perform an asymptotic worst-case analysis of the above methods. The main result is that the counting method gives the best performance for all cases but queries on cyclic databases, where the method does not even guarantee termination. The possible nontermination of the counting method represents a major obstacle to its effective use. In [10] this problem has been solved by combining the method with the magic-set method. In this paper, we have overcome the problem by introducing an extension of the method which behaves safely also with cyclic 1-bound CSL queries. An interesting open problem is whether the extended counting method can be used for larger classes of queries. Another area

of research is complexity analysis of other methods for query implementation (and, possibly, considering general queries).

APPENDIX

Before proving Theorems 1 and 2, we need some preliminary definitions and results.

Let LP be a logic program, and let D be a set of facts. Consider a fact $q(\mathbf{a})$, where $\mathbf{a}$ is a list of bound arguments. A *derivation tree* for $q(\mathbf{a})$ is defined as follows. Every leaf node of the tree is a fact in D. Every nonleaf node, say N, in the tree is a fact and is labeled by a rule in LP, say r, with P as head predicate and $P_1, \ldots, P_m$ as body predicates. The node N has children $N_1, \ldots, N_m$ and is *solved* in the rule r; thus, there is a substitution σ for all variables in r such that $P\sigma = N$ and $P_i\sigma = N_i$ $(1 \le i \le m)$. The substitution σ is called a *solving substitution* for N. The root of the derivation tree is $q(\mathbf{a})$.

It is easy to see that a fact $q(\mathbf{a})$ is inferred from LP $\cup D$ if and only if there is a (finite) derivation tree for $q(\mathbf{a})$.

Lemma 1. Let $Q = \langle G, \mathrm{LP}, D\rangle$ be a bound CSL query, and G be the query graph of Q.

(a) *If there are two arcs (n_1, n_3), (n_4, n_2) in A_u and A_d, respectively, and an answer path from n_3 to n_4, then there is an answer path from n_1 to n_2.*

(b) *If there is an answer path p from a node n_1 in N_u to a node n_2 in N_d with length $l > 1$, then there is an answer path of length $l - 2$ from n_3 to n_4, where n_3 and n_4 are the second and the $(l-1)$th node in the path p, respectively. Furthermore, say that $n_1 = [S_i, \mathbf{a}_1]$ and $n_3 = [U, \mathbf{a}_2]$; then $U = S_{i+}$.*

PROOF. Straightforward. □

Theorem 1. Let $Q = \langle G, \mathrm{LP}, D\rangle$ be a bound CSL query, and G_Q be the query graph of Q. If a tuple $\mathbf{b}$ is an answer of Q, then there exists an answer path from the source node $[S_1, \mathbf{a}]$ to the node $[S_1^-, \mathbf{b}]$ in G_Q.

PROOF. In order to prove the theorem, it is sufficient to show that, given any node $[S_i, \mathbf{a}_1]$ in N_u, if the fact $g(\mathbf{c}_1)$ is inferred from LP $\cup D$, where $\mathbf{c}_1(S_i) = \mathbf{a}_1$, then there is an answer path from $[S_i, \mathbf{a}_1]$ to $[S_i^-, \mathbf{b}_1]$ in G_Q, where $\mathbf{c}_1(S_i^-) = \mathbf{b}_1$. We prove the existence of such a path by induction on the number of recursive nodes (i.e., those with symbol g) in any derivation tree DT of $g(\mathbf{c}_1)$. We denote this number by s; furthermore, let σ denote the solving substitution for the root of DT.

Basis of the induction $(s = 1)$. Then the root $g(\mathbf{c}_1)$ of DT is labeled by a nonrecursive rule, say

$$r_q\colon g(\mathbf{x}_q) = C_q.$$

and the children of the root are the facts $P\sigma$ in D corresponding to predicates P in C_q. We have that $\mathbf{x}_q(S_i)\sigma = \mathbf{a}_1$ and $\mathbf{x}_q(S_i^-)\sigma = \mathbf{b}_1$. Therefore, since $[S_i, \mathbf{a}_1]$ is in N_u by assumption, the node $[S_i^-, \mathbf{b}_1]$ is in N_d and the arc $([S_i, \mathbf{a}_1], [S_i^-, \mathbf{b}_1])$ is in G_Q by the definition of query graph [see part (c) of the definition (Section 3.4)]. The above arc is an answer path.

Induction step. The theorem holds whenever the number of recursive nodes is less than s, where $s > 1$ (inductive hypothesis). Then the root $g(\mathbf{c}_1)$ of DT is labeled by the recursive rule, and its children are the fact $g(\mathbf{y})\sigma$ and the facts $P\sigma$ corresponding to the datum predicates P in the recursive rule. Therefore, for every predicate P in L^{S_i} or in $L^{S_i^-}$, $P\sigma$ is in D. Say that $g(\mathbf{y})\sigma = g(\mathbf{c}_2)$, where $\mathbf{c}_2(S_{i+}) = \mathbf{a}_2$ and $\mathbf{c}_2(S_{i+}^-) = \mathbf{b}_2$. We have that $\mathbf{x}_0(S_i)\sigma = \mathbf{a}_1$, $\mathbf{y}(S_{i+})\sigma = \mathbf{a}_2$, $P\sigma$ is in D for every predicate P in L^{S_i}, and $[S_i, \mathbf{a}_1]$ is in N_u by assumption. Hence, the arc $([S_i, \mathbf{a}_1], [S_{i+}, \mathbf{a}_2])$ is in A_u by the definition of query graph [see point (b) of the definition]. In addition, since the subtree of DT rooted at $g(\mathbf{c}_2)$ is a derivation tree with $s-1$ recursive nodes, by the inductive hypothesis there is an answer path from $[S_{i+}, \mathbf{a}_2]$ to $[S_{i+}^-, \mathbf{b}_2]$. Consider now the node $[S_{i+}^-, \mathbf{b}_2]$. Obviously, this node is in N_d; moreover, we also have that $\mathbf{x}_0(S_i^-)\sigma = \mathbf{b}_1$, $\mathbf{y}(S_{i+}^-)\sigma = \mathbf{b}_2$, and $P\sigma$ is in D for every predicate P in $L^{S_i^-}$. Hence, by the definition of query graph [see part (d) of the definition], the arc $([S_{i+}^-, \mathbf{b}_2], [S_i^-, \mathbf{b}_1])$ is in G_Q. It follows that, by Lemma 1(a), there is an answer path from $[S_i, \mathbf{a}_1]$ to $[S_i^-, \mathbf{b}_1]$. □

Lemma 2. Let Q be a 1-bound CSL query, and let $S_1, \ldots, S_k$ be the nodes of its binding graph.

(a) *The binding graph B_Q of Q is a cycle.*

(b) *If there is an answer path from a node $[S_i, \mathbf{a}_1]$ in N_u to a node $[U, \mathbf{b}_1]$ in N_d, then $U = S_i^-$.*

PROOF. (a): To prove this part of the lemma, it is sufficient to show that every node in B_Q has indegree equal to 1. Suppose not. Then there is a node S_j in B_Q with two incoming arcs, say (S_{j-1}, S_j) and (S_k, S_j). By definition of binding graph, the set of variables bound by S_{j-1} in r_0 equals the set of variables bound by S_k in r_0; thus $B_{S_{j-1}} = B_{S_k}$. Since $S_{j-1} \neq S_k$, there exists an argument in the head predicate of r_0 that is denoted by an index of S_{j-1} but not by an index of S_k. By assumption, all arguments of the head predicate of r_0 are variables, so the above argument is a variable, say X. Obviously, X is in $B_{S_{j-1}}$. But $B_{S_{j-1}} = B_{S_k}$; so the unbound head-predicate argument X is in B_{S_k}, and so the query is not 1-bound (contradiction).

(b): We proceed by induction on the length s of any answer path p leaving $n_1 = [S_i, \mathbf{a}_1]$ and entering $n_2 = [U, \mathbf{b}_1]$. If s is 1, then obviously $U = S_i^-$. Suppose now that $s > 1$. By Lemma 1(b) there is an answer path of length $s-2$ from n_3 to n_4, where n_3 and n_4 are the second and the $(s-1)$th node in the path p, respectively. Furthermore, $n_3 = [S_{i+}, \mathbf{a}_2]$. Therefore, by the inductive hypothesis, $n_4 = [S_{i+}^-, \mathbf{b}_2]$. By Lemma 2(a) there is only one node S_u such that $S_{u+} = S_{i+}$. Hence, $U = S_i^-$. □

Theorem 2. Let $Q = \langle G, \mathrm{LP}, D \rangle$ be a 1-bound CSL query, and G_Q be the query graph of Q. If there exists an answer path from the source node $[S_1, \mathbf{a}]$ to a node $[U, \mathbf{b}]$ in N_d, then the tuple $\mathbf{b}$ is an answer tuple of Q.

PROOF. By Lemma 2(b), $U = S_1^-$. In order to prove the theorem, it is sufficient to show that, given any node $[S_i, \mathbf{a}_1]$ in N_u, if there is an answer path from this node to a node $[S_i^-, \mathbf{b}_1]$ in N_d, say with length $2s+1$, then the fact $g(\mathbf{c}_1)$ is inferred from

LP $\cup D$, where $\mathbf{c}_1(S_i) = \mathbf{a}_1$ and $\mathbf{c}_1(S_i^-) = \mathbf{b}_1$. We carry out this proof by induction on s.

Basis of the induction ($s = 0$). The arc $([S_i, \mathbf{a}_j 1], [S_i^-, \mathbf{b}_1])$ is in A_f by the definition of answer path. By the definition of query graph [part (c)] there is a nonrecursive rule in LP, say

$$r_q\colon g(\mathbf{x}_q) :- C_q,$$

and a substitution σ such that $\mathbf{x}_q(S_i) = \mathbf{a}_1$, $\mathbf{x}_q(S_i^-) = \mathbf{b}_1$, and $P\sigma$ is in D for every P in C_q. It follows that a derivation tree for $g(\mathbf{c}_1)$ can be easily constructed. Therefore, $g(\mathbf{c}_1)$ is inferred from LP $\cup D$.

Induction step. The theorem holds whenever the length of an answer path is less than $2s + 1$ with $s > 1$ (inductive hypothesis). Since $s > 1$, by Lemma 1(b) there are two nodes, say $[S_{i+}, \mathbf{a}_2]$ and $[T, \mathbf{b}_2]$, such that the arcs $([S_i, \mathbf{a}_1], [S_{i+}, \mathbf{a}_2])$ and $([T, \mathbf{b}_2], [S_i^-, \mathbf{b}_1])$ are in A_u and A_d, respectively, and there is an answer path from $[S_{i+}, \mathbf{a}_2]$ to $[T, \mathbf{b}_2]$ with length $2(s-1) + 1$. By Lemma 2(b), $T = S_{i+}^-$. By the inductive hypothesis, the fact $g(\mathbf{c}_2)$, where $\mathbf{c}_2(S_{i+}) = \mathbf{a}_2$ and $\mathbf{c}_2(S_{i+}^-) = \mathbf{b}_2$, is inferred from LP $\cup D$. We now construct a derivation tree DT for $g(\mathbf{c}_1)$ as follows. The root is obviously $g(\mathbf{c}_1)$ and is labeled by the recursive rule r_0. A child of the root is $g(\mathbf{c}_2)$, which in turn is the root of a subtree coinciding with one of its derivation trees. [Note that at least one derivation tree exists for $g(\mathbf{c}_2)$, since this fact is inferred from LP $\cup D$.] The other children of the root of DT correspond to the database predicates in C_0 and are constructed as follows. Let σ_1 and σ_2 be two variable substitutions such that $\mathbf{x}_0(S_i)\sigma_1 = \mathbf{a}_1$, $\mathbf{y}(S_{i+})\sigma_1 = \mathbf{a}_2$, $P\sigma_1$ is in D for every P in L^{S_i}, $\mathbf{x}_0(S_i^-)\sigma_2 = \mathbf{b}_1$, $\mathbf{y}(S_{i+}^-)\sigma_2 = \mathbf{b}_2$, and $Q\sigma_1$ is in D for every Q in $L^{S_i^-}$ (such substitutions exist by the definition of query graph). Then there is a child $P\sigma_1$ for every P in L^{S_i} and a child $Q\sigma_2$ for every Q in $L^{S_i^-}$. By construction, in order to prove that DT is a derivation tree, we only need to show that there is a solving substitution σ for the root $g(\mathbf{c}_1)$ in the rule r_0. Let V, V_1, and V_2 be the domains of σ, σ_1, and σ_2, respectively. Obviously, $V = V_1 \cup V_2$ and $V_1 = B_{S_i}$, where B_{S_i} is the set of variables bound in r_0 by S_i. By the definition of 1-bound query, no variable occurring in V_2 appears in B_{S_i} or, therefore, in V_1. Hence, $V_1 \cap V_2 = \varnothing$. Therefore, we can set $\sigma(V_1) = \sigma_1$ and $\sigma(V_2) = \sigma_1$. It follows that σ is a solving substitution for the root of DT, and this concludes the proof. □

We want to thank Giorgio Ausiello for many inspiring discussions, and Jeff Ullman, who suggested a simpler proof of Theorem 3.

REFERENCES

1. Bancilhon, F., Maier, D., Sagiv, Y., and Ullman, J. D., Magic Sets and Other Strange Ways to Implement Logic Programs, in: *Proceedings of the 5th ACM SIGMOD-SIGACT Symposium on Principles of Database Systems*, 1986, pp. 1–15.
2. Bancilhon, F. and Ramakrishnan, R., An Amateur's Introduction to Recursive Query Processing Strategies, in: *Proceedings of the ACM SIGMOD Conference*, 1986, pp. 16–52.

3. Beeri, C., and Ramakrishnan, R., On the Power of Magic, in: *Proceedings of the 6th ACM SIGMOD-SIGACT Symposium on Principles of Database Systems*, 1987, pp. 269–283.
4. Henschen, L. J. and Naqvi, S. A., On Compiling Queries in Recursive First-Order Databases, *J. Assoc. Comput. Mach.* 31(1):47–85 (1984).
5. Haddad, R. W. and Naughton, J. F., Counting Methods for Cyclic Relations, in: *Proceedings of the 7th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, 1988.
6. Lloyd, J. W., *Foundations of Logic Programming*, Springer-Verlag, New York, 1984.
7. Saccà, D. and Zaniolo, C., On the Implementation of a Simple Class of Logic Queries for Databases, in: *Proceedings of the 5th ACM SIGMOD-SIGACT Symposium on Principles of Database Systems*, 1986, pp. 16–23.
8. Saccà, D. and Zaniolo, C., The Generalized Counting Method for Recursive Logic Queries, *Theoret. Comput. Sci.* 62:187–220 (1989).
9. Saccà, D. and Zaniolo, C., Implementation of Recursive Queries for a Data Language Based on pure Horn Clauses, in: *Proceedings of the 4th International Conference on Logic Programming*, Melbourne, 1987, pp. 104–135.
10. Saccà, D. and Zaniolo, C., Magic Counting Methods, in: *Proceedings of the ACM SIGMOD Conference*, San Francisco, 1987, pp. 49–59.
11. Ullman, J. D., *Principles of Database Systems*, Computer Science Press, Rockville, MD, 1982.
12. Ullman, J. D., Implementation of Logical Query Languages for Databased, *ACM Trans. Database Systems* 10(3):289–321 (1985).
13. Ullman, J. D., *Principles of Database and Knowledge-Base Systems*, Computer Science Press, Rockville, MD, 1988.